Implementing models of financial derivatives
Object Orientated Applications with VBA

NICK WEBBER
Implementing Models of Financial Derivatives
For other titles in the Wiley Finance series
please see www.wiley.com
Implementing Models of Financial Derivatives

Object Oriented Applications with VBA

Nick Webber
To clients of this book, may you enjoy it as much as I enjoyed writing it.
## Contents

Preface xv

**PART I A PROCEDURAL MONTE CARLO METHOD IN VBA** 1

1 **The Monte Carlo Method** 3
   1.1 The Monte Carlo valuation method 3
   1.2 Issues with Monte Carlo 8
   1.3 Computational issues 12
   1.4 Summary 16
   1.5 Exercises 16

2 **Levels of Programming Sophistication** 19
   2.1 What makes a good application? 19
   2.2 A high-level design 19
   2.3 Progressing towards the ideal 21
   2.4 Summary 22
   2.5 Exercises 22

3 **Procedural Programming: Level 1** 25
   3.1 Designing a Monte Carlo valuation application 25
   3.2 Deficiencies of the level 1 code 35
   3.3 Summary 36
   3.4 Exercises 36

4 **Validation and Error Handling: Level 2** 39
   4.1 Validation and error handling 40
   4.2 Encapsulating functionality 45
   4.3 The level 2 main() 48
   4.4 Summary 50
   4.5 Exercises 51
PART II  OBJECTS AND POLYMORPHISM

5  Introducing Objects: Level 3
   5.1  Objects in VBA 55
   5.2  An example: The StopWatch object 63
   5.3  Further helpful VBA features 65
   5.4  Objects in the Monte Carlo application 67
   5.5  Summary 78
   5.6  Exercises 78

6  Polymorphism and Interfaces: Level 4 81
   6.1  Polymorphism 81
   6.2  Interfaces in VBA 84
   6.3  Implementing a polymorphic stopwatch 86
   6.4  Polymorphism and the Monte Carlo application 88
   6.5  Assessment of the polymorphic design 100
   6.6  Summary 102
   6.7  Exercises 102

7  A Slice-Based Monte Carlo 107
   7.1  The revised Monte Carlo application object 107
   7.2  The option object 109
   7.3  The evolver object 113
   7.4  Summary 116
   7.5  Exercises 116

8  An Embryonic Factory: Level 5 119
   8.1  Events 119
   8.2  The Level 5 Monte Carlo application 122
   8.3  The Factory object 127
   8.4  Output 130
   8.5  Summary 133
   8.6  Exercises 133

PART III  USING FILES WITH VBA

9  Input and Output to File in VBA 137
   9.1  File handling in VBA 137
   9.2  The TextStream and FileSystemObject objects 138
   9.3  Intrinsic VB language functions 143
   9.4  Example: Reading and writing to sequential and random files 145
   9.5  Summary 151
   9.6  Exercises 151

10 Valuing a Book of Options 153
    10.1  Outline of the application 153
    10.2  Timings 174
PART IV POLYMORPHIC FACTORIES IN VBA

11 The VBE Object Library and a Simple Polymorphic Factory
   11.1 Using the VBE object library
   11.2 A simple factory illustration
   11.3 Summary
   11.4 Exercises

12 A Fully Polymorphic Factory: Level 6
   12.1 Conceptual features
   12.2 The polymorphic factory
   12.3 Using the Factory object
   12.4 Summary
   12.5 Exercises

13 A Semi-Polymorphic Factory: Meta-Classes
   13.1 The structure of the application
   13.2 Meta-class objects
   13.3 The semi-polymorphic factory
   13.4 Summary
   13.5 Exercises

PART V PERFORMANCE ISSUES IN VBA

14 Performance and Cost in VBA
   14.1 Arithmetic operations
   14.2 Procedure calls
   14.3 Data typing issues
   14.4 Summary
   14.5 Exercises

15 Level and Performance
   15.1 Variations of the level 0 application
   15.2 Effect of level on times
   15.3 Summary
   15.4 Exercises

16 Evolution and Data Structures
   16.1 Data structures in VBA
   16.2 Using VBA containers
   16.3 Numerical comparisons
   16.4 Summary
   16.5 Exercises
23.3 Numerical assessment 377
23.4 Summary 384
23.5 Exercises 386

PART VII  THE MONTE CARLO METHOD: CONVERGENCE AND BIAS 387

24 The Monte Carlo Method: Convergence and Bias 389
24.1 Reducing bias 389
24.2 Bias reduction methods 392
24.3 Bias and barrier options 396
24.4 Summary 398
24.5 Exercises 398

25 Discretization Methods 399
25.1 Discretization and convergence 399
25.2 Itô–Taylor discretization schemes 401
25.3 Schemes in 1-dimension 406
25.4 Predictor–corrector simulation 411
25.5 Numerical assessment for benchmark processes 413
25.6 Summary 416
25.7 Exercises 416

26 Applications to Models 417
26.1 The CIR process 417
26.2 Simulating discount factors 424
26.3 Summary 429
26.4 Exercises 429

27 Valuation in the Heston Model 431
27.1 Discretizing the Heston model 431
27.2 Convergence in the Heston model 435
27.3 Option valuation in the Heston model 436
27.4 Summary 446
27.5 Exercises 446

PART VIII  VALUING AMERICAN OPTIONS BY SIMULATION 447

28 Valuing American and Bermudan Options 449
28.1 American options 450
28.2 Monte Carlo and American options 455
28.3 Summary 461
28.4 Exercises 461

29 Estimating the Early Exercise Boundary 463
29.1 Approximating the continuation value function 463
29.2 Choices for basis functions 463
The purpose of this book is, as the title suggests, to acquaint the reader with the more advanced features of Visual Basic for Applications (VBA), and programming methods in general, in the context of numerical applications in valuing financial derivatives. Specifically it discusses error handling, objects and interfaces, file handling, events, polymorphic factories, design patterns and data structures and shows how they are used in Monte Carlo methods.

The context for the book is the reader who is developing applications from Excel and who does not have, or does not want, access to VBA outside that which accompanies Excel. Throughout, by “VBA” is meant VBA v6.X, implemented with Excel. This is accessible and widely used. VBA 2005, regarded here as a hybrid mixture of VB and C++, is not used, nor is VBA.Net.

VBA is one of the great standard tools of application implementation. Its ability to meld with Excel, and other Office applications, and its ability to facilitate extremely fast development, has led to its wide adoption even for serious applications. Here I am concerned chiefly with its ability to implement fast numerical methods for derivative valuation. Remarkably one finds that although it is slower than C++, it is not significantly slower.\(^1\) One can make a very strong case that the complexity of C++ overweights its speed advantage, and that VBA should be the routine vehicle of choice for numerical application design – except where speed really is the over-riding, dominant factor, and where very sophisticated C++ support (rather than just proficient and ordinarily sufficient levels of support) is available.

The reader is assumed to be familiar with the basics of VBA; procedures, declarations, logical structures, et cetera, and using VBA from within Excel, but perhaps not so familiar with objects in VBA.

Our topic is VBA for numerical applications, specifically the Monte Carlo numerical integration method. Our emphasis is thus very different from that of database or games designers who have their own priorities, distinct from ours. They may need to manage a large diverse range of objects, and be concerned with their interactions, just as we do, but the emphasis is different. Our objects come in a relatively small number of families, each with a distinct function within the application; there are things that do the doing and things that get done. There may be a large database of option specifications, but a relatively small number of objects with very particular functions within the valuation machinery. Computation is intense but of a qualitatively different sort to, for instance, image rendering.

This book has evolved over the years out of teaching material used for courses at the University of Warwick and at Cass Business School, and in practitioner courses. My own appreciation of VBA and my ability to use it effectively have developed together over this period.

\(^1\) The meaning of “significant” is a value judgement, but some timings are presented later.
RELATED READING

There are a number of good books on VBA. These include Kimmel et al. (2004), Green et al. (2007), Getz and Gilbert (2001) and Lomax (1998). Kimmel et al. and Green et al. are reference style books that are nevertheless written pedagogically. Kimmel et al. is written around Excel 2003 whereas Green et al., a later version, is for Excel 2007. Getz and Gilbert is an older book (it is based in Office 2000) but it emphasizes object-oriented VBA. Lomax is even older, but is still fresh and worthwhile.

VBA has been used in several books whose subject is financial derivatives of one sort or another. These include Jackson and Staunton (2001), Rouah and Vainberg (2007), Loeffler and Posch (2007) and Haug (2007). The emphasis in these books is more on the underlying models and applications, rather than on the effective use of VBA.

This book bridges the two categories. Like the more advanced VBA books it is object-oriented; like the derivatives books, it is about numerical methods applied to financial derivatives. There exist books such as Duffy (2004, 2007), Joshi (2004) and London (2004) that apply object-oriented C++ to derivatives pricing models. This book fills an analogous role to these for VBA, arguing, as we have indicated, that VBA should be considered as a competitive implementation language for a range of applications.

The focus in this book is on Monte Carlo methods although both lattice methods and PDE methods are touched upon. An excellent high-level treatment of Monte Carlo methods for derivative valuation is Glasserman (2004). Jäckel (2002) is less technical but is highly recommended; the author comes across as having been there and done that. Further good references are McLeish (2005) and Dagpunar (2007).

Finally, in a class of its own, I have to mention Numerical Recipes in C++ (Press et al., 2007). This book is a vade mecum for anyone in the numerics business. It is both a collection of coded numerical procedures and a textbook in its own right. The procedures it describes are widely applicable in many areas of science and computation, including those touched on here. Some of the more technical programs presented here adapt methods that can be found there. It is a strongly recommended buy for readers who wish to develop these aspects further.

STRUCTURE OF THE BOOK

This book is in eight parts. The first four parts focus on VBA. Each part introduces and discusses a new VBA feature and incorporates it into a developing, but plain, Monte Carlo application. The Monte Carlo method is used as a peg on which to hang some VBA. In stages, a simple procedural application is converted into a layered fully object-oriented application. Part I develops a very basic application. A simple procedural Monte Carlo method is constructed, and then error handling added in. Objects are introduced in Part II, including interfaces and run-time polymorphisms. Part III introduces files, demonstrating how the increasingly sophisticated application can input from file a book of options specifications and value them simultaneously. A polymorphic factory is constructed in Part IV.

Part V discusses performance-related issues, comparing, on the one hand, itty-bitty coding methods and, on the other, the costs of using the various built-in VBA data structures. It evaluates the performance of the Monte Carlo methods developed up to this point.

In the final three parts the focus is on the Monte Carlo application itself. The first of this group, Part VI, investigates a number of speed-up techniques, including stratified sampling, importance sampling, and the use of control variates. These are presented along with implementations and their effectiveness, alone and in combinations, assessed. Part VII looks at key practical issues linked by the concepts of convergence and bias. These include discretization, and option and model bias reduction methods. Finally, in a part to itself, valuation with the Longstaff and Schwartz least squares Monte Carlo method for American and Bermudan options is investigated.
A full set of appendices adds substantive material, including a discussion of lattice and PDE methods, a brief review of important root-finding methods, with implementations, and a primer on OOP.

In parallel with the exposition accompanying the development of the Monte Carlo application are a series of exercises. The reader is invited to develop a set of applications, several of which are presented first in appendices as low-level yukky applications, into high-level object-oriented structured applications. The applications are a simple trinomial application, a one-dimensional Crank-Nicolson PDE method, an implied volatility solver and an application to compute the value of \( \pi \). Building up these applications, shadowing the evolution of the Monte Carlo application, enables the reader to apply at first hand the techniques presented in the chapters, and to experience directly the challenging delights of coding high-level applications. How to program can be learned only by doing it, not by reading about it or by listening to lectures.

**ACKNOWLEDGEMENTS**

I would like to thank everyone who has contributed to the development of this book. These include my students – not only those who have taken my VBA courses but also those who have given me very valuable and detailed comments on its various drafts. In particular I am grateful to Kai Zhang and Pokpong Chirayukool for their thorough and careful reading of the manuscript, and their thoughtful suggestions. Part VII has benefited particularly from Kai’s comments and Part VIII from Pokpong’s suggestions. Between them they have corrected a large number of errors.

I would like especially to thank Alexandra Dias for reviewing the entire book as it was being written. Her constructive and insightful criticisms have been very greatly appreciated. Finally, I am grateful to the anonymous reviewers for contributing a set of very useful suggestions based on detailed readings of the manuscript. These have led to notable improvements in the book. Remaining errors and deficiencies are my responsibility.

Nick Webber
January 2010
This is an introductory part. Initial chapters introduce the Monte Carlo method in outline form, and discuss levels of program design.

Chapter 1 discusses the Monte Carlo method in abstract terms. It presents some of the mathematics lying behind the Monte Carlo methods that are later operationalized in code. It presents different evolution methods and data representation issues, but there is no actual coding.

Chapter 2 discusses issues in application design, setting the scene for the elaborations that follow. It briefly outlines the structure of an application that is developed through the first parts of the book.

In Chapter 3 we start to code up. This chapter constructs a purely procedural version of the Monte Carlo application. This has the properties of being utterly transparent but useless in practice; its faults are dissected and removed in subsequent chapters. Chapter 4 improves the application by introducing error handling. It also starts to move tentatively towards an object-oriented approach to programming by introducing a user-defined type to hold data in.

At this stage the application is still completely procedural. By the end of this part we will have gone about as far as it is sensible to go without using objects. Objects are introduced in Part II.
The Monte Carlo method is very widely used in the market as a valuation tool. It is used, through choice or necessity, with path-dependent options and in models with more than one or two state variables. It may be used in preference to PDE or tree methods, even in situations where these methods could work well, simply because of its generality and its robustness in contexts where a portfolio of options is being valued (rather than a single option at a time).

We start by rapidly reviewing the standard derivative valuation framework, and show how Monte Carlo works as a valuation method. Then we outline some of the factors that contribute to the design and implementation of a Monte Carlo valuation application. These are explored in greater detail as we progress through the book.


1.1 THE MONTE CARLO VALUATION METHOD

Suppose that in the market there is a European style option on an asset with value $S_t$ at time $t$, with payoff $H(S_T)$ at its maturity time $T$, for some payoff function $H : \mathbb{R} \to \mathbb{R}$. Write $O = (T, H)$ for this option. Suppose that the asset value is modelled as a stochastic process $S = (S_t)_{t \geq 0}, S_t \in \mathbb{R}^+$. For a European call option $O^c$ we have $O^c = (T, H^c_X)$ where $H^c_X = (S - X)^+$ for a strike price $X$.

The value $v_t$ of the option at time $t \leq T$ is given by the fundamental pricing equation (Harrison and Kreps (1979)).

$$v_t = \mathbb{E}_t \left[ H(S_T) \frac{P_t}{P_T} \right], \quad (1.1)$$

where $P = (P_t)_{t \geq 0}$ is the process followed by a numeraire $P_t$, and $\mathbb{E}_t$ takes expectations at time $t$ (with respect to an underlying filtration $\mathcal{F} = (\mathcal{F}_t)_{t \geq 0}$ of which little else will be said). Equation (1.1) assumes that processes are specified under the pricing measure with respect to $P_t$, so that $S_t/P_t$ is a martingale.

In this book we investigate simulation methods for computing (1.1), and are not so concerned with where (1.1) comes from. For instance, unless otherwise stated, we shall assume that processes are specified under the pricing measure, and we do not generally worry about change of measure or choice of numeraire.

In the Black–Scholes world, where the numeraire $P_t$ is the money market account, $P_t = \exp(\int_0^t r_s \, ds)$, and the short rate $r_t \equiv r$ is constant, equation (1.1) reduces to $v_t = e^{-r(T-t)} \mathbb{E}_t[H(S_T)]$. If, in addition, $S$ is a traded asset following a geometric Brownian motion (GBM) then under the pricing measure $\mathbb{P}$ associated with the numeraire $P_t$ its process is

$$dS_t = rS_t \, dt + \sigma S_t \, dz_t \quad (1.2)$$

for a Wiener process $z = (z_t)_{t \geq 0}$, where we have also assumed that the volatility $\sigma$ is constant. In this world the value $v_t^c$ of the European call option, $O^c$, is given by the Black–Scholes formula (Chapter 3, equation (3.2)).
More generally suppose there are $Q \geq 1$ underlying one-dimensional processes $S^q$, $q = 1, \ldots, Q$, and write $S = (S^q)_{q=1,\ldots,Q}$ for the $Q$-dimensional process they define. $S$ generates a filtration $\mathcal{F} = (\mathcal{F}_t)_{t \geq 0}$ on a sample space $\Omega$ where we can regard $\omega \in \Omega$ as representing a sample path for $S$ over an interval $[0, T_{\text{max}}]$ for some maximum time $T_{\text{max}}$. Write $S_t(\omega) = (S^q_t(\omega))_{q=1,\ldots,Q}$ for the value of $S$ at time $t$ in state $\omega$. We shall usually abbreviate this to $S_t$.

European options are determined by payoff functions $H$ defined on $\mathbb{R}^Q$. Let $O = (T, H)$ be a European style option written on $S$. The value $v_t$ at time $t$ of $O$ is

$$v_t = \mathbb{E}^P \left[ \frac{P_t(\omega)}{P_T(\omega)} | \mathcal{F}_t \right]$$

$$= \int_\Omega H(\omega) \frac{P_t(\omega)}{P_T(\omega)} dP$$

where $P$ is the risk-neutral measure on $\Omega$ corresponding to a numeraire $P$. Equation (1.3) rephrases (1.1) where we have written $H(\omega) \equiv H(S_T)$ and been more careful in exposing the dependence on $\omega$ of $P_t(\omega)$.

In practice, $H$ and $P_t$ will depend on $\omega$ only through a finite (and small) number of state variables observed at a discrete set of times $T = \{t_i\}_{i=0,\ldots,N} \subseteq [0, T_{\text{max}}]$ for some maximum time $T_{\text{max}}$.

**The Monte Carlo estimate**

Monte Carlo is a way of computing the integral (1.4). Suppose that for a domain $X \subseteq \mathbb{R}^Q$ we are given a suitably regular function $g : X \rightarrow \mathbb{R}$, and that we want to compute the integral

$$G(X) = \int_X g(x) \, dx.$$  (1.5)

Write $\mathcal{B}$ for the (Borel) measure on $\mathbb{R}^Q$ so that $\mathcal{B}(X)$ is the volume of a set $X \subseteq \mathbb{R}^Q$. The Monte Carlo integration method draws samples from $X$ uniformly under $\mathcal{B}$, taking $M$ draws $\{x_j\}_{j=1,\ldots,M}$, and constructs an approximation $\tilde{G}(X)$ to $G(X)$,

$$\tilde{G}(X) = \sum_{j=1}^M g(x_j) \Delta_{x_j}^M,$$  (1.6)

where $\Delta_{x_j}^M = \mathcal{B}(X)/M$ stands in for the volume element $dx$. As $M \rightarrow \infty$, $\tilde{G}$ converges to $G$. When the dimension $Q$ is large the Monte Carlo estimate $\tilde{G}$ is a computationally very efficient approximation to $G$.

The integral (1.4) has a structure slightly more specific than the general integral (1.5). It is an expected value of the form

$$G(X) = \int_X g(x) f(x) \, dx$$  (1.7)

for some density $f$, and for a European option

$$g(x) = H(x) \frac{P_t(x)}{P_T(x)}$$  (1.8)

where $x = S_T(\omega) \in X = (\mathbb{R}^+)^Q \subseteq \mathbb{R}^Q$. To investigate some consequences of this, suppose that there is a measure $\mathcal{F}$ on $X$ with distribution function $F$ and density $f(x)$ (which we presume exists) and, to start
with, suppose for simplicity that $Q = 1$. Consider $G^F(X) = \mathbb{E}_X^F[g(x)]$, the expected value of $g(x)$ under $F$ for $x \in X$. Set $U = F^{-1}(X)$. Then

$$G^F(X) = \int_X g(x) \, dF,$$

$$= \int_X g(x) f(x) \, dx,$$

$$= \int_U g(F^{-1}(u)) \, du.$$

(1.9)  
(1.10)  
(1.11)

Each of these three equivalent integrals can be approximated by a Monte Carlo integration:

<table>
<thead>
<tr>
<th>Integral</th>
<th>Approximation</th>
<th>Sampling distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\int_X g(x) , dF$,</td>
<td>$\sim \frac{1}{M} \sum_{j=1}^{M} g(x_j)$,</td>
<td>$x_j$ sampled under $F$</td>
</tr>
<tr>
<td>$\int_X g(x) f(x) , dx$,</td>
<td>$\sim \frac{B(X)}{M} \sum_{j=1}^{M} g(x_j) f(x_j)$,</td>
<td>$x_j$ sampled uniformly on $X$</td>
</tr>
<tr>
<td>$\int_U g(F^{-1}(u)) , du$,</td>
<td>$\sim \frac{B(U)}{M} \sum_{j=1}^{M} g(F^{-1}(u_j))$,</td>
<td>$u_j$ sampled uniformly on $U$</td>
</tr>
</tbody>
</table>

(1.12a)  
(1.12b)  
(1.12c)

One may either sample $X$ from the density $f(x)$ and compute the average of the $g(x)$, sample $X$ uniformly and compute the average of the $g(x)f(x)$ values or, equivalently, map on to $U \subseteq [0, 1]$ and integrate there.

When $Q > 1$ the integral and approximation in (1.12c) become a little more complicated. For $q = 1, \ldots, Q$ let $F_q$ be the $q$th marginal distribution function,

$$F_q(u) = \Pr[x_q \leq u].$$

(1.13)

Then under mild conditions $F(x_1, \ldots, x_Q) = C(F_1(x_1), \ldots, F_Q(x_Q))$ for a function $C : [0, 1]^Q \to [0, 1]$ called the copula of $F$. $C$ is a distribution function on $[0, 1]^Q$ with uniform marginals. In (1.12c) the integral becomes

$$\int_U g(F_1^{-1}(u_1), \ldots, F_Q^{-1}(u_Q)) \, dC$$

(1.14)

where $du$ is a volume element of $U \subseteq [0, 1]^Q$ and $(u_1, \ldots, u_Q) \in [0, 1]^Q$ is sampled under the distribution $C$.

Finally, the integral (1.4) can be approximated using (1.9), sampling $H(\omega)P_S(\omega)/P_T(\omega)$ under the measure $\mathbb{P}$. This means simulating $M$ sample paths $\{w_j\}_{j=1}^{M}$ for $S$ (under $\mathbb{P}$), computing

$$v_j = H(\omega_j)\frac{P_S(\omega_j)}{P_T(\omega_j)}, \quad j = 1, \ldots, M,$$

(1.15)

and taking the average of the $v_j$. 


6 Implementing Models of Financial Derivatives

This is essentially integrating using (1.12a), (1.12b) and (1.12c) can also be used. Using (1.12c) is called an inverse transform method.

Operationalizing this requires a number of approximations to be made. Fix a number of time steps \( N \), and a set of discretization times \( T = \{ t_i \}_{i=0}^{N} \), where \( 0 = t_0 < t_1 < \cdots < t_N = T \), and where we assume that \( \Delta t = t_{i+1} - t_i \) is a constant. Let \( \hat{S} = (\hat{S}_t)_{t=0}^{T} \) be a discrete \( Q \)-dimensional process, observed at times \( t_i \in T \), approximating \( S \). The Monte Carlo method implicitly determines the process \( \hat{S} \) through its choice of discretization method, and the discrete approximations \( \hat{H} \) and \( \hat{P} \) to \( H \) and \( P \).

Write \( \hat{S} = (\hat{S}_0, \ldots, \hat{S}_N) \) for a sample path of \( \hat{S} \), where \( \hat{S}_i \) is a realized value of \( \hat{S}_t \), so that \( \hat{S} \in \mathbb{R}^{Q \times (N+1)} \). We require

\[
\tilde{H} : \mathbb{R}^{Q \times (N+1)} \to \mathbb{R},
\]

\[
\tilde{P} : \mathbb{R}^{Q \times (N+1)} \to \mathbb{R},
\]

to approximate \( H \) and \( P \).

The Monte Carlo method generates a set of sample paths,

\[
\{ \hat{S}^j \}_{j=1}^{M} = \{ \hat{S}^j \}_{i=0}^{N}, \quad j=1, \ldots, M
\]

and approximates (1.4) by

\[
\hat{v}_t = \frac{1}{M} \sum_{j=1}^{M} \tilde{H}(\hat{S}^j) \frac{\tilde{P}_t(\hat{S}^j)}{\tilde{P}_T(\hat{S}^j)}.
\]

This is a path-by-path approximation. The set \( \hat{S}_t = (\hat{S}_t^1, \ldots, \hat{S}_t^M) \) is called the slice at time \( t_i \). It may be possible to compute (1.19) slice-by-slice instead of path-by-path. Where possible this may bring computational advantages, which are demonstrated later in the book.

The standard error

Since Monte Carlo is a probabilistic method the estimate \( \hat{v}_t \) in equation (1.19) has a distribution. The estimate \( \hat{v}_t \) should be unbiased, in that one hopes \( \mathbb{E}[\hat{v}_t] = v_t \), and efficient in the sense that, for any given \( M \), \( \text{var} [\hat{v}_t] \) should be as small as possible. The standard deviation of \( \hat{v}_t \) (or its sample estimate) is called the method’s standard error. Setting

\[
v^j = \tilde{H}(\hat{S}^j) \frac{\tilde{P}_t(\hat{S}^j)}{\tilde{P}_T(\hat{S}^j)},
\]

and assuming that successive \( v^j \) are independent, a sample estimate \( se(\hat{v}_t) \) for the standard error of \( \hat{v}_t \) is

\[
se^2(\hat{v}_t) = \frac{1}{M^2} \sum_{j=1}^{M} (v^j - \hat{v}_t)^2.
\]

As \( M \) increases \( se \) goes to zero with \( \sqrt{1/M} \). To construct a fast Monte Carlo method the aim is to get \( se \) small as quickly as possible. Speed-up methods are therefore also called variance reduction methods.
1.1.1 Example: A Black–Scholes European call option

A European call option is \( O^c = (T, H^X_c) \) where \( H^X_c = (S - X)^+ \) for a strike price \( X \) and \( S \in \mathbb{R}^+ \). In the Black–Scholes world, \( \tilde{P}_t(\tilde{S}) = e^{rt} \) so that

\[
\frac{\tilde{P}_t(\tilde{S}^j)}{\tilde{P}_T(\tilde{S}^j)} = e^{-r(T-t)}.
\]  

(1.22)

A Monte Carlo method generates \( M \) sample paths, \( \{\tilde{S}^j_0, \ldots, \tilde{S}^j_N\}_{j=1, \ldots, M} \), computes

\[
\tilde{H}^j = \tilde{H}^X_c(\tilde{S}^j) = (\tilde{S}^j_N - X)^+,
\]

and sets

\[
\tilde{v}_t = e^{-r(T-t)} \frac{1}{M} \sum_{j=1}^{M} \tilde{H}^j.
\]

(1.24)

1.1.2 Example: A knock-in barrier option

Suppose \( O^B = (T, H_B) \) is a knock-in barrier option with barrier level \( B \) on a single state variable following a GBM in a Black–Scholes world. Set

\[
\tau_B(\omega) = \min\{t \geq 0 \mid S_t(\omega) \leq B\}
\]

with value \( \infty \) if \( B \) is never hit. Suppose \( S_0 > B \) and let the payoff function be

\[
H_B(\omega) = (S_T(\omega) - X)^+1_{[\tau_B(\omega) \leq T]} \]

(1.26)

so that the option is a down-and-in call.

In this case one could set\(^1\)

\[
\tilde{H}_B(\tilde{S}) = (\tilde{S}_N - X)^+1_{[\tilde{r}_B(\tilde{S}) \leq T]}
\]

(1.27)

where

\[
\tilde{r}_B(\tilde{S}) = \min_{i=0, \ldots, N} \{t_i \mid \tilde{S}_i \leq B\}, \text{ or } \infty \text{ if } \tilde{S}_i > B \text{ for all } i.
\]

(1.28)

A Monte Carlo method generates \( \{\tilde{S}^j_0, \ldots, \tilde{S}^j_N\}_{j=1, \ldots, M} \), computes

\[
\tilde{H}^j = \tilde{H}_B(\tilde{S}^j) = \begin{cases} 
(\tilde{S}^j_N - X)^+, & \min\{\tilde{S}^j_0, \ldots, \tilde{S}^j_N\} \leq B, \\
0, & \text{otherwise,}
\end{cases}
\]

(1.29)

and sets

\[
\tilde{v}_t = e^{-r(T-t)} \frac{1}{M} \sum_{j=1}^{M} \tilde{H}^j.
\]

(1.30)

---

\(^1\) In practice a less naive, less biased, approximation would need to be used. For instance, in a GBM world a method based on El Babsiri and Noel (1998) could be used. We return to this in Part VII.
1.2 ISSUES WITH MONTE CARLO

In practice Monte Carlo is used to value and hedge a book of options with a model usually specified, like equation (1.2), as a set of SDEs. We briefly discuss the abstract structure of a Monte Carlo application, some practical considerations and some modelling aspects.

1.2.1 The structure of a Monte Carlo valuation

There are three components to the Monte Carlo valuation of a book of derivative securities.

1. The market component. This is the set of derivatives to be valued and the observables they are written on.
2. The model. This describes the way that state variables in the model evolve and the relationship between the state variables and the observables in the market.
3. The sampling mechanism. This specifies how, numerically, the SDEs followed by the state variables are evolved in discrete time.

Figure 1.1 illustrates the relationship between the three components. Each component, in its own way, is critical.

The model is expected to be able to recover the values of hedging instruments and to be sufficiently tractable to price a wide range of market products with some confidence. The sampling side is at the heart of getting a good distribution of values for the state variables. Finally, as a laudable instance of the dog wagging the tail, the market side is the raison d'être for the entire rigmarole.

The Monte Carlo method mediates between the sampling and modelling components by implementing a discretization of the SDE in the model. Similarly it connects the market and modelling components by integrating the one against the other.

The sampling component

The sampling side is purely mathematical and computational; it is independent of the financial model.

The output from the sampling side are increments to the drivers of the SDEs followed by the state variables of the model. Usually the distribution of the increments will be known, or at least be capable of being sampled. Whatever their distribution, these increments will be computed using some standard procedure from a set of uniform variates.

Uniforms sit at the bottom of a Monte Carlo procedure; they are its foundation, its bedrock. They are atomic in that (for our purpose) they cannot be decomposed into further components.

The model component

The model exists to service the needs of market participants and, insofar as there is a wide variety of needs, so there is a wide variety of models. There are HJM and market models, the SABR and Heston models, factor models and string models, diffusion models and Lévy process models, bridge distributions and time changes; some areas from time to time settle upon a market standard model but these change through time.

Models are usually specified in terms of SDEs driven, most generally, by Lévy processes. Sometimes the state variables are themselves asset prices or rates observed in the market. Sometimes they are not, so that values of market observables have to be extracted from the model. For instance in the fixed income market a 3-factor Gaussian affine model may enable the process followed by the short rate to be obtained. Unfortunately since the short rate does not exist in any practical sense, the values of assets that do exist,
Figure 1.1 The structure of a Monte Carlo valuation scheme
such as bond prices, need to be computed. In the case of a Gaussian affine model there are explicit formulae for their prices; in other factor models there are not and numerical methods must be used.

At some stage a set of SDEs has to be simulated. If the SDEs cannot be solved as functions of their drivers then some kind of discretization method will be needed to pass from the increments generated by the sampling side to sample paths of the state variables.

An important practical property that a state variable distribution should have to enable it to be implementable with a Monte Carlo method is that it be closed under convolutions. This means that increments to the variable add up to bigger increments within the same family of distributions. If this property did not hold then changing the length of a time step would cause the mathematics to change non-trivially.

The market component

The market throws out problems and challenges. If there is a demand in the market for a product then the modelling side had better keep up. The need to match a volatility surface has been a major impetus in the development of models in the fixed income and FX markets.

A derivative product specifies in its contract the relationship between its payoff and the values of observables in the market. Quite often the contractual details, although absolutely necessary to get right, are finicky. For instance, the computation of an average, or of a closing price, or indeed of a day count can be complex. Models usually abstractify away these inconvenient features with simplifying assumptions.

There is a limit on how far that can go before the effect becomes noticeable. Nevertheless we shall suppose that there is a simple relationship between the payoff to a derivative and the value of a market observable (or a series of values).

1.2.2 Practical requirements

From a practical viewpoint there are three vital ingredients to a Monte Carlo valuation system. These go significantly beyond the theoretical embodiment of the method in equation (1.19). A method must be able to calibrate, it must be possible to obtain hedge ratios, and it must be fast.

Calibration

Calibration is the name given to the procedure used to find parameter values for a model. It is usually done by requiring that parameter values be chosen so that some set of market prices, perhaps of liquid instruments used for hedging, be matched as closely as possible by model prices.

Calibration is primarily a property of the model, not the Monte Carlo method per se, but because Monte Carlo values are probabilistic they will not exactly equal model prices unless they are made to do so. In any case a decent model will have to recover the prices of the instruments that are used to hedge.

Under this heading also comes the requirement that instruments valued simultaneously should have prices consistent with one another. Arbitrage between prices must not be possible.

Hedging

Hedging is at least as important as valuation. Being able to get out hedge ratios is absolutely necessary, so calibrating accurately to the value of hedging instruments is vital. Usually their prices will be liquid.

2 But not emphasized in this book.
Sometimes, however, it is the availability of adequate and suitable hedging and pricing methods that increase the liquidity of a product in the market.

**Speed**

As a numerical integration method Monte Carlo works by generating a sample from the state space, computing the value of the integrand at each point in the sample, and taking the average. Computing the value of the integrand is usually not a problem; it is much harder to get a good sample of the state space. For valuing derivatives this means getting a good sample of paths (or slices) followed by the state variables in the valuation model.

Usually (but not always) from a model one is given directly, or obtains, a set of SDEs for the state variables in the model. The SDEs are normally driven by Lévy processes (although perhaps not time homogeneous.) Often the Lévy processes are just Wiener processes or jump-diffusion processes, but not always.

There are immediately two issues.

1. Given a sample path of the driving processes, how is a set of sample paths for the SDE obtained?
2. How in the first place is a sample path for the driving processes obtained?

The first issue is all about discretizing an SDE. One is given increments of the driving process and from them one has to manufacture increments to the SDE. Sometimes, for instance for a GBM, there is an exact solution to the SDE, so that the SDE can be sampled exactly; but usually there is not, and a discrete approximation has to be used.

The second is about obtaining samples from underlying distributions. This is usually straightforward although efficiency may be issue. Some less common distributions and related functions\(^3\) may not have cheap sampling methods.

In either case the important thing is to match a target distribution as closely as possible. In the first case this is the infinite dimensional sample space \(\Omega\). In the second it is, with any luck, a much nicer finite dimensional distribution – maybe even univariate normal. These issues are discussed at much greater length in Parts VI and VII.

**1.2.3 Modelling**

This section briefly mentions some aspects of the modelling component that affect the Monte Carlo method. It is not in the scope of this book to investigate a range of models in detail although some models are reviewed *en passant* at various points.

**Number of state variables**

A big advantage of Monte Carlo is that it is almost as easy to simulate many state variables as it is to simulate just one. Some other methods, such as lattice and PDE methods, suffer from dimensionality problems which prevent them from being used effectively with more than a very small number of state variables. This does not apply to Monte Carlo; it is a powerful practical motivation for the adoption of Monte Carlo as a valuation mechanism when realism, accuracy, or plain necessity, require more than one or two state variables to be present in a model.

\(^3\) For instance, at the time of writing the inverse of the beta distribution function.
Examples of situations where more than one state variable is needed include:

(1) instruments paying off on more than one observable;
(2) additional stochastic volatility factors introduced to enable a model to fit better to an implied volatility surface;
(3) a range of equity, FX and debt instruments where interest rate risk is significant and has be modelled alongside FX or default risk.

Classic examples include Libor market models where each forward Libor rate may be a separate state variable, or at least where a large number of drivers may be required to capture adequately the behaviour of the set of forward Libors. Here a Monte Carlo method is more or less essential, but difficulties can then arise when attempting to value options with early exercise features. See Part VIII.

**Realism and tractability**

Realism, in the sense of the ability to fit market data, is crucial, but comes at a cost. Often the cost is so great that practicality requires only an acceptable fit, for loose definitions of ‘acceptable’. Realism often implies complexity and complexity implies reduced tractability.

Heston, as a stochastic volatility extension of GBM, fits better to the implied volatility surface than plain GBM, often making it, in theory, the better model to use. Unfortunately it is a much harder model to implement in general than plain GBM. Specific issues with Monte Carlo include problems with discretization leading to a trade-off between bias and speed. The SABR model is used extensively, even though it may fit worse than Heston, simply because it is more tractable.

**Modelling observables**

Models have to calibrate to observable quantities, but their state variables need not be observable. For instance, there are both theoretical and practical advantages in using a Libor market model, in which the state variables are market observable forward rates, compared to a Gaussian affine term structure model in which state variables are abstract quantities. In a Gaussian affine model the values of observable quantities must be computed. The model loses a direct connection with what is being modelled, and with that it loses intuition. The main advantage of a Gaussian affine model is its tractability and range of applicability, but these are offset by its need to be calibrated to the market. Since its state variables are observable a LMM calibrates automatically – a huge advantage.

For Monte Carlo the issue is very pertinent. Having to calibrate by repeated expensive Monte Carlo valuations may be completely infeasible. In the Heston model semi-explicit formulae exist for vanilla products so that calibration is vastly simplified. Monte Carlo methods can then be used with a calibrated Heston model to value non-vanilla products.

1.3 **COMPUTATIONAL ISSUES**

A basic Monte Carlo method has been easy to describe and, as we see in Chapter 3, is very easy to implement. Of course it will run slowly, it is likely to be biased, or to have other issues with convergence, and in any case is likely to be limited to a specific option type.

The issues involved in making Monte Carlo run faster, run better, and run flexibly fall into two categories: issues with the method and issues with the implementation. Parts I to V look at implementation issues
and Parts VI and VII at issues with the method. For the moment we introduce some general ideas that elaborate on some of the issues raised in section 1.2.

### 1.3.1 The Monte Carlo method

The Monte Carlo method that forms the focus of Parts I to V is very basic; not only is a plain method relatively slow but it is also likely to be biased. Speeding-up the method involves generating a better sample of paths. Reducing bias involves improving the discretization method. Techniques to do this have a largely theoretical basis, founded in mathematics, developed in theorems, described in equations, and realized in code.

**The generating method**

The set of techniques available to speed-up convergence of a Monte Carlo method include fundamental methods such as the use of control variates and importance sampling. It also includes sampling techniques such as stratified sampling and using low discrepancy sequences.

**The discretization method**

Converting a description of asset price evolution in continuous time into a discrete time version is discretization. There is huge literature on this. A standard reference is Kloeden and Platen (1995). It is essential to use a discretization technique that avoids significant bias, and so converges to an unbiased estimate of the underlying continuous time solution. There is no point in applying speed-up techniques to a Monte Carlo method if you are converging faster towards the wrong solution.

### 1.3.2 Implementation issues

These issues are much more nitty-gritty. You have a theoretical method to hand, but how do you program it? There are issues at three levels; top-most design; intermediate level operational issues; and low-level data representational issues.

1. **Top-level design issues.** Numerical applications can be written at various levels of programming sophistication. We shall eventually arrive at a fully object-oriented design (or as full as seems expedient with VBA). The design is sketched in Chapter 2, and elaborated in most of the remaining chapters in the first four parts of this book.
2. **Intermediate level operational issues.** This is about the direction of evolution, evolution type, and storage requirements and type. We elaborate a little on this in section 1.3.3.
3. **Low-level data representational issues.** What structures are used to represent data in the implementation? See Chapter 16.

These issues are interrelated. Stratified sampling is implemented most effectively (with European style average rate options for instance) using a binary chop evolution direction. This, however, requires more intermediate data to be stored than either forwards or backwards evolution.

Backwards evolution must be used to value American style options. However using binary chop or backwards evolution requires a bridge discretization to be known; if it is not, then only forward evolution may be implemented.

In the remainder of this chapter we discuss a framework for intermediate issues.
1.3.3 Intermediate level issues

We assume that there is an evolver, a function that computes random draws from the conditional distribution \( \tilde{S}_{i+1} | \tilde{S}_i \). We denote this by \( \delta \), so that \( \delta(\tilde{S}_i) \) is a draw from the distribution \( \tilde{S}_{i+1} | \tilde{S}_i \). It is the method used to discretize \( S \) that determines \( \delta \), and hence determines the process \( \tilde{S} \).

Note that we do not assume that \( \tilde{S}_{i+1} | \tilde{S}_i = S_{t_{i+1}} | \tilde{S}_i \) in distribution. How close the equality holds largely determines the degree of bias in the discretization.

We discuss discretization methods in Part VII. For now we note that an example of a discretization method (and a very poor one) is the Euler method: given a \( Q \)-dimensional process \( dS_t = \mu(S_t) dt + \sigma(S_t) dz_t \) and a time step \( \Delta t_i = t_{i+1} - t_i \) it sets

\[
\hat{S}_{i+1} = \delta(\hat{S}_i) = \hat{S}_i + \mu(\hat{S}_i) \Delta t_i + \sigma(\hat{S}_i) \sqrt{\Delta t_i} \varepsilon_t
\]  

(1.31)

where \( \varepsilon_t \sim N(0, 1) \in \mathbb{R}^Q \) are normal IID increments.

From a computational perspective in computing a set \( \{\hat{S}_j\} \) there are three issues of importance:

1. What sort of data structure is being returned?
2. What is the direction of evolution?
3. What data is being stored?

From this viewpoint, whether a method is low discrepancy, or stratified, or Brownian bridge, or uses a control variate, is (in the language of OOP) an implementation detail; here we prefer to call it a Monte Carlo method detail.

1.3.4 Evolution type

In general, suppose \( X = (X_t)_{t \geq 0}, X_t \in \mathbb{R}^Q \), is a stochastic process of dimension \( Q \) with state space \( S = \mathbb{R}^Q \). Let \( T = \{t_i\}_{i=0, \ldots, N}, 0 = t_0 < \cdots < t_N = T_{\text{max}} \), be a set of discretization times and set \( x_i = X_{t_i} \) as usual. A Monte Carlo sample path is a vector \( \hat{X} = (\hat{X}_0, \ldots, \hat{X}_N) \), where \( \hat{X}_i = (\hat{X}_{i,1}, \ldots, \hat{X}_{i,Q}) \in \mathbb{R}^Q \) is the value of the discretized process at time \( t_i \), and \( \hat{X}_0 = X_0 \) is the initial value of the process.

Suppose that \( M \) sample paths are generated. Write \( \hat{X}_i^j, j = 1, \ldots, M, i = 1, \ldots, N \), for the value at time \( t_i \) of the discrete process along the \( j \)th sample path, and \( \hat{X}_{i,q} \) for the value of its \( q \)th coordinate, then

\[
\Xi = \{ \hat{X}_{i,j}^j \}_{i=0, \ldots, N, q=1, \ldots, Q} \subseteq \mathbb{R}^{M \times (N+1) \times Q}
\]

(1.32)

is the entire set of reals generated in the simulation.

\( \Xi \) is the set used to do the Monte Carlo numerical integration, but in implementing a method there is a great deal of choice in how \( \Xi \) is computed. A scheme determines how the set \( \Xi \) is sliced when constructing the Monte Carlo method: what is stored, what is evolved, what is returned.

For a (random) evolution operator \( \delta : \mathbb{R}^Q \rightarrow \mathbb{R}^Q, (\hat{X}_i) = \hat{X}_{i+1} \) on \( \hat{X} \), set

\[
\delta_i : \mathbb{R}^{i \times Q} \rightarrow \mathbb{R}^{(i+1) \times Q}, (\hat{X}_0, \ldots, \hat{X}_{i-1}) \mapsto (\hat{X}_0, \ldots, \hat{X}_{i-1}, \delta(\hat{X}_{i-1})),
\]

(1.33)

and define

\[
\delta_{(n)} = \delta \circ \cdots \circ \delta,
\]

(1.34)
to be the \( n \)-fold composition of \( \delta \), with \( \delta(0) = 1 \),

\[
\delta_N = \delta'_N \circ \cdots \circ \delta'_1 : \mathbb{R}^Q \to \mathbb{R}^{(N+1) \times Q},
\]

(1.35)

and

\[
\delta^M = (\delta, \ldots, \delta) : \mathbb{R}^{M \times Q} \to \mathbb{R}^{M \times Q},
\]

(1.36)

to be the extension of \( \delta \) to \( \mathbb{R}^{M \times Q} \).

We have \( \delta(n)(\hat{X}^j) = \hat{X}^{j \leftarrow n} \). \( \delta_N \) generates a sample path of \( \hat{X} \), and \( \delta^M \) moves a slice forwards by one time step.

There are broadly four ways of constructing \( \Xi \). These are to construct and return a sequence:

1. **Element-wise.** A single value at a time, \( \hat{X}_0, \hat{X}_1, \ldots, \hat{X}_N, \hat{X}_0^1, \ldots, \hat{X}_N^M \),

\[
\begin{align*}
\hat{X}_0^1 & \rightarrow \hat{X}_1^1 \rightarrow \cdots \rightarrow \hat{X}_N^1, \\
\hat{X}_0^2 & \rightarrow \hat{X}_1^2 \rightarrow \cdots \rightarrow \hat{X}_N^2, \\
& \vdots \\
\hat{X}_0^M & \rightarrow \hat{X}_1^M \rightarrow \cdots \rightarrow \hat{X}_N^M.
\end{align*}
\]

(1.37)

2. **Path-wise.** One path at a time, \( \hat{X}^1 \) to \( \hat{X}^M \), where \( \hat{X}^j = (\hat{X}_0^j, \ldots, \hat{X}_N^j) \) is the outcome of the \( j \)th application of \( \delta_N \) to \( \hat{X}_0 \),

\[
\begin{align*}
\hat{X}_0^1 & \rightarrow (\hat{X}_0^1, \hat{X}_1^1, \ldots, \hat{X}_N^1), \\
\hat{X}_0^2 & \rightarrow (\hat{X}_0^2, \hat{X}_1^2, \ldots, \hat{X}_N^2), \\
& \vdots \\
\hat{X}_0^M & \rightarrow (\hat{X}_0^M, \hat{X}_1^M, \ldots, \hat{X}_N^M).
\end{align*}
\]

(1.38)

3. **Slice-wise.** One slice at a time, \( \hat{X}_0, \ldots, \hat{X}_N \), where \( \hat{X}_i = (\hat{X}_i^1, \ldots, \hat{X}_i^M) = \delta^M(\hat{X}_{i-1}) \),

\[
\begin{pmatrix}
\hat{X}_0^1 \\
\hat{X}_0^2 \\
\vdots \\
\hat{X}_0^M
\end{pmatrix}
\rightarrow
\begin{pmatrix}
\hat{X}_1^1 \\
\hat{X}_1^2 \\
\vdots \\
\hat{X}_1^M
\end{pmatrix}
\rightarrow
\cdots
\rightarrow
\begin{pmatrix}
\hat{X}_N^1 \\
\hat{X}_N^2 \\
\vdots \\
\hat{X}_N^M
\end{pmatrix}.
\]

(1.39)

4. **Holistic.** \( \Xi \) as a lump: \( \Xi = \delta^M_N(\hat{X}_0) \), where \( \delta^M_N = (\delta_N, \ldots, \delta_N) \times 1^M : \mathbb{R}^Q \to \mathbb{R}^{M \times (N+1) \times Q} \) and \( 1^M : \mathbb{R}^Q \to \mathbb{R}^{M \times Q} \) is the diagonal operator \( X \mapsto (X, \ldots, X) \)

\[
\hat{X}_0 \rightarrow
\begin{pmatrix}
\hat{X}_0^1, & \hat{X}_1^1, & \ldots & \hat{X}_N^1 \\
\hat{X}_0^2, & \hat{X}_1^2, & \ldots & \hat{X}_N^2 \\
& & \vdots & \\
\hat{X}_0^M, & \hat{X}_1^M, & \ldots & \hat{X}_N^M
\end{pmatrix}.
\]

(1.40)

Element-wise evolution is the simplest but also the lowest level. This is not necessarily a bad thing, but if it means that an element-wise program is fixed into a mold that cannot later accommodate changes to the method or to the option being valued, then it is bad. Perhaps, not surprisingly, element-wise evolution
Implementing Models of Financial Derivatives

is inappropriate for more complex applications because of the overhead of shifting around individual numbers. It is more efficient to pass around a set of values as a slice or path.

There is nothing necessarily wrong with path-wise evolution. Conceptually this approach generates one history at a time. You can value your options in this history and then move on to the next. One objection against path-wise evolution is that it may always generate an entire path even if, for a knock-out option for instance, the option payoff maybe known before the final time. It does unnecessary computation; it is wasteful. As far as that goes, it is true. If the only option you had in your book was a single knock-out, then path-wise is not optimal for your purpose. However, in real life you do not have a single option: you have a book. The more options you have, even if they are all knock-outs of one variety or another, the more likely it is that for the book as a whole the entire path will be needed. By the time you get this far it becomes too awkward to keep track of whether you can stop evolving or not. You bite the bullet and generate an entire path at a time.

An alternative, but equally natural, conceptual approach is to generate values slice-wise. The simplest idea here is to move a slice forwards through time one step at a time. Now the concept is to look at alternative presents and to move forwards with these through time. Some methods (stratified sampling with a bridge for instance) do not go relentlessly forwards through time but move backwards and forwards generating values that fill up a sample path in non-chronological order. For these methods there are computational advantages in slice-wise evolution (although they may also work path-wise).

The holistic approach gives you a splodge of alternative worlds. Here is the multi-verse, take your pick. Again there is nothing wrong with this. Some methods (for example, some varieties of moment matching) require this approach. Of course the downside is that you have to store and return everything. For $M = 50000$, $N = 100$ and $Q = 3$ this is $15 \times 10^6$ Doubles. Not so bad these days, but increase $M$ or $N$ by too much and you are in trouble.

Given a choice of evolution type one still has to address (as we shall in Chapter 16) the lower level issue of how precisely sample paths or slices are to be stored.

All four evolution methods – element-wise, path-wise, slice-wise and holistic – are used at various points in the book. For instance, element-wise evolution is used in Chapters 3, 4 and 6; path-wise evolution is used, mostly for elegant variation, in Chapter 5 and again in Chapter 10; slice-wise evolution is used in Chapters 7, 8, 12 and 13; and holistic evolution is needed in some types of moment matching method where the completed sample is adjusted, post evolution, to ensure that it has certain properties (such as possessing the exact theoretically correct moments).

1.4 SUMMARY

We have introduced a number of ideas in this chapter. The basic Monte Carlo method has been described in mathematical terms, and some of the issues surrounding its implementation have been discussed. We explore these in much greater detail, and with much greater pragmatism, in subsequent chapters.

1.5 EXERCISES

In later chapters it is assumed that you are familiar with the basics of VBA so these exercises are designed to warm-up your VBA. Some exercises in future chapters build on solutions constructed here.

1. Implement the following formulae in VBA.

   (a) The Black–Scholes formula is given in equation (3.2), page 25. Write a Function, BlackScholes(), to compute the Black–Scholes formula. It should take $S_0$, $r$, $\sigma$, $X$
and $T$ as arguments. Although not usually recommended, for the moment you should use Application.NormSDist to compute values of the standard normal distribution function.\footnote{The spreadsheet LibraryProcedures.xls contains a Function normal_cdf() that computes the standard normal distribution function much more efficiently. See Appendix C.}

(b) Consider a down-and-out barrier call (DOC) option with maturity time $T$, strike $X$, and down-barrier level $H$, on an asset with value $S_t$ following a geometric Brownian motion under risk-neutrality with volatility $\sigma$ and riskless rate $r$. Let $Pv(x) = e^{-r(T-t)}x$ and $\nu = (2r/\sigma^2) - 1$. For $H < X$ the value $DOC_t$ of the option at time $t < T$ is given by

\[
DOC_t = S_t N(d_1(X)) - Pv(X) N(d_2(X)) - \left( \frac{H}{S_t} \right)^{\nu+2} S_t N\left( d_1\left( \frac{XS_t^2}{H^2} \right) \right) + \left( \frac{H}{S_t} \right)^{\nu} \left( \frac{H}{S_t} \right)^{\nu+2} S_t N\left( d_2\left( \frac{XS_t^2}{H^2} \right) \right),
\]

(1.41)

and for $H \geq X$ by

\[
DOC_t = S_t N(d_1(H)) - Pv(X) N(d_2(H)) - \left( \frac{H}{S_t} \right)^{\nu+2} S_t N\left( d_1\left( \frac{S_t^2}{H} \right) \right) + \left( \frac{H}{S_t} \right)^{\nu} \left( \frac{H}{S_t} \right)^{\nu+2} S_t N\left( d_2\left( \frac{S_t^2}{H} \right) \right).
\]

(1.42)

where $N$ is the standard normal distribution function and

\[
d_1(x) = \frac{1}{\sigma \sqrt{T-t}} \ln \left( \frac{S_t}{Pv(x)} \right) + \frac{1}{2} \sigma \sqrt{T-t},
\]

(1.43)

\[
d_2(x) = d_1(x) - \sigma \sqrt{T-t},
\]

(1.44)

(for instance, see Joshi (2003)). Write a Function, DOC(), to evaluate this formula.

Suppose that the Function is to be used in an application where it is evaluated many times for different values of $S$ and $T$ (but the same values of $r$, $\sigma$, $H$ and $X$). Write a version of DOC(), DOCfast(), optimized for performance in these circumstances.

You suspect that you will also be asked to implement the whole range of up-and-out, up-and-in, down-and-in, and down-and-out barrier call and put option valuation formulae. Look up formulae for these options (for instance, in Wilmott (1998) or Haug (2007)). To save yourself time in the future, how might you write DOC() now to make it easy to extend later? Is this sensible?

(c) Let $B_t(T)$ be the value at time $t$ of a pure discount bond maturing at time $T$ with value 1. Let $\tau = T - t$ be the time to maturity, $r_\infty$ a (constant) long rate, and $r_t$ the short rate at time $t$. In the Vasicek term structure model the value of $B_t(T)$ is $B_t(T) = \exp(-\tau r_t(T))$ where

\[
r_t(T) = r_\infty + (r_t - r_\infty) \frac{1 - e^{-\alpha \tau}}{\alpha \tau} + \frac{\alpha^2 \tau}{4\alpha^2} \left( \frac{1 - e^{-\alpha \tau}}{\alpha \tau} \right)^2
\]

(1.45)

with $r_\infty = \mu - \sigma^2/2\alpha^2$, for certain parameters $\alpha$, $\sigma > 0$ and $\mu$. 

\[
\]
You have an application that for some reason needs to compute equation (1.45) very frequently. Write a Function, taking \( r, \alpha, \sigma \) and \( \mu \) as arguments, that does this as cheaply as possible.

(d) A continuously compounded average rate call option with strike \( X \) and maturity time \( T \), starting at the current time \( t = 0 \), written on a geometric Brownian motion with initial value \( S_0 \), short rate \( r \) and volatility \( \sigma \), where the average \( a_t \) is computed geometrically,

\[
a_t = \exp \left( \frac{1}{T - t} \int_t^T \ln(S_u) \, du \right),
\]

has value \( A_t \),

\[
A_t = e^{-\delta(T-t)} S_t N(d_1) - e^{-r(T-t)} X N(d_2),
\]

where

\[
\delta = \frac{1}{2} \left( r + \frac{1}{2} \hat{\sigma}^2 \right),
\]

\[
\hat{\sigma} = \frac{1}{\sqrt{3}} \sigma,
\]

\[
d_1 = \frac{1}{\hat{\sigma} \sqrt{T-t}} \ln \left( \frac{e^{-\delta(T-t)} S_t}{e^{-r(T-t)} X} \right) + \frac{1}{2} \hat{\sigma} \sqrt{T-t},
\]

\[
d_2 = d_1 - \hat{\sigma} \sqrt{T-t}.
\]

Implement this formula as a VBA Function.

2. Suppose an option can have either a call payoff or a put payoff. The client specifies which by entering a code letter on a spreadsheet front-end. An application reads in the character but needs to validate it, establishing that it is acceptable. Write a utility Function, GetChar(), with signature

\[
\text{GetChar}(X \text{ As Long, Y As Long, valids As String}) \text{ As String },
\]

that reads in a String from cell \((X, Y)\) on the front-end and tests to see if it is a single character appearing as one of the acceptable characters in the String \(\text{valids}\). Test it for the case when acceptable characters are either “p” or “c” so that \(\text{valids}\) is the String "pc".

3. Write some code to test the user’s knowledge of the times-tables. The application randomly selects two integers, \(a\) and \(b\), in the range 1 to 12. It prompts the user with these who must then suggest a value for \(c = a \times b\). The application then prints a congratulatory message if the suggestion is correct and an encouraging message if it is correct only within epsilon. The code should be able to present more than one problem in sequence. Make sure that your interface could be used plausibly by a 5 year old.

---

\(^5\) For instance it might have to evaluate it repeatedly with different parameter values to calibrate to a market term structure.
Much of this part is concerned with programming techniques, exploiting VBA features, and assessing the damage or delight this causes to speed and clarity. In this chapter we look at a grand design.

### 2.1 WHAT MAKES A GOOD APPLICATION?

A number of factors contribute towards a good application. Of course the application must provide basic functionality, but it is equally important to recognize that the strength of an application resides not only in what it happens to be able to do at the moment, but also in how easy it is to adapt its functionality to changing requirements.

Possibly the most important design principle is that of decoupling. As far as possible the left hand of an application must not know what the right hand is doing; if so then we can change what the right hand is doing without changing the left hand. In a decoupled application the effect of any change is purely local. Even adding large chunks of functionality, if done polymorphically,\(^1\) will not cause anything else in the world to have to adapt to accommodate it.

### 2.2 A HIGH-LEVEL DESIGN

In a fully fledged numerical application there will be a succession of layers, each of which is responsible for some component of the application’s functionality. Figure 2.1 shows the structure we aim at in this book. It is Platonic; an ideal form whose shadow we may glimpse from time to time.

Solid lines represent predefined links hard-wired in. Dashed lines are links that can be set by other parts of the application, and dotted lines indicate those parts of the application that do the setting.

There are four layers. The top-most layer, the invoker, is the calling procedure, \texttt{main()}, that fires the application proper. It comes equipped with an error channel. In our case clicking a button on an Excel spreadsheet causes \texttt{main()} to run.

Next comes the first application layer. It reads in environmental data from elsewhere and has its own error channel. These links are shown as hard-wired, but they could be set by the invoker at the level above. The environment file contains settings for the application as a whole, for instance where to look for the specifics of the particular Monte Carlo method and its input/output channels.

The second application layer is a factory layer. It is responsible for creating the application itself, tailoring it according to specifications read in from locations given by the environment file.

Finally, the bottom layer is the actual Monte Carlo application. The factory sets the input and output streams and logging streams as required.

We examine these concepts in much more detail as we build up the application step-by-step in the first four parts of this book.

---

\(^1\) Concepts such as \textit{polymorphism} and \textit{encapsulation} are discussed later in the book.
The sequence of events that the application goes through when its button has been clicked are:

1. `main()` is run. It sets up an error channel and creates the application object.
2. As it is being constructed, the application object reads in settings from the environment file. It uses these to create the factory and to set a link between the factory and the Monte Carlo method specification.
3. The factory is asked to create the Monte Carlo application and to link it to its data input (option specifications) and output files.
4. `main()` requests the application object to `run()`; the application object asks the Monte Carlo application to `run()`.
5. The Monte Carlo application does its thing, reading in option data and outputting results.
6. Monte Carlo ends. Destructors are called for all objects as the stack unwinds, and `main()` ends.

Data read in to the bottom numerical application layer may require objects to be created. For instance, if option specifications are read in from file to be valued successively by the numerical method then option objects may need to be constructed. In this case the factory may not only set up a channel between an input file and the numerical application, but it may also set up a link to the factory itself.

At an even higher level, input data may determine which category of numerical method is used for a specific option, case by case. There is then a link between the in-file and the application manager layer.

The reason why this type of hierarchy is needed is to decouple as much as possible everything from everything else. By identifying and separating away bits of functionality, encapsulating it, and preventing other things from interfering with it, we end up with a stable yet resilient application. It may not be recognizable as the program we started off with but it will be moving towards industrial strength.

Decoupling is central to the development in the first part of this book. At every stage we look to see what are core responsibilities and what responsibilities can be delegated elsewhere. This entails a considerable degree of abstraction. One starts with a very concrete piece of numerics. Then one tries to work out what is really going on. Not only does this help the programming, it also illuminates the underlying mathematical theory.

### 2.3 PROGRESSING TOWARDS THE IDEAL

We work towards Figure 2.1 design in stages, distinguishing seven levels of sophistication in code design.

1. **Yukky.** Level 0: Not even a recognition of the existence of design criteria, let alone their implementation. May have just a single monolithic `main()` procedure.
2. **Basic procedural code.** Level 1: Avoids global variables but has no user-defined data types. Has user-defined functions but there is no validation or error handling. All code is in a single standard module. It has a `main()` that lies in the numerical application layer.
3. **Validation and error handling.** Level 2: Exploits error trapping. Code is split between more than one standard module, reflecting its purpose. User-defined data types may be used, but not objects. Code re-use is feasible. This level is intended to be the furthest it is possible to go without using objects. It still lives in the numerical application layer but now has an error channel.
4. **Non-polymorphic objects.** Level 3: Has objects to manage basic tasks, but the design is not polymorphic.
   - In VBA this means that interfaces are not used. It has an application manager object and may have an environment file.
5. **Polymorphic objects.** Level 4: This is polymorphic, which in VBA means it uses interfaces, but it does not exploit design pattern concepts.
6. **Embryonic factory.** Level 5: It has a factory whose job is to create objects needed elsewhere in the application. It has all four layers but is not fully decoupled, particularly for object creation.
7. **Design patterns.** Level 6: Uses design patterns to achieve, as closely as possible in VBA, a fully polymorphic application. Certain design patterns can be implemented, others merely mimicked. Its factory is fully polymorphic. The design begins to resemble Figure 2.1 with an environment file to tell the factory layer where to look to find the specification of the numerical application.

Further levels could be added but that is enough for us.
Input objects shall be polymorphic so other data sources could be substituted. Output shall be to either the front-end or to file. The specification of the numerical procedure contains information such as, for a Monte Carlo method, the number of time steps and sample paths to use. More generally the environment file could contain information on which of several factories is to be used and which numerical method (Monte Carlo, or PDE solver, or a lattice method) is to be employed.

2.4 SUMMARY

We have sketched a set of levels building up to a grand design. Over the next few parts we develop this succession of ever more sophisticated applications, following the levels sketched above. Each stage adds in another component vital to a decoupled high-level application. Although its packaging becomes quite sophisticated, the Monte Carlo method itself remains largely a toy application.

2.5 EXERCISES

These exercises continue the general VBA exercises started in Chapter 1.

1. The $m$th central moment $\mu'_m$ of a probability distribution, $m \geq 2$, is $\mu'_m = \mathbb{E}[(X - \mu)^m]$ where $\mu = \mathbb{E}[X]$ is the mean of the distribution. Given a set of values $\{x_i\}_{i=1,\ldots,N}$ generated from the distribution the empirically computed $m$th central moment $\hat{\mu}'_m$, $m \geq 2$, is

$$\hat{\mu}'_m = \frac{1}{N} \sum_{i=1}^{N} (x_i - \bar{\mu})^m$$

with $\bar{\mu} = (1/N) \sum_{i=1}^{N} x_i$. For each of the following distributions and generators write an application that: (i) generates $N$ successive values from the distribution; (ii) from the generated values computes the first $M$ empirical central moments of the series; (iii) outputs these values alongside the theoretic moments from the distribution.

Is it possible to assess whether the suggested method of generation is any good?

(a) How good is the built-in VBA random number generator Rnd? Use Rnd to generate variates from the distribution $U[a, b]$. Rnd generates uniform variates $U[0, 1]$ so variates from $U[a, b]$ can be obtained by rescaling the output from Rnd.

For $u \sim U[a, b]$, with mean $\mu = \frac{1}{2}(a + b)$, the $m$th theoretical central moment $\mu'_m = \mathbb{E}[(X - \mu)^m]$, $m \geq 2$, is

$$\mu'_m = \frac{(a - b)^m + (b - a)^m}{2^{m+1}(m + 1)}.$$  \hspace{1cm} (2.2)

(b) Consider the normal distribution $N(\mu, \sigma^2)$ with mean $\mu$ and variance $\sigma^2$. In VBA it is possible (but not recommended\(^2\)) to generate a sample $X$ from the normal distribution with mean $\mu$ and standard deviation $\sigma$ with the fragment

$$X = \mu + \sigma \times \text{Application.NormSInv(Rnd)}.$$  \hspace{1cm} (2.3)

\(^2\)A Function normal1() in the spreadsheet LibraryProcedures.xls generates normal variates much more efficiently. See Appendix C.
For the normal distribution function the $m$th theoretical central moment is $\mu'_m$, 
\[
\mu'_m = 0, \quad m \text{ odd}, \\
\mu'_m = \frac{(2n)!}{2^n n!} \sigma^{2n}, \quad m = 2n \text{ even. (2.4)}
\]

(c) Occasionally one hears a recommendation that to generate a standard normal variate one should first generate 12 standard uniforms $\{u_i\}_{i=1,\ldots,12}$ and return as a standard normal variate the quantity 
\[x = \sum_{i=1}^{12} u_i - 6.\] Does this work?

2. In the Merton (1976) jump-diffusion model the asset process $S_t$ follows an underlying geometric Brownian motion but may also occasionally jump. When a jump occurs $S_t$ jumps to a value $S_t \left(e^J - 1\right)$ where the log-proportional jump size $J \sim N(\eta, \nu^2)$ is normally distributed with mean $\eta$ and variance $\nu^2$. Jump times follow a Poisson process with intensity $\lambda$. Under risk neutrality the diffusion part of $S_t$ has drift $\mu = r - \lambda \kappa$ and volatility $\sigma$, where $\kappa = \mathbb{E}[e^J - 1] = e^{\eta + \frac{1}{2} \nu^2} - 1$.

It is possible to value vanilla options in this model. Let $c^M_t$ be the value at time $t$ of a Merton European call option with strike $X$ and maturity time $T$. Write $BS_t(S, r, \sigma)$ for the value of the standard Black–Scholes formula for a European call with strike $X$ and maturity time $T$, with asset value $S$ at time $t$, riskless rate $r$ and volatility $\sigma$. Define $r_n$, $\sigma_n$ and $\lambda'$ as
\[
\begin{align*}
    r_n &= r - \lambda \kappa + n \frac{\eta + \frac{1}{2} \nu^2}{T - t}, \\
    \sigma_n^2 &= \sigma^2 + n \frac{\nu^2}{T - t}, \\
    \lambda' &= \lambda (1 + \kappa),
\end{align*}
\]
then $c^M_t$ is given by an infinite series
\[
c^M_t = \sum_{n=0}^{\infty} e^{-\lambda'(T-t)} \frac{\left(\lambda'(T-t)\right)^n}{n!} BS_t(S_t, r_n, \sigma_n).
\]

Implement this formula (truncating it at some level). How rapidly does the formula converge in $n$? How rapidly does it converge in execution time?
This page intentionally left blank
This chapter builds a very simple Monte Carlo application to value a European call option in the Black–Scholes framework. The first objective is unprepossessing but it will generalize to more useful applications. It is assumed that the reader is familiar with the basics of VBA but may be uncertain how to put together an application.

The Monte Carlo method introduced here, and embroidered in future chapters, is very basic. More elaborate, faster, Monte Carlo techniques will be introduced and implemented later in the book.

There is a single state variable $S_t$. In the Black–Scholes framework $S_t$ follows a geometric Brownian motion with SDE

$$dS_t = rS_t \, dt + \sigma S_t \, dz_t$$

(3.1)

where $r$ is the risk-free instantaneous short rate, $\sigma$ is volatility of returns, and $z = (z_t)_{t \geq 0}$ is a Wiener process. The value $c_t$ at time $t$ of a vanilla European call option maturing at time $T$ with strike $X$ is

$$c_t = S_t N(d_1) - \text{PV}(X) N(d_2)$$

(3.2)

where

$$\text{PV}(X) = e^{-r(T-t)} X,$$

(3.3)

$$d_1 = \frac{1}{\sigma \sqrt{T-t}} \ln \left( \frac{S_t}{\text{PV}(X)} \right) + \frac{1}{2} \sigma \sqrt{T-t},$$

(3.4)

$$d_2 = d_1 - \sigma \sqrt{T-t},$$

(3.5)

and $N$ is the standard normal distribution function. We usually assume that valuation takes place at time $t = 0$.

### 3.1 DESIGNING A MONTE CARLO VALUATION APPLICATION

When building a fresh procedural application in VBA the initial sequence of construction tasks is to

1. create a front-end;
2. create a main stub with I/O, and a button to run it;
3. build a program outline.

Once this has been done the main implementation work can start, but the first three steps are necessary prerequisites. We construct a Monte Carlo application to value a European call option when the underlying asset follows a GBM. The completed application is MC_project_v1 in the spreadsheet MC_example_v1.xls.

---

1 We assume that the reader is familiar with the basic finance. A standard reference is Hull (2008).
**Creating a front-end**

This is the first step. The usefulness of first creating a front-end is that (i) it clarifies exactly the inputs and outputs to the application from the front-end and (ii) input and output have specific locations to refer to.

In this case there are three categories of input data. Firstly, the asset data: the initial asset value $S_0$, the riskless rate $r$, and the volatility $\sigma$; secondly, the option data: the exercise price $X$, and the time to maturity $T$; and finally the method data: the number of time steps $N$, and the number of sample paths $M$.

Three numbers are output at the end of a run: the option value, $c$, its standard error, $se$ (which must always be reported for a Monte Carlo method), and the time taken to compute the result. A counter is output as the run progresses.

Figure 3.1 shows a typical front-end. Boxes are used to provide structure to the front-end, grouping together related items. It is clear where each item of input and output belongs. The cost of prettifying the front-end is slight, but it massively enhances user-friendliness and usability.

The option parameter values shown in the figure are not unreasonable, and neither are the Monte Carlo parameter values, $N$ and $M$. Of course in this case the exact solution to the SDE (3.1) is available, and the option cashflows occur at a single predetermined time, so we could use long-step Monte Carlo with $N = 1$. But just to illustrate the method: we don’t.

**Creating a stub**

The entry `Sub`, here and elsewhere, is called `main()`. The next step is to insert a standard module into the project and create a stub `main()` procedure. A button will be created on the front-end. Clicking it will run the application.

Figure 3.2 shows the stub. Banner comments are put in from the beginning. These mark the start and end of the code module, separate out functions, and split the code into chunks.\(^2\) A button can now be

---

\(^2\) In later examples, to save space on the printed page, we are likely to abbreviate banner comments from five lines down to three lines, or to one line separators, or drop them altogether. In almost all cases full banners are retained in the code itself.
Option Explicit

Public Sub main()
End Sub

Option Explicit

Figure 3.2 Monte Carlo application: the main() stub

Figure 3.3 Adding in a button

added to the front-end and attached to main(). Figure 3.3 shows the code run by the button. It just calls main(). Here the standard module has been named MC_example_v1.

Every module will of course have Option Explicit set, so that variable declaration is required. To save space in illustrations presented in the text we shall usually drop this line from the figures – but it is always there in the code.

For a description of how to do this please refer to books such as Kimmel et al. (2004) or Green et al. (2007), and see a brief description in Appendix F.
Add in the basic I/O

Before adding in the substantive Monte Carlo functionality, code for I/O can be inserted into `main()`. Figure 3.4 shows the additional code. Each variable is declared and given its value on the same line. The advantage of first constructing the front-end is clear: there are concrete locations to read in from and write to. What are currently dummy variables are declared for the option value, `val`, and its standard error, `se`.

From the beginning an attempt is made to make the code pretty. This is not mere aesthetics but concrete pragmatism; code that is easy to read is likely to be easier to maintain.

Note the use of indenting. Indenting will of course be adopted as a programming standard. For anything that has an inside, those insides will be indented.

Even in these few lines of code there is a lot wrong. There is no validation; any data in input cells of the wrong type, or outside a valid range, will break the application. This serious defect will begin to be addressed in the level 2 version of the code.

Variables are usually declared and initialized on the same line (as far as is sensible). It is a standard programming paradigm to declare variables only when they can be initialized. Historically, some languages required variable declaration to take place only at the start of blocks of code. This is no longer necessary in VBA and should be avoided. Declaring and initializing in-line, using the statement separator `:`, is both clear and compact. Since we make extensive use of :, a short digression on its properties may be helpful.

The : statement separator

The : separator has two related uses: (1) as a label delimiter and (2) as a statement separator. Figure 3.5 illustrates its various uses. On line 3.5a is a `GoTo` statement that, when executed, causes control to branch to line 3.5b, starting with the `EvilLabel: label:` label. The `: label:` is used to delimit the label; what follows after, on that line or the next, is the code to be executed.

The second use as a statement separator is illustrated on line 3.5c. This declares in-line the variable `S_0` and then sets its value. Care needs to be taken, however, in situations like 3.5d. Here the statement `a = b` is executed only if `done` is `True`; it is part of the `If` statement, not part of the line following the `If` statement.

---

4 Or rather, assigned initial values. Unlike C++, variables are initialized when they are created only with default values.

5 This is just to illustrate :. `GoTo` should never be used (except in error handling when there is no choice).
Procedural Programming: Level 1

[Code example]

'XXXXXXXXXXXXXXXXXXXXX First example: Label delimiter XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
If done Then GoTo EvilLabel 'a. Never do this. Jumps to the label
'Stuff
EvilLabel: Call SomeSub 'b. Execution branches to here

'XXXXXXXXXXXXXXXXXXXXX Second example: Statement separator XXXXXXXXXXXXXXXXXXXXXXXXXXX
Dim S_0 As Double: S_0 = Cells(12, 6).Value 'c. Statement separator

'XXXXXXXXXXXXXXXXXXXXX Third example: Beware XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
If done Then b = 3: a = b 'd. a = b is part of the If statement

'XXXXXXXXXXXXXXXXXXXXX Fourth example: Select Case XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Select Case TypeChar
Case "c": Call PayoffCall 'e. In-line Case delimiter
Case "p": Call PayoffPut
End Select

Figure 3.5 Using the : separator

The final example shows : again being used as a statement separator. Here it is used in Select statement to separate a Case from the line executed in that Case. This is useful when each Case involves only a single statement.

Cell coordinates or range names?

The code in Figure 3.4 refers to cells by their coordinates. It would also be possible, and very reasonable, to endow cells with range names and to input and output from and to named locations. For instance, cell (12, 6) could be given the name S0. Line 3.4a could then be replaced by

Dim S_0 As Double: S_0 = Range("S0").Value '.

The advantage of doing this is that the client could rearrange the front-end without needing to alter the code in main(). Of course the code still needs to know the name that the cell goes by, but overall the front-end and the code have greater decoupling than is the case when coordinates are used explicitly.6

The spreadsheet MC_example_v0_names.xls is a version of the application built in this chapter that uses range names instead of cell coordinates. In general, as a coding convention, we continue to use coordinates rather than names.

Put in outline code for generating sample paths

We return to the Monte Carlo example. The next step is to add in code for generating sample paths. First insert at position (a) in Figure 3.4 the code in Figure 3.6. This is a stub for iterating through N steps along M sample paths. The For and Next statements are inserted before filling in the contents of the loop. The loop variables i and j are declared just before the points at which they are used.7

Now the code to perform the sample path evolution can be added in. At position (b) in Figure 3.6 insert the declaration-initialization

Dim path_S As Double: path_S = S_0

and at position (c) the line

path_S = next_S(path_S)'.

---

6 I am grateful to an anonymous reviewer for pointing this out.
7 By default For loops increment in steps of 1. We discover in Chapter 14 that loops run faster with an explicit Step. We ignore this for now.
path_S holds successive values generated along each path. It is initialized to $S_0$ at the start of each path.

next_S() is a function to compute the next value along the sample path. This, for the moment, is just a placeholder. We know that we shall need code to perform this task, but we can defer until later the implementation of the required functionality. Later, when we return to it, we shall also settle on the exact arguments that next_S() needs.

This is a common programming device. Functionality to evolve sample paths is required but we do not yet know what it is to be – so we give it a name and a placeholder and come back afterwards to fill in the implementation.\(^8\)

**Adding in code to compute payoffs**

At position (d) in Figure 3.6 insert the line

\[
\text{Dim payoff As Double: payoff = Compute_PO(path_S, X)}.
\]

payoff holds the payoff value which is computed with the function Compute_PO(). This function is so simple and unambiguous that it can be added immediately. Figure 3.7 shows the form it takes for the payoff to a European call. It calls the function my_max() that returns the maximum of two Doubles. Of course it is far cheaper to supply one’s own utility function than to call the application function Application.Max().

\[
\begin{align*}
\text{Private Function Compute_PO(S As Double, X As Double) As Double} \\
\text{Compute_PO} = \text{my_max}(0, S - X) \\
\text{End Function} \\
\text{Private Function my_max(a As Double, b As Double) As Double} \\
\text{If a > b Then my_max = a Else my_max = b} \\
\text{End Function}
\end{align*}
\]

\(^8\) This is a low-level example of the methodology of “programming to an interface”. This is explored more fully later.
Both data and procedures may be Public, with visibility throughout any open project; Friend, with scope restricted to their project; or Private, with scope confined to their own module. Here and elsewhere all procedures are declared with the narrowest possible scope. This means Private whenever possible; we prefer Friend to Public and Private to Friend. Compute_PO() is declared as Private since it has no relevance outside the module in which it is defined. my_max() is also declared to be Private, but as a useful utility function it will soon be put into a utility library module as a Public function.

Here again it is far better to assign the task of computing the payoff to a function (despite the overhead of a function call) instead of directly writing in the code to compute the payoff. This encapsulates functionality that could subsequently change.

**Computing the option value and standard error**

Although we do not yet know exactly how sample paths will be generated, they shall indeed be generated and their payoffs computed. Write \( p_j, j = 1, \ldots, M \), for the payoff along the \( j \)th sample path, computed in line (3.9). Set

\[
 s_1 = \sum_{j=1}^{M} p_j, \quad s_2 = \sum_{j=1}^{M} p_j^2, \tag{3.10}
\]

then the Monte Carlo estimates \( \hat{c} \) of the option value \( c \) and the standard error \( se \) of \( \hat{c} \) are

\[
 \hat{c} = e^{-rT} \frac{1}{M} s_1, \tag{3.11}
\]

\[
 se = e^{-rT} \frac{1}{M} \sqrt{s_2 - \frac{1}{M} s_1^2}. \tag{3.12}
\]

In the Monte Carlo application two variables, acc_vals and acc_squs, representing \( s_1 \) and \( s_2 \), are declared, initialized and accumulated (Figure 3.8).

From acc_vals and acc_squs the option value and its standard error are computed by operationalizing equations (3.11) and (3.12). The code in Figure 3.9 does this. It needs to be inserted at position \( (e) \) in Figure 3.8.

```dimacsembler
Dim acc_vals As Double: acc_vals = 0 'additional code
Dim acc_squs As Double: acc_squs = 0
Dim j As Long
For j = 1 To M 'for each sample path
    Dim path_S As Double: path_S = S_0
    Dim i As Long
    For i = 1 To N 'for each time step
        path_S = next_S(path_S, drift, sqrt)
    Next i
    Dim payoff As Double: payoff = Compute_PO(path_S, X)
    acc_vals = acc_vals + payoff
    acc_squs = acc_squs + payoff * payoff
Next j
'Position (e)
```

**Figure 3.8** Computing the option value and its standard error (i)
32 Implementing Models of Financial Derivatives

![Image](image_url)

**Figure 3.9** Computing the option value and its standard error (ii)

Note that one does not write `acc_squs = acc_squs + payoff ^ 2` but `acc_squs = acc_squs + payoff * payoff`. The power function `^` is expensive; to compute `c = a^b` the power function seems to implement `c = exp(b ln(a))`. Use multiplication where possible instead of `^` (see Chapter 14).

**Incrementing sample paths**

The only thing left to do is to write the function `next_S()`. The solution to (3.1), expressing $S_t$ as a function of the value $z_t$ of the Wiener process at time $t$, is

$$S_t = S_0 \exp\left(\left( r - \frac{1}{2} \sigma^2 \right) t + \sigma z_t \right).$$  \hfill (3.13)

Since there are $N$ steps over the time period $T$, the step length is $\Delta t = T/N$ and over a single step

$$S_{t+\Delta t} = S_t \exp\left(\left( r - \frac{1}{2} \sigma^2 \right) \Delta t + \sigma \sqrt{\Delta t} \, \varepsilon_t \right)$$ \hfill (3.14)

where $\varepsilon_t \sim N(0, 1)$ is standard normal IID.

The function `next_S` implements equation (3.14). Since $\mu_{\Delta t} = (r - \frac{1}{2} \sigma^2) \Delta t$ and $\sigma_{\Delta t} = \sigma \sqrt{\Delta t}$ can be precomputed they are passed as arguments (named `mu_dt` and `sig_dt`) to `next_S()`. The result is given in Figure 3.10. The function `GetNormal()` returns a standard normal IID variate $\varepsilon_t$ (see below).

Code to compute $\mu_{\Delta t}$ and $\sigma_{\Delta t}$ needs to be inserted into `main()`, as in Figure 3.11, and the call to `next_S()` has to be amended to include the additional arguments, changing

$$path_S = next_S(path_S)$$ \hfill (3.15)

to

$$path_S = next_S(path_S, drift, sgrt)'.$$ \hfill (3.16)

**Adding in a timer**

For any numerical work you need to have a rough idea of how long the code is going to take to run. For a Monte Carlo method if it takes $x$ seconds to run with $N$ time steps and $M$ sample paths, then *ceteris paribus* with $pN$ time steps and $qM$ sample paths it will take about $pqx$ seconds to run.

```vba
Private Function next_S(S As Double, mu_dt As Double, sig_dt As Double) As Double
next_S = S * Exp(mu_dt + sig_dt * GetNormal)
End Function
```

**Figure 3.10** The Function `next_S()`
To compute an elapsed time add in the code shown in Figure 3.12. This records the start time, before the code to be timed begins, then at the end computes the elapsed time. Timing issues are discussed in more detail in Chapter 5.

*Adding in a counter*

It is reassuring to a client of a Monte Carlo application – or any heavy numerical application – to know how it is progressing. For Monte Carlo it is easy and convenient to output a counter to show how many sample paths have been generated so far to let you know if you have enough time to go for a cup of coffee, or not. We add in the line

\[
\text{If } j \div 1000 = j \mod 1000 \text{ Then } \text{Cells}(8, 7).\text{Value} = j
\]

(3.17)

at the start of the \( j \)-loop. This outputs a value of \( j \) every 1000 sample paths. Output to the front-end is expensive but the test is cheap; printing out only every 1000 paths is reasonable.

*Generating normal variates*

Code to generate normal variates is shown in Figure 3.13. GetNormal() uses polar rejection to generate two normal variates at a time. It calls stat_ran0() which generates uniform variates. Both GetNormal() and stat_ran0() are based upon code from Press et al. (2002), modified to use Static variables instead of global variables. We do not discuss the mathematics underlying these two functions. The interested reader is referred to, for instance, Press et al. (2002).

Two features are:

1. GetNormal() uses explicit casting of \( 2\# \) * et cetera as a Double.\(^{10}\) This makes a big difference to run times. It also uses \( u * u \) instead of \( u ^ 2 \). Again, this is much faster. These performance issues and others are explored in Chapter 14.
2. stat_ran0() uses the Static initialization idiom, described in Chapter 4. This sets the values of certain constants, and also sets the initial value of a Seed.

---

\(^9\) In VBA Static variables are expensive; despite this they are used here to avoid using globals.

\(^{10}\) The \# character in \( 2\# \) causes the compiler to treat the literal as a Double. The literal 2, with no \#, is treated as an Integer.
Private Function GetNormal() As Double
    Static Spare_normal_flag As Boolean 'Initialised to False
    Static Spare_normal As Double 'Initialised to 0
    If Spare_normal_flag Then
        Get_normal = Spare_normal
        Spare_normal_flag = False
        Exit Function
    End If
    Dim u As Double, v As Double
    Dim W As Double: W = 1#
    Do While W >= 1# 'Finds two N(0,1) variates by polar rejection
        u = 2# * stat_ran0() - 1#
        v = 2# * stat_ran0() - 1#
        W = u * u + v * v
    Loop 'Have not rejected. Construct the normals
    Dim C As Double: C = Sqr(-2# * Log(W) / W)
    Get_normal = C * u
    Spare_normal = C * v 'Store the spare normal for the next call
    Spare_normal_flag = True
End Function

Private Function stat_ran0() As Double 'Based on ran0 in Press et al.
    Static NOT_FIRST_TIME As Boolean 'Initialised to False
    If NOT_FIRST_TIME = False Then 'NOT_FIRST_TIME idiom
        Static IA As Long: IA = 16807 'Give fixed values
        Static IM As Long: IM = 2147483647
        Static IQ As Long: IQ = 127773
        Static IR As Long: IR = 2836
        Randomize 'Sets seed for Rnd
        Static Seed As Long: Seed = Int(Rnd * IM_1) 'Sets Seed with Rnd
        NOT_FIRST_TIME = True
    End If
    Dim k As Long: k = Int(Seed / IQ)
    Seed = IA * (Seed - k * IQ) - IR * k
    If Seed < 0 Then Seed = Seed + IM
    stat_ran0 = AM * Seed
End Function

Figure 3.13 Generating a normal variate

Randomize is a VBA function that randomly sets the seed for the VBA random number generator Rnd. When, as here, Rnd is called with no argument, it returns a uniformly generated Single in the range [0, 1). Rnd should not be used directly to generate uniforms; in serious applications clients must control the random number generators they use. In this book Rnd is used only in non-sensitive situations such as, in this example, setting the value of the seed used by the proper random number generator.

GetNormal() and stat_ran0() are library utility functions. Later they are put into a separate library utility module. See Appendix C for a description of the library modules included with this book.

This version of the application is now complete. It is run by clicking the button. The complete Sub main() is shown in Figure 3.14.
Public Sub main()
    Cells(9, 7).Value = ""
    Dim S_0 As Double: S_0 = Cells(12, 6).Value
    Dim rr As Double: rr = Cells(13, 6).Value
    Dim sigma As Double: sigma = Cells(14, 6).Value
    Dim X As Double: X = Cells(17, 6).Value
    Dim T As Double: T = Cells(18, 6).Value
    Dim N As Long: N = Cells(12, 9).Value
    Dim M As Long: M = Cells(13, 9).Value
    Dim dt As Double: dt = T / N
    Dim drift As Double: drift = (rr - 0.5 * sigma * sigma) * dt
    Dim sgrt As Double: sgrt = sigma * Sqr(dt)
    Dim acc_vals As Double: acc_vals = 0
    Dim acc_squs As Double: acc_squs = 0
    Dim e_time As Double: e_time = Timer
    Dim j As Long
    For j = 1 To M 'for each sample path
        If j / 1000 = j \ 1000 Then Cells(8, 7).Value = j
        Dim path_S As Double: path_S = S_0
        Dim i As Long
        For i = 1 To N 'for each time step
            path_S = next_S(path_S, drift, sgrt)
        Next i
        Dim payoff As Double: payoff = Compute_PO(path_S, X)
        acc_vals = acc_vals + payoff
        acc_squs = acc_squs + payoff * payoff
    Next j
    Dim val As Double: val = acc_vals / M
    Dim se As Double: se = Sqr(acc_squs - acc_vals * acc_vals / M) / M
    Dim dis As Double: dis = Exp(-rr * T)
    val = dis * val
    se = dis * se
    e_time = Timer - e_time
    Cells(17, 9).Value = val
    Cells(18, 9).Value = se
    Cells(9, 7).Value = e_time
End Sub

Figure 3.14 The level 1 Sub main()

3.2 DEFICIENCIES OF THE LEVEL 1 CODE

The structure of the program is shown in Figure 3.15. The design – if it can be called that – is as basic as can be. The button runs main(). main() reads in and writes out to the front-end, doing a Monte Carlo in between.

Referring back to Figure 2.1, we can see that by comparison there is almost nothing in Figure 3.15. We have noted that there is no validation or error trapping, and there is a minimal possibility of code re-use. Even so, the construction of this example illustrates some ideas that apply more broadly. For instance, do the things you know about, and hive off the things you don’t yet know about until later. Create functions to do self-contained things that you can fill in later.

Functions are good ideas because (a) it is possible to change the way a function works (its implementation detail) without altering the client code that calls it and (b) the functionality itself can be altered. For
instance, a call payoff function can be replaced by a put payoff function, or perhaps the GBM process for the underlying asset can be replaced by a Heston process.

These ideas cannot be fully worked through unless a language admits polymorphism. This is possible to some extent with VBA through the use of interfaces. However there are a number of levels to go through before that stage is reached.

Before moving on we make an observation: the program syntax and structure is simple enough to be transcribable more or less directly into other languages. It contains nothing that is not common to other languages. In particular it can be translated directly into a broadly comparable C++ program. This enables a performance comparison to be made for this program between VBA and C++.

For 100 time steps and 500 000 sample paths the VBA version took 67.8 seconds. In C++ the transcribed program took 14.4 seconds to run. These numbers represent a direct comparison between execution times across languages (and compilers). C++ is faster, but not staggeringly so. Running in a little under a quarter of the time is not to be sneezed at, but C++ is not faster by orders of magnitude.

Here, advanced features of C++ are not required. Given the relative simplicity of programming in VBA, and its natural I/O to an Excel spreadsheet, it is not clear that one is automatically better off programming in C++. Quite the reverse; the simplicity of VBA makes it the language of choice for this simple style of programming.

An even more basic version of the application, one with no functions at all (other than to generate random numbers) is included on the CD as MC_project_v0 in MC_example_v0.xls. Timing comparisons (see Chapter 14) show it to be about 15% faster than MC_project_v1; but please do not use it for anything.

### 3.3 SUMMARY

We have walked through the implementation of a very basic Monte Carlo method. Some of its deficiencies have been highlighted. The chief problems are probably its lack of re-usability and its complete lack of validation. The lack of validation is put right straightaway in Chapter 4.

### 3.4 EXERCISES

The first two exercises ask you to modify the Monte Carlo application developed in this chapter. Exercise 2 will enable it, eventually, to simulate more general processes than geometric Brownian motion.

---

11 Cpp_example_v0. You will need a C++ compiler before this will run.
12 Dev C++ v5 on the same platform as the VBA, compiled with “best” optimization.
Procedural Programming: Level 1 37

The remaining exercises introduce four streams that are continued and elaborated upon in later chapters of this book. The streams are: (i) an application to value $\pi$ by series expansions; (ii) an implied volatility application; (iii) a lattice application; and (iv) a PDE solver application. Exercises in later chapters build upon solutions in earlier chapters.

1. As the number of sample paths $M$ increases so the standard error $se$ of a Monte Carlo method reduces. Investigate the behaviour of $se$ as $M$ increases. In particular estimate experimentally the rate at which $se$ decreases in $M$.

2. The Monte Carlo method presented here uses equation (3.13), the exact solution to equation (3.1), to evolve $S_t$. When the exact solution is not known an approximation must be used. Suppose that $S = (S_t)_{t \geq 0}$ has the process

$$dS_t = \mu(S_t) \, dt + \sigma(S_t) \, dz_t$$

(3.18)

for functions $\mu$ and $\sigma$, and let $\varepsilon_t \sim N(0, 1)$ be a standard normal variate. Two simple approximations are the Euler and the Milstein:

Euler: \[ S_{t+\Delta t} = S_t + \mu(S_t) \Delta t + \sigma(S_t) \sqrt{\Delta t} \, \varepsilon_t, \]

Milstein: \[ S_{t+\Delta t} = S_t + \mu(S_t) \Delta t + \sigma(S_t) \sqrt{\Delta t} \, \varepsilon_t + \frac{1}{2} \sigma(S_t) \sigma'(S_t) (\varepsilon_t^2 - 1) \Delta t, \]

(3.19)

where $\sigma'(S_t) = \partial \sigma(S_t) / \partial S$.

For a geometric Brownian motion these become

Euler: \[ S_{t+\Delta t} = S_t + r S_t \Delta t + \sigma S_t \sqrt{\Delta t} \, \varepsilon_t, \]

Milstein: \[ S_{t+\Delta t} = S_t + r S_t \Delta t + \sigma S_t \sqrt{\Delta t} \, \varepsilon_t + \frac{1}{2} \sigma^2 S_t (\varepsilon_t^2 - 1) \Delta t, \]

(3.20)

which approach the exact solution as $N \to \infty$.

Monte Carlo option values produced by evolving $S_t$ with these approximations are biased; the bias decreases as $N \to \infty$.

Investigate the extent of the bias in option prices for the Euler and Milstein approximations for fixed $T$ as the value of $N$ varies. For a given value of $M$ how great does $N$ need to be before the bias is reduced to an acceptable fraction of the standard error?

3. Pi stream. The following series converge to values that are functions of $\pi$. They can be used to compute approximations to the value of $\pi$.

\[
1 + \frac{1}{3^2} + \frac{1}{5^2} + \frac{1}{7^2} + \ldots = \frac{\pi^2}{8},
\]

(3.21)

\[
1 + \frac{1}{3} \left( 1 + \frac{2}{5} \left( 1 + \frac{3}{7} (1 + \ldots) \right) \right) = \frac{\pi}{2} \quad \text{(Beeler et al.)},
\]

(3.22)

\[
\sum_{k=0}^{\infty} \frac{(k!)^2}{(2k + 1)!} = \frac{2\pi}{9\sqrt{3}} \quad \text{(Gosper)}.
\]

(3.23)

Write a level 1 application, pi_app_v1.xls, to compute the value of $\pi$ by implementing Functions that evaluate these series. The Functions should take an argument to determine the number of terms to be
summed. Arrange for the user to specify on the front-end which series is to be used and the number of terms to be summed. The application should output the approximate value of $\pi$ and the true value (for comparison) given by the built-in function `Application.Pi()`.

4. **Implied volatility stream.** In exercise 1(a), Chapter 1, you wrote code to implement the Black–Scholes formula. Suppose that in the market at time $t$ you can observe the value $c_t$ of a European call option with strike $X$ and time of maturity $T$, the current value of the underlying $S_t$, and an interest rate $r$. The option’s implied volatility, $\sigma^{IV}$, is the value of $\sigma$ that, when put into the Black–Scholes formula, recovers the market value of the option,

$$c_t = BS_t(S, r, \sigma^{IV}, X, T).$$  \hspace{1cm} (3.24)

The bisection algorithm for finding roots of functions is presented in Appendix H. Construct a solver, `IV_app_v1.xls`, based on the bisection algorithm, that computes implied volatilities for the Black–Scholes formula. A tolerance level needs to be specified so that the algorithm can determine when it should terminate. As the tolerance level reduces, how does the number of iterations increase?

5. **PDE stream.** The spreadsheet `CN_pde_v1.xls` (see Appendix G) implements a level 1 Crank–Nicolson PDE solver. How much faster can you make it run?

   Some of the coding in `CN_pde_v1.xls` is inefficient and there is a significant design flaw that leads to unnecessary replicated computation. Can you find this and remove it?

6. **PDE stream.** Extend the code in `CN_pde_v1.xls` to enable the application to value single barrier options. Benchmark it to the values of down and out barrier call options (Chapter 1, exercise 1(b)) and investigate its convergence properties.

7. **Lattice stream.** The spreadsheets `Lattice_application_v0.xls` and `Lattice_application_v0_faster.xls` (see Appendix F) implement a level 0 trinomial lattice for a geometric Brownian motion. Create a level 1 version of `Lattice_application_v0_faster.xls`, `Lattice_application_v1.xls`, by

   (a) identifying which parts of its functionality could conveniently be put into procedures;

   (b) implementing the design by writing these procedures.

   How does the performance of the level 1 version compare to that of the level 0 version?

8. **Lattice stream.** Extend the code in `Lattice_application_v1.xls` to enable it to value single barrier options. Benchmark it to the values of down and out barrier call options (Chapter 1, exercise 1(b)) and investigate its convergence properties.
Validation and Error Handling: Level 2

In Chapter 3 a level 1 implementation of a Monte Carlo application was presented. It is poor on various counts: the chief problems are (i) its inability to facilitate code re-use, (ii) the difficulty of maintenance, (iii) its complete lack of validation and (iv) its restricted functionality.

Of course it is just a throw-away piece of code; maintenance is not an issue if the code is not going to be used in any other way, nor do you need validation.

The objective of this chapter is to create something that is a little more than an end in itself; to produce an application with some flexibility and extensibility; where validation both exists and, because of the ‘worthwhileness’ of the application, is needed; and to make the code cleaner with greater cohesion and encapsulation. More specifically we shall:

1. Implement validation and error handling.
2. Separate out and encapsulate into functions bits of code related to the same task, and clear away variables by localizing them within functions.
   Input and output can be separated off into dedicated functions; the formulae for the option value and its standard error can be turned into functions; and the variables representing $dt$, $drift$ and $sgrt$ are used only in $next_S()$ and can be tidied away into $next_S()$ (although this is dangerous).
3. Give the data greater structure by creating a user-defined data type. Input is made a lot tidier by defining a data type.
4. Split the code between separate, potentially re-usable, standard modules.

Procedures in modules intended to serve as libraries will normally be declared to be Public. All others will have their scope restricted as appropriate. The spreadsheet MC_example_v2.xls contains the new application.

We start by looking at validation and error handling. VBA has two mechanisms that allow an object further down in the stack to communicate with objects further up the stack. The first is by throwing an error using the Err.Raise method of the Err object. The error thrown can be caught further up the stack by a procedure implementing an On Error statement. Control stays with the procedure that intercepts the error.

The second mechanism is by raising an event. Events are raised by objects and can be caught by objects containing the source object declared "WithEvents". After the event has been processed control returns to the source object. We explore this mechanism in Chapter 8.

We shall talk of errors being thrown and caught (by analogy with usage in languages like C++) and of events being raised. However, since VBA calls the Err.Raise method to throw an error we shall sometimes also refer to errors being raised. It will be clear from the context whether it is an error or an event that is being raised.

When an error is thrown the information that can be transmitted with it is restricted to a single Long (the error number) and two Strings (see below). This severely constrains the usefulness of throwing errors. By contrast when an event is raised it can take arbitrary arguments with it, making events potentially a more powerful mechanism; however their usefulness is restricted because control must return to the object raising the event.1

1 And in practice their implementation in conjunction with polymorphic objects seems problematical. See Chapter 8.
Here we discuss the first mechanism. Although in principle it could be used routinely to exit from within the stack, in practice its use should be reserved for its intended purpose of signalling and handling error conditions.

## 4.1 VALIDATION AND ERROR HANDLING

Procedures operate with and on data. Either explicitly or implicitly (not good) that data must belong to a set of allowed, valid, states. This constitutes a contract between a procedure and the data on which it operates. If the data fails to comply with the contract, the procedure may have undefined behaviour, or just collapse, weeping. For instance, the Black–Scholes formula rather expects to have a positive time to maturity and a strictly positive asset value; it might feel it needs a positive interest rate. If a procedure does not receive its arguments in the form it requires then it has to decide what to do about it.

For items with an internal state – procedures with Static variables, globals in a standard module, data members in a class module – there will be only a restricted range of allowed states. These too need to be checked.

Validation refers to the process by which an object or procedure checks values and types of the variables defining its state or on which it operates, preferably at the point at which their values and types become known, to ensure that the object or procedure is in and remains in a valid state. In a strongly typed language like C++ a mismatch in type results in a compile-time error; in a less strongly typed language like VBA some type mismatches may not result in an error until run-time.

At its crudest level validation can be as simple as ensuring that something that should be a whole number is a whole number, or something that should be a String is a String. It is a very good idea to have utility validation procedures to confirm these basic properties.

We discuss two validation Functions, check_double() and check_long(), two Functions that call them, get_double() and get_long(), and a helper Sub ForwardError(). These procedures use the Err object, so we briefly discuss this first.

### 4.1.1 The Err object

The Err object is one of a number of predefined VBA objects. It is a singleton; one and only one instance ever exists and it is globally accessible from procedures anywhere in an application. Its state gives information about the current error condition.

It has six Properties (which for the moment can be regarded as Public data members) of which three will be discussed here and used in our implementations. These three are listed in Table 4.1. If an error has been raised, by the occurrence of a VBA error condition or explicitly by the application through a call to the Err.Raise method, these properties contain the error’s number, its source (a String identifying the originator of the error), and a description of the error.

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Err.Number</td>
<td>Long</td>
<td>The last error number</td>
</tr>
<tr>
<td>Err.Source</td>
<td>String</td>
<td>Source of last error</td>
</tr>
<tr>
<td>Err.Description</td>
<td>String</td>
<td>Description of last error</td>
</tr>
</tbody>
</table>

2 Of course, Option Explicit will always be set.
3 The other three properties concern DLL errors and help files; we do not use them.
The Err object has two methods, Err.Clear and Err.Raise. Err.Clear resets the Err object (sets the error number to zero and each String to an empty String). Err.Raise raises a (user-defined) error. It is this latter method that most concerns us.

### 4.1.2 Using the Err object

Figure 4.1 displays the functions check_double() and check_long(). check_double() tests its argument to discover if it is numeric. If it is then it is returned as a Double; otherwise check_double() throws a user-defined error. Similarly check_long() tests whether its argument is a Long.

check_double() and check_long() take a Variant as their argument, of necessity as they make no assumptions about type, using the VBA functions IsNumeric(), CDbl() and CLng() to validate. IsNumeric() returns True if its argument is numeric, and is guaranteed no-throw. CDbl() converts a numeric argument into a Double, but throws if its argument is not numeric, and similarly for CLng(). By testing their argument before executing CDbl() or CLng(), check_double() and check_long() take control and prevent the code from breaking.

In check_double() the line that raises the error is

```
Call Err.Raise(vbObjectError + 1114, "check_double", "Field is non-numeric"). (4.1)
```

Err.Raise() has three arguments (excluding two optional arguments relating to a possible helpfile). The first sets the error number Property, the second the error source Property, and the third the error description Property.

vbObjectError is a VBA constant whose value is used to determine whether an error is a VBA error or a user-defined error. Application error numbers are the range vbObjectError + 1 to vbObjectError + 65534. Any other error number refers to a VBA error. User-defined error numbers should therefore be offset by vbObjectError to distinguish them from VBA errors. Line (4.1) raises the user-defined error 1114; its source is “check_double” and its description is “Field is non-numeric”.

```
Public Function check_double(x As Variant) As Double
'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  'XX check_double()  
'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  If Not IsNumeric(x) Then
    Call Err.Raise(vbObjectError + 1114, "check_double", "Field is non-numeric")
  End If
  check_double = CDbl(x)
End Function
'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
'XX check_long()
'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Public Function check_long(x As Variant) As Long
  If Not IsNumeric(x) Then
    Call Err.Raise(vbObjectError + 1114, "check_long", "Field is not a long")
  End If
  check_long = CLng(x)
  If check_long <> CDbl(x) Then
    Call Err.Raise(vbObjectError + 1115, "check_long", "Field is not a long")
  End If
End Function
```

![Figure 4.1 Validation functions](image)

4 There is nothing to prevent an application raising a VBA error, but this should be done only with circumspection.
A useful utility is `RaiseError()` which throws if an attempt is made to run it,

```vba
Private Sub RaiseError(num As Long, source As String, mess As String)
    Call Err.Raise(vbObjectError + num, source, mess)
End Sub'
```

(4.2)

Its purpose is just to wrap a call to `Err.Raise`. This is syntactic sugar, hiding away `Err` object gubbins, but making code easier to read and to understand is important.

**Error trapping**

When an error is thrown it can be caught further up the stack. Error catching is also referred to as error trapping. To be able to trap errors a procedure must contain an `On Error` statement, of which there are two versions. The first is

```vba
On Error GoTo Error_label',
```

(4.3)

When an error is detected on a subsequent line, (4.3) causes control to pass to the label `Error_label:`. This is the version usually used and its idiomatic use is described below.

The second version,

```vba
On Error Resume Next'
```

(4.4)

is used appropriately in particular situations, and will be described elsewhere in this book as need arises.

The idiom for invoking error handling is shown in Figure 4.2. If an error is thrown, the `On Error` statement directs control to the label `Error_Handling:`. An error handling `Sub`, `ErrorHandler()`, is called. The `Resume Clean_up` statement causes control to return to the label `Clean_up:`. It is very important that whenever an error is thrown the state of the application is returned to a valid state, preferably the same (valid) state it was in before the operation that caused the error condition. Code following the `Clean_up:` label is meant to do just that.

The usual exit point for the `Sub` is the `Exit Sub` statement; the `End Sub` statement is never reached.

Figure 4.2 Idiom for error handling

5 The `Resume` statement is valid (that is, does not throw a compile-time error) only in a section of code invoked during error handling (that is, following on from a label branched to by an `On Error GoTo Error_label` statement). Two other forms of the `Resume` statement are possible. `Resume` (by itself) causes control to return to the line that threw the error; `Resume Next` causes control to pass to the line following the one that threw the error. We discuss these uses elsewhere as necessary.
The GoTo statement is evil; error handling is the only circumstance where it should be used.

Figure 4.3 gives an example of error trapping. The Functions get_double() and get_long() are designed to do input from the front-end, calling the appropriate validation functions. get_double() takes as arguments the coordinates of a cell on the front-end. It reads in from that cell and, if the contents are numeric, returns as a Double the value that it finds there. If the contents are not numeric it calls the ForwardError() Sub which throws an error. This error can be trapped by an On Error statement further up the stack.

It would be possible to provide alternative versions of these validators that, instead of cell coordinates, take range names as arguments, but we have not done so here.

---

6 Here a technical computing term referring to something you should never ever do. Unless you have to.
get_double() and get_long(), and the validators check_double() and check_long(), are put into a separate standard module, Validators, as Public procedures. The Validators module is designed to contain utility validation procedures that can be used in more than one application. Having written a set of useful general-purpose procedures one wishes that the effort does not go to waste, and makes them available for use by other clients.7

A full set of library utility modules is described in Appendix C. The module LibValidators contains all the procedures described here, and others.

check_double() and check_long() are called by get_double() and get_long(). get_double() and get_long() trap errors raised by check_double() and check_long(), re-raising them with a slightly more specific description to be caught higher up the stack. If an error is thrown, for instance by check_double() in get_double(), control passes to the label Bad_type:. The Sub ForwardError() re-throws the error so there is no need for a Resume statement.

ForwardError(), also displayed in Figure 4.3, takes as arguments an array of Longs, representing a set of anticipated user-defined error numbers, and a String intended to contain the name of the procedure that calls ForwardError(). If the active error is one of those anticipated, then it is re-thrown with the String argument substituting for the error source. If the error is unanticipated then it is re-thrown with the error description modified to include the String. In this instance 1114 and 1115 are error numbers defined in check_long() for a non-numeric field and a non-Long numeric respectively.

This technique, of catching an error, processing it in some way, and then re-throwing it, is very common. If an error occurs then action may need to be taken at a number of levels within the stack as procedures institute recovery at their own levels.

The final component of error handling, after an error has been raised and caught, is to do something about it. This is very much application dependent. The code in Figure 4.2 calls the Sub ErrorHandler() to do all that. In our context the error handling Sub ErrorHandler() need not do anything sophisticated (nor need the clean-up code yet do anything at all). Figure 4.4 shows the error handler we shall use. All it does is print a message box describing the error. ErrorHandler() uses the VBA constant vbCrLf.

```
'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
'XX error handling Sub
'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Public Sub ErrorHandler()
    Dim ErrorNum As Long: ErrorNum = Err.Number - vbObjectError
    Const ErrorBnd As Long = 65535
    Dim msg As String
    If ErrorNum > 0 And ErrorNum < ErrorBnd Then
        msg = "User defined error" & vbCrLf & _
            "AppErr is: " & ErrorNum & vbCrLf & "Source: " & Err.source & vbCrLf & "Description: " & Err.Description
    Else
        msg = "Visual Basic error" & vbCrLf & _
            "Number is: " & Err.Number & vbCrLf & "Source: " & Err.source & vbCrLf & "Description: " & Err.Description
    End If
    Call MsgBox(msg, , "Object Error")
End Sub
```

Figure 4.4 The error handling Sub

7 The qualified name of check_double(), for instance, is Validators.check_double(). We do not routinely use the qualified name unless it is necessary to disambiguate a name clash.
This is the carriage return character Chr(13). We prefer to use predefined constants than the messier equivalents.

All errors, VBA or user-defined, are caught by the handler. It deals with them separately but only slightly differently.

The error handling Sub is put into a separate standard module, ErrorHandling. In the level 3 version of the code it becomes a separate object.

### 4.2 ENCAPSULATING FUNCTIONALITY

Various parts of the main() of Figure 3.14, (page 35) can be re-written as Functions or Subs. Output can be hived off straight away, as can the tasks of computing \( \hat{c} \) and \( se \). Implementations of procedures for this functionality are given in Figure 4.5.

The output Sub Output_results() is trivial, simply outputting its arguments to the front end. The Sub ClearCells() is an additional bit of functionality that will blank out the output cells at the start of a run. A knowledge of the locations being written to is confined to ClearCells() and Output_results().

acc_vals and acc_squs are updated in the Sub UpdateAccVals() that takes them ByRef. Compute_val() and Compute_se() take acc_vals and acc_squs as arguments and from them compute the option value and its standard error. They also take a user defined type (UDT) as an argument. We discuss this in the next section. Each procedure separately computes a discount factor (such duplication is poor; it indicates a design flaw).

```vba
'XX Accumulates values and squared values (arguments are ByRef)
Private Sub UpdateAccVals(vals As Double, squs As Double, pay As Double)
vals = vals + pay
squs = squs + pay * pay
End Sub

'XX Compute_val() and Compute_se()
Private Function Compute_val(acc_vals As Double, data As MC_data) As Double
Dim M As Long: M = data.M
Dim discount As Double: discount = Exp(-data.rr * data.T)
Compute_val = discount * acc_vals / M
End Function

Private Function Compute_se(acc_vals As Double, acc_squs As Double, data As MC_data) As Double
Dim M As Long: M = data.M
Dim discount As Double: discount = Exp(-data.rr * data.T)
Compute_se = discount * Sqr(acc_squs - acc_vals * acc_vals / M) / M
End Function

'XX output_stuff
Private Sub Output_results(val As Double, se As Double, e_time As Double)
Cells(17, 9).Value = val
Cells(18, 9).Value = se
Cells(9, 7).Value = e_time
End Sub

Private Sub ClearCells()
Cells(17, 9).value = ""
Cells(18, 9).value = ""
Cells(9, 7).value = ""
End Sub
```

Figure 4.5 Splitting off functionality
The main Monte Carlo functionality will also be put into a separate procedure, away from input, output and the timing function. We return to this below.

**Input with a user-defined type**

Defining and using a *Type* makes passing data around an application much easier. It explicitly constructs a set of data elevated to legitimacy by its reification as a *Type*, and it enables a single argument to be passed to a procedure rather than many.

Figure 4.6 gives a *Type* declaration. In VBA user-defined *Types* are POD (plain old data). They hold readable and writable data with no attached functionality.

An instance of `MC_data` can be declared and passed around the application as an argument. For instance, a separate `Sub`, `Read_in_data()` in Figure 4.7, can be written to take responsibility for inputting from the front-end. The input lines in the `main()` of Figure 3.14 can be replaced by a single line

\[
\text{Dim data As MC_data: Call Read_in_data(data)} \quad (4.5)
\]

and subsequently `M`, for instance, can be replaced by `data.M`.

The advantage of doing this, apart from helping to tidy up `main()`, is that knowledge of input locations is confined to `Read_in_data()`. Client code need have no idea of where `Read_in_data()` is getting its input from.

There are two very distinct types of data in this UDT. The first type are variables specifying the process, the option, and the Monte Carlo parameters. The second type has a single instance: the value of `NC` that stores the frequency that a counter is output to the front-end. `NC` has nothing to do with Monte Carlo

```
Private Type MC_data
    sigma As Double 'Stock volatility
    rr As Double 'Stock short rate
    S_0 As Double 'Initial stock price
    T As Double 'Final time
    X As Double 'Strike
    N As Long 'Number of time steps
    M As Long 'Number of sample paths
    NC As Long 'Counter output interval
End Type
```

**Figure 4.6** The user-defined type, `MC_data`

```
Private Sub Read_in_data(ByRef data As MC_data)
    data.sigma = get_double(14, 6)
    data.rr = get_double(13, 6)
    data.S_0 = get_double(12, 6)
    data.X = get_double(17, 6)
    data.T = get_double(18, 6)
    data.N = get_long(12, 9)
    data.M = get_long(13, 9)
    data.NC = 1000 'Hard-wired in
End Sub
```

**Figure 4.7** The input `Sub`
**per se.** It is an environmental variable that affects how the application interacts with the client. We see further examples of environmental settings later in the book. As a special case its value is, for the moment, just hard-wired in to Read_in_data().

**Encapsulating locals into a function**

dt, drift and sgrt are used only in next_S() and can be confined to next_S() as Static variables. This gives us the Function next_S_v2() (Figure 4.8). It is necessary to pass our instance of MC_data as an argument to next_S_v2(). There it is used to initialize Static data held by next_S_v2() using the NOT_FIRST_TIME idiom.

**The NOT_FIRST_TIME idiom**

The NOT_FIRST_TIME idiom is important. It was used in the random number generator stat_ran0() (Figure 3.13, page 34), and it recurs in procedures throughout this book.

The NOT_FIRST_TIME idiom is a technique for initializing Static data in a procedure. A Boolean sentinel, NOT_FIRST_TIME, toggles whether the values are set or not. The variables declared as Static, protected by NOT_FIRST_TIME, have their values set only the first time the procedure is run.

In Figure 4.8 only if NOT_FIRST_TIME is False are dt, et cetera, initialized and NOT_FIRST_TIME set to True. This ensures that dt, et cetera, are set only once. Note that the scope of dt, et cetera, is not restricted to the If-statement block but extends to the end of function.8

We refer to this technique as the Static initialization idiom or the NOT_FIRST_TIME idiom. Testing the value of a Boolean is normally much cheaper than assigning a value, so this idiom should always be used with Static data that needs to be initialized.

next_S_v2() is cleaner than next_S(), but a little slower. One argues that the slight speed hit is fully compensated for by the increased clarity afforded to main().

Note that there is a major hazard with using this idiom in VBA: the lifetime of the Static variables extends beyond the run-time of the main() lasting, in fact, until the project is reset. This can be very, very, dangerous. Suppose, for instance, that the value of a variable set by this idiom depends on the value of a parameter read in from the front-end. If between runs one changes the value of the parameter, one would

```vba
Private Function next_S_v2(S As Double, ByRef data As MC_data) As Double
    Static NOT_FIRST_TIME As Boolean 'The sentinel, initialised to False
    If NOT_FIRST_TIME = False Then 'When called for the first time.
        Dim sig As Double: sig = data.sigma
        Static dt As Double: dt = data.T / data.N
        Static drift As Double: drift = (data.rr - 0.5 * sig * sig) * dt
        Static sgrt As Double: sgrt = sig * Sqr(dt)
        NOT_FIRST_TIME = True 'set the sentinel
    End If
    next_S_v2 = S * Expdrift + sgrt * GetNormal)
End Function
```

Figure 4.8 The evolver function

---

8 This contrasts to the situation in C++ where variables declared in an if-block have scope restricted to that block.
expect the application to use the new value; but it does not. At least, the values of Static variables that depend on it, set with the Static initialization idiom, are not changed. For instance, consider changing the value of the time to maturity parameter, \( T \). \( \text{next\_S\_v2()} \) uses the Static initialization idiom to set the value of \( \Delta t \) from \( T \). If the value of \( T \) is changed the value of \( \Delta t \) does not change as it should, until the project is reset. The application will run but it will produce erroneous output.

The solution is to limit the use of this idiom to initialization that does not involve changeable data. It is safe to use the idiom for \( \text{Stat\_ran0()} \), for instance, since \( \text{Stat\_ran0()} \) uses the Static initialization idiom only with Const expressions.

Using objects largely does away with the need for this idiom; in fact its use in \( \text{next\_S\_v2()} \) is a clear signal that the procedure should be replaced by an object.

**Outputting a counter**

We replace the line in \( \text{MC\_example\_v1} \) with a simple Sub (Figure 4.9). It has an output location hard-wired in but the output frequency, interval, is supplied as an argument.

To use the counter place the line

\[
\text{Call OutputCounter}(j, \text{data.NC})
\]

at the head of the \( j \) loop. This outputs a counter at intervals of data.NC sample paths.

**Separating procedures between standard modules**

Several procedures have already been put in their own modules: validators into Validators, the error handler into ErrorHandling. We can also put \( \text{my\_max()} \) into another standard module, here called UtilityStuff. Various other general purpose mathematical functions could also be added, as and when they are written. It seems a good idea to have a separate module for the random number generators \( \text{GetNormal()} \) and \( \text{Stat\_ran0()} \).

We end up with the level 2 code split between five modules; ErrorHandling, MC\_application\_v2, RandomNumberGenerators, UtilityStuff and Validators. See MC\_example\_v2.xls. Four of these are potentially re-usable.

### 4.3 THE LEVEL 2 main()

The Monte Carlo application is now in its own procedure, \( \text{MC\_app()} \). This takes three arguments: the UDT with the data it needs, and two arguments ByRef to return the option value and its standard error. The new main() does error handling, input and output, and leaves the application to do its own thing. The only arithmetic calculation it does is computing the elapsed time. The Subs main() and MC\_app() are given in Figure 4.10. The structure of the application is given in Figure 4.11.

The real work is performed by \( \text{MC\_app()} \). It is far more compact than the corresponding level 1 code. It does fewer itty-bitty things itself, instead delegating them to other procedures. Its role is becoming...
Sub main()
On Error GoTo Error_Handling
Call ClearCells
Dim data As MC_data: Call Read_in_data(data)
Dim e_time As Double: e_time = Timer
Dim oval As Double, se As Double: Call MC_app(oval, se, data)
Call Output_results(oval, se, Timer - e_time)
Clean_up:
Exit Sub
Error_Handling: 'Error handling bit
Call ErrorHandler
Resume Clean_up
End Sub
Private Sub MC_app(oval As Double, se As Double, data As MC_data)
Dim acc_vals As Double: acc_vals = 0
Dim acc_squs As Double: acc_squs = 0
Dim j As Long
For j = 1 To data.N 'for each sample path
    Call OutputCounter(j, data.NC)
    Dim path_S As Double: path_S = data.S_0
    Dim i As Long
    For i = 1 To data.N 'for each time step
        path_S = next_S_v2(path_S, data) 'Danger. cleaner but slower than v1
        Dim payoff As Double: payoff = Compute_PO(path_S, data.X)
        Call UpdateAccVals(acc_vals, acc_squs, payoff)
    Next i
    oval = Compute_val(acc_vals, data)
    se = Compute_se(acc_vals, acc_squs, data)
Next j
End Sub

Figure 4.10 main() and MC_app()

Figure 4.11 Flow in the level 2 application
that of organizing other things that do the real work. Each little Sub or Function has its own separate responsibility. \texttt{MC\_app()}’s is to organize.

Although there are important differences in the way that the code is broken into procedures (or not) the only additional VBA feature between this and the level 1 application in Chapter 3 is the presence of an error handling procedure.

Error handling, here as elsewhere, is unsophisticated. Procedures that may throw have \texttt{On Error} statements to enable them to raise errors gracefully.

No clean-up is required here; both the \texttt{Clean\_up:} label and the \texttt{Resume Clean\_up} statement could be elided, but we stick to the full idiom.

The whole thing is much tidier than the level 1 version, and the separation of code into different modules according to function is a considerable improvement, but the code left in \texttt{MC\_application\_v2} is nowhere near a satisfactory state. There are some obvious signs.

The use of the \texttt{Static} initialization idiom (and the use of sentinels in general) is almost a hack. It helps to localize some variables to within a \texttt{Function} but that is about all it can do. For instance, look at the variables \texttt{acc\_vals} and \texttt{acc\_squs}. These are initialized before the main loop, incremented within it, and used to compute \texttt{val} and \texttt{se} after the main loop. One would like to be able to construct a procedure with the functionality to compute \texttt{val} and \texttt{se} when required (and only when required) but with \texttt{acc\_vals} and \texttt{acc\_squs} localized within it. This is not possible given what we know so far.

As another example, in \texttt{main()}, \texttt{e\_time} is initialized before \texttt{MC\_app()} is executed and updated after it. One would like to have a single entity from which one could both request the elapsed time and also cause it to be reset. The \texttt{Static} initialization idiom does not help us here either.

These examples have in common the need to do more than one thing to the same set of data. What is required is not only a way of allowing a procedure to maintain data between calls, which can be accomplished with \texttt{Statics}, but also to enable more than one procedure to act on a single set of data.

Traditionally, in the absence of other mechanisms, two techniques have been used. The first is to have a single procedure, containing \texttt{Static} data, that is passed a parameter that toggles which of several possible actions the procedure performs. The second is to create a separate standard module containing \texttt{Private} global variables and both \texttt{Private} and \texttt{Public/Friend} procedures. The \texttt{Public/Friend} procedures are exposed to other modules; the data remains \texttt{Private}. See Appendix E for a detailed example.

These techniques “work” but they are very restrictive. For a start you can have only one instance of a procedure or module, whereas you may want to be able to have many instances. These techniques also promote poor design and in any case are not sufficiently flexible to encompass, for instance, polymorphism. Something better is needed, and that thing is objects.

### 4.4 SUMMARY

We have described error trapping and some idioms for using it, and it has been implemented along with the next level up of the Monte Carlo application.

We have noted the desirability of tying together data and functionality in some way. This is precisely the purpose of objects.

In the next chapter we introduce the use of objects to construct a level 3 version of the Monte Carlo application. Objects provide precisely the functionality needed (i) to have different procedures acting on the same data and (ii) to enable more than one instance to exist simultaneously.
4.5 EXERCISES

These exercises ask you to convert level 1 applications into level 2 applications by incorporating error handling and input validation, and encapsulating data into one or more UDTs. In particular cases there may be further issues to consider.

1. **Pi stream.** Convert the level 1 application, pi_app_v1.xls from exercise 3, Chapter 3, into a level 2 application, pi_app_v2.xls. Is there a case for introducing a UDT? What form should validation take in this application?

2. **Implied volatility stream.** From the level 1 Black–Scholes implied volatility application you developed in exercise 4, Chapter 3, add in error handling, validation, and appropriate UDTs. For this application, IV_app_v2.xls, what particular validation might be required? How many UDTs does the application need?

3. **PDE stream.** Starting from the faster version of the level 1 Crank–Nicolson PDE solver, in CN_pde_v1.xls (Chapter 3, exercise 5), produce a level 2 version. As well as a UDT to facilitate input can you identify any other UDTs that could be useful in the application?

4. **Lattice stream.** The level 1 version of the lattice application, Lattice_application_v1.xls, has no validation. What validation is appropriate for the lattice application? Implement this, and other level 2 features, to construct a level 2 lattice application, Lattice_application_v2.xls.

5. The module LibValidators in LibraryProcedures.xls contains a number of validation procedures. See Appendix C. What further validation procedures might be useful? How would you implement them?
This page intentionally left blank
This part moves on from the basic procedural level code of Part I to introduce objects. Chapter 5 starts in a non-polymorphic context. Various objects are identified and implemented, both structural and those specific to Monte Carlo.

In Chapter 6 the application goes polymorphic. Appropriate VBA interfaces are discussed and implemented and some problems with the application are identified.

A vector-based, slice-wise, Monte Carlo is constructed in Chapter 7. This illustrates a number of design issues and forms the foundation for extension in later chapters. In particular, extensions met in Part VI enable certain speed-up methods to be implemented efficiently.

Chapter 8, the final chapter of this part, investigates event generation and interception VBA. The chapter discusses this feature only to discard it for our numerical application. It also introduces the very important notion of a factory, albeit non-polymorphic. This is developed radically in further parts.
This page intentionally left blank
We commented at the end of Chapter 4 on the deficiencies of procedurally based code. Level 2 represents just about the furthest that our application can sensibly go without introducing objects. This chapter makes the application object oriented, although the power of a polymorphic design has to wait until Chapter 6.

First we describe how objects are implemented in VBA. Section 5.2 illustrates with a simple example: a stopwatch object to time code execution. Some helpful VBA functions are presented in section 5.3 and section 5.4 presents two versions of an OOP Monte Carlo application. The first is a basic extension of the non-OOP application given in Chapter 4; the second starts to develop a more sophisticated OOP architecture by introducing an object whose sole role is to manage other objects.

## 5.1 OBJECTS IN VBA

An object is a user-defined type, differing from a Type as, is addition to containing data, it can also have functionality. Procedures belonging to an object are sometimes called methods. More than one instance of an object can be instantiated.

A discussion on objects and object-oriented programming is given in Appendix E. A reader unfamiliar with OOP concepts should read this chapter and the next in conjunction with the appendix.

The most important aspects of objects are that (i) they encapsulate state, (ii) they contain procedures that provide functionality with or on the state, and (iii) they enable polymorphism. Objects in VBA have limited functionality compared to objects in other languages, such as C++. For instance, in VBA constructors (subs that are run when an object is instantiated) cannot take arguments. This severely limits the degree to which data encapsulation is possible; in particular, data defining the state of an object requires – directly or indirectly – a setter function. This means that copy and conversion constructors, and copy assignment (terms from C++), must employ Public/Friend setters, breaking encapsulation.\(^1\) Nor is it possible in VBA 6.X to overload operators, or to have (encapsulated) meta-class data. See Appendix E for a fuller discussion of these issues.

The programming style that uses objects idiomatically is called object-oriented programming (OOP). VBA is a not a fully OOP language: it lacks features that enable a proper concept of inheritance; it cannot make objects behave like primitive types; and it lacks the native ability to have meta-class date. There is limited scope to implement design patterns; for instance, although it is possible to implement the singleton pattern, the factory pattern, and registration and call-back, this can be done only with difficult work-arounds. Having said that, some of the patterns can be mimicked; see Part IV.

Encapsulation is a concept that has come up several times. It is directly relevant to objects, where data, representing state, is held within an object. We have often and shall often refer to encapsulation in a broader sense to mean separating off data or functionality with the intention of making it harder for interfering busy-bodies to get their hands on it.

Polymorphism is possible in VBA. This hugely important capability is discussed in Chapter 6. One feature that objects in VBA have, which is absent in C++, is the ability to raise events. For the moment we put events to one side; they are met, discussed and discarded (for our purposes) in Chapter 8.

---

\(^1\) In fact objects themselves in VBA are never copied or assigned – only references to them.
We distinguish between defining an object, declaring references to it, and instantiating it. The object definition is where the data, procedures and properties within the object are defined. This happens in the object’s class module. An object declaration is the point at which the name of a reference to an object is introduced to the program. The instantiation is the point at which an object of some type is actually created. In VBA, unlike C++, only references to objects can be declared, not instantiations of objects themselves.

Before an object can be used it must be instantiated, before it can be instantiated a reference to it must be declared, and before a reference can be declared the object must be defined (in a class module). We discuss how references to object are declared, how objects are instantiated and used, and then look at the details of how objects are defined.

### 5.1.1 Using objects: a stopwatch example

There are several built-in functions in VBA that set and return time values. For instance `Timer()` returns the elapsed time since midnight in seconds as a `Single` and `Now()` returns the current system time as a Variant of sub-type `Date`. Other, higher-resolution, timing mechanisms are available through the Windows API. Some failings of `Timer()` are discussed briefly in section 5.2.

These functions perform simple primitive tasks. They are used in client code to provide higher level services. We saw an example of this in Chapter 3, Figure 3.14, where the `Timer()` function is used to time code execution. However, the client code now contains messy details unrelated to its core functionality. It would be much better to be able to tidy away this functionality, and to enable it to be easily re-usable. Here we illustrate how an object can be used to perform this task, and we explain how the object is defined in section 5.2.

An example of using an object is given in Figure 5.1. It shows a stopwatch object, `StopWatch`, being declared and used to time code execution. The purpose of defining a `StopWatch` object is to make this task easy, decoupling code elsewhere from an knowledge of exactly how the stopwatch works internally, and facilitating the re-use of the code.

We use the stopwatch extensively in subsequent chapters: in the Monte Carlo application, in other heavy numerical applications, and in timing exercises to assess code performance. The idea is to replicate as closely as possible the way a physical stopwatch works so that using the `StopWatch` object is intuitive.

In the figure an instance of a stopwatch is created on line 5.1a. Just like any other type the declaration uses a `Dim` statement. What is new is the `Set` statement. This causes the reference, `watch`, to be set to a new instance of a `StopWatch`.

```vba
Sub main()
  '...
  Dim watch As StopWatch: Set watch = New StopWatch 'a
  watch.Start_Timer 'b
  'Run the time consuming stuff
  watch.Stop_Timer 'c
  Call OutputTime(watch.Elapsed_Time) 'd
  Set watch = Nothing 'e
  '...
End Sub
```

**Figure 5.1 Using the StopWatch**

---

2 The function `Time()` can be used both to return and to set the system time. The ability to set the system time is dangerous; it is better to use `Timer()`.
The timer is started on line 5.1b, just before the code to be timed is executed, and stopped on line 5.1c. The statement

\[
\text{watch.Start_Timer}
\]

(5.1)

runs the StopWatch's Start_Timer procedure. The time is requested on line 5.1d and, using a utility Sub, OutputTime(), not displayed, is output.

Finally, and importantly, the StopWatch is destroyed on line 5.1e. The line

\[
\text{Set watch = Nothing}
\]

(5.2)

removes the association between the name and the object it refers to and causes the object to be removed from memory.

The clean way that the StopWatch object is used should be contrasted with the timer used in Chapter 3. This littered main() with a variable. The OOP stopwatch exposes nothing to its client. The benefits of the StopWatch object are clear. Not only does it supply the required functionality but it does so in an intuitive and transparent way. A further strong plus is that it is highly re-usable; its class module can be added in anywhere (or made available as part of a library).

### 5.1.2 Declaring references and instantiating objects

References to an object must be declared using a Dim statement. Objects themselves differ from predefined and user-defined types in that they must be instantiated in a separate statement. Instantiation may happen, using the New keyword, in one of two ways:

1. Using a Set statement with New, when the object is immediately instantiated.
2. Using Dim with New, when the object will be instantiated the first time it is used.

Suppose that a Butterfly object type has been defined. Line (5.3) shows a reference to a Butterfly object being declared (in the Dim statement) and the object instantiated (in the Set... New statement).

\[
\text{Dim RedAdmiral As Butterfly: Set RedAdmiral = New Butterfly ' .}
\]

(5.3)

This form of the Set statement creates a new instance of a Butterfly and sets RedAdmiral to be a reference to it.

There is an alternative form of the Set statement in which a name is assigned a reference to a pre-existing object. For instance, in the fragment

\[
\begin{align*}
\text{Dim RedAdmiral As Butterfly: & Set RedAdmiral = New Butterfly} \\
\text{Dim RA As Butterfly: & Set RA = RedAdmiral}
\end{align*}
\]

(5.4)

RA is Set to be a reference to the same instantiation as RedAdmiral. If RedAdmiral is subsequently re-assigned to be a reference to a different instantiation of Butterfly, or is Set to Nothing, RA is not re-assigned and continues to refer to the same object.

The other way of instantiating an object is through Dim... New. Lines (5.5) illustrate.

\[
\begin{align*}
\text{Dim YellowBrimstone As New Butterfly ' a} \\
\text{ 'Various stuff not involving YellowBrimstone ' b} \\
\text{YellowBrimstone.FlappWings(54) ' c.}
\end{align*}
\]

(5.5)
Line (5.5a) declares YellowBrimstone to be a reference of type Butterfly. On line (5.5c) the FlapWings() method of YellowBrimstone is executed (with an argument of 54 for some reason). It is only on line (5.5c) that a New Butterfly is instantiated, and YellowBrimstone is set to be a reference to it, without the need of a Set statement. Sometimes this form of instantiation is convenient, for instance, if immediately on being declared, an initialization procedure of the object is called. If the first use of an object does not occur until a point some distance beyond the Dim statement, we strongly prefer the form in (5.3).³

There is one further special use of the Set statement; this is to cause references to objects to be destroyed. An example is line (5.6):

\[
\text{Set RedAdmiral} = \text{Nothing} \quad '. \tag{5.6}
\]

If RedAdmiral is a reference to a Butterfly object then the reference is disassociated from it. If RedAdmiral is the last reference to that particular Butterfly object then that object is destroyed and removed from memory.

It is good practice always to explicitly Set object references to Nothing, even if they would in any case be destroyed by going out of scope. The reason is ease of maintenance; code becomes clearer and much less error prone if you always Set and then Set to Nothing as a matched pair.⁴

This provides another good illustration of why one should not use Dim... New. You do not know where the object is subsequentially instantiated. If after line (5.5a) the first mention of YellowBrimstone is the statement Set YellowBrimstone = Nothing then it is instantiated on that line and immediately destroyed. Is this what the programmer intended? Probably not. It is safer by far to instantiate explicitly as in line (5.3).

**Some notation**

On line (5.5c) the FlapWings() method of the YellowBrimstone object of type Butterfly is executed. Suppose you want to refer to the FlapWings() method of the Butterfly object type in general, rather than in association with a specific instantiation. We use the notation

\[
\text{Butterfly::FlapWings()} \tag{5.7}
\]

for this purpose. This refers to the FlapWings() method of the Butterfly object type. If another object type, Avian say, also has a FlapWings() method the notation allows us to disambiguate between the two cases, writing either Butterfly::FlapWings() or Avian::FlapWings().

Using :: in this way with VBA is purely notational. Unlike C++, where :: is the scope resolution operator, in VBA :: has no syntactic significance.

**5.1.3 Using objects**

Objects contain data and procedures. They can be declared to be Public, Private or Friend. Public members are part of the object’s interface. They are callable from anywhere, including from outside the immediate project. Private members are part of the object’s implementation detail and are callable only within the class module. Friend members are visible only within their own project. The scope of Public members can be restricted to their project with an Option Private statement.

³ Although there would have to be a good reason for the declaration to occur so far in advance of the declaree being used.
⁴ It is bad practice not to. Perhaps not as serious as not delete-ing new-ed memory in C++, but bad enough.
An object’s client, here a procedure containing a reference to the object, may (i) execute a Public
procedure (a Function, Sub or Property) and (ii) get or set the value of a Public data member, should
there be any.
Calling a Public Function or Sub or accessing a data member is done using the same syntax as used
to access the data in a Type, with the qualifying character ‘.’. For instance suppose the Butterfly object
has a Public Sub named MakeChrysalis() and a Public Function named IsAirborne() returning
a Boolean, then these are called with

\[
\begin{align*}
\text{Call RedAdmiral.MakeChrysalis()} \\
\text{Dim InFlight As Boolean: InFlight = RedAdmiral.IsAirborne(183)}
\end{align*}
\]

(with an arbitrary argument to IsAirborne()). Just like ordinary variables, references to objects can be
passed around an application as arguments to procedures. It is also possible to have arrays and collections
of object references (see Chapter 14).

5.1.4 Defining objects

A VBA object is defined in a class module. Each class module defines a single object type whose name
is the name of the class module. To define a new object type insert a new class module into the project
and rename it with the object name (in the Properties window).

In a class module there will be

1. a data section declaring Private data. (There should rarely (if ever) be any Public data. The
   convention employed in this book is that names of Private data members always end in an underscore
   character.\(^5\) Constants, arrays, UDTs, and fixed-length strings cannot be declared to be Public.)
2. an events section defining and declaring the events recognized by the object (see Chapter 8);
3. a constructor and destructor, if needed;
4. definitions of Properties;
5. definitions of Public methods;
6. definitions of Private methods.

A variable must be declared before it can be used, so the data section is normally at the top of a
class module.

Constructors and destructors

There are two special methods – the object’s constructor and destructor. These have the signatures

\[
\begin{align*}
\text{Private Sub Class.Initialize()}
\text{Private Sub Class.Terminate()} \,' .
\end{align*}
\]

In VBA the constructor and destructor are just procedures that are run when objects are created or destroyed;
the act of running them does not of itself cause an object to be created or destroyed. The constructor,

---

\(^5\) For reasons that will become apparent it is preferable not to use the underscore character in the middle of a name (interface
implementation and event handling add suffixes and prefixes onto user-defined names – connected to them with underscore characters).
Hence we will prefer the name StockPrice_, for instance, to stock_price_. Nevertheless we continue to use names containing
underscore characters when there is no risk in doing so.
Implementing Models of Financial Derivatives

always named $\text{Class\_Initialize}()$, is run when an object of this type is instantiated. The destructor, always named $\text{Class\_ Terminate}()$, is run when the last reference to an object of the type is Set to Nothing or goes out of scope. The destructor is also called automatically, if required, if an error is thrown in one of the object’s member procedures but is caught only higher up the stack in another object. This means that dangling references are cleared up.

An important use of a destructor is to Set to Nothing references to objects that have been declared in the object, often by an object’s constructor or by a hand-crafted initializer procedure. The constructor can be used, to some extent, to initialize data members, which may include references to objects. Constructors take no argument; they correspond to default constructors in C++.

**Properties**

Property statements are intended as getters and setters for Private data held by objects. There are three sorts of Property statement, Property Get, Property Let and Property Set. The intended use of Property Get is to return the value of a data member; Property Let is to set the value of a data member; and Property Set is to Set the value of (a reference to) an object data member.

Properties are much the same as ordinary member procedures, but they differ in three ways. First, a Property Get must always return a value and a Property Let/Set must always take at least one argument. Second, a Property Let or Property Set when invoked can appear only on the left-hand side of an assignment. Third, unlike ordinary procedures, Property Gets and Property Let/Sets may share the same name. This last distinction is the important one. It makes Properties easy to use as getters and setters.

An example of defining a Property Get and a Property Let is given in Figure 5.2. This illustrates the use of a Property to implement a getter and setter pair. The class $\text{StockObject}$ contains a Private data member, $\text{StockValue}_\_\$, which the Properties Get and Let. The Properties are given a suggestive name, although the names of a Property Get or a Property Let/Set could be anything, and in a matched pair need not be the same.6

The use of the Property defined in Figure 5.2 is illustrated in the client code in line (5.10):

```
Dim An\_obj as StockObject: Set An\_obj = New StockObject
'...and later...
An\_obj.StockValue = 150 'executes Let
value = An\_obj.StockValue 'executes Get
```

The Property Let statement is being executed on line 3, and the Property Get on line 4. Because they have the same name the usage is natural.

---

6 Although they should be. When used as getter and setters, their usual function, it is evil to give Properties names that are misleading or unintuitive; indeed anything other than a name directly related to the name or function of the underlying data member, or set of data members, must be avoided.
Property Gets and Property Let/Sets need not come in matched pairs. A data member may have one without the other, or of course none at all.

A Property Get may take arguments. If a Property Get does take arguments then those of a corresponding Property Let/Set must be compatible.

Properties are Public by default; they can be declared as Private to restrict their scope to their module, or as Friend.

Properties are useful. They should be used to avoid having Public data members, and they allow additional code to be run when values are Get or Let/Set. For instance, the Property Let in Figure 5.2 could validate by checking if its argument is positive. Other examples occur throughout the book. A Property Get that only returns the value of a data member, with no additional functionality, is referred to as a vanilla getter.

5.1.5 Objects, references and lifetimes

Objects in VBA are conceptually fixed in memory and, in a real sense, inaccessible. By this we mean that once they are created somewhere in memory they stay there. They do not move; they cannot be copied or assigned.

You cannot transfer state from one object to another without breaking encapsulation. The only way to access them is indirectly, through references. References are mutable. These can be assigned and re-assigned and passed around an application at will. An object is destroyed automatically when, and only when, the last reference to it goes out of scope.

It is important to be clear in VBA about what is an object and what is a reference to an object. Consider the lines

\[
\begin{align*}
\text{Dim CoffeeTable As Table: Set CoffeeTable = New Table} & \quad \text{'}a \\
\text{Dim DinningTable As Table: Set DinningTable = CoffeeTable} & \quad \text{'}b.
\end{align*}
\]

Line (5.11a) instantiates a Table and assigns to CoffeeTable a reference to it, and (5.11b) assigns to DinningTable a reference to the object that CoffeeTable refers to; these two lines do not assign a Table to CoffeeTable and DinningTable.

Nevertheless we shall often injure reality and describe (5.11a), for instance, as assigning a Table to CoffeeTable.

Reference counting

When more than a small number of objects are being instantiated care must be taken to ensure that the application keeps track of all references to them. This is where reference counting, as an example of meta-class data, helps.

Meta-class data is data that is common to and shared by every instance of a class. For an application that values a book of options the Long representing the number of option objects that have been instantiated is an item of meta-class data that belongs to the option objects as a group, but not to any particular option object. In an application where there are several types of option objects (perhaps one type for average rate options and another for vanilla European options, et cetera) each option type might have an identifier code (for instance AVR for an average rate option, VAN for a vanilla European, et cetera). The String containing the identifier code is a piece of meta-class data common to every option of that type, belonging to no option in particular.

\[7\] There is nothing, apart from good sense, to prevent an object containing a reference to itself. Even if all references external to it are Set to Nothing the object is not destroyed, but sits there in memory, unreachable. This is the nearest that VBA comes to dangling references. It is only really a potential worry when using a decorator pattern.
Implementing Models of Financial Derivatives

In VBA one can mimic meta-class data either by defining a **Public** global variable in a standard module, or by putting a variable into a helper object (see Chapter 13). For the moment let us assume, to our horror, that the first course is taken.

Suppose we have some class, `AClass`, for which we need to keep some meta-class data. To be concrete suppose that we want (i) to keep track of how many `AClass` objects are currently instantiated and (ii) to assign to each a unique identifier so that we can tell two instances apart. The first idea is related to what in other languages is called reference counting, although here it should be called instance counting;\(^8\) the second means that the objects take care of their own identity, client code does not have to keep track of it for them. We say that objects that support (i) and (ii) are using tracking.

Define two global variables, `IDAClassEvil` and `NumAClassEvil`, both Longs. The first is to enable us to construct a unique ID and the second to count the number of `AClass` objects currently instantiated.

In `AClass`'s definition is the code shown in Figure 5.3. Whenever an `AClass` is instantiated its constructor increments both `IDAClassEvil` and `NumAClassEvil` and sets the `AClass`'s ID number `IDnum_` to `IDAClassEvil`. `NumAClassEvil` is decremented in `AClass`'s destructor so that the value of `NumAClassEvil` is exactly the number of `AClass` objects in existence. `IDAClassEvil` just keeps on increasing, giving each `AClass` a unique identifier.\(^9\)

Figure 5.4 shows tracking in use. The commentary in the figure describes what is going on. `MsgBox`es print out IDs as the procedure progresses. The statements assign references, not objects. Objects disappear only when all references to them are removed, explicitly by being `Set` to `Nothing` or being re-assigned, or implicitly by going out of scope; it is much better to be explicit.

`a_ref`, `b_ref` and `c_ref` initially point to separate objects. When `b_ref` is re-assigned to refer to object 1 (line 5.4a), object 2 loses its only reference and is destroyed. When `a_ref` is re-assigned to refer to object 3 (line 5.4b), `b_ref` is still left referring to object 1 so no object is destroyed. When `c_ref`...
is disassociated (line 5.4c) a_ref still refers to object 3; object 3 is finally destroyed when a_ref is re-assigned to object 1 (line 5.4d).

Occasionally it may be necessary to ensure that only a single instance of an object can ever exist. Achieving this in code is called the singleton pattern. The singleton pattern cannot be achieved directly in VBA but it could, for instance, be mimicked by tracking. The constructor could, for instance, throw (using the error handling idioms of Chapter 4 to clear up any objects that it has created) if an attempt is made to instantiate an occurrence of the object when NumAClassEvil is already non-zero.

A better method of applying the singleton idiom is given in Appendix E.

5.2 AN EXAMPLE: THE StopWatch OBJECT

In section 5.1.1 we saw a stopwatch object being used. Now we show how the stopwatch can be implemented. The class module StopWatch, defined in Figure 5.5, implements it. The StopWatch object illustrates some of the features we have seen so far. It has Private data, a constructor, a Property Get, and Public Subs that operate on the data.

It is natural to define the StopWatch as an object because (i) its data is persistent, that is, its values must be maintained in between function calls, (ii) several different procedures act on the same set of data, (iii) if multiple timers are needed several different instantiations of the StopWatch can exist at any one time, and (iv) it is easy to re-use.

The Public methods and the Property Get have signatures

```
Public Sub Start_Timer()
Public Sub Stop_Timer()
Public Sub Reset_Timer()
Public Property Get Elapsed_Time() As Double '.
```

Here it is safe to use the underscore character in procedure names, but I should not. In Chapter 6 where a polymorphic stopwatch is described the procedure names have to be changed to remove the underscore character.
Stop_Timer() and Start_Timer() pause and restart the stopwatch. Reset_Timer() stops the watch and resets it to zero, and Elapsed_Time() passes the time from the stopwatch on to the client.

There are no Private member functions. The Private data members are

\[
\begin{align*}
\text{Private elapsed\_time\_ As Double} \\
\text{Private Timer\_running\_ As Boolean} \\
\text{Private current\_begin\_time\_ As Double}
\end{align*}
\]

(5.13)

Timer\_running\_ is False if the watch is stopped or has been reset; True if it is currently running. elapsed\_time\_ is the accumulated time that the stopwatch has been running up until the last time it was stopped; current\_begin\_time\_ is the time that the watch was last started (after being stopped or reset). When the StopWatch is created it is not running and there is no elapsed\_time\_ so it has a constructor that sets Timer\_running\_ and current\_begin\_time\_ to False and 0 respectively.

The Property Get, Elapsed\_Time(), is an example of a getter which does more than simply return the value of a data member. It looks to see if the stopwatch is running (that is, if Timer\_running\_ is True or False). If the stopwatch is running it returns

\[
\text{Timer - current\_begin\_time\_ + elapsed\_time\_},
\]

(5.14)
the sum of the time elapsed since the stopwatch was last started plus the accumulated time up to the time it was last stopped. If the stopwatch is not currently running it returns \texttt{elapsed\_time}. This is exactly the behaviour of a physical stopwatch.

The \texttt{Public} methods operate in the obvious way. \texttt{Reset\_Timer()} just sets \texttt{Timer\_running} to \texttt{False} and the \texttt{elapsed\_time} to zero. \texttt{Start\_Timer()} does nothing if the stopwatch is already running and if the stopwatch is stopped it sets \texttt{Timer\_running} to \texttt{True} and sets the value of \texttt{current\_begin\_time} to the current time. Similarly, \texttt{Stop\_Timer()} sets \texttt{Timer\_running} to \texttt{False} and updates the value of \texttt{elapsed\_time}.

The elapsed time returned by the \texttt{StopWatch} is a \texttt{Double}. If the elapsed time is needed in Hour:Minute:Second format the output will have to be formatted this way by the client. Note also that the \texttt{StopWatch} uses the \texttt{Timer()} function to return the number of seconds that has elapsed since midnight. If a timing operation starts before midnight but ends after midnight the \texttt{StopWatch} in its current form will return a bad time. Kimmel \textit{et al}. (2004) describe a high-precision timer object. This should be used in preference to the \texttt{StopWatch} object given here when very accurate times are required; also it does not reset at midnight. A third timing mechanism is provided by Getz and Gilbert (2001). All three stopwatches are implemented in the Monte Carlo application described in section 5.4. They share the same interface but the internal implementation details are different. We compare them in more detail in Chapter 6.

We saw the stopwatch in use in Figure 5.1. It is an extremely useful utility object. An obvious extension to the object described here is to turn the \texttt{StopWatch} into a multi-timer, enabling it to have more than one timer running simultaneously. See exercise 1.

### 5.3 FURTHER HELPFUL VBA FEATURES

There are a few further features that it is convenient to mention here. These are the \texttt{Object} type, the \texttt{Is} and \texttt{Me}, the \texttt{IsObject()} function, and the \texttt{With} statement.

**The Object type**

It is possible to declare an object to be of generic \texttt{Object} type with

\[
\text{Dim RedAdmiral As Object } . \tag{5.15}
\]

\texttt{RedAdmiral} is now a reference to some kind of object, but which kind is yet to be determined. Later it can be \texttt{Set} to a specific object type, for example

\[
\text{Set RedAdmiral = New Butterfly } . \tag{5.16}
\]

or

\[
\text{Set RedAdmiral = RefToAButterfly } . \tag{5.17}
\]

Objects of type \texttt{Object} can also be passed as arguments to procedures. This enables run-time polymorphism, since a procedure need not know the specific type of object it is being passed, but it is much better to use interfaces (see Chapter 6). Using an interface means that the procedure knows exactly what \texttt{Public} member functions can be called. Using \texttt{Objects} breaks type. Because it can be anything there is nothing to prevent it being anything. We limit our use of \texttt{Object} to circumstances in which there seems to be no easy alternative. \texttt{Object} may not be evil but it is certainly reprobate.

In any case, using the \texttt{Object} type to achieve polymorphism is expensive.
The *Is* keyword

This is used to test whether two references refer to the same object, or whether an object reference is Nothing. The syntax is

\[
\text{Dim IsSame As Boolean: IsSame = RedAdmiral Is SomeButterfly} \quad \text{'}a, \\
\text{If RedAdmiral Is Nothing then Call MsgBox("Swatted")} \quad \text{'}b.
\]

In the first line IsSame is True only if RedAdmiral and SomeButterfly both refer to the same object. The line (5.18b) tests if RedAdmiral is Nothing.

Is can also be used with the key word TypeOf in If statements. The line

\[
\text{Dim IsBut As Boolean: IsBut = TypeOf RedAdmiral Is Butterfly} \quad (5.19)
\]

sets IsBut to True if the object RedAdmiral is indeed of type Butterfly.

The *Me* keyword

This is used to create and pass as an argument to a procedure a reference to the calling object, just like the C++ *this* keyword. Examples of this extremely important feature occur in Chapter 6 and further discussion is deferred until then.

The *IsObject()* function

VBA supplies a function, *IsObject()* , to determine whether the type of its argument is an object or not. It takes a variant as its argument. It returns True if the argument has been declared as an object type, and False otherwise.

The *With* statement

A With/End With block is a construction that enables multiple method and Property calls to be made on an object without explicitly qualifying the method with the reference name. This leads to succinct code and greater clarity. In Figure 5.6 the WriteLine method is applied successively to object txt_f in a With block.\(^{11}\) The object is named in the With statement and afterwards inside the With block it is implicit that its methods are being called. Methods of other objects can still be called but they need to be qualified.

```vba
Set txt_f = File_Sys.CreateTextFile("d:\testfile.txt", True)
With txt_f
    .WriteLine "Freude, schoener Goetterfunken,"
    .WriteLine "Tochter aus Elysium,"
    .WriteLine "Wir betreten feuertrunken,"
    .WriteLine "Himmlische, dein Heiligtum."
End With
```

**Figure 5.6** Fragment: using the With statement

\(^{11}\) Here the File_Sys.CreateTextFile method is using the VBA FileSystemObject to create the text file written to in the With statement (see Chapter 9).
5.4 OBJECTS IN THE MONTE CARLO APPLICATION

Now we turn our attention to implementing the Monte Carlo application with objects. Upon discovering OOP there is a temptation to turn everything into an object. We are unable to resist, citing pedagogical intent.

When designing an application, and the objects within it, it is important that each object is cohesive and self-contained. An object should be responsible for one task and one task only, that is its alone, removing knowledge of its implementation from the rest of the application. Other parts of the application see only the object’s interface, and hence can access the functionality of the task that the object is responsible for.

There are various bits of functionality in the level 2 Monte Carlo application that can be rewritten as objects. The StopWatch we have already seen. Other obvious objects include input and output objects, an object to generate normal variates, an error handler object, and a payoff object. Perhaps less obvious, but conceptually at least as important and satisfying, is an object to accumulate results from each sample path. This embryonic set of objects will be developed considerably in later chapters.

Input and output do not have a lot to do with Monte Carlo per se, so it is easy to justify the separation of this functionality. Also, generating normal variates is a useful general-purpose activity that a Monte Carlo application should use, but should not be restricted to the Monte Carlo application alone.12

Anticipating the focus of Chapter 6 we put the computation of payoffs into its own object. Here this is a bit of an over-kill, but once we want to have a number of interchangeable payoffs the structure makes a lot more sense.

We present two versions of the basic object-oriented application. The first, version 3.1, is just a variation of the level 2 application where objects replace procedures, as outlined above (but without an object corresponding to the MC_app() procedure). It substitutes method calls for procedures but it has the same structure as the level 2 application shown in Figure 4.11.

The second, version 3.2 (Figure 5.16 in section 5.4.2), introduces a level of indirection between main() and the Monte Carlo application. An intermediate object, AppObWrapper, takes over the responsibility for creating the objects associated with the application, including input and output. main() is responsible only for setting up error handling and for creating the AppObWrapper object.

All the objects used here are of the stand-alone, non-polymorphic, variety. The power of objects only really becomes apparent when we go polymorphic in Chapter 6. Even so, the current example illustrates the VBA approach to objects and some design features.

5.4.1 Level 3 Monte Carlo: First version

The level 3 main() is shown in Figure 5.7. It is in the spreadsheet MC_example_v3a.xls. Comparing 5.7 with Figure 4.10 (page 49) the first thing that comes to mind is that it is longer. This is because it has swathes of code devoted to creating and Setting to Nothing a set of objects. Lines 5.7a to 5.7g instantiate and initialize the objects used in the simulation; these are all Set to Nothing in lines 5.7o to 5.7u. This clutter is removed in version 3.2.

Level 3.1 implements a path-wise Monte Carlo. Complete paths are generated in one go, so in Figure 5.7 there is a loop only for the sample paths, and no time step loop. The generator object generates an entire sample path but keeps it to itself. On line 5.7i it is asked to return just the final value of S, which is passed over to the payoff object.

12 In a numerical application there may be performance issues associated with making a large number of method calls. We investigate this further in Chapter 14.
Public Sub main() 'Level 3.1
On Error GoTo Error_Handling
  Dim err_handler As ErrorHandler: Set err_handler = New ErrorHandler 'a
  Dim data_in As InputManager: Set data_in = New InputManager 'b
  Dim data_out As New OutputManager: Call data_out.SetValues(data_in) 'c
  Dim acc As New ResultAccumulator: Call acc.SetValues(data_in) 'd
  Dim gen As New PathGenerator: Call gen.SetValues(data_in) 'e
  Dim pay_off As New Payoff: Call pay_off.SetValues(data_in) 'f
  Dim watch As New StopWatch: Call watch.Start_Timer 'g
  Dim j As Long
  For j = 1 To data_in.M 'for each sample path 'h
    Call data_out.OutputCounter(j)
    Dim final_S As Double: final_S = gen.Get_new_final_S 'i
    Dim cash As Double: cash = pay_off.pay_off(final_S)'j
    Call acc.update(cash) 'k
  Next j
  data_out.c_value = acc.Get_c_value 'l
  data_out.standard_error = acc.Get_se 'm
  data_out.Elapsed_Time = watch.Elapsed_Time 'n
Clear_Up:
  Set watch = Nothing 'o
  Set pay_off = Nothing 'p
  Set gen = Nothing 'q
  Set acc = Nothing 'r
  Set data_out = Nothing 's
  Set data_in = Nothing 't
  Set err_handler = Nothing 'u
Exit Sub
Error_Handling:
  Call err_handler.Handle_error 'v
Resume Clear_Up
End Sub

Figure 5.7  The version 3.1 main()

Here some objects have been declared using Dim...New. This is only because they are immediately instantiated on the same line when their SetValues() method is called. Even here using Dim...New has a bad taste.13

We discuss each object in turn, discussing its place in the Scheme Of Things.

The ErrorHandler object

The class module ErrorHandler is shown in Figure 5.8. The reader will note that Sub Handle_error() is essentially identical to Sub ErrorHandle() in Figure 4.4; the only difference is that it has been elevated to the status of a Public method in the object ErrorHandler (and the name has changed).

In Figure 5.7 an instance of ErrorHandler is created on line 5.7a with the Set...New statement and a reference, err_handler, assigned to it. If an error occurs, control passes to Error_Handling: and the method err_handler.Handle_error is Called on line 5.7v. Control then returns to Clear_Up:.

13 I coded this way only because the layout looked elegant.
Public Sub Handle_error()
Dim msg As String
Dim ErrorNum As Long: ErrorNum = Err.Number - vbObjectError
If ErrorNum > 0 And ErrorNum < 65535 Then
    msg = "Application defined error" & vbCrLf & 
         "AppErr is: " & ErrorNum & vbCrLf & 
         "Source: " & Err.source & vbCrLf & 
         "Description: " & Err.Description
Else
    msg = "Visual Basic error" & vbCrLf & 
         "Number is: " & Err.Number & vbCrLf & 
         "Source: " & Err.source & vbCrLf & 
         "Description: " & Err.Description
End If
Call MsgBox(msg, , "Object Error")
End Sub

Figure 5.8 The ErrorHandler object

There is now some clearing up to be done. All references, including that to the error handler itself on line 5.7u, have to be Set to Nothing before main() exits.

**The InputManager object**

The input manager is responsible for inputting from the front-end. Its purpose is to relieve the Monte Carlo application of knowledge of, and responsibility for, this functionality. For instance, only InputManager knows the locations on the front-end where the data is found.

The InputManager object is created in main() on line 5.7b, and on lines 5.7d, 5.7e and 5.7f is passed as an argument to an initializer method of application objects that need access to parameters read in from the front-end; these are the Payoff object, the ResultAccumulator object and the PathGenerator object. Finally it is Set to Nothing on line 5.7t.

Its definition is given in Figure 5.9. Its Private data members are in correspondence with the inputs from the front-end.14 These are read in by the Private method Read_in_data(). This is called by InputManager’s constructor, so that data is read in from the front-end automatically when an InputManager object is created. The only other elements of the object are a sequence of Properties, a getter for each data member.

The InputManager object is a hyped-up version of the MC_data UDT defined in Chapter 4. It has the same data, now Private and accessed with getters, set in the class constructor.

An initialization method is necessary only because, in VBA 6.x, a constructor cannot take arguments; it is far better, in languages where it is permitted, to be able to initialize while constructing.

**The Payoff object**

This is an easy object. Its definition is given in Figure 5.10. It has a single data member, X_ corresponding to the strike X, initialized in the interface method SetValues() on line 5.7f of main(). The only other function is the Friend interface method pay_off(), that given a value of S returns the payoff for that value, called on line 5.7j. It is Set to Nothing on line 5.7p.

---

14 Except for the output counter interval which is hard-wired in.
Private sigma_ As Double 'Stock volatility
Private rr_ As Double 'Stock short rate
Private S_0_ As Double 'Initial stock price
Private T_ As Double 'Final time
Private X_ As Double 'Strike
Private N_ As Long 'Number of time steps
Private M_ As Long 'Number of sample paths
Private NC_ As Long 'Interval for output counter

Private Sub Class_Initailize()
  Call Read_in_data
End Sub

Friend Property Get sigma() As Double: sigma = sigma_: End Property
Friend Property Get rr() As Double: rr = rr_: End Property
Friend Property Get S_0() As Double: S_0 = S_0_: End Property
Friend Property Get X() As Double: X = X_: End Property
Friend Property Get T() As Double: T = T_: End Property
Friend Property Get N() As Long: N = N_: End Property
Friend Property Get M() As Long: M = M_: End Property
Friend Property Get NC() As Long: NC = NC_: End Property

Private Sub Read_in_data()
  sigma_ = get_double(14, 6)
  rr_ = get_double(13, 6)
  S_0_ = get_double(12, 6)
  X_ = get_double(17, 6)
  T_ = get_double(18, 6)
  N_ = get_long(12, 9)
  M_ = get_long(13, 9)
  NC_ = 1000 'Hard-wired in
End Sub

Figure 5.9 The InputManager object

Private X_ As Double
Friend Sub SetValues(ByRef data As Input_manager)
  X_ = data.X
End Sub
Friend Function pay_off(S As Double) As Double
  pay_off = my_max(0, S - X_)
End Function

Figure 5.10 The Payoff object
The Payoff object is an example of a type of object called a functor.\(^\text{15}\) A functor has a single application method, here called pay_off() but often just called run(). It may also have structural methods, like SetValues(), but that is it. The idea is that a functor is, like Payoff, just a function turned into an object.

**The OutputManager object**

This object is responsible for output. Only the output manager knows where on the front-end results are to be output to. In main() it is created on line 5.7c and destroyed on line 5.7s. In lines 5.7l, 5.7m and 5.7n it accepts values of item to output from the ResultAccumulator and StopWatch objects. These lines call its Property Let methods.

The OutputManager object’s definition is given in Figure 5.11. It has four data members. Three correspond to the three items to be output: the option value, the standard error and the elapsed time. The fourth is the counter output interval used by the OutputCounter() method, set by the SetValues() method.

```vba
Private c_value_ As Double
Private standard_error_ As Double
Private elapsed_time_ As Double
Private interval_ As Long
```

```vba
Private Sub Class_Initialize()
    Cells(17, 9).Value = ""
    Cells(18, 9).Value = ""
    Cells(8, 7).Value = ""
    Cells(9, 7).Value = ""
End Sub
```

```vba
Private Sub Class_Terminate()
    Call Write_out_data
End Sub
```

```vba
Friend Sub SetValues(ByRef data As InputManager)
    interval_ = data.NC
End Sub
```

```vba
Friend Property Let c_value(ByVal x As Double)
    c_value_ = x
End Property
```

```vba
Friend Property Let standard_error(ByVal x As Double)
    standard_error_ = x
End Property
```

```vba
Friend Property Let Elapsed_Time(ByVal x As Double)
    elapsed_time_ = x
End Property
```

```vba
Friend Sub OutputCounter(j As Long)
    If j / interval_ = j \ interval_ Then Cells(8, 7).Value = j
End Sub
```

```vba
Private Sub Write_out_data()
    Cells(17, 9).Value = c_value_
    Cells(18, 9).Value = standard_error_
    Cells(9, 7).Value = elapsed_time_
End Sub
```

Figure 5.11  The OutputManager object

\(^{15}\) In C++ a functor is an object that overloads operator(). This is not possible in VBA, but the definition given here captures the same notion as that envisaged in C++.
method. Each of the three output data members has a Property Let to set its value. The object has a constructor that erases the current contents of the output cells, and a destructor that calls a Private method, Write_out_data(), that outputs its data members to their destination cells. Because output happens in the OutputManager’s destructor, no explicit procedure call is required to accomplish it.

Outputting the counter is a different category of output to outputting the option value, et cetera. The counter is pure monitoring information; the others are actual results from the application. One can – and we shall – argue that the counter should be treated differently to the other data items, but not yet. For the moment we just slam it into the same object as the others.

The remaining three objects are, from the Monte Carlo perspective, the most directly relevant.

The PathGenerator object

The PathGenerator object has the responsibility of generating values of the underlying asset, S, for the final time, T. Line 5.7e of main() declares a reference, gen, to an instance of a PathGenerator object in the Dim... New statement, and on the same line the object is created when the method gen.SetValues(), which initialized the object, is called. The object is used on line 5.7i when gen.Get_new_final_S(), its second and last method that delivers its functionality, is called. Finally it is Set to Nothing on line 5.7q.

Figure 5.12 gives the definition of the PathGenerator object. It has two interface methods, SetValues(), which initializes it, and Get_new_final_S(). SetValues() takes the InputManager as its argument and uses it to set the values of its Private data members, N_ and S_0_. It also computes the values of two further data members, drift_ and s_root_t_, that are used by Get_new_final_S().

The final data member is a reference norm_gen_ to a NormalGenerator object (see below). An object contained by another object is called a composited object. PathGenerator has both a constructor and a destructor to manage norm_gen_; it follows the idiom of assigning the reference in the constructor and destroying it in the destructor. In C++ this is called the RAII (resource acquisition is initialization) idiom; see below.

norm_gen_ is used to generate normal variates in the Function Get_new_final_S(). Get_new_final_S() generates an entire path of values of S but returns only the value for the final value.

The NormalGenerator object

This object is again straightforward. Figure 5.13 displays its definition. Its functionality is the Public method GetNormal() that returns a normal variate. We have already seen how it is used by the PathGenerator object.

The advantage to having an object to generate a normal variate is that it avoids the use of expensive Statics. Instead the variables that were previously Static, Spare_normal_flag, IA_, et cetera, are Private data members whose values are Set in the constructor, Class_Initialize(). This is a much clearer than the previously used, rather contrived, Statics idiom. Of course there is now a method call instead.

Note that there is a pair of Properties to Get and Let the seed for the uniform generator. This is a potentially useful feature for advanced users who might want to control the setting of this parameter; however it is a potentially dangerous feature in less knowledgeable hands. We include these Properties here for illustration. Where they are not present they can be added easily by users who need them.

The ResultAccumulator object

The final object is the ResultAccumulator object whose definition is given in Figure 5.14. This is the only object among those we have defined so far (apart from the StopWatch) whose functionality cannot really be substituted for with an ordinary procedure.
Figure 5.12  The PathGenerator object

`PathGenerator` looks after the accumulation of results from each sample path and from them computes the option value and its standard error. This involves three separate tasks; initializing the accumulators, updating them when a result arrives and, from them, computing the values required.

Its data members are the accumulators, `acc_values_` and `acc_squares_`, the number of results that have been accepted, `M_`, and the discount factor, `discount_`. `acc_values_`, `acc_squares_` and `M_` are initialized (to zero) in the `ResultAccumulator`'s constructor, and `discount_` is computed in the initializer method `SetValues()`.

When the client calls the `update(value)` method with a new result, `value`, it is accumulated on to `acc_values_` and `acc_squares_` which compute the sum of values and the sum of values squared, respectively. When the client requests an option value or a standard error the `Property Get c_value()` and `se()` are called. These are non-trivial. They compute the option value and standard error, respectively, before returning them.

Note that the option value and its standard error are not stored; they are computed at need. `c_value` and `se` can be called at any time to return the results so far.
' NormalGenerator
Private Spare_normal_flag_ As Boolean
Private Spare_normal_ As Double
Private Const IA_ As Long = 16807 'Give fixed values
Private Const IM_ As Long = 2147483647
Private Const AM_ As Double = 1 / IM_
Private Const IQ_ As Long = 127773
Private Const IR_ As Long = 2836
Private Seed_ As Long

Private Sub Class_Initialize()
    Spare_normal_flag_ = False
    Spare_normal_ = 0
    Randomize 'Randomize sets the seed for Rnd
    Seed_ = Int(Rnd * IM_) 'Initialises Seed
End Sub

Friend Property Get seed() As Long: seed = Seed_: End Property
Friend Property Let seed(sd As Long): Seed_ = sd: End Property

Public Function GetNormal() As Double
    Dim u As Double, v As Double
    If Spare_normal_flag_ Then
        GetNormal = Spare_normal_
        Spare_normal_flag_ = False
        Exit Function
    End If
    Dim W As Double: W = 1#
    Do While W >= 1#
        u = 2# * ran0() - 1#
        v = 2# * ran0() - 1#
        W = u * u + v * v
    Loop 'Not rejected; construct the normal variates
    Dim C As Double: C = Sqr(-2# * Log(W) / W)
    GetNormal = C * u
    Spare_normal_ = C * V 'Store the spare normal for the next call
    Spare_normal_flag_ = True
End Function

Private Function ran0() As Double 'Adapted from Press et al.
    Dim k As Long: k = Int(Seed_ / IQ_)
    Seed_ = IA_ * (Seed_ - k * IQ_) - IR_ * k
    If Seed_ < 0 Then Seed_ = Seed_ + IM_
    ran0 = AM_ * Seed_
End Function

Figure 5.13 The NormalGenerator object

Using ResultAccumulator is straightforward. It is created and destroyed on lines 5.7d and 5.7r. On line 5.7k its update() method is called to register a new payoff. Finally on lines 5.7l and 5.7m its Property Gets are called to give the option value and the standard error to the OutputManager object.

This version of ResultAccumulator sticks closely to the previous non-OOP examples. In this example interest rates are non-stochastic so that the discount factor is constant – the same for every sample path. It can be precomputed and applied, as in Figure 5.7, only at the final step to compute the option value and its standard error. In models where the interest rate is stochastic, and the discount factor different for each sample path, each payoff must be discounted by its own discount factor.
Introducing Objects: Level 3

Figure 5.14  The ResultAccumulator object

Assessment of the version 3.1 design

Inspecting main() in Figure 5.7, we see that all unnecessary functionality has been stripped away and put into separate objects, each responsible for one – and only one – task.

The declaration section now declares objects, not primitive data items, initializing them with a uniform syntax. It is clear what each object does. There is still a main loop to generate payoffs along successive sample paths, epitomizing the Monte Carlo method, but the task of sample path generation has been pushed away into a separate PathGenerator object. The functionality associated with computing results is also removed to its own object.

The penultimate section of main() cleans up the objects earlier created and, finally, there is the error handling section.

The only procedure left in the application standard module is main(). Everything else has gone, tidied away, into its own functional compartment.

What is so good about this? As long as the interface is left unaltered the implementation of the interface in any object can change – the innards of Public functions, the number and type of Private functions, and the Private data members – without affecting any other part of the application. The implementation of the PathGenerator can change without affecting one iota the ResultAccumulator and vice versa; the uniform random number generator can change without affecting the OutputManager object.

Functionality is completely encapsulated within objects, and as such can be interchanged between applications. The InputManager and OutputManager are pretty much tied to just this application, but the other objects can be re-used elsewhere.
Note however that, as advertised, the structure of this level 3 application is identical to the level 2 version given in Figure 4.11. Adding objects has made no difference.

The second version of the design begins to illustrate how objects can be used in an application to perform structural roles in addition to providing basic functionality.

5.4.2 Level 3 Monte Carlo: Version 3.2

The difference between version 3.2 (MC_example_v3b.xls) and version 3.1 is the interposition of an object, AppObWrapper, in between main() and the Monte Carlo. The definitions of none of the other objects are changed. The new main() is given in Figure 5.15. It implements just two bits of functionality. Its main burden is to act as a sink for errors thrown elsewhere in the application. It instantiates an ErrorHandler object and the standard idiom for error trapping that goes with it. This functionality is wrapped around a composited AppObWrapper object.

Its second task is to execute the run() method of the AppObWrapper object. This object is responsible for instantiating and destroying the objects that do the real work. It is given in Figure 5.16. It has, as Private data members, all of the objects previously instantiated in main() (apart of course from the ErrorHandler still in main()). These are Set and initialized in AppObWrapper’s constructor and are destroyed in the destructor.16 The destructor also asks the ResultAccumulator to give OutputManager the option value and its standard error and also passes it the elapsed time. These are output when the OutputManager goes out of scope as before.

The run() method is now looking very lean and streamline. It is no longer encumbered by extraneous object management stuff. That is all put where it belongs, in the AppObWrapper’s constructor and destructor.

**RAII**

The idiom of instantiating objects or setting references in a constructor and destroying the object or Setting to Nothing a reference in the corresponding destructor is extremely important. It is so important as to have its own acronym, RAII, short for “resource acquisition is initialization”. The idea is that whatever

```'
Public Sub main()  'Level 3.2
On Error GoTo Error_Handling
    Dim err_handler As ErrorHandler: Set err_handler = New ErrorHandler
    Dim the_app As AppObWrapper: Set the_app = New AppObWrapper
    Call the_app.run
Clear_up:
    Set the_app = Nothing
    Set err_handler = Nothing
Exit Sub
Error_Handling:
    Call err_handler.Handle_error
    Resume Clear_up
End Sub
'
```

Figure 5.15 The level 3.2 main()

16 There is now no need to use Dim...New. This construction is immediately removed.
resource an object needs it should acquire in its constructor and release in its destructor. Conversely, if there is a resource that an application requires then there should be an object whose task is to manage it by grabbing it in its constructor and releasing it in its destructor. In our case the resource is a set of objects that need instantiating and destroying.

**Assessment of the version 3.2 design**

Version 3.1 introduced objects. Now version 3.2 has introduced a separate object just to look after them. There is now an additional layer in our application (Figure 5.17). `main()` delegates the Monte Carlo
Numerics to a layer lower down in the application. The structure is looking quite elegant. All the messy stuff is hived off out of the way into `AppObWrapper`'s constructor and destructor.

At the moment there is only one layer lower down, but you can see where this is heading. The current design, version 3.2, is far superior to the old level 1, but there is plenty to do yet. We could, for instance, decouple the part of the application that sets up the I/O from the part that does the Monte Carlo numerics proper. This is coming imminently in Chapter 6.

### 5.5 SUMMARY

Using objects enables code to be re-used. Actually, at the moment things are not quite suitable for re-use. For that one needs separately defined interfaces enabling proper polymorphic design, but that is coming up in Chapter 6.

But for now note the following: the rewrites of `main()` accomplished in this chapter clarifies the nature of the tasks that it is undertaking. It can be seen to do three things; of course the first is to run the Monte Carlo application, but now two other tasks, not specifically related to the Monte Carlo application, can be identified. `main()` has the second task of managing a set of objects, setting them and destroying them; in version 3.1 it does this itself, in version 3.2 it delegates the task to the `AppObWrapper` object. Third, as in level 2, it is responsible for setting up and managing error handling.

In Chapter 6 we move on to simplify the application yet further. The `AppObWrapper` looks after both the object resources used specifically to determine the Monte Carlo application, and also the objects used to support the application – the input and output channels. In the next version the Monte Carlo numerics are decoupled from the I/O streams.

### 5.6 EXERCISES

What objects should there be in an application? As well as standard structural objects, what application-specific objects are either worthwhile, or necessary or elegant? These exercises ask you to consider not only how objects should be implemented but also which objects are appropriate in an application.

1. The `StopWatch` object (Figure 5.5) can time only a single operation at a time (without instantiating more than one `StopWatch`). How could you adapt the `StopWatch` to enable a single instantiation to time more than one operation at once?
2. **The complex number exercise.** Construct a complex number object that can be used to facilitate complex arithmetic in VBA. Which complex number operations would you want to implement? How would you implement them in VBA?

Note that VBA does not allow you to overload operators. It is illegal to write code like

\[
\begin{align*}
\text{Dim } a \text{ As Complex: Set } a = \text{New myComplex} & \quad \text{'a,} \\
\text{Dim } b \text{ As Complex: Set } b = \text{New myComplex} & \quad \text{'b.} \\
\text{Dim } c \text{ As Complex: Set } c = a + b & \quad \text{'c, Illegal.}
\end{align*}
\]

It is necessary to work around this restriction.

3. **The string object exercise.** VBA has a primitive `String` type and lots of built-in procedures that work with `Strings`. As an exercise construct a string object, `StrObj`, that contains a `String` as a Private data member but which has methods to operate on it, endowing it with its own functionality.

Implement the methods listed in Table 5.1.

4. **Pi stream.** It is natural to convert the \( \pi \) functions into object functors. Do this, adding in other structural objects, to rewrite `pi_app_v2.xls` as a level 3 application, `pi_app_v3.xls`.

5. **Implied volatility stream.** From the implied volatility application, `IV_app_v2.xls`, implement the functionality of the bisection algorithm as an object and convert the Black–Scholes function into a functor. Add in other objects as appropriate to produce a level 3 application `IV_app_v3.xls`. Implement it so that code like the following can execute:

\[
\begin{align*}
\text{Dim } fn \text{ As BSFunctor: Set } fn = \text{New BSFunctor} & \quad \text{'a,} \\
\text{Dim } sv \text{ As BisectionSolver: Set } sv = \text{New BisectionSolver} & \quad \text{'b,} \\
\text{Call } sv.\text{SetValues(whatever): Call } fn.\text{SetValues(ifneeded) } & \quad \text{'c,} \\
\text{Call } sv.\text{FindRoot(fn, target) } & \quad \text{'d,} \\
\text{Dim } root \text{ As Double: root = sc.GetRoot()} & \quad \text{'e,} \\
\text{Dim } fval \text{ As Double: fval = sc.FnValue()} & \quad \text{'f.}
\end{align*}
\]

### Table 5.1 Methods for the `StrObj` object

<table>
<thead>
<tr>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sub construct(s As String)</td>
<td>Initializes the <code>StrObj</code> with the <code>String</code> <code>s</code></td>
</tr>
<tr>
<td>Function convertToStrg() As String</td>
<td>Returns the <code>String</code> defining the state of the <code>StrObj</code></td>
</tr>
<tr>
<td>Function size() As Long</td>
<td>Returns the number of characters in the <code>StrObj</code></td>
</tr>
<tr>
<td>Function at(N As Long) As String</td>
<td>Returns character at the <code>N</code>th position in the <code>StrObj</code></td>
</tr>
<tr>
<td>Function find(s As StrObj) As Long</td>
<td>Returns the start position of <code>s</code> in the <code>StrObj</code>, if it is a sub-<code>StrObj</code></td>
</tr>
<tr>
<td>Sub replace(N As long, L As long, s As StrObj)</td>
<td>Replaces characters in positions <code>N</code> to <code>N + L - 1</code> by <code>s</code></td>
</tr>
<tr>
<td>Function count(ch As String) As Long</td>
<td>Counts how many times the character <code>ch</code> appears in the <code>StrObj</code></td>
</tr>
<tr>
<td>Function copy(N As long, L As long) As StrObj</td>
<td>Returns the characters in positions <code>N</code> to <code>N + L - 1</code> as a <code>StrObj</code> object</td>
</tr>
<tr>
<td>Sub delete(N As long, L As long)</td>
<td>Deletes the characters in positions <code>N</code> to <code>N + L - 1</code> from the <code>StrObj</code></td>
</tr>
<tr>
<td>Sub append(s As StrObj)</td>
<td>Appends <code>s</code> to the end of the <code>StrObj</code></td>
</tr>
</tbody>
</table>
Also implement the method of false position and Ridders’ method (Appendix H) as objects that can work with the Black–Scholes functor.


   What objects should be in a level 3 PDE application? These should include a slice object and an evolver object, with a solver object called by the evolver to perform the tri-diagonal inversion, as well as structural objects. The evolver object should perform both the backwards evolution step for continuation values and the American comparison step, if required. Boundary values can be set in the slice object, but are (partially) determined by the option object. How should they be set in your code?

7. **Lattice stream.** Identify the objects in the lattice application Lattice_application_v2.xls. These should include slice objects in addition to objects representing options and the process, and the structural objects. Implement an object-oriented level 3 version of the lattice application as Lattice_application_v3.xls.
Polymorphism and Interfaces: Level 4

The previous level 3 versions of the application used objects, but did not exploit anything like their full power. The level 4 application, MC_example_v4.xls, introduces polymorphism. In VBA this means defining and using interfaces. The level 4 application also confirms the role of main() as a top-level manager rather than part of the application itself.

We extend the functionality of the previous version of the numerical side of the application to the computation of values of barrier options. In a major extension to the level 3 versions we introduce an option object. This has a composited payoff object but now we have the power to value options much more complex than the European options we have been using to date. We give two related examples: double barrier knock-out and knock-in options. Sample paths in this case – a very simple one-off option valuation problem – are evolved step by step so that at each step a test can be made of whether the option has knocked in or out.1

We saw in Chapter 5 that the structure of the application was improved by adding in a separate object to manage the instantiation of other objects in the application. We now go a step further by adding in a second layer of indirection between main() and the Monte Carlo application, specifically to manage input and output objects.

We start by giving an illustration of polymorphism in client code, then we catch up with some explanation of the VBA mechanisms for implementing polymorphism. Section 6.4 presents a polymorphic Monte Carlo application. The design is assessed in section 6.4.

6.1 POLYMORPHISM

Ordinarily if you want to change the functionality of some code you have to rewrite it. One way of minimizing the amount of rewriting that is needed is to put things inside functions. Then you may be able to rewrite what goes on inside the function without having to change the code that calls the function. For instance, consider the line

\[ \text{Dim a As Double: a = DoWonderfulStuff()} \]

for a Function, DoWonderfulStuff(), returning a Double. Changing the functionality of DoWonderfulStuff() changes the effect of line (6.1) without having to rewrite the line itself. You may keep several versions of DoWonderfulStuff(), toggling them in and out of a project depending on what it is you want the code to do. This is clumsy, but it leads on to the idea of polymorphism. Instead of a plain function call the client code calls methods from some object – but precisely which object the methods are being called from is toggled in and out somewhere else.

For instance, we mentioned in Chapter 5 three versions of a StopWatch object. They all had the same interface methods but differed slightly in the accuracy and speed of their operation. Table 6.1 gives a more concrete comparison.2 It lists for each StopWatch object its resolution (in seconds) and the time taken, relative to the plain StopWatch, to start and stop the stopwatch and to get the elapsed time.

The plain StopWatch is cheap and cheerful. Its resolution is poor and it is the slowest of the three stopwatches. Its sole advantage is that it uses only core VBA functionality and no Windows API functions.

---

1 This is not efficient when you are valuing a book. Later we return to discuss the choice between path-wise and slice-wise evolution.
2 Times are produced by the spreadsheet TimerTimer.xls.
The Getz and Gilbert stopwatch, StopWatchGandG, is one-third faster and its resolution is acceptable for most applications. The high resolution stopwatch StopWatchGBBA of Green et al. is no slower than StopWatch but its resolution is very sharp indeed.

There are circumstances where one might need the ability to switch between each type of stopwatch. Figure 6.1 shows how this can be done polymorphically. The details are explained later in the chapter.

---

**Table 6.1** Comparison of stopwatch objects. (Times in seconds)

<table>
<thead>
<tr>
<th>Stopwatch object</th>
<th>Resolution</th>
<th>Relative speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>StopWatch</td>
<td>1/18</td>
<td>1</td>
</tr>
<tr>
<td>StopWatchGandG</td>
<td>1/1000</td>
<td>0.62</td>
</tr>
<tr>
<td>StopWatchGBBA</td>
<td>1/199507</td>
<td>0.99</td>
</tr>
</tbody>
</table>

---

Figure 6.1 Using a polymorphic stopwatch
main() times the execution of the function compute_poly(). A stopwatch is declared (as type IStopWatch) and instantiated (in GetWatch()) on line 6.1b. It is switched on and off on lines 6.1c and 6.1d, and the time is read off on line 6.1e.

The key to this code lies on line 6.1b. IStopWatch is an interface type (details below). Each of the three different stopwatch objects is set up so that it can be declared to be of this type. The Function GetWatch() returns a reference to an IStopWatch object. The reference it returns could be to any one of the three stopwatch types: the client code in main() does not know which stopwatch it is dealing with.

GetWatch() is quite simple. Depending on the value of the character s_type, supplied by the user on line 6.1a, a Select statement toggles which stopwatch is instantiated and returned as the IStopWatch object.

main() is truly polymorphic. It has no idea which stopwatch object it is using, or indeed what the stopwatch is doing. Its functionality changes but it needs no rewriting. The crucial feature is that the three individual stopwatch objects have the same interface methods: StartTimer(), StopTimer(), ElapsedTime() and ResetTimer(). The mechanism which enables them to be declared as IStopWatch also ensures that they share the same interface methods. This means that client code does not have to worry about which stopwatch it is dealing with. It knows what methods it can use, even if it does not know exactly what each method does.

In this example the three stopwatches perform more or less identical functions; it is really only the internal implementation (and its characteristics) that is different. This is referred to as the strategy pattern: using polymorphism, as here, to toggle between different implementations of essentially the same or similar functionality. In the Monte Carlo application developed later in this chapter polymorphism is used not to do the same thing in different ways but to give the application radically extended functionality. This unleashes the real power of polymorphism.

Do not use Run()

Before moving on, the application Run() function needs to be mentioned, and dismissed. Run() takes as arguments a String containing the name of a macro and a list of arguments to pass to the macro. It returns the value of the macro with those arguments. For instance the line

\[
\text{Dim val As Double: val = Run("compute\_poly", a, b, c, x)}
\]

causes the function call compute\_poly(a, b, c, x) to be executed and its value returned.

The String passed to Run() is a variable, so this is a mechanism that can enable one to write polymorphic code; indeed there are examples of this use in several well-known books.

Please avoid using Run() in any remotely time-sensitive code. It may be convenient as a mechanism for the one-off execution of a piece of code you want to be able to toggle, but even for this purpose use an If-statement to select what to run.

Table 6.2 gives timings for executing a small Function (which computes the value of a quadratic polynomial) either directly, or indirectly by passing its name and its arguments to Run(). The results are
alarming. Calling Run() adds an overhead of around $2 \times 10^{-5}$ seconds per call. This is terrible; this cost cannot be carried in a speed-sensitive context. $10^7$ calls – quite plausible in a numerical application – adds 200 seconds to the run time. Avoid the extremely expensive Run(); please use proper polymorphism if only on cost grounds alone.

### 6.2 Interfaces in VBA

Interfaces are VBA’s method of implementing polymorphism. Objects conforming to the same interface can substitute for one another. In principle, using an interface is easy:

1. Define an interface.
2. Define object types conforming to the interface.
3. Declare data members whose type is that of (a reference to) the interface, and procedures taking the interface type as an argument.

A data member can be Set to refer to, and the procedure can be passed, any object conforming to the interface. This is the essence of polymorphism: a reference to an interface type can be replaced by a reference to any object conforming to that interface.

An interface is a class module, ordinarily without any implementations; that is, it contains member data declarations and procedure stubs only (but see Figure 6.4).

By convention interface class modules are given names prefixed by “I”. We also adopt the convention of prefixing the name of all objects conforming to an interface with the name of the interface, for instance ButterflyRedAdmiral conforms to the interface IButterfly. For reasons that will become apparent, the names of neither interface nor conforming objects should contain the underscore character, nor procedures within them.

An example of an interface class module is IPayoff (given in Figure 6.2) showing an interface for payoff objects. It has stubs for a Property Let SetValues(), taking an InputManager as its argument, and for a Payoff() Function taking and returning a Double. These are exactly the Public interface (with a small ‘i’) procedures defined for the Payoff object in Chapter 5. SetValues() is intended to be used to initialize data members of any conforming object, and Payoff() returns the payoff value.

An example of an object conforming to this interface is PayoffCall given in Figure 6.3. There are three things to notice about it. First, it has an Implements statement at the start of the class module, before any procedure definitions. Second, it has a pair of procedures, IPayoff_SetValues() and IPayoff_Payoff(), whose names are the same as those in IPayoff but prefixed by IPayoff_ and whose signatures are identical to the corresponding procedures in IPayoff. Third, every stub procedure in IPayoff has a corresponding procedure in PayoffCall. These are sufficient and necessary conditions for PayoffCall to implement the interface IPayoff.

![Figure 6.2](image-url)  
An interface for a set of payoff objects

---

3 An object may conform to more than one interface. We cross this nomenclature bridge when we come it.
The underscore character has a special use. In procedure names in conforming objects it connects the name of the interface type to the name of the stub being implemented. VBA gets confused if the stub name (or interface name) also contains the underscore character. For instance, had the SetValues() stub in IPayoff been named instead Set_Values(), then in PayoffCall the corresponding method would have to be named IPayoff_Set_Values(). Unfortunately a compile time error results in PayoffCall at the point where this name is introduced. For this reason the underscore character cannot be used in names of stubs in interface objects, nor in the name of the interface object itself.

Note that although the stub procedures in IPayoff are declared to be Public their implementations in PayoffCall are Friend. This is legal.\(^4\) Also, PayoffCall is not prevented from defining additional procedures, but it must implement everything in IPayoff, using the syntax illustrated in Figure 6.3.

In general, if an object Implements an interface ISomeThing, then it must implement every procedure in the interface, with the same signature; and if ISomeThing contains a data member, then the conforming object must define both a Property Get and a Property Let/Set for it.

Failure to conform to an Implements interface results in a compile time error, so all the stubs in the interface are guaranteed to be implemented in all conforming objects. An object supplying an interface, implemented by other objects, need not be anything special. It does not seem to be a hard and fast requirement that its procedures are just stubs. (See Figure 6.4.) We can now formally define a term we have just been using informally: an object conforms to an interface if it Implements it.

---

\(^4\) But note that run-time errors result if stubs in interface modules are declared as Friend.
As many objects as you like can be written to conform to the same interface. For instance, Figure 6.5 shows the PayoffPut object. The difference between this and PayoffCall is only in the definition of the IPayoff_Payoff() method; they conform to the same interface and can substitute for one another in situations where a reference to an object conforming to the IPayoff interface is expected.

An example of an object with an IPayoff data member is OptionEuro, displayed in Figure 6.6. It conforms also to an interface, IOption, but that does not concern us for the moment. Here we are interested in the data member pay_, declared to be of type IPayoff. OptionEuro has a Property Let, IOption_SetValues(), that does a number of things, including setting the value of pay_ using the SetPayoff() method of data. This latter method presumably returns a reference to an object conforming to IPayoff; for instance, it could be a reference to a PayoffPut object or to a PayoffCall object. pay_ is destroyed in the destructor as usual.

The pay_ data member is used in the Property Get method, IOption_OValue(). It calls the Payoff() method of pay_; because pay_ conforms to IPayoff the Payoff() method, with the anticipated signature, is guaranteed to exist.

Note that OptionEuro has no idea which particular IPayoff conforming object it is executing. This is polymorphism; the same piece of code can do many different things.

We shall often refer to an object being derived from an interface class. This terminology is convenient but misleading. The analogy with fully object-oriented languages goes only so far. In VBA all methods in the interface class must be implemented. In C++ a class derived from a base class inherits public and protected methods from it but only virtual methods must be implemented in the derived class (although other methods may be over-ridden).

### 6.3 IMPLEMENTING A POLYMORPHIC STOPWATCH

Figure 6.1 showed how a polymorphic stopwatch could be used in client code. Now we describe the changes in the stopwatch objects themselves and the IStopWatch interface. We focus on the polymorphic features rather than the details of the stopwatch functionality.

The interface object, IStopWatch, is shown in Figure 6.7. It has stubs for the four methods we need for a stopwatch interface. Figure 6.8 shows the implementation of the StopwatchGandG object. It now...

---

5 More detail about the option object itself is given in section 6.4.3.

6 The names of the stopwatch methods have been changed from those seen in Chapter 5 to remove the underscore character.
Implements IOption

Implements IStopWatch, forcing it to conform to the IStopWatch interface, enabling it to be executed polymorphically. It has four methods, IStopWatch_StartTimer(), IStopWatch_StopTimer(), IStopWatch_ElapsedTime() and IStopWatch_ResetTimer(), corresponding to the four stubs in IStopWatch.
StopWatchGandG is very similar to the plain StopWatch. The only difference is that it uses the Windows API function GetTickCount(),\(^7\) that counts the number of elapsed ticks (milliseconds), instead of Timer(), and in IStopWatch_ElapsedTime() the elapsed time has to be scaled by 0.001 to convert the number of millisecond ticks into seconds.

### 6.4 POLYMORPHISM AND THE MONTE CARLO APPLICATION

The first task is to identify which parts of the application may be able to have, or should have, their functionality substituted for, and hence for which parts it makes sense to have an interface to conform to.

There are several obvious candidates. We shall want to substitute one payoff for another. Currently the asset follows a geometric Brownian motion; we may wish to value an option with a different process for the underlying. We introduce an option object; the option may be vanilla European, or it could be an average rate option, or a barrier option, or an American option, or an option of some other type.

---

\(^7\) For an explanation of the syntax of the declaration of GetTickCount() see Green et al. (2007).
Table 6.3 Interfaces for the Monte Carlo application

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
<th>Conforming objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPayoff</td>
<td>Immediate payoffs</td>
<td>PayoffCall, PayoffPut, PayoffBond</td>
</tr>
<tr>
<td>IOption</td>
<td>Options</td>
<td>OptionEuro, OptionKO, OptionKI</td>
</tr>
<tr>
<td>IGenerator</td>
<td>Evolved processes</td>
<td>GeneratorGBM</td>
</tr>
<tr>
<td>IStoppingTime</td>
<td>Barrier conditions</td>
<td>STDoubleBarrier</td>
</tr>
<tr>
<td>IAccumulator</td>
<td>Accumulator objects</td>
<td>AccumulatorTerminal</td>
</tr>
<tr>
<td>IApp</td>
<td>Applications</td>
<td>AppMC</td>
</tr>
</tbody>
</table>

With this in mind we define the interfaces listed in Table 6.3. Missing from the table are categories such as error handlers, random number generators and stopwatches. These could also be made polymorphic and we may get around to them later.

We shall illustrate by defining only a few objects conforming to each interface; sometimes just one object. In fact it is good practice, where sensible, to define an interface for every object that might eventually need one, whether it currently seems necessary or not. This reflects a classic design principle: “program to an interface”. If you have an interface that your objects must conform to then your object design will be structured, is likely to be more stable, is extendible by adding in additional conforming objects, and at least one source of potential incompatibility with the rest of the application is removed.

Before going any further, to make the discussion more concrete, we present the application front-end. This is shown in Figure 6.9. It has fields not only for the numerical values of variables, but also for making choices from the various different conforming objects that can be specified. The payoff field, for instance,
Implementing Models of Financial Derivatives

could be one of $c/p/b$, for a call, a put, or a bond payoff, and the option field one of $o/e$, where $o$ is for a knock-out option and $e$ is for a vanilla European option.\footnote{Strictly, barrier options are also European. Hopefully my choice of notation will not cause confusion.}

Note that the field for the process type, for instance, may take only the value $g$. Later it would be possible to add in other process types; this is possible because within the code the only assumption made is that the process conforms to an interface (in particular the IGenerator interface).

The counter interval parameter is now read in from the front-end, but it is separated out from the other parameter settings.

The IPayoff interface was presented earlier in section 6.2. We go on to discuss several other aspects of the application in turn, starting with main(), then the IApp interface and its conforming object, AppMC, and then, at the other interface, classes and conforming objects.

6.4.1 The main() procedure and the AppObWrapper object

The new structure is shown in Figure 6.10. There is no change to main() from version 3.2.

In the new design the Monte Carlo numerics bit is a separate polymorphic object, AppMC of type IApp, contained as a composite object in AppObWrapper, Set in its constructor and destroyed in its destructor. All the Monte Carlo functionality has been taken away from AppObWrapper and put into the IApp object. When AppObWrapper is run the request is dispatched on to the Monte Carlo object to do its stuff.

AppObWrapper constructs objects for input/output and makes them available to the AppMC object. The AppMC object is decoupled from the details of I/O.

There are changes to the other objects to accommodate the new polymorphic structure, discussed below. The most substantial change is to the InputManager. This now takes on an aspect of a factory object in that it returns objects of an interface type depending on what the user has specified on the front-end.

The new AppObWrapper is shown in Figure 6.11. It is now pretty much a shell as far as Monte Carlo is concerned. Its responsibility is nevertheless both well-defined and essential: to decouple the numerics

![Figure 6.10](image-url)
Polymorphism and Interfaces: Level 4

from input and output. The InputManager and OutputManager are not yet polymorphic (they will be) but in principle the AppObWrapper could be passing the numerics arbitrary input and output streams, as long as they satisfy the contract that the numerics part expects. Naturally when the I/O goes polymorphic the interfaces define exactly the necessary contract.

An important aspect of the design is the way that object creation takes place. AppObWrapper instantiates an IApp object and then calls its SetValues() procedure. This initiates a cascade of object creation.

There is a new object: a Monitor object. Its responsibility is to send monitoring output to the client. Here this just means that it outputs the step counter instead of the OutputManager, but we now have the potential to output other types of monitoring information to the client as well. Separating out the counter from the real application output is a useful piece of decoupling from the perspective of the OutputManager. It does not have to worry about where the weird monitoring stuff has to go.

The AppObWrapper owns the Monitor object. It passes a reference to it to the IApp object in its SetValues() method for it to use as it pleases.

We shall return to the application object in section 6.4.2, but first we shall look at the new InputManager and OutputManager, and the new Monitor object.
The **InputManager**

The responsibility of the **InputManager** is to supply other objects in the application with data, with some sanity-check validation and, with the advent of polymorphism, to manufacture and initialize objects.

The structure of **InputManager** adds to the previous version in Chapter 5. As before there are **Private** data members for each numeric field, with a **Property** **Get** for each, and **InputManager**’s constructor reads in the numeric data.

The previous version had only to read in numerical values. Now this version has to read in values that determine what objects are to be created, and then to create them. In Chapter 8 we take the latter responsibility away from the **InputManager** and place it into a purpose-built object factory. 9

There are additional **Functions** that return references to newly created objects, one for each interface class. They are **SetPayoff()**, **SetEvent()**, **SetOption()**, **SetProcess()** and **SetAccumulator()**, returning an **IPayoff**, **ISтопpingTime**, **IOption**, **IGenerator** and an **IAccumulator** object respectively. We show only one of these **Functions**, **SetOption()**, in Figure 6.12; the others are identical in structure.

**SetOption()** reads in a character from the front-end, calling the validator **get_char()**, which throws if the input cell (at location (26, 6) on the front-end) does not contain one of the allowed values given in its **String** argument. The **Select** statement assigns to **SetOption** a reference to an object of the selected type. Before the reference is returned, the object’s **Property Let**, **SetValues()**, is called, assigning **Me** to it.

This is an example of using **Me** to pass a reference to a calling object through to an object whose procedure is being called. An example of a typical **SetValues()** procedure, **OptionEuro::SetValues()**, was given in Figure 6.6. The **IOption** object, in this case an **OptionEuro**, requests the **InputManager** to give it a reference to an **IPayoff** object and also sets the value of a parameter it needs.

The **InputManager** is also used in Figure 6.11. The **AppObWrapper** constructor assigns the **InputManager** to the application objects’s **SetValues()** method, just like Figure 6.12.

The **OutputManager**

This requests output values from the application object and the stopwatch and outputs them. It is similar to the previous version but now it has a single **Property Let**, **SetValues()**, instead of the previous three. **SetValues()** takes an **IApp** as an argument, calling the **IApps** **OValue()** and **SEValue()** **Property** **Gets** to set the values of the option and its standard error (Figure 6.13). Output happens in the destructor as before. It no longer has to deal with the step counter.

---

9 Although not a proper design pattern-type factory. We achieve this in Chapter 12.
Polymorphism and Interfaces: Level 4

Figure 6.13  OutputManager::SetValues()

Figure 6.14  The Monitor object

The Monitor object

Its sole responsibility is to take over the counter output from the OutputManager. Its implementation (Figure 6.14) is as expected.

6.4.2 The IApp interface and AppMC

Input and output are now no longer the responsibility of the numerics object; this is reflected in the data members of the new AppMC object. It brings together only what is needed to do the numerics in a Monte Carlo method: an option, a generator and some structure to relate the two and get some numbers out.

The Monte Carlo application implements the IApp interface, shown in Figure 6.15. It is minimal. It assumes that the application needs an initializer Property Let, SetValues(), and two Property Gets, OValue() and SEValue(), called by the OutputManager. The only other procedure is the run() method called by the AppObWrapper.
The conforming object, AppMC, is shown in Figure 6.16. The run() method is the central Monte Carlo procedure. It loops around for each sample path, and within this, for each time step. This is a change from the level 3 applications, reverting to the way the level 2 and indeed the level 1 applications were structured. In fact it turns out that this element-wise scheme runs very slowly, chiefly because it incurs additional function calls on every step. We discuss evolution schemes further in Chapter 14.
Three objects are required by the Monte Carlo method; an IOption object to compute payoffs, an IGenerator object to evolve sample paths, and an IAccumulator object to accumulate results. At the start of each sample path the generator and the option are re-initialized. Then on each time step the generator returns the next value along the sample path and passes it over to the option object (line 6.16a). On 6.16b the option is asked whether it knows its payoff. If it does, control passes to 6.16c where the option value is given to the accumulator object; in any case, at the conclusion of each sample path it is presumed that the option knows its value.

The OptionEuro object never knows its payoff until the final maturity time, but the knock-out option, OptionKO, knows in advance whether it has been knocked out, and hence whether it already knows its payoff.

The data members needed by the Monte Carlo application object are the number of sample paths, M_, the number of time steps, N_ and the three composited objects, the IOption object opt_, the IGenerator object gen_ and the IAccumulator object acc_. These are initialized in SetValues() where the relevant methods of the InputManager are called. Following the standard idiom, the object references are Set to Nothing in the destructor.

The two Property Get_s, OValue() and SEValue(), simply dispatch the request for values on to the accumulator object.

6.4.3 The objects used by the application

We consider in turn each of the three composited objects used by the latest version of the Monte Carlo application.

The IOption interface

What role does an option play in an application? What functionality should it have? Here we endow an option object with five procedures: a Property Get, OValue(), to return the option’s cashflow once the value is known;\(^\text{10}\) a Property Get, Finished(), whose return value is True if the option has enough information to establish its payoff along the sample path, otherwise False; a Property Let, RegisterValue(), that gives the option the next asset value along a sample path; a Property Let, SetValues(), that initializes the option; and a Sub, Initialise(), that re-initializes the option at the beginning of each sample path.

These are all very reasonable requests to make of an option, but there is a quasi-philosophical point to make. The application is constrained to be able to value only options that are capable of conforming to the interface. Here the list of interface procedures (and their signatures) does not just describe an option, rather, it defines what an option is within the application. By definition, only those objects conforming to the IOption interface are options.

Figure 6.17 gives the IOption interface. It is very much as you would expect from the previous discussion, and from the example in section 6.2. There we gave the definition of OptionEuro; here we give the definition of two other conforming objects, OptionKO and OptionKI.

Knock-out barrier options

A double barrier knock-out option expires worthless if the asset value \(S_t\) ever exceeds an upper barrier level \(U\), or is less than a lower barrier level \(D\). If it is not knocked out it receives a payoff at maturity time \(T\).

\(^{10}\) This assumes that there is only a single cashflow. More generality is needed and is possible.
Public Sub Initialise(): End Sub
Public Property Let SetValues(ByRef data As InputManager): End Property
Public Property Get OValue() As Double: End Property
Public Property Let RegisterValue(s As Double): End Property
Public Property Get Finished() As Boolean: End Property

Figure 6.17 The IOption interface

Implements IOption

Private pay_ As IPayoff
Private evt_ As IStoppingTime
Private OValue_ As Double
Private Finished_ As Boolean
Private last_s_ As Double
Private S0_ As Double

Private Sub Class_Terminate()
    Set pay_ = Nothing
    Set evt_ = Nothing
End Sub

Friend Property Get IOption_OValue() As Double
    If Finished_ Then
        IOption_OValue = OValue_
    Else
        IOption_OValue = pay_.Payoff(last_s_)
    End If
End Property

Friend Property Let IOption_RegisterValue(s As Double)
    last_s_ = s
    Finished_ = evt_.TestEvent(s)
    If Finished_ Then OValue_ = 0#
End Property

Friend Property Get IOption_Finished() As Boolean
    IOption_Finished = Finished_
End Property

Friend Property Let IOption_SetValues(ByRef data As InputManager)
    Set pay_ = data.SetPayoff
    Set evt_ = data.SetEvent
    S0_ = data.S_0
    Call reset
End Property

Friend Sub IOption_Initialise()
    Call reset
End Sub

Private Sub reset()
    last_s_ = S0_
    Finished_ = False
    OValue_ = 0#
End Sub

Figure 6.18 The knock-out option object: OptionKo
Our reification, OptionKO (Figure 6.18) has two composited objects: its payoff pay_ of type IPayoff, previously discussed, and a stopping time object of type IStoppingTime.

The stopping time object evt_, here of type STDDoubleBarrier, is present to test whether the current asset value has breached the barriers. The payload function is the TestEvent() procedure that returns True if the event has occurred, that is, if the option has knocked out. The definition of STDDoubleBarrier is given in Figure 6.19 (we skip the interface; it is clear from the definition what this is).

pay_ and evt_ are Set in IOption_SetValues() and destroyed in the destructor. At each step the asset value is passed to the stopping time object. If it ever returns True then the knock-out barrier option returns a zero value.

Knock-in barrier options
A knock-in barrier option is valued very similarly to a knock-out option. A knock-in option can be in one of two states: either it has been knocked in or it has not yet been knocked in. The state information is stored within the object as a Boolean. As a sample path is evolved forwards, the object tests whether it has been knocked in and updates its state accordingly.

The object OptionKI represents a knock-in option. Its definition is shown in Figure 6.20.

It defines a Boolean KnockedIn_ to record the option’s state. It is True only if the stopping time object, evt_, ever returns True. reset() initializes its value by testing the initial value of S. Finished_ is always false since the knock-in option’s payoff is not known until time T.

At each step the RegisterValue() method is called. This tests to see if the option has now knocked-in. At the final time, when OValue() is called, a payoff is computed only if the option has knocked-in; otherwise the payoff is zero.

The interface methods
IOption_RegisterValue(), IOption_Finished(), and IOption_OValue() provide the application’s functionality.

When the generator computes the next value of the underlying it is passed to the option object via the RegisterValue() Property. What happens next is up to the particular option object. Both OptionKO and
Implementing Models of Financial Derivatives

OptionKI store the value (in last_s_) and ask the StoppingTime whether the barriers have been breached. If they have, then the payoff is either switched on or switched off, with KnockedIn_ and Finished_ being set appropriately. The Property Get IOption_OValue(), returns the conditional OValue_ once this value can be computed, at the knock-out time in the case of OptionKO, or at the final maturity time for OptionKI.

There is one further procedure, IOption_Initialise(), called at the start of each sample path, that resets all the data members to their starting values.

The IGenerator interface

There is only a single derived object, GeneratorGBM, shown in Figure 6.21. The functionality of GeneratorGBM is straightforward: to generate successive values of $S_t$. Comparing its definition with that of PathGenerator in Figure 5.12, we see that it is, if anything, simpler. It now generates one value of $S_t$ at a time, rather than an entire path. The previously generated value of $S_t$, last_s_, is stored so that it can be incremented on the next step, and there is an initializer Sub that resets the value of last_s_ at the beginning of every sample path.

The IAccumulator interface

This is also straightforward. The interface IAccumulator is given Figure 6.22 and its single derived object AccumulatorTerminal in Figure 6.23. There is no substantive difference between the implementation for AccumulatorTerminal given here and that of ResultAccumulator in Figure 5.14 of Chapter 5. The only difference is that here an interface has been provided, so if at a later date it becomes necessary to substitute a different accumulator object (for instance, one that for some reason printed out intermediate results to file) it would be easy to do so without changing any application code.

---

11 We omit the interface object on the grounds that its definition is clear from that of the conforming object.
Implements IGenerator

Private N_ As Long
Private S_0_ As Double
Private last_s_ As Double
Private drift_ As Double
Private s_root_t_ As Double
Private norm_gen_ As NormalGenerator

Private Sub Class_Initialize()
    Set norm_gen_ = New NormalGenerator
End Sub

Private Sub Class_Terminate()
    Set norm_gen_ = Nothing
End Sub

Friend Property Get IGenerator_NextS() As Double
    last_s_ = next_S(last_s_)
    IGenerator_NextS = last_s_
End Property

Friend Property Let IGenerator_SetValues(ByRef data As InputManager)
    N_ = data.N
    S_0_ = data.S_0
    last_s_ = S_0_
    Dim dt As Double: dt = data.T / data.N
    drift_ = (data.rr - 0.5 * data.sigma * data.sigma) * dt
    s_root_t_ = data.sigma * Sqr(dt)
End Property

Friend Sub IGenerator_Initialise()
    last_s_ = S_0_
End Sub

Private Function next_S(s As Double) As Double
    next_S = s * Exp(drift_ + s_root_t_ * norm_gen_.GetNormal)
End Function

Figure 6.21 The GeneratorGBM object

Figure 6.22 The IAccumulator interface
Just like ResultAccumulator, AccumulatorTerminal is set in a world where interest rates are deterministic. The discount factor (there is only one) can be precomputed and used to discount the option value and the standard error at a final step.

### 6.5 ASSESSMENT OF THE POLYMORPHIC DESIGN

How easy is it to understand the application? Is it any good? Sensible questions. Looking at the application described above it might seem to be far more complicated than the level 1 application in Chapter 3. In some ways (well, actually in quite a few ways) that is true. However, the purpose of the design is to remove from the programmer of the AppMC application object any need to worry about error trapping, and input and output, and to give him or her as much flexibility as possible to adapt the application to future needs.

If further input variables are needed then the InputManager will have to change to reflect this, but in languages permitting registration and call-back even this can be minimized. The idea is that the

---

12 Objects would register their output with the OutputManager, passing it a reference to a call-back procedure that supplied it with the needed parameter(s). All the work is done in the application object.
programmer is free to program the application object as he or she likes, as long as it conforms to the \texttt{IApp} interface.

\textbf{Running the application}

Referring to Figure 6.10, there is now a four-level-deep set of procedures before the application object runs. When the button on the front-end is clicked the following calls are made

\begin{verbatim}
Sub Front_End.CommandButton1_Click()
  ↓
Sub MC_example_v4.main()
  ↓
Sub AppObWrapper.run()
  ↓
Sub AppMC.run()
\end{verbatim}

This is nothing unusual. In serious industrial applications the stack can be many layers deep. In our case another layer will be added in Chapter 8.

As each layer is entered, the relevant constructor is called and the object initialized as required. The application is then run. Finally, as each layer goes out of scope, the destructors are called. These \texttt{Set to Nothing} the composited objects in each layer, and perform other functionality as required.

The programmer only needs to worry about the final function call. The other layers sitting above it perform essential house-keeping, but these are things that the programmer of the end-application should not need to worry about.

\textbf{Is it any good?}

The current design is a big improvement over the level 3 versions. It is now polymorphic and reasonably general. It is easy to extend its functionality by adding additional conforming objects. It is possible to value nested options with complex payoff patterns, and one can add processes with, for instance, stochastic volatility, by substituting for the generator object.

There are still some things wrong with it – in fact, two main things. Firstly, the architecture of the application (and it should now be clear why applications of this sort are referred to as having an architecture, and why engineers work on them) handles output inefficiently. In a Monte Carlo application it is useful and desirable to print out a counter showing how far the application has progressed. Right now it is clunky to try to add this in.

This is where VBA events might help out. We illustrate using these for output in Chapter 8, but abandon them for this purpose immediately afterwards.

The second issue concerns the Monte Carlo bit itself. By now this may seem to be a small part of the application, but it is the essential part. One would like to change the Monte Carlo application algorithm to run a decent, fast, Monte Carlo instead of the na\"ive version we have been toying with until now; it should be possible to incorporate control variates or stratified sampling, and one might like to run a least squares Monte Carlo for Bermudan options.
Table 6.4  Double barrier knock-out values: convergence in $N$, times $t$ in seconds

<table>
<thead>
<tr>
<th>$N$:</th>
<th>50</th>
<th>100</th>
<th>200</th>
<th>400</th>
<th>800</th>
<th>1600</th>
<th>3200</th>
<th>6400</th>
<th>12 800</th>
<th>25 600</th>
<th>51 200</th>
<th>102 400</th>
</tr>
</thead>
<tbody>
<tr>
<td>$se$:</td>
<td>0.016</td>
<td>0.016</td>
<td>0.015</td>
<td>0.015</td>
<td>0.014</td>
<td>0.014</td>
<td>0.014</td>
<td>0.014</td>
<td>0.014</td>
<td>0.014</td>
<td>0.014</td>
<td>0.014</td>
</tr>
<tr>
<td>$t$:</td>
<td>5</td>
<td>9</td>
<td>18</td>
<td>36</td>
<td>71</td>
<td>142</td>
<td>282</td>
<td>562</td>
<td>1125</td>
<td>2242</td>
<td>4503</td>
<td>9258</td>
</tr>
</tbody>
</table>

It is not only the speed of the Monte Carlo application that is an issue. The option objects presented here have incorporated knock-in and knock-out features, and we are able, merrily, to value them. Unfortunately a standard Monte Carlo method examines the barrier condition only at the times at which it computes asset values. A consequence is that knock-in options are heavily under-valued and knock-out options heavily over-valued relative to the continuously monitored case.

Consider valuing a double knock-out call option\(^{13}\) with a continuously monitored barrier and $X = 100$, $T = 1$, $D = 80$, $U = 120$ on an asset with $S_0 = 100$, $r = 0.05$, and $\sigma = 0.2$. There is an explicit solution available in this case\(^{14}\) giving a value of around 1.1147. Monte Carlo values produced by MC_example_v4.xls for values of $N$ increasing from 50 to 102 400 are given in Table 6.4. Times, $t$, are in seconds.

These results are appalling. Only with $N = 12 800$, and a computation time of 1125 seconds, is the result within one standard error of the explicit solution, and even for this number of time steps it is not clear that the result is still not significantly high biased.

The table demonstrates that a plain Monte Carlo method is not an acceptable valuation method for a continuously monitored barrier option. A bias correction method, discussed in Part VI, is needed.

6.6 SUMMARY

Polymorphism in VBA is achieved by using interfaces. These are clean and elegant, although their implementation in VBA has some slight problems (as we shall see).

We have been able to establish run-time polymorphism, but object creation is still tied to Select statements. It is not until Part IV that we put this right.

In fact the current structure, resulting in \texttt{AppMC::run()}, is quite slow. In the next chapter we devise an alternative data structure that begins to represent the data more effectively and reduces the execution time considerably.

We revisit the architecture in later chapters to accommodate the requirements of an improved, faster, Monte Carlo to be explored in Part VI.

6.7 EXERCISES

Now it is time for the exercise stream applications to go polymorphic. This makes them enormously more powerful but mistakes made at this stage in coming up with a suitable set of interfaces can be costly to rectify later.

1. The CEV process for $S_t$ is

$$dS_t = r S_t \, dt + \sigma S_t^{\gamma} \, dz_t$$  \hfill (6.6)
Polymorphism and Interfaces: Level 4

Table 6.5  CEV process: sets of values for $\sigma$ and $\gamma$

<table>
<thead>
<tr>
<th>Set</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\gamma$</td>
<td>0.5</td>
<td>0.75</td>
<td>1.0</td>
<td>1.25</td>
<td>1.5</td>
</tr>
<tr>
<td>$\sigma$</td>
<td>2</td>
<td>0.63</td>
<td>0.2</td>
<td>0.063</td>
<td>0.02</td>
</tr>
</tbody>
</table>

for a parameter $\gamma \geq 0$ (so that $\gamma = 1$ is a geometric Brownian motion). In general, equation (6.6) does not have an exact solution, so when simulating a CEV process an approximate discretization scheme must usually be used.

The Euler and Milstein discretization schemes were given in equation (3.19), page 37. Create a CEV generator object as part of an IGenerator hierarchy with methods to enable its conforming objects to be simulated by either the Euler or Milstein schemes, as well using an exact solution (if available).15

Value a set of European call options with time to maturity $T = 1$ and strikes $X = 90, 95, 100, 105$ and $110$, on an asset following a CEV process with $S_0 = 100, r = 0.05$, for four sets of values for $\sigma$ and $\gamma$, given in Table 6.5. The values are fixed so that when $S \sim 100$ the total volatility $\sigma S^\gamma$ is approximately the same level for each pair. Take care to ensure that the number of time steps is sufficiently large so that any discretization bias is masked by Monte Carlo noise.

Compute implied volatilities for these options, and for each $(\gamma, \sigma)$ pair plot the implied volatility smile.

2. Under risk-neutrality suppose an asset $S_t$ has the process

$$dS_t = \mu(S_t)\,dt + \sigma(S_t)\,dz_t$$

(6.7)

for deterministic functions $\mu(S)$ and $\sigma(S)$ of $S$. Let $\mu', \mu'', \sigma'$, and $\sigma''$ denote derivatives with respect to $S$. For a time step $\Delta t$ let $\Delta w$ and $\Delta y$ be independent draws from the normal distribution $N(0, \Delta t)$ with mean 0 and variance $\Delta t$. The Itô–Taylor 1.5 strong approximation16 is given by

$$\Delta S_t = S_{t+\Delta t} - S_t = \mu \Delta t + \sigma \Delta w + \frac{1}{2} \sigma' (\Delta w^2 - \Delta t)$$

$$+ \left( \mu \sigma' + \frac{1}{2} \sigma'^2 \sigma'' \right) \left( \frac{1}{2} \Delta w \Delta t - \frac{1}{2 \sqrt{3}} \Delta y \right)$$

$$+ \frac{1}{2} \sigma \mu' \Delta t \left( \Delta w + \frac{1}{\sqrt{3}} \Delta y \right) + \frac{1}{2} \left( \mu \mu' + \frac{1}{2} \sigma^2 \mu'' \right) \Delta t^2$$

$$+ \frac{1}{2} \left( \sigma (\sigma')^2 + \sigma'^2 \sigma'' \right) \left( \frac{1}{3} \Delta w^2 - \Delta t \right) \Delta w.$$  

(6.8)

Add an Itô–Taylor method to the IGenerator interface constructed in exercise 1. For small values of $N$ assess the degree of bias of this discretization compared to the Euler and Milstein schemes when valuing the European calls with the CEV process from exercise 1.

3. Pi stream. It is clear that the $\pi$ objects in the level 3 pi application, pi_app_v3.xls, should belong to a polymorphic hierarchy. Implement this as an interface IPi in a level 4 application pi_app_v4.xls.

---

15 If an object is asked for its exact solution when one is not known, the object may raise an error.

16 See Kloeden and Platen (1995) and Chapter 25.
4. **Implied volatility stream.** Up to this point the implied volatility application has required a deep coupling between the root-finding algorithm and the function whose root is to be found. Polymorphism offers a mechanism to remove this coupling.

(a) Modify the bisection procedure to enable it to take a (polymorphic) object as an argument. Modify the Black–Scholes formula to enable it to conform to this interface, so that the root-finding algorithm can run with the Black–Scholes formula without knowing it is Black–Scholes. Call the new application IV_app_v4.xls.

(b) Make a barrier option valuation function (Chapter 1, exercise 1(b)) a member of the same hierarchy so that it too can be passed to the root-finding algorithm. On the front-end let the user determine which valuation function is having its implied volatility found.

(c) Turn the bisection procedure into an object from a polymorphic hierarchy so that something along the lines of the following code runs:

```vba
Dim rootfinder As IRootFinder: Set rootfinder = New BisectionRootFinder
Dim fun As IRootConforming: Set fun = New BlackScholes
Call rootfinder.SetRange(lb, ub)
Dim root As Double: root = rootfinder.run(fun) '.
```

Compare the execution times for the polymorphic root finder compared to the coupled non-polymorphic root finder.

You may find it useful to use a utility function like `CastRootConforming()`,

```vba
Public Function CastRootConforming(obj As Object) As IRootConforming
    Set CastRootConforming = obj
End Function
```

in your implementation. (See the discussion of casts in Chapter 12.)

(d) Implement the method of false position and Ridders’ root-finding algorithms (see Appendix H) as part of the `IRootFinder` hierarchy. Compare their performance, with Black–Scholes, against that of the bisection algorithm.

5. **PDE stream.** The Crank-Nicolson method you constructed as CN_pde_v3.xls has a plain evolver that, for American options, incorporates the early exercise decision by a direct comparison between continuation values and exercise values. Appendix G describes the SOR method. This replaces the direct comparison by an integrated procedure that is accurate to second order in time.

Introduce an interface `IEvolver`, and other interfaces as necessary, to make the application polymorphic. Name the new version CN_pde_v4.xls. Implement both the SOR method and the current plain evolver to conform to the `IEvolver` interface. For the SOR method replicate the convergence analysis of Appendix G to establish empirically its convergence rate.

6. **Lattice stream.** Appendix F describes both a plain trinomial lattice and a “faster” lattice that uses pruning to reduce significantly the amount of computation required. Other, faster, higher order lattice methods also exist.
### Table 6.6  Branching specification for the Alford and Webber lattice

<table>
<thead>
<tr>
<th>$j$</th>
<th>$0$</th>
<th>$1$ and $-1$</th>
<th>$2$ and $-2$</th>
<th>$3$ and $-3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p^j$</td>
<td>$0.4766698866$</td>
<td>$0.2339147378$</td>
<td>$0.0269381893$</td>
<td>$0.0008121295$</td>
</tr>
</tbody>
</table>

(a) By making the slice object from the application Lattice_application_v3.xls polymorphic, allow the lattice application to switch transparently between pruned and unpruned evolution. Name the new application Lattice_application_v4.xls.

(b) Suppose that under the risk-neutral measure $S$ obeys a geometric Brownian motion with drift $r$ and volatility $\sigma$. In the Alford and Webber heptanomial lattice method a value $\hat{S}$ for the discrete lattice process at time $t_l$ branches to seven values $\hat{S}^j$, $j = -3, \ldots, 3$, at time $t_{l+1}$,

$$
\hat{S}^j = \hat{S} \exp \left( \left( r - \frac{1}{2} \sigma^2 \right) \Delta t + \sigma z^j \right), \quad j = -3, \ldots, 3, \quad (6.11)
$$

where $z^j = j \sqrt{\kappa \Delta t}$ for $\kappa = 1.432760571$. The branching probabilities are specified in Table 6.6. The lattice recombines. For details of the derivation see Alford and Webber (2001). Create a polymorphic hierarchy for the lattice evolver, that both the trinomial and the heptanomial evolvers can conform to, consistent with the slice interface devised for exercise 6(a).

Compare the rate of convergence of the pruned and unpruned heptanomial lattice to that of the trinomial lattice.

7. In the Monte Carlo application MC_example_v4.xls, suppose that for some reason you wanted to output a running update on the current option value and standard error as the Monte Carlo simulation progressed. How could you modify the application to do this?
Now that we have a polymorphic option object with a well-defined interface we can revise the implementation of the Monte Carlo method to take advantage of it. This chapter focuses on the Monte Carlo application itself, rather than the supporting super-structure. As usual we elaborate on the previous version. Functionality is separated out and handed over to interchangeable specialist objects.

The new Monte Carlo method is, in the language of Chapter 1, slice-wise, that is, a vector of states is evolved along $N$ time steps. At each step a generator object constructs a slice of values and passes it over to an option object. The option object works out its payoff and hands it out when requested to do so.

This version is significantly faster that the level 4 application of Chapter 6, mainly because of the change in the way that data is represented within the application. The important point is not just the increase in speed; moving away from element-wise to slice-wise evolution is also a major conceptual improvement.

The version described here accepts slices in chronological order, from time $t_1$ up to time $t_N = T$. Later, when we discuss bridge methods, the slices will be constructed in non-chronological order.

This leads on to a working definition of degree of path dependency from a computation perspective. What is the amount of state information required to be stored by an option object in order to price itself irrespective of the order in which slices arrive, at any time leading up to its maturity? An average rate option requires one piece of state information (its running average)\(^1\) whereas a barrier option requires up to two state-dependent items, its payoff at time $T$ and a Boolean to indicate whether it has knocked-in or knocked-out.

This is an alternative notion of path-dependency to that of an effective dimension where the type of the path-dependent statistic and the number of time-dependent values that contribute towards it is important; clearly an average rate option is ‘more’ path-dependent in this sense than a barrier option.\(^2\)

First we look at the new AppMC object, then at the new option hierarchy and finally at the object responsible for evolving slices from one step to the next.

### 7.1 THE REVISED MONTE CARLO APPLICATION OBJECT

The level 4 application object, AppMC in Figure 6.16 (page 94), looped through each sample path in turn and for each sample path successive values of the state variable were generated in turn. This is a standard element-wise evolution. The new AppMC, Figure 7.1, has only a time step loop.\(^3\) An evolver object, evl_, generates a slice of values for the underlying state variable with its EvolveNextSlice() method. The slice is returned ByRef, as is the time the slice is for, slice_time_. The counter i is passed to the evolver for it to use if it wants to.

It is legal in VBA for functions to have arrays as return types, but passing and returning arrays as ByRef arguments is usually cheaper.

The evolver is entirely responsible for deciding the order in which slices are constructed. For the moment the slices are constructed in chronological order, but it turns out to be highly advantageous in a bridge method to be able to relax this.

---

\(^1\) It also needs to know the span over which the running average has been computed, but this is not state dependent.

\(^2\) This can be made more formal but we do not pursue this further. See Caflisch, et al. (1997).

\(^3\) This is MC_example_v4b.xls.
'Implementing Models of Financial Derivatives

Implements IApp

' data section

Private N_ As Long 'Number of time steps
Private opt_ As IOption 'The option
Private evl_ As IEvolver 'The slice evolver
Private OValue_ As Double 'The option value (eventually)
Private SEValue_ As Double 'Its standard error
Private slice_() As Double 'The slice at the given time
Private slice_time_ As Double 'The time the slice is generated for
Private mon_ As Monitor

' structural

Private Sub Class_Terminate()
    Set opt_ = Nothing
    Set evl_ = Nothing
    Set mon_ = Nothing
End Sub

Friend Property Get IApp_OValue() As Double
    IApp_OValue = OValue_
End Property

Friend Property Get IApp_SEValue() As Double
    IApp_SEValue = SEValue_
End Property

Friend Property Let IApp_SetValues(ByRef data As InputManager, ByRef mon As Monitor)
    N_ = data.N 'Number of time steps
    Set mon_ = mon 'Set the monitor
    Set evl_ = data.SetEvolver
    Set opt_ = data.SetOption
    ReDim slice_(1 To data.M) As Double
End Property

' other interface

Friend Sub IApp_run()
    Call reset
    Dim i As Long
    For i = 1 To N_ 'For each time step
        Call mon_.OutputCounter(i)
        Call evl_.EvolveNextSlice(slice_, i, slice_time_) 'a
        Call opt_.ReceiveNextSlice(slice_, i, slice_time_) 'b
    Next i
    Call register
End Sub

Friend Sub IApp_reset()
    Call evl_.Initialise
    Call opt_.Initialise
End Sub

Private Sub register()
    OValue_ = opt_.OValue
    SEValue_ = opt_.SEValue
End Sub

Figure 7.1 The AppMC object
Once constructed, the slice and its construction time, slice_time_, (and the counter i), are passed over to the option object opt_ by calling its ReceiveNextSlice() method. The option does its thing with the slice, using it to contribute towards its final payoff, perhaps storing it if necessary for later use.

Once each of the N_ slices (one for every time step in some order) have been fed to the option object, it is asked for its value and its standard error. Just like the implementation in Chapter 6, the AppObWrapper, unchanged from then, passes at the end of AppObWrapper::run() the AppMC object on to the output object so that the option value and its standard error can be extracted.

The rest of the AppMC object is straightforward. AppMC::SetValues() instantiates evl_ and opt_. They are Set to Nothing in its destructor. The only other methods are the Private Subs reset() and register(). At the start of a run reset() calls the Initialise() methods of the evolver and option objects, just in case they need to do anything (and because it looks nicely balanced with the register() method which asks the option object for its value and standard error).

### 7.2 THE OPTION OBJECT

The interface to which options now conform is changed only a little from level 4 but it is a crucial change. Instead of receiving individual values from the evolver one at a time it now receives them as a slice. Other changes are the presence of an Initialise() method and an SEValue() getter. The new IOption interface is shown in Figure 7.2.

An example of a conforming object is OptionAverage, shown in Figures 7.3 and 7.4, that reifies an average rate option (this version includes the current value of the underlying in the calculation of the average).

The payoff to an average rate option is a function of the average asset value over the life of the option. Suppose the option is European style, created at time t and maturing at time T > t. In continuous time the average asset value is \( A_T \),

\[
A_T = \frac{1}{T-t} \int_t^T S_s \, ds. \tag{7.1}
\]

The payoff to a fixed strike continuously averaged average rate call option is \( H_T = (A_T - X)^+ \) at time T.

In discrete time the asset value is observed at times \( T = \{ t_i \}_{i=0,...,N} \) with \( t_0 = t, \ t_N = T, \) and \( t_i < t_{i+1} \) for all \( i \). The discrete time average is computed as

\[
\hat{A}_T = \frac{1}{|T|} \sum_{t \in T} S_t \tag{7.2}
\]

\[
= \frac{1}{N+1} \sum_{i=0}^{N} S_{t_i}. \tag{7.3}
\]

---

**Figure 7.2** The IOption interface
In this case the average includes the initial value $S_{t_0}$ but in general it need not. The dates $T$ are referred to as reset dates.

Our average rate option object assumes that the time steps in the Monte Carlo method correspond to reset dates. As the number of time steps $N$ increases, the Monte Carlo option value converges to the value of the continuously averaged average rate option.
Figure 7.4 The OptionAverage object: ReceiveNextSlice() and Compute_Values()

Figure 7.3 shows the routine parts of the OptionAverage object, largely common to all options. There is a payoff object, identical to those we have seen before, and various other standard data members.

New is a Boolean, done_, that says whether the option value and standard error have yet been computed. If they have not yet been computed when the getters for these values are called, the Private Sub Compute_Values() is asked to compute them (Figure 7.4).

There is an array AccValues_ that is updated whenever the option is passed a new slice in the ReceiveNextSlice() method. The option stores accumulated values in AccValues_, not the average to date. Note that an average rate option does not require slices to be received in any particular order.

The main average rate option specific functionality is in the ReceiveNextSlice() method, that just increments the AccValues_ array, and the Compute_Values() Sub. All that Compute_Values() does is to compute the option payoff for each sample path and work out the option value and standard error, and set done_.

Note that Compute_Values() traps errors it might encounter when working out the standard error, in particular, should radix be negative. This check is not normally required, but if deterministic low discrepancy sequences are used radix might be just ever so slightly negative.

Note also that most of the code in Compute_Values() is common to every option conforming to IOption and is not specific to OptionAverage. This indicates a design flaw. We discuss and put it right in the level 5 iteration of the application.

Only one discount factor is computed; the cashflow to the option is received at only one time, its maturity. In general there is no reason why the option should not be passed an array of interest rates or

---

4 Throwing with the utility Sub, RaiseError() (lines (4.2), page 42).
discount factors, computed either deterministically, or as a sample path generated by a separate interest rate simulator.

OptionEuro is very similar in construction to OptionAverage; the fact that this so underlines naturalness of the design. Instead of having an array AccValues_, it has instead an array OValues_ of values at time $T$. When the option is passed a slice for time $T$ it calls a Sub Compute_O_Values() that computes the payoffs at the final time and puts them into OValues. All other slices are ignored. This is not as expensive as it might seem. As far as the option goes it just means that its ReceiveNextSlice() method is called $N$ times rather than just once. The real waste, if there is any, comes from the work done by the evolver object to generate slices. Even this is not too bad. If the process requires short-step evolution then the slices have to be generated anyway, and if long-step evolution is possible then it is up to the client to specify how many, or how few, slices are generated.

OptionEuro’s ReceiveNextSlice() method is displayed in Figure 7.5. The structural side of OptionEuro is more or less the same as before apart from a name change and the presence of a Boolean, ready_, which is set when the slice for time $T$ is received. The Compute.Values() method

```
'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX OptionEuro: ReceiveNextSlice XXXXXXXXXXXXXXXXXXXXXXXXXXX
Friend Sub IOption_ReceiveNextSlice(ByRef SVals() As Double, i As Long, t As Double)
    If t = t_ Then Call compute_O_values(slice)
End Sub
Private Sub compute_O_values(ByRef SValues() As Double)
    Dim i As Long
    For i = 1 To M_
        OValues_(i) = pay_.Payoff(SValues(i))
    Next i
    ready_ = True
End Sub
```

**Figure 7.5** The OptionEuro object: ReceiveNextSlice()

```
'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX OptionKI: ReceiveNextSlice XXXXXXXXXXXXXXXXXXXXXXXXXXX
Friend Sub IOption_ReceiveNextSlice(ByRef SVals() As Double, i As Long, t As Double)
    Call UpdateStarted(slice, t)
    If t = t_ Then Call compute_O_values(slice)
End Sub
Private Sub compute_O_values(ByRef SValues() As Double)
    Dim i As Long
    For i = 1 To M_
        OValues_(i) = pay_.Payoff(SValues(i))
    Next i
    ready_ = True
End Sub
Private Sub UpdateStarted(ByRef SValues() As Double, t As Double)
    Dim j As Long
    For j = 1 To M_
        If Not Started_(j) Then
            Started_(j) = evt_.TestEvent(SValues(j))
        End If
    Next j
End Sub
```

**Figure 7.6** The OptionKI object: ReceiveNextSlice()
differs from `OptionAverage::Compute_Values()` only in that it tests to see if `ready_` is set (and if it isn’t, throws) and line 7.4a is replaced by

$$\text{value} = \text{OValues}_i\text{.} \quad (7.4)$$

`OptionKI` and `OptionKO` follow a nearly identical pattern. We look only at `OptionKI`. In addition to the data members in `OptionEuro` there is now a stopping time object, just as in level 4, and two `Booleans`: an array `started_` with value `True` if the option has been knocked-in along that path, and a scalar `ready_` with value `True` if the option’s payoff at the final time has been calculated.

When a new slice is received (`ReceiveNextSlice()` in Figure 7.6) two things happen. First, a check is made to see if the option has knocked-in and `started_` updated. Second, if the new slice is for time $T$ the array of final, knocked-in, values is computed and `ready_` set to `True`.

The only other substantive difference is that line 7.4a is replaced by

$$\text{If Started}_i\text{() Then value} = \text{OValues}_i\text{() Else value} = 0 \quad (7.5)$$

which sets the payoff to the knocked-in value if it has been knocked in, and zero otherwise.

## 7.3 THE EVOLVER OBJECT

The evolver generates one slice at a time. It returns slices to AppMC on line 7.1a in any order it pleases. In case AppMC can use it, the $i$ counter is passed to it. The evolver interface object `IEvolver` is given in Figure 7.7. It defines a single substantive procedure, `EvolveNextSlice()`. We describe only a single conforming object, `EvolverW`, which acts as a façade for an `IWiener` and an `IGenerator` object. `EvolverW` is shown in Figure 7.8. Apart from looking after the `IWiener` and `IGenerator` objects its only responsibility is to respond to an `EvolveNextSlice()` request by dispatching it on in turn to its composited objects. These do the real work. The `IWiener` object generates slices from a Wiener process and from them the `IGenerator` object generates values of the underlying.

### The Wiener object

It is the responsibility of this object to return slices from a Wiener process. For the moment we have only a single object, `WienerPlain`, conforming to the interface, `IWiener`. As the name suggests there is nothing fancy about it. The payload procedure is `EvolveNextSlice()` (Figure 7.9; we omit the `IWiener` definition). As it generates slices sequentially, it checks that the slice counter $i$ it is passed is one more than the previous one. It asks its composite `NormalGenerator` object for a slice of normal variates which it uses to update the slice of previously stored Wiener path values. When we look at faster
Monte Carlo, this naive generator is replaced by more sophisticated versions, for instance a Brownian bridge object.

The NormalGenerator object, nor_, is the same as the level 4 version but with an additional method that returns an array of standard normals. The method call,

\[ \text{Call nor_}.\text{VectNorm(N_Slice_, i, N_)} \]

passes to VectNorm() two additional parameters, the step counter \( i \) and the total number of steps, \( N_\). These arguments are currently ignored by the VectNorm(). Later they will be used to inform nor_ of the relative importance of the slice it is returning. This will allow the NormalGenerator to return ‘better’ samples for steps of greater importance.

The Generator object

In level 4 the generator object, GeneratorGBM, had a composite NormalGenerator object of its own to generate increments to a GBM process. This is no longer sufficient. To get potentially better samples we move the responsibility for getting increments away from the IGenerator object and give it instead to the IWiener object. All that the IGenerator object now does is to obtain samples (from a GBM) from a Wiener process. In the case of a GBM it is possible to solve the SDE so that obtaining GBM samples is trivial. For processes that do not possess easy solutions – for instance, CEV processes or processes with stochastic volatility – the generator object would have to do much more work.

The IGenerator interface is given in Figure 7.10. Compared to level 4, instead of having a NextS() method it has a EvolveNextSlice() method. The implementation of EvolveNextSlice() in GeneratorGBM is shown in Figure 7.11. It accepts an array of samples from a Wiener process at time \( t \), and returns, \textit{in situ}, a sample from a GBM for time \( t \).
A Slice-Based Monte Carlo

'WienerPlain object.

Implements IWiener

Private nor_ As NormalGenerator
Private M_ As Long
Private N_ As Long
Private t_ As Double
Private dt_ As Double
Private rt_dt_ As Double
Private W_Slice_() As Double
Private N_Slice_() As Double
Private Last_i_ As Double

Private Sub Class_Initialize()
    Set nor_ = New NormalGenerator
End Sub

Private Sub Class_Terminate()
End Sub

Friend Property Let IWiener_SetValues(ByRef data As InputManager)
    N_ = data.N
    M_ = data.M
    t_ = data.t
    dt_ = t_ / N_
    rt_dt_ = Sqr(dt_)
    ReDim W_Slice_(1 To M_) As Double
    ReDim N_Slice_(1 To M_) As Double
    Call IWiener_Initialise
End Property

Friend Sub IWiener_Initialise()
    Dim j As Long
    For j = 1 To M_ Step 1
        W_Slice_(j) = 0#
        N_Slice_(j) = 0#
    Next j
    Last_i_ = 0
End Sub

Friend Sub IWiener_EvolveNextSlice(ByRef slice() As Double, i As Long, t As Double)
    If i <> Last_i_ + 1 Then Call RaiseError(1234, "WienerPlain", "non-sequential")
    Call nor_.VectNorm(N_Slice_, i, N_)
    Dim j As Long
    For j = 1 To M_ Step 1
        slice(j) = W_Slice_(j) + rt_dt_ * N_Slice_(j)
        W_Slice_(j) = slice(j)
    Next j
    t = i * t_
    Last_i_ = i
End Sub

Figure 7.9 The WienerPlain object
Implementing Models of Financial Derivatives

Public Property Let SetValues(ByVal data As InputManager): End Property
Public Sub Initialise(): End Sub
Public Sub EvolveNextSlice(ByVal slice() As Double, i As Long, t As Double)
End Sub

Friend Sub IGenerator_EvolveNextSlice(ByVal vec() As Double, i As Long, t As Double)
Dim drift_t As Double: drift_t = drift_ * t
Dim j As Long
For j = 1 To M_
    vec(j) = S_0_ * Exp(drift_t + sigma_ * vec(j))
Next j
End Sub

Figure 7.10 The IGenerator interface

Friend Sub IGenerator_EvolveNextSlice(ByVal vec() As Double, i As Long, t As Double)
Dim drift_t As Double: drift_t = drift_ * t
Dim j As Long
For j = 1 To M_
    vec(j) = S_0_ * Exp(drift_t + sigma_ * vec(j))
Next j
End Sub

Figure 7.11 GeneratorGBM.EvolveNextSlice()

Why have a separate evolver object? Three reasons. First, it tidies up AppMC. Second, it decouples AppMC from any knowledge of the underlying processes and their sampling. Third, since it is polymorphic it can change; the generating process does not have to be separated into a Wiener part and a generator part, nor need there be a Wiener part at all – innovations could be gamma for all that AppMC knows.

7.4 SUMMARY

This chapter has described a new structure for the Monte Carlo method. It has two components. First, a new option object and, second, a new evolution method (further adjustments to the option object will be necessary as we progress further). We are now firmly focused on slice-wise evolution, evolving an array through time rather than step-by-step along a single sample path.

In the next few chapters we develop the structure further. However, this shall concentrate on the process of object creation and the provision of meta-class data that enables the application to track individual objects.

Currently the application works only for a single state variable. The data structure being evolved will also need to adapt away from a simple array.

7.5 EXERCISES

This set of exercises asks you to extend the definition of the option object to enable you to value options whose payoffs depend on discrete reset dates.

A set of reset dates is \( T = \{ T_k \}_{k=1}^K \) with \( t \leq T_1 < \ldots < T_K \leq T \) such that the option payoff is determined in some way, directly or indirectly, by the values of the underlying asset only on dates \( t \in T \).

You will need to modify the front-end to enable the user to specify the set \( T \). For instance you may restrict \( T \) so that \( T_{k+1} - T_k = \Delta T \) is constant for all \( k \). If so, then you can specify the set \( T \) by the triple \( (T_1, \Delta T, K) \).

Most methods for valuing discretely reset options require asset values to be generated for dates \( t \in T \). This usually means that each reset date must coincide with a time step, so that for all \( k \), \( T_k = t_k \) for
some $i_k$. For a fixed final maturity time $T$ and a time step $\Delta t = T/N$, the number of time steps $N$ cannot be chosen arbitrarily. $N$ must be compatible with $T$. Make sure your code validates for this.

For the following options assume, unless otherwise stated, that the option has strike $X = 100$, maturity time $T = 1$, and that it is written on an asset obeying a geometric Brownian motion with current value $S_0 = 100$, $r = 0.05$, and $\sigma = 0.2$.

The exercises ask you to investigate how the option price varies as $K$ changes. Suppose that, for a given $K$, the dates $T_k = \frac{k-1}{K} T$, $k = 1, \ldots, K$, are equally spaced between time $0 = t_0$ and time $T$. Use values for $K$ in the set $K = \{2, 3, 5, 7, 9, 11, 13, 17, 21, 26\}$ and values of $N$ that are multiplies of 1200. With this specification of $T$ these $N$ are compatible with $K$ in $K$.

1. **Discretely reset average rate options.** A discretely reset arithmetic average rate option pays off against an arithmetic average value $a^A$,

$$ a^A = \frac{1}{|T|} \sum_{t \in T} S_t, $$

(7.7)

computed from stock values observed at times in the set $T$. A geometrically averaged average rate option pays off against the geometric average $a^G$,

$$ a^G = \left( \prod_{t \in T} S_t \right)^{1/|T|} $$

(7.8)

$$ = \exp \left( \frac{1}{|T|} \sum_{t \in T} \ln(S_t) \right). $$

(7.9)

For example, a fixed strike average rate call option has payoff $H(a^A) = (a^A - X)^+$ at maturity time $T$.

Implement discretely reset arithmetically averaged and geometrically averaged average rate option objects. Extend the Monte Carlo, lattice and PDE methods to be able to value them. As $K$ increases, how quickly do the values of the discretely reset options converge to a limiting value of a continuously reset average rate option?

Note: geometrically averaged average rate options have explicit formula. The formula for a continuously reset geometric average rate call option was given in exercise 1(d), Chapter 1. Suppose a discretely reset geometric average rate call option resets at times $T_1 < \ldots < T_N = T$ with $T_{k+1} - T_k = \Delta T$ constant. Then the value $c$ of the option at time $t = T_1 - \Delta t$ is

$$ c_t = e^{-r(T-t)} \left( e^{\hat{\sigma} + \frac{1}{2} \tau^2} S_t N(d_1) - X N(d_2) \right) $$

(7.10)

where

$$ \hat{\sigma} = \frac{N+1}{2N} \left( r - \frac{1}{2} \sigma^2 \right) T, $$

(7.11)

$$ \tau^2 = \sigma^2 \frac{(N+1)(2N+1)}{6N^2} T, $$

(7.12)

$$ d_1 = \frac{1}{\tau} \left( \ln \left( \frac{S_t}{X} \right) + \hat{\sigma} \right) + \tau, $$

(7.13)

$$ d_2 = d_1 - \tau. $$

(7.14)

\(^5\) See for example Haug (2007) and Appendix B.
2. **Discretely reset barrier options.** A discretely reset barrier option checks the barrier condition only at dates \( t \in T \). For instance, an up-and-in barrier call option with strike \( X \) and barrier level \( U \) maturing at \( T \) has payoff
\[
(S_T - X)^+, \quad \text{if there exists } t \in T \text{ such that } S_t \geq U,
0, \quad \text{otherwise.} \tag{7.15}
\]
Implement a discretely reset down-and-out barrier option object. Extend the Monte Carlo, lattice and PDE methods to be able to value it. As \( K \) increases, how quickly do the values of the discretely reset option converge to a limiting value of a down-and-out continuous barrier option?

3. **Bermudan options.** Bermudan options can be exercised early but only at times \( t \in T \). For the lattice and PDE methods this means that the American comparison must be made at times \( t \in T \) and at no other times.

Implement a Bermudan option object. Extend the lattice and PDE methods to be able to value it. As the number of exercise opportunities increases how quickly do the values of the Bermudan option converge to a limiting value of an American put?\(^6\)

4. In the Monte Carlo method the plain evolver steps forwards through time in strict sequence from time \( t_0 \) to time \( t_N \). A backwards Brownian bridge evolver first constructs asset values directly for time \( t_N \) and then steps backwards in sequence from time \( t_N \) to time \( t_{N-1} \) and so on until time \( t_1 \). Implement a backwards Brownian bridge `IEvolver` conforming evolver object.

To evolve for time \( t_0 \) directly to time \( t_N \) use the exact solution for geometric Brownian motion. For \( \varepsilon \sim N(0, 1) \) a standard normal variate generate \( S_N \) from \( S_0 \) by
\[
S_N = S_0 \exp \left( \left( r - \frac{1}{2} \sigma^2 \right) (t_N - t_0) + \sigma \sqrt{t_N - t_0} \varepsilon_N \right). \tag{7.16}
\]
Then, given two points \( S_i \) for time \( t_i \) and \( S_k \) for time \( t_k > t_i \), the Brownian bridge technique generates a value \( S_j \) for time \( t_j, t_i < t_j < t_k \), as follows.

Set
\[
\omega_k = \frac{1}{\sigma} \left( \ln \left( \frac{S_k}{S_i} \right) - \left( r - \frac{1}{2} \sigma^2 \right) (t_k - t_i) \right) \tag{7.17}
\]
and compute \( \omega_j \) as
\[
\omega_j = \frac{t_j - t_i}{t_k - t_i} \omega_k + \sqrt{\frac{(t_j - t_i) (t_k - t_j)}{(t_k - t_i)^2}} \varepsilon_i, \quad \text{for } \varepsilon_i \sim N(0, 1). \tag{7.18}
\]
Then generate \( S_j \) as
\[
S_j = S_i \exp \left( \left( r - \frac{1}{2} \sigma^2 \right) (t_j - t_i) + \sigma \omega_j \right). \tag{7.19}
\]

\(^6\) Whose value you should establish independently.
The implementations we have seen so far have progressed from the ridiculous to the (by comparison) sublime. There remain several features of VBA that have yet to be exploited and a number of important design idioms to present. In this chapter we examine one of each of these; a significant VBA feature, events, and a key design idiom, the factory pattern (albeit non-polymorphic).

Events are used to send information back through a chain of calling procedures to an object that can handle the event. Control then passes back to the source object. In the context of the Monte Carlo application we might want to use events to send intermediate results and counters back to a receiving object. This could handle them by outputting to the front-end. In fact we end up using an alternative and perhaps cleaner mechanism of passing an output object around the application along with the factory.

Events are flexible. They can be used to report any event an application cares to define, for instance, in conjunction with the Err object, to implement more sophisticated error reporting.

The factory pattern is an important design pattern that, when it can be fully implemented, decouples code that creates objects from a knowledge of the objects they are creating. The design in this chapter still has to know about every possible object that it could create, and so is heavily coupled to the rest of the application. In Part IV we show how a better, fully decoupled, factory can be built.

The level 5 version of the application, MC_example_v5, does two things. It introduces a factory and it also reconfigures the Monte Carlo procedure itself. One obvious change is in the front-end (Figure 8.1). To prepare for the more sophisticated Monte Carlo techniques later in the book the application has been adapted to enable it to run more than one entire Monte Carlo valuation at a time, and the front-end reflects this. Why would anyone want to do this? Because for the more advanced, faster, Monte Carlo the standard error may heavily overestimate the true standard deviation of the Monte Carlo value. The only way to find the standard error is to run the Monte Carlo a number of times and compute it directly as the standard deviation of the Monte Carlo outcomes. We need this ability later and it is convenient to build it in now.¹

It also means, as we see below, that more than one object needs to be able to output. The simple output structure currently in place cannot cope.

In Figure 8.1 the N Runs cell allows the user to specify how many replications of the Monte Carlo valuation are to take place. Results from successive runs are output in successive rows on the right-hand side. An overall option price and a standard error (computed as the standard deviation of the individual runs) is output in the row above. Times are output for the individual runs, and the overall time.

First we look at the syntax of events; then at how the level 5 application is put together. We then look at a factory and at output. Finally (eventually) we look at the improved Monte Carlo application. By now, the application as a whole is becoming so abstract that the numerics component looks almost like an after-thought.

### 8.1 EVENTS

There are three parts to defining and using events. First, a source object has to define an event. Second, a receiving object has to be able to intercept the event and, third, the event has to be raised. Any object

¹ Luckily you don’t have to do this every time you want a Monte Carlo estimate, just enough times so that you know roughly what it is going to be.
can intercept an event as long as it contains a reference to the object raising the event. In particular, more than one object can intercept the same event as long as they contain references to the same source object.

To be able to raise an event the source object has to do two things:

1. declare an event using the `Event` keyword;
2. raise the event using the `RaiseEvent` keyword.

To intercept an event the sink object must

1. contain a reference to the source object, declared with the `WithEvents` keyword;
2. define a procedure that is executed when the source raises an event.

**Defining and raising events**

Both defining and raising events is straightforward. Suppose an object `SourceObject` contains the code in Figure 8.2. This declares with the `Event` keyword an event called `PassingALong()` that takes a `Long` as an argument (events can take arguments of any type, just like ordinary procedures). The event is raised inside the procedure `SomeSub()`. When the `RaiseEvent` statement is executed the event is raised with, in this case, the argument 93.

An object’s constructor and destructor, `Class_Initialize()` and `Class_Terminate()`, are special events that are fired only when an object is created or the last reference to it is removed.\(^2\) Events cannot be raised (or intercepted) by procedures in standard modules.

\(^2\) Not quite true. They can be raised in code, but to do so is evil. It is much better to define your own separate initializer and clear-up procedures.
Intercepting events

It is also simple to intercept an event. Suppose the object SinkObject contains the code in Figure 8.3. A SourceObject, AnObj_, is declared WithEvents so that handling its events is enabled. AnObj_ has to be Set, perhaps in a constructor as in Figure 8.3, or perhaps through an assignment. When the object that AnObj_ references fires the event PassingALong(), the Sub AnObj__PassingALong() intercepts it. AnObj__PassingALong() must have the same signature as PassingALong().

The name of the intercepting Sub cannot be chosen arbitrarily. It is formed by concatenating the reference name, AnObj_, with the event name, PassingALong(), joined with an underscore.

Once AnObj_ has been declared in SinkObject its name appears in the left-hand drop-down menu for SinkObject in the VBA IDE, and when that item is selected the events it can raise appear in the right-hand drop-down menu. Figure 8.4 shows an object mess_ declared WithEvents. The right-hand drop-down menu lists the five events it can fire. Selecting NewRun, for instance, inserts its stub, with the name mess__NewRun(), into the class module.

Summary of Events

Events are a powerful feature of VBA. They are at their most useful in applications that are event driven, that is, that respond to client requests or other asynchronous intervention. For a purely numerical application their use is problematical. It is true that from a client perspective one may well wish to set a numerical application off and then tell it to go away and to come back when it has finished. The numerics signals it has finished by raising an event, sending its results back to the client. But from within the numerics itself there seems to be little need to be event-driven.

We conclude that events do not suit our present purposes and move on.
The level 4 version of our Monte Carlo application handles output rather clumsily. The OutputManager object is created in the AppObWrapper (Figure 6.11, page 91) and receives values only from Property Gets defined for the IApp object. Everything has to pass through the IApp interface. This is not necessarily a bad thing in a stable application, but one may wish to add in a bit more flexibility.

There is a saying that to solve most programming problem you just have to add in another layer of indirection. There is quite a bit of truth in this. The further we go the more layers of indirection are introduced. One ends up with a hierarchy of objects, each responsible for some aspect of functionality; many layers of indirection one on top of the other. To give us the ability to manage output more flexibly we add a Factory layer.

The new structure is shown in Figure 8.5. Input and output are shown being directed to the front-end. In fact we allow the output object to be polymorphic and in Chapter 9 give an object to output to file as an alternative.

The AppObWrapper object is responsible for creating the objects that read in data and output results. It initiates further object creation commissioned in the Factory. In outline the Factory reads in the Monte Carlo specifications from the front-end. It uses these to set up the Monte Carlo application and the input and output channels, and links them to the Monte Carlo application.

We look at the Factory object in detail in section 8.3 and at output in section 8.4. We look first at AppObWrapper and the Monte Carlo application objects.

Figure 8.4 mess_'s events in the RHS drop-down window

8.2 THE LEVEL 5 MONTE CARLO APPLICATION

The level 4 version of our Monte Carlo application handles output rather clumsily. The OutputManager object is created in the AppObWrapper (Figure 6.11, page 91) and receives values only from Property Gets defined for the IApp object. Everything has to pass through the IApp interface. This is not necessarily a bad thing in a stable application, but one may wish to add in a bit more flexibility.

There is a saying that to solve most programming problem you just have to add in another layer of indirection. There is quite a bit of truth in this. The further we go the more layers of indirection are introduced. One ends up with a hierarchy of objects, each responsible for some aspect of functionality; many layers of indirection one on top of the other. To give us the ability to manage output more flexibly we add a Factory layer.

The new structure is shown in Figure 8.5. Input and output are shown being directed to the front-end. In fact we allow the output object to be polymorphic and in Chapter 9 give an object to output to file as an alternative.

The AppObWrapper object is responsible for creating the objects that read in data and output results. It initiates further object creation commissioned in the Factory. In outline the Factory reads in the Monte Carlo specifications from the front-end. It uses these to set up the Monte Carlo application and the input and output channels, and links them to the Monte Carlo application.

We look at the Factory object in detail in section 8.3 and at output in section 8.4. We look first at AppObWrapper and the Monte Carlo application objects.
The **AppObWrapper** object

The level 5 `main()` is identical to the level 4 and level 3.2 `main()` in Figure 5.15 (page 76). It sets up error handling and instantiates an `AppObWrapper` object, which it asks to run().

The new `AppObWrapper` object is shown in Figure 8.6. Compared to the level 4 version of `AppObWrapper` in Figure 6.11 (page 91) it has now handed over to the factory the task of setting up I/O. The factory instantiates input (always of type `InputManager`) and output objects. There are two sorts of output objects: a `Monitor` object and a polymorphic outputter object conforming to the `IOutputter` interface. The factory gives references to the I/O objects to whatever objects in the application need them.

The `AppObWrapper` object requests access to the outputter in its constructor by calling `fct_.Outputter`. It also asks from the `Factory::OuterApp()` method a reference `app_` to an `IApp` object, initiating a cascade of object creation. `AppObWrapper::run()` outputs a start message, sets the stopwatch going, and asks its `app_` to `run()` (Figure 8.6).

The **AppRepeater** object

In Figure 8.5 the Monte Carlo application object is shown within an outer object (there labelled `AppRep`). It is in the outer object that the repeated Monte Carlo valuation runs are set in motion. Not one but two objects conform to the `IApp` interface. The first is the Monte Carlo application `AppMC`, but there is another, `AppRepeater`, shown in Figure 8.7. The `AppRepeater` object runs the `AppMC` object `N` runs times.

The `Factory` object sets up both the `AppRepeater` and the `AppMC` objects. In Figure 8.6 the `AppObWrapper` object calls the `Factory::OuterApp()` property to obtain a reference to an
Private out_ As IOutputter
Private wch_ As StopWatch
Private app_ As IApp

Private Sub Class_Initialize()
Set wch_ = New StopWatch
Dim fct_ As Factory: Set fct_ = New Factory 'Set up the Factory
Set app_ = fct_.OuterApp 'Set the application
Set out_ = fct_.Outputter 'Set the outputter
Set fct_ = Nothing 'Releases the Factory
End Sub

Private Sub Class_Terminate()
Set wch_ = Nothing
Set out_ = Nothing
Set app_ = Nothing
End Sub

Public Sub run()
Call out_.StartRun
    Call wch_.Start_Timer
    Call app_.run
    Call wch_.Stop_Timer
    Call out_.FinalResults(app_, wch_.Elapsed_Time)
End Sub

Figure 8.6 The AppObWrapper object

AppRepeater; the AppRepeater object calls Factory::Application() Property (in SetValues(), Figure 8.7) which returns a reference to an AppMC. Factory also gives the AppRepeater a reference to the IOutputter object and to the Monitor.

When AppRepeater::run() is called it loops around NRuns_ times. On each loop it calls the AppMC::run() method. The AppRepeater object sends a run counter to the Monitor at the start of each run and sends results to the IOutputter at the end of each iteration.

The decorator pattern

The IApp interface\(^3\) is undemanding. There are getters for the option value and its standard error, the utility setter SetValues(), and a single verb, run(). That is all that an object conforming to IApp has to provide. As far as the client of an IApp object is concerned it could be doing anything, as long as it delivers the interface.

In our case the AppRepeater object has an IApp object as a composited data member, instantiated by a call to Factory::Application() that returns an AppMC object. The application would run just as well if the AppObWrapper object had a composited AppMC instead of the AppRepeater it actually has.

We have a structure in which an object of one conforming type is wrapping another object conforming to the same type. This is called the decorator pattern. The outer object adds or modifies the functionality of the inner object. In our case the outer object, IApp::AppRepeater, decorates the inner object, IApp::AppMC, by making it run N Runs times.

\(^3\) Identical to the level 4 version in Figure 6.15, page 93, except that the SetValues() method now takes the Factory as its argument.
Implements IApp

Private out_ As IOutputter
Private wch_ As StopWatch
Private app_ As IApp
Private acc_ As IAccumulator
Private NRuns_ As Long
Private OValue_ As Double
Private SEValue_ As Double
Private mon_ As Monitor

Private Sub Class_Initialize()
Set wch_ = New StopWatch
End Sub

Private Sub Class_Terminate()
Set app_ = Nothing
Set acc_ = Nothing
Set wch_ = Nothing
Set mon_ = Nothing
Set out_ = Nothing
End Sub

Public Property Get IApp_OValue() As Double
IApp_OValue = OValue_
End Property

Public Property Get IApp_SEValue() As Double
IApp_SEValue = SEValue_
End Property

Public Property Let IApp_SetValues(ByRef fact As Factory)
Set out_ = fact.Outputter
Set mon_ = fact.MonitorChannel
Set app_ = fact.Application
Set acc_ = fact.Accumulator
NRuns_ = fact.Inputter.NRuns
End Property

Public Sub IApp_run()
Dim counter As Long
For counter = 1 To NRuns_
    Call mon_.OutputRunCounter(counter)
    Call reset
    Call wch_.Start_Timer
    Call app_.run
    Call wch_.Stop_Timer
    Call register
    Call out_.ResultsReady(app_, wch_.Elapsed_Time)
Next counter
Call GetValues
End Sub

Private Sub reset()
    wch_.Reset_Timer
End Sub

Private Sub register()
    acc_.RegisterValue = app_.OValue
End Sub

Private Sub GetValues()
    OValue_ = acc_.OValue
    SEValue_ = acc_.SDValue
End Sub

Figure 8.7 The AppRepeater object
Implementing Models of Financial Derivatives

' AppMC, MC: Forwards evolution of a vector

Implements IApp

' data section

Private opt_ As IOption
Private evl_ As IEvolver
Private N_ As Long
Private OValue_ As Double
Private SEValue_ As Double
Private slice_() As Double
Private slice_time_ As Double
Private mon_ As Monitor

' structural

Private Sub Class_Terminate()
    Set opt_ = Nothing
    Set evl_ = Nothing
    Set mon_ = Nothing
End Sub

Friend Property Get IApp_OValue() As Double
    IApp_OValue = OValue_
End Property

Friend Property Get IApp_SEValue() As Double
    IApp_SEValue = SEValue_
End Property

Friend Property Let IApp_SetValues(ByRef fact As Factory)
    Set evl_ = fact.Evolver
    Set opt_ = fact.OptionType
    Set mon_ = fact.MonitorChannel
    N_ = fact.Inputter.NSteps
    ReDim slice_(1 To fact.Inputter.NPaths) As Double
End Property

' other interface

Friend Sub IApp_run()
    Call reset
    Dim i As Long
    For i = 1 To N_
        Call mon_.OutputCounter(i)
        Call evl_.EvolveNextSlice(slice_, i, slice_time_)
        Call opt_.ReceiveNextSlice(slice_, i, slice_time_)
    Next i
    Call register
End Sub

Private Sub reset()
    Call evl_.Initialise
    Call opt_.Initialise
End Sub

Private Sub register()
    OValue_ = opt_.OValue
    SEValue_ = opt_.SEValue
End Sub

' end of file

Figure 8.8  The AppMC object
The AppMC object

There is only one difference between the level 5 AppMC object and the level 4 AppMC object of Chapter 7: the new AppMC::SetValues() takes the Factory as its argument. Figure 8.8 shows the new AppMC object in its entirety. The Monitor object is set by the Factory in AppMC::SetValues() procedure along with the IOption and IEvolver objects. Two objects – AppRepeater and AppMC – are able to output via the same Monitor object: AppMC::run() asks the Monitor to output a step counter; AppRepeater::run() sends a run counter to the Monitor. The Monitor object is discussed in section 8.4.

8.3 THE Factory OBJECT

The Factory is used by the rest of the application to generate objects on demand. It has a procedure for each interface type that returns a reference to a new object conforming to that type. This is much like one of the responsibilities of the level 4 InputManager object. Here in level 5 the emphasis is not on input per se but on object creation, of which inputting parameter values is just one part. The Factory object is central; data describing the Monte Carlo method is peripheral. Indeed as a throw-back to level 3 the Factory object has an InputManager as a composited object whose responsibility is just to read in parameter values from the front-end. When InputManager (Figure 8.9) was introduced in level 3 it encapsulated input to the application. In level 4 it took on the responsibility for object creation. Now it plays a role subservient to the Factory object. It reads in both parameter values and ID characters for objects. The latter are used by the factory to make the appropriate objects.

Input could be made polymorphic, perhaps to read in option specifications from file. This would take us into a discussion of serialization – the task of representing an object in a form that can be read in from file. Preserving the values of primitive data members is easy; what is harder is encoding a specification of the type of object to which the data belongs. We investigate this in Chapter 10.

In any case the Factory object no longer needs to know about where input originates. Since another object handles input, the Factory object is decoupled from a knowledge of it.

Figure 8.10 shows the Factory object definition. It has, as composited data members, an InputManager, an IOutputter, and a Monitor object. Each of these objects has a getter Property. The IOutputter and Monitor handle output. They are discussed in more detail in section 8.4 below.

The main procedures of the Factory object are Property Gets. Two of these, Inputter() and Outputter(), are just the getters for the InputManager and IOutputter. The remainder are a series of Factory Properties that create polymorphic objects from an interface type.

Every time a Factory Property is called it creates a new object, of the type determined by the InputManager, and passes a reference to itself to the newly created object. This can then, if it wants, call Inputter() to set the values of its data members, Outputter() to allow it to Set a reference to the output channel to enable it to output, and Factory Properties to enable it to create composited objects.

A typical example is Factory::Payoff(). It asks the InputManager for a character determining the type of the IPayoff conforming object it has to create. The Select statement executes Set...New. The newly created object’s SetValues() Property is then called to enable it to initialize its data members, including Setting any composited objects, and then the reference is returned.

There is some validation from the library Function, Validators.check_char(). This is similar to the get_char() Function in Figure 6.12 (page 92). It takes a character (read in by the InputManager from the front-end) and checks it against a String of acceptable characters. If the character does not appear in the String, then check_char() throws.

Clearly there is some telepathy going on here. The user typing in characters on the front-end has to know which ones to enter. There is no mechanism (at level 5) to enable objects to let the front-end know what characters they accept.
Private sigma_ As Double 'Stock volatility
Private r_ As Double 'Stock short rate
Private S_0_ As Double 'Initial stock price
Private T_ As Double 'Final time
Private X_ As Double 'Strike
Private U_ As Double 'upper barrier
Private D_ As Double 'lower barrier
Private N_steps_ As Long 'Number of time steps
Private N_basis_ As Long 'Number of basis fns
Private N_paths_ As Long 'Number of paths
Private N_runs_ As Long 'Number of complete runs
Private p_type_ As String 'Payoff type
Private e_type_ As String 'StoppingTime type
Private o_type_ As String 'Option type
Private g_type_ As String 'Generator type
Private NC_ As Long 'Interval for output counter

Private Sub Class_Initialize()
Call Read_in_data
End Sub

Private Sub Read_in_data()
S_0_ = Check_strictly_positive(get_double(12, 6), "S_0")
r_ = Check_strictly_positive(get_double(13, 6), "r")
sigma_ = Check_strictly_positive(get_double(14, 6), "sigma")
X_ = Check_strictly_positive(get_double(17, 6), "X")
T_ = Check_strictly_positive(get_double(18, 6), "T")
U_ = Check_strictly_positive(get_double(19, 6), "U")
D_ = Check_strictly_positive(get_double(20, 6), "D")
N_steps_ = Check_strictly_positive(get_long(12, 9), "# steps")
N_paths_ = Check_strictly_positive(get_long(13, 9), "# paths")
N_runs_ = Check_strictly_positive(get_long(14, 9), "# runs")
p_type_ = get_char(24, 6, "") 'Payoff type
e_type_ = get_char(25, 6, "") 'StoppingTime type
o_type_ = get_char(26, 6, "") 'Option type
g_type_ = get_char(24, 9, "") 'Generator type
NC_ = Check_strictly_positive(get_long(27, 9), "NC")
End Sub

Figure 8.9 The InputManager object
Private inp_ As InputManager
Private out_ As IOutputter
Private mon_ As Monitor

Private Sub Class_Initialize()
    Set inp_ = New InputManager
    Set mon_ = New Monitor: Call mon_.SetValues(inp_)
    'Set out_ = New OutputterFile
    Set out_ = New OutputterFrontEnd
End Sub

Private Sub Class_Terminate()
    Set inp_ = Nothing
    Set out_ = Nothing
    Set mon_ = Nothing
End Sub

Friend Property Get Inputter() As InputManager
    Set Inputter = inp_
End Property

Friend Property Get Outputter() As IOutputter
    Set Outputter = out_
End Property

Friend Property Get MonitorChannel() As Monitor
    Set MonitorChannel = mon_
End Property

Friend Property Get Application() As IApp
    Set Application = New AppMC: Application.SetValues = Me
End Property

Friend Property Get Accumulator() As IAccumulator
    Set Accumulator = New AccumulatorTerminal: Accumulator.SetValues = Me
End Property

Friend Property Get OuterApp() As IApp
End Property

Friend Property Get Evolver() As IEvolver
    Set Evolver = New EvolverW: Evolver.SetValues = Me
End Property

Friend Property Get Process() As IGenerator
    Select Case check_char(inp_.GType, "g", "Process")
        Case "g": Set Process = New GeneratorGBM
    End Select
    Process.SetValues = Me
End Property

Friend Property Get Payoff() As IPayoff
    Select Case check_char(inp_.PType, "bcp", "Payoff")
        Case "c": Set Payoff = New PayoffCall
        Case "p": Set Payoff = New PayoffPut
        Case "b": Set Payoff = New PayoffBond
    End Select
    Payoff.SetValues = Me
End Property

'****** Remaining factory properties are omitted ******'
An example of using the Factory object, given in Figure 8.6, is in the definition of the AppObWrapper object. In the constructor AppObWrapper::Class.Initialize() the line

\[
\text{Set app\_} = \text{fct\_.OuterApp} \quad (8.1)
\]

requests the Factory object (referenced by fct\_) to instantiate a new instance of the application object and assigns app\_ as a reference to it. Following the same steps as the Payoff() method, the Factory Property Get OuterApp() returns a reference to a new AppRepeater, an object conforming to IApp. The Factory object calls the IApp Property Let SetValues(), and passes to it a reference to itself, before the reference is returned.

Figure 8.11 shows the AppRepeater::IApp_SetValues() Property, repeated from Figure 8.7. It uses the Factory reference it has been passed to initialize its data members; these in turn can use the Factory object to initialize their composited objects. This particular example shows the AppRepeater object calling both the Inputter() and Outputter() Properties of the Factory.

When the fact.Application method is called, and before the IApp reference is returned, its SetValues() method is called. The object whose reference is assigned to app\_ is fully initialized before it leaves the Factory.

To set its NRuns\_ data member, the line

\[
\text{NRuns\_} = \text{fact.Inputter.NRuns} \quad (8.2)
\]

is executed. This doubly qualified function calls the NRuns Property of the Inputter Property of the Factory object. It could be, and later is, replaced by a singly qualified method call to the Factory object.

Objects created by the Factory object should define a SetValues() Property. This is not absolutely necessary but it makes life easier for everyone if it is assumed that one exists, even as a stub. Later we create an interface that every Factory creatable object must conform to. SetValues() is a part of that interface.

### 8.4 OUTPUT

Output occurs through two channels: the monitor and the outputter. When output is required an object calls the appropriate procedure of a composited IOutputter object. The Monitor object displays the values of counters to enable the progress of the application to be monitored; the outputter outputs mainly results from the application. Outputter objects implement the interface IOutputter so that outputting results can be handled polymorphically.

Five things can happen that trigger output, so we define five corresponding items of output. The events are:

1. **The application starts up.** This triggers a message from the outputter. When outputting to the front-end we use it to blank-out output cells. When outputting to file it does nothing (see Chapter 9).
2. **A new run begins.** The run number is output by the Monitor object.

3. **A counter increments.** A counter records the sample path or time step. This is output by the Monitor to show how the run is progressing.

4. **New results are ready.** New results from a single Monte Carlo run are ready. They are dealt with by the outputter, and sent to the front-end or to file.

5. **Final aggregate results are ready.** Outputs aggregate values and standard deviations.

The relationship between output events is illustrated schematically in Figure 8.12. There are two concentric cycles of activity outputting to either the Monitor or to the outputter.

The Monitor object is displayed in Figure 8.13. It handles the new run and counter increment events. IOutputter objects, whose interface is given in Figure 8.14, handle the start run, results ready and final results events.

The StartRun() and FinalResults() methods of the outputter object are called (by AppObWrapper) at the beginning and end of a series of runs. At the start of each run a run counter is output (in

![Figure 8.12 Pattern of output events](image)

![Figure 8.13 The Monitor object](image)
The IOutputter interface

```vba
Public Sub StartRun(): End Sub
Public Sub ResultsReady(counter As Long, ByRef app As IApp, T As Double): End Sub
Public Sub FinalResults(ByRef app As IApp, T As Double): End Sub
```

end of file

Figure 8.14  The IOutputter interface

The OutputterFrontEnd object

```vba
'XX  OutputterFrontEnd. Outputs to the front end
Implements IOutputter 'Output interface
'XX data section XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Private c_value_ As Double
Private standard_error_ As Double
Private elapsed_time_ As Double
'XX interface XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Public Sub IOutputter_StartRun()
    Call ZeroCells
    Call ZeroNextRow(1)
End Sub
Public Sub IOutputter_ResultsReady(counter As Long, ByRef app As IApp, T As Double)
    Call RegisterValues(app, T)
    Call WriteOutRow
    Call ZeroNextRow(counter_ + 1)
End Sub
Public Sub IOutputter_FinalResults(ByRef app As IApp, T As Double)
    Call RegisterValues(app, T)
    Call WriteOutData
    Call MsgBox("Application has finished", , "Monte Carlo application")
End Sub
'XX private XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Private Sub RegisterValues(ByRef app As IApp, T As Double)
    c_value_ = app.OValue
    standard_error_ = app.SEValue
    elapsed_time_ = T
End Sub
Private Sub WriteOutRow()
    Cells(11 + counter_, 12).value = counter_
    Cells(11 + counter_, 13).value = c_value_
    Cells(11 + counter_, 14).value = standard_error_
    Cells(11 + counter_, 15).value = elapsed_time_
End Sub
Private Sub WriteOutData()
    Cells(9, 13).value = c_value_
    Cells(9, 14).value = standard_error_
    Cells(9, 15).value = elapsed_time_
End Sub
Private Sub ZeroCells()
    Cells(9, 13).value = ""
    Cells(9, 14).value = ""
    Cells(9, 15).value = ""
End Sub
Private Sub ZeroNextRow(i As Long)
    Cells(11 + i, 12).value = ""
    Cells(11 + i, 13).value = ""
    Cells(11 + i, 14).value = ""
    Cells(11 + i, 15).value = ""
End Sub
'XX end of file
'XX end of file
```

Figure 8.15  The OutputterFrontEnd object
AppRepeater) and at the end of each run the results of that run. During each run a further counter
is output (in AppMC) showing the progress of that run.

The IOutputter declares these output methods. We have described only a single IOutputter con-
forming object, OutputterFrontEnd. Output to file is discussed in Chapter 9. OutputterFrontEnd
prints results to the front-end as they are generated. It is displayed in Figure 8.15.

Given the discussion in previous chapters there is not too much to add. It now outputs not just a single
result, but a series of results from each run, and aggregate results. When it is passed results from a new
run, it prints them out to a row determined by a counter passed along with the results. At each step
it blanks out the next row; this is supposed to give the user a visual clue as to how far the runs have
progressed.

8.5 SUMMARY

This chapter has looked at output in some detail, investigating and rejecting events, although investigating
output to file is deferred. A Factory object has been constructed and used.

The level 5 application is beginning to get somewhere but there is still some way to go. Output to file is
still absent and the Factory is not in a good state. It is coupled to all other objects in the application. It
will take a fair amount of effort but we shall show how it can be done in Part IV. Before then, in Part III,
we explore I/O to file.

8.6 EXERCISES

1. The Monte Carlo application has a single object, the InputManager, to both read in data values and
to store inputted values for later access by client objects. Is it sensible to have a separate object whose
purpose is solely to carry input data, without itself reading it in, making it available upon request to
other parts of the application?

(a) Modify the application MC_example_v5.xls to endow it with a polymorphic InputCarrier object.

(b) Design an OutputCarrier object to store output data (and descriptions). This should be updated
by objects with output and be passed to an OutputManager object to perform the actual output.

2. The chapter has shown how a (non-)polymorphic factory can be built to take over the role of object
creation in an application. This set of exercises asks you to adapt the current set of level 4 applications
to include a factory.

(a) Pi stream. Create a level 5 version of the pi application, pi_app_v5.xls.

(b) Implied volatility stream. Add a factory to IV_app_v4.xls to get IV_app_v5.xls.

(c) PDE stream. Construct CN_pde_v5.xls, a level 5 version of the Crank–Nicolson PDE application.

(d) Lattice stream. Incorporate a factory to the lattice application to make Lattice_application_v5.xls.
This page intentionally left blank
This part discusses file handling in VBA. It is almost a prerequisite for a half-way decent application to have the need and the ability to be able to read and write to and from file. Applications ordinarily work with data; if they can process one set of data they should also be able to process multiple sets, and that is where files come in. Files store data, conveniently, so that applications can get it when they want it.

We have an options valuation application that can value a single option at a time. In practice it would be an unusual institution that owned a single option; it has an options book containing perhaps hundreds or thousands of options (owned or monitored) which it needs to value and to compute hedge ratios for. Specifications for these options will be held on file, or placed on file, where the application can access them.

Chapter 9 presents the two VBA file-handling mechanisms. It introduces first the FileSystemObject and TextStream objects, and then the VB intrinsic file-handling functions. Various I/O idioms are described and illustrated in a simple example.

Chapter 10 adds file handling to our Monte Carlo application, allowing it to read in from file (or from a spreadsheet) a set of option specifications. These are then valued all together and the results output back to file or to a spreadsheet.
Using VBA with Excel enables applications to exploit the front-end as a source of data and as a place to send output. This is very convenient; it is a cheap and easy built-in I/O channel that is all you need to use for many applications.

However there are also many situations where I/O to a spreadsheet is not going to work well. For instance, if you want to read in from a large database it might not be very sensible to first import the data into Excel. You would prefer to read in directly from the database rather than having to go through a tedious intermediate step of first importing to a spreadsheet. In any case size limitations on individual spreadsheets could mean that they are too small to hold all the data you need to read in.

If you want your output to be passed on to other applications it is likely to be quite clumsy to first send it to Excel and then, from there, translate it into a format that the destination application can read. You would want to output directly to a medium that is easily accessible by the destination application. Also, writing out to an Excel front-end is very slow; if you need to output large volumes of data it is much faster to write to a file.

For one reason or another you find you need to be able to import data from file and to send results to file.

This chapter looks at the ways you can do this. There are two main mechanisms in VBA: using the FileSystemObject object to create and write to TextStream objects, and using VBA intrinsic functions for general random access and sequential files. We look at each mechanism in turn, ending with an example of using VBA intrinsic functions. In Chapter 10 we go further, setting up a Monte Carlo application that prices simultaneously as many options as you like, reading in their specifications from file.

We start with a general discussion about files and what operations you would like to perform on them. There are several important issues associated with file handling that we do not attempt to address. For instance, not discussed here are multiple user access, remote storage, et cetera.

9.1 FILE HANDLING IN VBA

Files are places for storing data. Their structure can be complicated and specific, with an internal format and encoding that only particular applications can comprehend and manage, or general, such as text files using a standard encoding, that many applications have the ability to read.

Access to the contents of files can be in one of several modes:

1. Sequential. The file must be worked through from beginning to end in strict sequence. It is not possible to jump backwards and forwards in the file.
2. Random. The file is a collection of records. One can read and write to records out of sequence.
3. Binary. Reading and writing can be from and to specific byte positions.

Streams in C++ are sequential channels; VBA TextStream objects are essentially just sequential text files. Database operations often need to be random access as particular individual records are sought,
accessed and processed. Our numerical work does not really need random access. At most we assume that sets of input data are presented to an application in sequence, and results are output, in sequence.

Binary mode access is the lowest level access. Here you can do anything, but it is completely unstructured. We do not use it.

To exchange data with other applications – for instance, to read in tab and comma delimited data from file – it is simplest to use sequential mode. If you need to write your own files, to re-use yourself rather than to share with other applications, then you could consider using random access mode. You devise your own output Type and use Put and Get (see section 9.3) to write it out to file and read it in again.

There are standard operations that need to be performed on files, falling into five categories. These are:

1. creating and deleting files;
2. opening and closing connections to files;
3. reading and writing from file;
4. getting and setting file attributes;
5. locational operations.

To access a file for reading and writing it must first be opened: a communications channel must be established with the file. While the file is open the operating system will (should) prevent other applications from interfering with it (for instance, by deleting it unexpectedly without your knowledge). Once you have finished with a file you should close it, freeing it to enable other applications to use it.

Locational operations involve moving files around the directory structure, creating folders, and other related actions. We are concerned chiefly with first three categories, and some of the fourth, but nothing to do with the fifth.

File I/O is inherently unsafe. It can go wrong, apparently unpredictably. A file might not be found where it is expected to be; when you wish to use it, it might be already open and locked by another user; it may not be formatted in quite the way the application attempting to read it expects; when you want to delete it, it might not be there.

For these reasons, and others, it is essential to check and to error trap appropriately when performing operations with files. If you want to create a new file you should make sure it doesn’t exist before you create it; similarly if you want to delete a file, or access it for reading or writing, you need first to make sure that it exists.

9.2 THE TextStream AND FileSystemObject OBJECTS

In Chapter 8 we presented an output object, OutputterFrontEnd, that wrote to the Excel front-end. After a discussion of the FileSystemObject and TextStream objects we describe an OutputterFile object that outputs to file. It uses the VBA FileSystemObject object to create and write to a TextStream object. As the name TextStream suggests, this is a file to which text can be streamed, much like std::cout in C++. Before examining the OutputterFile object we spend some time reviewing the FileSystemObject object it uses. We present just a sketch of the system. A standard VBA reference book (such as Kimmel et al., 2004) should be consulted for a deeper treatment.

9.2.1 The FileSystemObject object

The FileSystemObject object provides access to VBA’s file-handling functionality. It gives a single point of entry for file-handling. We use only TextStream files but this section digresses a little on the system as a whole.
The `TextStream` object is well suited to our use. The only operations we require for the `OutputterFile` object are to send successive numerical results to file and in particular random access is not needed. The resulting file can be read by any text editor, or read back into Excel.

The construction in (9.1) creates a `FileSystemObject` object:

```vba
Dim File_Sys_ As Object
Set File_Sys_ = CreateObject("Scripting.FileSystemObject")
```

File_Sys_ should be declared as an Object as on line (9.1a). The `CreateObject()` function on line (9.1b), taking for our purposes a magic argument, returns a reference to a `FileSystemObject` object. In (9.1) the object is late-bound. If a reference to the Microsoft Scripting Runtime library is set (in Tools | References) then an early bound object can be declared with

```vba
Dim File_Sys_ As Scripting.FileSystemObject
Set File_Sys_ = New Scripting.FileSystemObject
```

The `FileSystemObject` object can create `TextStream` files (to create other file types the VBA `Open` statement has to be used). There are various objects and collections that the `FileSystemObject` object can access. These are displayed in Table 9.1.

Drives is `FileSystemObject`'s sole Property. It returns a Collection object\(^1\) containing all the available Drive objects. The `FileSystemObject` object has many methods, operating of the objects listed in Table 9.1. The methods can be classified into seven main categories: getters, name getters, movers, copiers, creators, deleters, and existential. They are summarized in Table 9.2. `Spath` and `Dpath` stand for paths to the source object and destination respectively. Some methods have additional optional arguments. The getter methods return the object type suggested by their name. The delete methods cannot be undone.

The `CreateTextFile` method creates and opens a `TextStream` object for reading or writing. There is a second `FileSystemObject` method, `OpenTextFile`, not shown in the table, that opens a `TextStream`.

### Table 9.1 `FileSystemObject` objects and collections

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>File</td>
<td>Object</td>
<td>Access to the properties of a file</td>
</tr>
<tr>
<td>Files</td>
<td>Collection</td>
<td>Collection object of files within a folder</td>
</tr>
<tr>
<td>Folder</td>
<td>Object</td>
<td>Access to the properties of a folder</td>
</tr>
<tr>
<td>Folders</td>
<td>Collection</td>
<td>Collection object of folders within a folder</td>
</tr>
<tr>
<td>Drive</td>
<td>Object</td>
<td>Access to the properties of a drive</td>
</tr>
<tr>
<td>Drives</td>
<td>Collection</td>
<td>Collection object of available drives</td>
</tr>
</tbody>
</table>

### Table 9.2 Summary of `FileSystemObject` methods

<table>
<thead>
<tr>
<th>Operation</th>
<th>Files</th>
<th>Folders</th>
<th>Drives</th>
<th>Argument</th>
<th>Return</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exists?</td>
<td>FileExists</td>
<td>FolderExists</td>
<td>DriveExists</td>
<td>path</td>
<td>Boolean</td>
</tr>
<tr>
<td>Create</td>
<td>CreateTextFile</td>
<td>CreateFolder</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Delete</td>
<td>DeleteFile</td>
<td>DeleteFolder</td>
<td>–</td>
<td>path</td>
<td>–</td>
</tr>
<tr>
<td>Copy</td>
<td>CopyFile</td>
<td>CopyFolder</td>
<td>–</td>
<td>Spath, Dpath</td>
<td>–</td>
</tr>
<tr>
<td>Move</td>
<td>MoveFile</td>
<td>MoveFolder</td>
<td>–</td>
<td>Spath, Dpath</td>
<td>–</td>
</tr>
<tr>
<td>Getter</td>
<td>GetFile</td>
<td>GetFolder</td>
<td>GetDrive</td>
<td>–</td>
<td>object</td>
</tr>
<tr>
<td>Name</td>
<td>GetFileName</td>
<td>GetParentFolder</td>
<td>GetDriveName</td>
<td>path</td>
<td>String</td>
</tr>
</tbody>
</table>

\(^1\) See Chapter 16.
object for reading and can create it if it does not already exist, and a third, OpenAsTextStream, that opens a file for reading or writing but does not create one.

In addition to these methods there are a few other miscellaneous methods that do not fit the scheme of Table 9.2, but these shall be ignored here.

The various methods and Properties and their relation to the six file system objects and Collections are summarized in Table 9.3. In the table fso stands for a FileSystemObject object, fo is a Folder object, fos a Folders object, fi a File object, fis a Files object, d a Drive object and ds a Drives object.

For each object type the table gives three things: the methods and Properties that return them; their own Properties; and their own methods. The Properties are not really touched upon. The individual objects — File, Folder, Drive — have a large number of Properties between them, too many to enumerate here. The Collection objects have Collection methods and Properties: the Count, Item, Copy, Delete and Move methods, and Properties belong to the Collection object, and are not specific to the file system.

CreateTextFile is both a Folder method and a FileSystemObject method.

### 9.2.2 Using the TextStream object

The TextStream object is ideal for easily reading and writing text files. It cannot be used for random access. Standard file methods and Properties apply, but the TextStream object has its own additional methods and Properties. TextStream file objects are useful, and are very good at what they do, but they are limited in functionality.

The structure of a TextStream object is that of a stream of characters, including the newline character, Chr(13). The characters in between successive newline characters constitute a line. Reading and writing can be character by character or line by line. Reading and writing is always from or to the current position in the file.

In section 9.2.1 we described the methods used to create and open a TextStream object. It also has a Close method to close it. We omit the syntax for these file methods. Some are illustrated below but we refer the reader to accounts elsewhere for a fuller description.

---

**Table 9.3 File system summary**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Object type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name:</td>
<td>File</td>
</tr>
<tr>
<td></td>
<td>Folder</td>
</tr>
<tr>
<td></td>
<td>Drive</td>
</tr>
<tr>
<td>Returned by:</td>
<td>fso.GetFolder</td>
</tr>
<tr>
<td></td>
<td>fos.Item</td>
</tr>
<tr>
<td></td>
<td>fos.GetSpecialFolder</td>
</tr>
<tr>
<td></td>
<td>d.RootFolder</td>
</tr>
<tr>
<td></td>
<td>fi.ParentFolder</td>
</tr>
<tr>
<td></td>
<td>fo.ParentFolder</td>
</tr>
<tr>
<td>Properties:</td>
<td>Access to individual object Properties</td>
</tr>
<tr>
<td>Methods:</td>
<td>Copy/Delete/Move</td>
</tr>
<tr>
<td></td>
<td>OpenAsTextStream</td>
</tr>
<tr>
<td></td>
<td>CreateTextFile</td>
</tr>
<tr>
<td>Name:</td>
<td>Files</td>
</tr>
<tr>
<td></td>
<td>Folders</td>
</tr>
<tr>
<td></td>
<td>Drives</td>
</tr>
<tr>
<td>Returned by:</td>
<td>fo.Files</td>
</tr>
<tr>
<td></td>
<td>fo.SubFolders</td>
</tr>
<tr>
<td></td>
<td>fso.Drives</td>
</tr>
<tr>
<td>Properties:</td>
<td>Count/Item</td>
</tr>
<tr>
<td>Methods:</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>AddFolder</td>
</tr>
</tbody>
</table>
TextStream methods divide into three types: reading, writing and position; there are in addition a number of Property Gets returning information. Write ts for a TextStream object. Table 9.4 summarizes its methods and Property. Examples include WriteLine() that appends a newline character to its output and WriteBlankLines(N) that writes N newline characters. The file can be traversed only in one direction: forwards.

9.2.3 The OutputterFile object

We return to the Monte Carlo application of Chapter 8 to give it an object, OutputterFile, that outputs to file. It is shown in Figure 9.1. The IOutputter interface has a new method, a SetValues() Property Let, to enable the OutputterFile object to be passed a path specification from the front-end.

OutputterFile creates and writes to a TextStream file. As usual an adapted RAII idiom is used. The TextStream object, text_file_, is created in OutputterFile’s SetValues() method (line 9.1c) with the name (and path) specified by the o_path_ argument. o_path_ is passed to OutputterFile by the factory on line 9.1a. The second argument to CreateTextFile(), with value True, means that if the file already exists it will be overwritten. text_file_ is released in its destructor (line 9.1e). The file is explicitly closed on line 9.1d, although it would be closed automatically when it goes out of scope on line 9.1e.

The actual writing to file is done in the Private WriteOutRow() and WriteOutData() methods. These construct output lines and then call WriteLine() to output to text_file_. WriteLine() sends a String followed by a newline character so that the next line indeed starts on the next line. The WriteBlankLines(1) method writes a single blank line to file. Output lines contain tab (Chr(9) or vbTab) delimited numbers. Comma delimiters could be used instead, or indeed just spaces.

To use the new outputter a small change is made to the Factory from Chapter 8. In the factory constructor the outputter out_ is set to be the value of a Property, SetOutputter(). This is shown in Figure 9.2. The InputManager reads in a character that specifies whether output is to be to file or to the spreadsheet front-end. SetOutputter() returns the appropriate outputter. The InputManager also now reads in from the front-end a String specifying the path to the output file. Allowing the user to specify an output path in this way is much more convenient than hard-wiring a path into OutputterFile’s constructor, say.
'XX OutputterFile
'Implements IOutputter ‘Outputter interface
' data section
Private o_path_ As String 'Path to the output file
Private File_Sys_ As Object 'File system objects
Private text_file_ As Object
Private c_value_ As Double 'Output values
Private standard_error_ As Double
Private elapsed_time_ As Double

Public Property Let IOutputter_SetValues(ByRef fact As Factory)
o_path_ = fact.Inputter.out_path 'a
Set File_Sys_ = CreateObject("Scripting.FileSystemObject") 'b
Set text_file_ = File_Sys_.CreateTextFile(o_path_, True) 'c
End Property

Private Sub Class_Terminate()
text_file_.Close 'd
Set text_file_ = Nothing 'e
Set File_Sys_ = Nothing 'f
End Sub

Public Sub IOutputter_StartRun() 'Stub
End Sub

Public Sub IOutputter_ResultsReady(ByRef app As IApp, T As Double)
Call RegisterValues(app, T)
Call WriteOutRow(counter)
End Sub

Public Sub IOutputter_FinalResults(ByRef app As IApp, T As Double)
Call RegisterValues(app, T)
Call WriteOutData
Call MsgBox("Application has finished", , "Monte Carlo application")
End Sub

Private Sub RegisterValues(ByRef app As IApp, T As Double)
c_value_ = app.OValue
standard_error_ = app.SEValue
elapsed_time_ = T
End Sub

Private Sub WriteOutRow(counter As Long)
Dim data_line As String: data_line = ""
data_line = data_line & CStr(counter) & vbTab
data_line = data_line & CStr(c_value_) & vbTab
data_line = data_line & CStr(standard_error_) & vbTab
data_line = data_line & CStr(elapsed_time_)
Call text_file_.WriteLine(data_line)
End Sub

Private Sub WriteOutData()
Dim data_line As String: data_line = ""
data_line = data_line & CStr(c_value_) & vbTab
data_line = data_line & CStr(standard_error_) & vbTab
data_line = data_line & CStr(elapsed_time_)
Call text_file_.WriteBlankLines(1)
Call text_file_.WriteLine(data_line)
End Sub

Public Sub IOutputter_StartRun() 'Stub
End Sub

Public Sub IOutputter_ResultsReady(ByRef app As IApp, T As Double)
Call RegisterValues(app, T)
Call WriteOutRow(counter)
End Sub

Public Sub IOutputter_FinalResults(ByRef app As IApp, T As Double)
Call RegisterValues(app, T)
Call WriteOutData
Call MsgBox("Application has finished", , "Monte Carlo application")
End Sub

Private Sub RegisterValues(ByRef app As IApp, T As Double)
c_value_ = app.OValue
standard_error_ = app.SEValue
elapsed_time_ = T
End Sub

Private Sub WriteOutRow(counter As Long)
Dim data_line As String: data_line = ""
data_line = data_line & CStr(counter) & vbTab
data_line = data_line & CStr(c_value_) & vbTab
data_line = data_line & CStr(standard_error_) & vbTab
data_line = data_line & CStr(elapsed_time_)
Call text_file_.WriteLine(data_line)
End Sub

Private Sub WriteOutData()
Dim data_line As String: data_line = ""
data_line = data_line & CStr(c_value_) & vbTab
data_line = data_line & CStr(standard_error_) & vbTab
data_line = data_line & CStr(elapsed_time_)
Call text_file_.WriteBlankLines(1)
Call text_file_.WriteLine(data_line)
End Sub

Figure 9.1 The OutputterFile object: Interface
9.3 INTRINSIC VB LANGUAGE FUNCTIONS

The FileSystemObject object enables clients to use TextStream files. If you need to store data with a complex structure then it is likely to be slightly easier to use VB intrinsic functions than to use TextStream files; if you need to work with random access files then you have to use intrinsic functions. We review these here.

9.3.1 VB Intrinsic functions

The intrinsic functions are much less systematic than the FileSystemObject and TextStream methods; haphazard in fact. Table 9.5 summarizes some of the important file handling functions. Only a few of these will be described in any detail, and that mainly through examples. For a detailed and systematic treatment please refer to a standard text book such as Lomax (1998).

Open is used both to open files and to create them (if they do not already exist). Close and Reset each close files; Close closes a specific file, Reset closes all currently open files. FileCopy overwrites the destination file (whose name must be specified) if it already exists. Name renames a file; Dir returns the name of a file with a matching set of attributes (including a path).\(^2\)

ChDir and ChDrive are DOS-like commands that set the current default folder (directory) and drive. They do not actually open anything but they seem to fit into this slot in the table.

There are a number of functions that get and set attributes of files and folders. Some that get information are listed in Table 9.6. A file’s attributes are its system properties: whether it is hidden, or read-only, et cetera. These can be set with the SetAttr function.

When a file is open a pointer is maintained that points to the current read or write position. Several functions deal with this pointer. Loc returns its position and Seek sets its position. The EOF function returns true if the pointer is at end-of-file, False otherwise.

---

\(^2\) The use of Dir should be avoided; it can be tricky to use safely. FileSystemObject methods should be used instead.

---

### Table 9.5 Summary of VB intrinsic functions: file and folder methods

<table>
<thead>
<tr>
<th>Operation</th>
<th>Files</th>
<th>Folders</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create</td>
<td>Open</td>
<td>MkDir</td>
</tr>
<tr>
<td>Delete</td>
<td>Kill</td>
<td>RmDir</td>
</tr>
<tr>
<td>Copy</td>
<td>FileCopy</td>
<td></td>
</tr>
<tr>
<td>Open</td>
<td>Open</td>
<td>ChDir, ChDrive</td>
</tr>
<tr>
<td>Close</td>
<td>Close, Reset</td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Dir, Name</td>
<td>CurDir</td>
</tr>
</tbody>
</table>

---
Implementing Models of Financial Derivatives

**Table 9.6** Getting information about files and folders

<table>
<thead>
<tr>
<th>Function</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetAttr</td>
<td>File and folder attributes</td>
</tr>
<tr>
<td>FileAttr</td>
<td>The access mode of a file</td>
</tr>
<tr>
<td>FileDateTime</td>
<td>Last modification date and time</td>
</tr>
<tr>
<td>FileLen, LOF</td>
<td>The length of a file</td>
</tr>
</tbody>
</table>

**Table 9.7** Read and write methods

<table>
<thead>
<tr>
<th>Access type</th>
<th>Writing data</th>
<th>Reading data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential, comma delimited</td>
<td>Write #</td>
<td>Input #</td>
</tr>
<tr>
<td>Sequential, tab/space delimited</td>
<td>Print #</td>
<td>Line Input #</td>
</tr>
<tr>
<td>Random</td>
<td>Put</td>
<td>Get</td>
</tr>
<tr>
<td>Binary</td>
<td>Put</td>
<td>Get</td>
</tr>
</tbody>
</table>

When a file is opened a number is given to the channel that communicates with it. Thereafter the file is referred to by that number: it is the file’s handle. A number once assigned to a file should not be assigned to any other, until the file has been closed and the association destroyed. The function `FreeFile` returns an available reference number.

**9.3.2 Functions for reading and writing**

There are seven VBA read and write functions. Six are listed in Table 9.7.

A seventh function, `Input` (without the #), is used to read in sequentially a fixed number of raw characters, returning a String. We do not use it. There is another function, `Format`, that converts text and numerical expressions into formatted Strings. Although not a file function *per se* it is potentially useful in this context, but it is a very complicated, old style, function.

Reading and writing functions come in pairs; files written with one function should be read in with the corresponding paired function. Which pair you choose depends upon the access mode you specify, how the file is formatted (for instance comma or tab delimited) and what you need from the file.

**Tab and space-delimited sequential files**

`Print #` sends values to file, inserting spaces between successive values. Data written by `Print #` should be read in with `Line Input #`. This reads in an entire line (delimited with `Chr(13)`)) from a sequential file (Opened with `For Input`), returning a String or Variant. It can be also be used to read files written using `Write #`.

**Comma-delimited sequential files**

Write these with `Write #` and read with `Input #`. `Write #` writes an entire line at a time, inserting commas between successive values (and does other formatting too) so that the file it writes can be read correctly by `Input #`. 
Input # expects comma-delimited input. It casts its input, but it is not able to correctly convert Dates. Dates must read in as Strings and then converted to Dates using ordinary VBA library functions. Input # requires the data it reads in to be in exactly the format it expects, otherwise it throws.

Random access files

You must use Put and Get with random access files. They are set up to read and write a single variable at a time, but that variable can be an instance of a UDT. It is usual to devise a UDT just in order to be able to Put it and Get it from file. Put writes data in a structured format, and in particular writes UDTs correctly (although care needs to be taken with UDTs containing variable length Strings). Get is set up to read in records written to file in the format used by Put, so use Get to read data written using Put.

Random access files are not portable. Get can read them but other applications cannot easily do so. To exchange data with other applications it is better to stick with boring but portable TextStream files.

9.4 EXAMPLE: READING AND WRITING TO SEQUENTIAL AND RANDOM FILES

We give an example to illustrate the use of VB intrinsic functions. Data is held in a comma-delimited file, ukmm.csv. The first few lines of this file are shown in Figure 9.3. The figure also shows the layout of lines from a tab-delimited version, ukmm.dat. Each line holds a date followed by seven Doubles.

The example is in the workbook 1_File.xls. ukmm.csv is input sequentially into VBA, then written out into a random access file. Lastly, it is read back in from the random access file and a few records written to the front-end.

It is convenient to define a Type, Daily_record,

Private Type Daily_record 'A user defined type
    record_date As Date
    m3 As Double
    '...five Double items omitted
    y10 As Double
End Type ',

(9.3)

to hold the data stored in a single record (line) of ukmm.csv. Using Daily_record one can write much cleaner code, much less error prone, exploiting the power of the Put-Get functions.

Figure 9.3  ukmm.csv and ukmm.dat: record layout
Public Sub main()

On Error GoTo error_label

Dim Input_file_name As String: Input_file_name = GetInputPath()
Dim Output_file_name As String: Output_file_name = GetOutputPath()

Dim RateData() As Daily_record
Call ReadInFromFile(Input_file_name, RateData) 'fills up RateData

Call CheckIfExists(Output_file_name) 'Does the file exist?
Call WriteToRandom(Output_file_name, RateData) 'writes out RateData

Dim MoreData() As Daily_record
Dim N_recs As Long: N_recs = 10 'Reads in N_recs
Call ReadFromRandom(Output_file_name, N_recs, MoreData) 'fills up MoreData
Call WriteToFrontEnd(MoreData) 'writes out MoreData

Exit Sub

error_label:
ErrorHandler
End Sub

Private Function GetInputPath()
Dim loc As String: loc = Cells(9, 5).Value
Select Case loc
  Case "h": GetInputPath = "D:\ukmm.dat"
  Case "f": GetInputPath = Cells(10, 5).Value
  Case Else: Call RaiseError(12, "GetInputPath", "Bad character")
End Select
End Function

Private Function GetOutputPath()
Dim loc As String: loc = Cells(9, 5).Value
Select Case loc
  Case "h": GetOutputPath = "D:\ukmm.rdn"
  Case "f": GetOutputPath = Cells(11, 5).Value
  Case Else: Call RaiseError(12, "GetOutputPath", "Bad character")
End Select
End Function

Figure 9.4  File I/O: a test harness

main() is given in Figure 9.4. Paths to the input and output files are returned by the functions GetInputPath() and GetOutputPath(). If the user has specified hard-wired paths then default values are used, otherwise strings specifying the input and output paths are read in from the front-end.3

An array of Daily_record, RateData, is passed ByRef to the function ReadInFromFile(), which fills it with data read in from the input file. After checking if the output file already exists with CheckIfExists(), WriteToRandom() writes RateData out to a random access file. ReadFromRandom() then reads a small set of records back in again. Finally WriteToFrontEnd() writes them to the front-end.

3 Readers of this book need to ensure, here and elsewhere, that file locations are correctly specified.
Private Sub ReadInFromFile(file_name As String, ByRef R_data() As Daily_record)
On Error GoTo error_label
Dim fnum As Long: fnum = FreeFile() 'a
Open file_name For Input As #fnum 'b
ReDim R_data(0 To 0) As Daily_record 'c
Dim rec As Daily_record
Do Until EOF(fnum)
  Call GetRecord(fnum, rec) 'd
  R_data(UBound(R_data)) = rec 'e
  ReDim Preserve R_data(LBound(R_data) To UBound(R_data) + 1) 'f
Loop
ReDim Preserve R_data(UBound(R_data) - 1) 'g
Close fnum 'h
Exit Sub
error_label:
Close fnum
Call RaiseError(1111, "ReadInFromFile()", "error")
End Sub

Private Sub GetRecord(fnum As Long, ByRef rec As Daily_record)
On Error GoTo error_label
Dim date_string As String
Input #fnum, date_string, rec.m3, rec.m6, rec.y1, rec.y2, rec.y3, rec.y5, rec.y10
rec.record_date = CDate(date_string) 'Type conversion to date
Exit Sub
error_label:
Call RaiseError(1111, "GetRecord()", "error")
End Sub

Figure 9.5 Inputting from a comma-delimited sequential file

9.4.1 Reading from a sequential file

Figure 9.5 shows the function ReadInFromFile() and its helper function GetRecord(). The entire body of ReadInFromFile() (and GetRecord()) is contained within an error-trapping block. If an error is detected it is thrown and control passes to the error label where it is re-thrown. The error is eventually passed to main()'s ErrorHandler procedure.4

ReadInFromFile() obtains a handle, fnum, from FreeFile() on line 9.5a. The file is opened and associated with the handle on line 9.5b, and referred to by that handle number thereafter.

The Open statement is

\[
\text{Open file_name For Input As #fnum '}. \quad (9.4)
\]

This form of the Open statement opens the file denoted by file_name for sequential input; in this form the statement throws if the file does not exist.

A standard idiom is used to read in data into the array R_data. R_data is ReDimed on line 9.5c to have a single (empty) record at index position 0. The procedure then reads in from file in a Do loop that terminates when EOF is reached. When a new record is provided by GetRecord() it is inserted into the last, empty, slot in R_data. R_data is then ReDim Preserveed (line 9.5e) to have an empty slot

---

4The error-handling implemented here is crude; it is an illustrative placeholder only.
appended on to the end. When EOF is hit, the final, unused, empty slot is removed on line 9.5g. If there are no records in the file, line 9.5g throws.

GetRecord() inputs with the line

\[ \text{Input } \# \text{fnum, date_string, rec.m3, rec.m6, } _{\text{ rec.y1, rec.y2, rec.y3, rec.y5, rec.y10 }'.} \quad (9.5) \]

Input # assumes that fields in the input file are comma delimited. It expects to be able to read in and cast into the types given in its argument list. If it cannot cast as expected, then it throws. It is therefore extremely important that the input file contains no invalid formatting.

Since Input # cannot format dates, the date field is input as a String and then converted by hand to a Date using CDate().\(^5\)

GetRecord() works for comma-delimited records. What if the records arrive tab or space delimited? It is no longer possible to read in using Input #; it is now necessary to read in an entire line using Line Input # and parse it. Figure 9.6 shows how this is done in the function GetRecord2(). On line 9.6a an entire line of the input file is read in by Line Input # and put into the String l_str. The VBA function Split() splits Strings into sub-strings determined by a delimiting character supplied as an argument, here the tab character vbTab. The result is put into a String array sp_str. Each String in sp_str is then converted by hand into the type expected by the items in the Daily_record rec.

As ever, if there is any invalid formatting in the input file the process stops working.

### 9.4.2 Outputting to a random access file

The next section of main() outputs to a random access file. The procedures CheckIfExists() and WriteToRandom() are given in Figure 9.7. The lines

\[ \text{Dim f_name As String: f_name = Dir(Output_fname) 'a} \]
\[ \text{If f_name = "" Then Exit Sub 'b} \]

test if the file specified by Output_fname exists (with default attributes). Dir() returns an empty String if the file does not exist, otherwise it returns the file name. What you do if you find the file you want

\[ \text{Dim l_str As String: Line Input } \# \text{fnum, l_str 'a} \]
\[ \text{Dim sp_str() As String: sp_str = Split(l_str, vbTab) 'b} \]
\[ \text{Dim lb As Long: lb = LBound(sp_str)} \]
\[ \text{Dim ub As Long: ub = UBound(sp_str)} \]
\[ \text{If 1 + ub - lb <> 8 Then Call RaiseError(1111, "GetRecord2()", "error") 'c} \]
\[ \text{rec.record_date = CDate(sp_str(lb)) 'd} \]
\[ \text{rec.m3 = CDbl(sp_str(lb + 1)) '...five lines omitted} \]
\[ \text{rec.y10 = CDbl(sp_str(lb + 7))} \]

\[ \text{Exit Sub} \]
\[ \text{error_label:} \]
\[ \text{Call RaiseError(1111, "GetRecord2()", "error")} \]
\[ \text{End Sub} \]

\[ \text{Figure 9.6 Inputting from a tab-delimited sequential file} \]

\[ ^5 \text{This may be the only occasion in the book that the Date type is used.} \]
Input and Output to File in VBA

149

'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
'XX
CheckIfExists
'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Private Sub CheckIfExists(Output_file_name As String)
On Error GoTo error_label
Dim f_name As String: f_name = Dir(Output_fname)
If f_name = "" Then Exit Sub

'a
'b

Dim res As Integer: res = MsgBox("Output file exits. Delete?", vbYesNo)
Select Case res
Case vbYes: Call Kill(Output_file_name)
Case vbNo: Call RaiseError(1111, "CheckIfExists()", "exists")
End Select
Exit Sub
error_label:
Call RaiseError(1111, "CheckIfExists()", "error")
End Sub
'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
'XX
WriteToRandom
'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Private Sub WriteToRandom(Output_fname As String, ByRef R_data() As Daily_record)
On Error GoTo error_label
Dim fnum As Long:

fnum = FreeFile()

Dim lb As Long:
Dim ub As Long:

lb = LBound(R_data)
ub = UBound(R_data)

Open Output_fname For Random As #fnum Len = Len(R_data(lb))
Dim i As Long
For i = lb To ub
Put #fnum, , R_data(i)
Next i
Close fnum

'a

'b

'c
'd

Exit Sub
error_label:
Close fnum
Call RaiseError(1111, "WriteToRandom()", "error")
End Sub

Figure 9.7 Outputting to a random access ﬁle

to write to already exists depends on you and the requirements of your application. Here the procedure
throws if the client does not want to delete it. Alternatively a line like (9.7)
If File_name <> Dir(Path_name) Then Kill File_name

(9.7)

would quietly delete a pre-existing ﬁle.6
In the procedure WriteToRandom() the output ﬁle Output_fname is Opened For Random in random
access mode with the statement
Open Output_fname For Random As #fnum Len = Len(R_data(lb)) '.

(9.8)

This form of Open statement creates the ﬁle if it does not already exist.
In this mode I/O is record by record, and Open needs to know how big a record is. This is done in
line (9.8) by setting the Len clause. Since Daily_records are being written, the Open statement sets the
Len clause to the size of a Daily_record, returned by the Len() function. Len() returns the number
of bytes required to store its argument, but if its argument is a String it returns the number of characters
in it. If its argument is a UDT containing variable length Strings, then the value returned by Len can
only be used with care.
6

Kill deletes the named ﬁle but throws an error if the ﬁle is open or is read-only.


Implementing Models of Financial Derivatives

The actual writing is done by the Put statement

\[
\text{Put } \#\text{fnum}, , \text{R_data(i)} ' . \tag{9.9}
\]

Despite being Opened in random access mode, the Put statement in line (9.9) is effectively outputting sequentially: there is a missing optional argument that would specify the record number where Put would begin writing.

9.4.3 Inputting from a random access file

The final section of main() reads in from a file written to in random access mode. The procedure ReadFromRandom() is given in Figure 9.8. It reads in the first \( N \) records from the file file_name using Get.\(^7\) The file is Opened on line 9.8b with

\[
\text{Open file_name For Random As } \#\text{fnum Len = sz}' . \tag{9.10}
\]

This opens the file for both reading and writing. The alternative statement

\[
\text{Open file_name For Random Access Read As } \#\text{fnum Len = sz} \tag{9.11}
\]

opens the file for read access only.

As access is random a record length has to be supplied. \( sz \) is the size of a Daily_record, returned by \( \text{Len()} \). On line 9.8c the function \( \text{LOF()} \) returns the length of the file in bytes; dividing through by the size of an individual record gives the number of records in the file.\(^8\)

![Figure 9.8 Inputting from a random access file](image)

\(^7\)Unless there are too few records in the file.

\(^8\)\text{my}_\text{Lmin()} \) is a utility function that returns the minimum of two longs.
Records written to a file with **Put** should be read by the paired input function **Get**. The statement

\[
\text{Get \#fnum, , rec (9.12)}
\]

reads in a record’s worth of data from file (that is, data of length \( sz \)). There is a missing, optional, argument in line (9.12): a record number. When this argument is missing, the next record in sequence is read in. Hence **ReadFromRandom()** is treating the input file as if it were sequential, albeit with a record structure.

### 9.5 SUMMARY

We have investigated the two parallel file systems supported by VBA: **TextStream** files as part of the **FileSystemObject** object, and file handling with VB intrinsic functions. **TextStream** files are convenient but limited in functionality. Intrinsic file handling is flexible and powerful, not particularly elegant, but necessary when you run up against the limitations of **TextStream** files.

In the next chapter we see how the ability of intrinsic functions to manage formatted input can be used to advantage in a Monte Carlo application that reads in successive option specifications from file.

### 9.6 EXERCISES

The first set of exercises are file-handling warm-up exercises. Then you are asked to add in file I/O to the four application exercise streams.

1. A file contains tab-delimited numerical data in a series of rows. Write an application to read in one row of data at a time, compute some basic statistics from the numbers in the row, and output the statistics to another file. For each row you should compute the number of numbers, and their average, maximum, minimum and standard deviation.

2. The file **BadData.csv** is supposed to contain Black–Scholes pricing date in comma-delimited form. Each row should contain

\[
S, r, \sigma, X, T, \text{TypeCode}, \quad (9.13)
\]

in that order, where TypeCode is either the character “c” for a call or “p” for a put. Unfortunately the file is badly formatted. The rows may not contain the data intended and there may be errors in the data.

Write an application to read in from the file (one row at a time), validate the data in the row, compute the Black–Scholes option value and output the value to file along with the input data. If errors are detected the application should behave sensibly. This means: try to keep on going if possible, and produce sensible error messages.

3. Compare the costs of reading and writing to four different file types: (i) a text stream file, (ii) a tab-delimited sequential file, (iii) a comma-delimited sequential file and (iv) a random access file that is read from and written to in sequential order. To perform the benchmark you should create an empty file and then time how long it takes to write \( N \) Doubles to file, and then time how long it takes to read them back in again. Time output and input separately. \( N \) should be large enough so that the operations take several seconds to complete.

What do you conclude from your comparison?
4. **Pi stream.** Modify the pi application to output partial sums to file. If the client requests a value $\pi_N$ of $\pi$ for $N$ terms, output to file the set of pairs $(i, \pi_i)$ for $i = 1, \ldots, N$.

5. **Implied volatility stream.** Modify the implied volatility application to enable it to read in European option pricing data from file. The $i$th row in the input file should contain values for $v_i, X_i, T_i, S_i, r_i, \text{TypeCode}_i$

\[
\begin{align*}
&v_i, X_i, T_i, S_i, r_i, \text{TypeCode}_i \\
&\text{where TypeCode}_i \text{ is either the character “c” for a call or “p” for a put, } v_i \text{ is the market value of the option whose strike is } X_i \text{ and time of maturity is } T_i, \text{ and } S_i \text{ and } r_i \text{ are the asset value and riskless rate.}
\end{align*}
\]

The application should:

(a) Compute and output to file an implied volatility $\sigma_{IV}^i$ for each option individually.

(b) Find a global value of implied volatility, $\sigma_{IV}$, that best fits every option price, perhaps according to a least squares criterion,

\[
\sigma_{IV} = \arg \min_{\sigma} \left\{ \frac{1}{N} \sum_{i=1}^{N} (v_i - \text{BS}(S_i, r_i, \sigma, X_i, T_i, \text{TypeCode}_i))^2 \right\},
\]

where BS is the standard Black–Scholes pricing formula with the obvious interpretation.

Equation (9.15) is a minimization problem. It can be solved using the golden section search method described in Appendix H.

Note that in this case one might wish to check that the options are all written on the same asset so that, for instance, one might require $S_i \equiv S$ and $r_i \equiv r$ for all $i$.

6. **PDE stream.** Modify the application to read in problem specifications from file. Rows in the file are in the form of vectors $S, r, \sigma, X, T, \text{TypeCode}, \text{ExCode}, N, M, S_{\text{max}}, S_{\text{min}}, \text{EvolverCode},$

\[
\begin{align*}
&S, r, \sigma, X, T, \text{TypeCode}, \text{ExCode}, N, M, S_{\text{max}}, S_{\text{min}}, \text{EvolverCode},
\end{align*}
\]

where ExCode is either the character “e” for a European option or “a” for an American option, and EvolverCode is either the character “p” for plain evolution or “s” for SOR evolution. All the other symbols have their usual meaning applied to the Crank–Nicolson method.

The application should read in a specification, compute the option value, output the computed values to file (together with the problem specification), and then continue with the next row, until end of file is hit.

7. **Lattice stream.** Modify the application to read in problem specifications from file. Each row of the file should be a vector with the structure

\[
\begin{align*}
&S, r, \sigma, X, T, \text{TypeCode}, \text{ExCode}, N, \text{LatticeCode}, \text{SliceCode},
\end{align*}
\]

where ExCode is as in exercise 6, LatticeCode is either the character “3” for a trinomial lattice or “7” for the heptanomial lattice and SliceCode is either the character “u” for a plain unpruned slice or “p” for a pruned slice. All the other symbols have their usual meaning applied to the lattice method.

As in exercise 6 the application should read in a specification, compute the option value, output the computed values to file (together with the problem specification), and then continue with the next row, until the end of file is hit.
10 Valuing a Book of Options

The purpose of this chapter is to construct a Monte Carlo application that can value simultaneously a whole book of option specifications. Option specifications can be input from either file or from a spreadsheet, and valuation output can be to file or to a spreadsheet.

In Part VI this framework needs to be extended to enable control variates and importance sampling to be included as part of the option specification, and other speed-ups can also be used.

A number of features are introduced that at this stage are not strictly required. An IOptionWrapper object, for instance, wraps each option object. These features are place-holders for significantly more complex structures that can follow later.

There are three contributions from this chapter. The first is that it builds on Chapter 9 by illustrating some chunky I/O to and from file, in a polymorphic setting; the second is in the design modifications needed to allow the application to value an option book; the third is the introduction of a path object to encapsulate the concept of a path (in the path-wise Monte Carlo used here).

10.1 OUTLINE OF THE APPLICATION

The new application is MC_example_v5b.xls. The overall structure is very similar to MC_example_v5 from Chapter 7. There are three main changes to the application.

To allow the application to value a book the single, simple, option object is replaced by an array of option objects (each wrapped inside an IOptionWrapper object). This is kept by an options specification manager object that looks after it and keeps it organized. The single accumulator object also is replaced by an array.

I/O is expanded considerably. The main I/O objects from MC_example_v5, the polymorphic IOutputter and IInputter objects, are replaced by non-polymorphic wrapper objects containing polymorphic composited I/O objects. We get a stable non-polymorphic interface, but with polymorphism supplied by the composite objects.

Finally, by introducing a path object a great deal of tidying away and conceptual decoupling can be achieved. The path object presented here is quite basic but it is the natural home for some fairly radical potential future expansion.

There are also some more minor changes: the factory has to be modified somewhat to deal with the new structures, and the method is path-wise rather than slice-wise.

The application uses a set of enumerative constants and UDTs to specify certain aspects of the application, defined in the code module TypesModule. They are listed in Figure 10.1 together with a number of helper functions. There are UDTs for option specifications and for process specifications, and there are Enums for the payoff type, and vestigial Enums, containing a single substantive Enum constant, for the Wiener object, process object and accumulator object types.

Using UDTs can be useful, convenient, safe and beneficial, but using Enums in this manner is quite poor. It betrays a procedural mindset and code that is bound to be incredibly heavily coupled. The Enums are designed to facilitate the use of Select statement in the factory. If you must use Select then Enums are very handy, but if the use of Select is itself misguided, a poor design feature, then Enums are a backwards improvement.
Public Enum PayType 'Enumerates payoff types
    voidtype 'no type
    calltype 'European call
    puttype 'European put
    stocktype 'stock
End Enum

Public Enum WieType 'Enumerates Wiener generator types
    voidtype 'no type
    plain_Wiener 'plain
End Enum

Public Enum ProcType 'Enumerates process types
    voidtype 'no type
    gbm 'GBM
End Enum

Public Enum acctype 'Enumerates accumulator types
    voidtype 'no type
    plain_acc 'Plain accumulator
End Enum

Public Type ProcSpec 'Process specification
    p_type As ProcType 'Process type
    S0 As Double'Initial stock price
    r As Double 'Stock short rate
    sig As Double 'Stock volatility
End Type

Public Type OptSpec 'Option specification
    o_type As PayType 'Payoff type
    T As Double'Final time
    X As Double 'Strike
End Type

Public Function StrgToPayType(inp As String) As PayType
    Select Case inp
        Case "": StrgToPayType = PayType.voidtype
        Case "call": StrgToPayType = calltype
        Case "put": StrgToPayType = puttype
        Case "stock": StrgToPayType = stocktype
        Case Else: Call RaiseError(26, "StrgToPayType", "Bad PayType")
    End Select
End Function

Public Function StrgToWieType(inp As String) As WieType
    Select Case inp
        Case "": StrgToWieType = WieType.voidtype
        Case "p": StrgToWieType = plain_Wiener
        Case Else: Call RaiseError(26, "StrgToWieType", "Bad WieType")
    End Select
End Function

Public Function StrgToProcType(inp As String) As ProcType
    Select Case inp
        Case "": StrgToProcType = ProcType.voidtype
        Case "g": StrgToProcType = gbm
        Case Else: Call RaiseError(26, "StrgToProcType", "Bad ProcType")
    End Select
End Function

Public Function StrgToAccType(inp As String) As AccType
    Select Case inp
        Case "": StrgToAccType = AccType.voidtype
        Case "p": StrgToAccType = plain_acc
        Case Else: Call RaiseError(26, "StrgToAccType", "Bad AccType")
    End Select
End Function

Figure 10.1  Enumerative constants and UDTs
They really should be avoided. At the moment though we do not yet know how to set up a polymorphic factory – this happens only in Part IV – so we proceed, regarding this as an exercise in programming technique, deprecated or not.

The helper functions make things worse. They hard-wire in some identifier Strings. Using them will involve telepathy.

Even the fact of employing a UDT can be dangerous. It explicitly or, worse, implicitly limits what the application can do. When this a conscious design choice it is fine, but even so great care must be taken.

Consider the option specification, OptSpec (Figure 10.1). An option is to be specified by a maturity time, and strike and a payoff type, for that is what the OptSpec allows. This is a highly constrained specification; it is too narrow to encompass average rate or barrier or, unambiguously, American options. Clearly OptSpec could be extended very easily, and later we do away with the current limitation, but this has an ad hoc feel to it. What is really needed is a deeper concept of what an option is – but this would take us too far from the current topic.

For the time being we consider only GBM for the underlying process; the ProcSpec Type contains items for the initial asset value \( S_0 \), the short rate \( r \), and the asset returns’ volatility \( \sigma \), as well as a ProcType Enum. It is quite adequate for us for the time being.

The reader will have noted and wondered about the existence of a voidtype Enum constant. Its presence is a confession that the data structure is not fully thought through. In fact as the application stands, voidtype could (should) just be removed from the types where it occurs.

Why is it there? Because later on options are given the opportunity to specify a control variate (see Chapter 20). If they have no control variate to specify then the control variate payoff type (for instance) is set to voidtype. Of course, this is a very weak design; at best it should be regarded as a real-world make-do solution.1

First we look at changes to the valuation side of the application, and then at I/O.

### 10.1.1 The valuation side

The invocation chain is standard. However, instead of an IOutputter object, AppObWrapper has a composite OutputManager object. This holds a composite IOutputOption object that outputs results polymorphically. Output is discussed in section 10.1.2 below.

The AppMC object is shown in Figure 10.2. It is similar in overall form to the MC_example_v5 version of AppMC (page 126) but its composite objects differ in their implementations and functionality. The previous version held two Doubles as data members, one for the option value and the other for its standard error. The new version replaces these by arrays of Doubles, Ovalues(_) and SEvalues(_), which have entries for each option in the book. It has a composite object, pth_, and instead of a single IOption object it has an IOptionSpecs object to manage a set of option objects.

Inside IOptionSpecs each IOption object is contained in an IOptionWrapper object. The set of IOptionWrapper objects are contained in an array. IOptionSpecs also holds an array of corresponding accumulator objects.

The program flow through the AppMC::run() method is shown in Figure 10.3. The arrows show the direction of flow as data is computed and passed on. The path object, pth_, looks after individual paths. An IEvolver object, evl_, generates sample paths for the underlying state variable (here, as usual, just a GBM). On each iteration of the main loop in AppMC::run(), pth_ is passed on to evl_. This sends it to its composited Wiener generator object and then on to the path generator. pth_ is then sent to the options object, specs_, conforming to IOptionSpecs, where it is handed to each IOptionWrapper object in

---

1 Otherwise known as a bodge that will be hugely expensive to fix when the time comes.
Implementing Models of Financial Derivatives

' AppMC
' data section

Private pth_ As IPath 'Wraps a path
Private evl_ As IEvolver 'Generates a path
Private specs_ As IOptionSpecs 'The book of option specifications
Private M_ As Long 'number of paths
Private mon_ As Monitor 'Prints a counter
Private Ovalues_() As Double 'Option values
Private SEvalues_() As Double 'se values

Private Sub Class_Terminate()
    Set evl_ = Nothing
    Set pth_ = Nothing
    Set specs_ = Nothing
    Set mon_ = Nothing
End Sub

Friend Property Let IApp_SetValues(fact As Factory)
    Set evl_ = fact.Evolver
    Set specs_ = fact.OptionSpecs
    Set pth_ = fact.Path
    Set mon_ = fact.GetMonitor
    M_ = fact.M
End Property

Friend Sub IApp_OValues(ByRef a() As Double): a = Ovalues_: End Sub
Friend Sub IApp_SEValues(ByRef a() As Double): a = SEvalues_: End Sub

Private Sub reset(): End Sub 'just a stub
Private Sub register()
    Call specs_.OValues(Ovalues_)
    Call specs_.SEValues(SEvalues_)
End Sub

Figure 10.2 The application object, AppMC

turn. These pass it to their compositive IOption objects. These compute payoff values that are immediately given to the corresponding accumulator object. After $M_\text{ paths}$ have been generated and processed, $\text{specs}_\text{ }$ is asked for its results.

We discuss each object in more detail. We look first at the path object, then at the IEvolver object, and finally at the IOptionSpecs object.
The IPath object

The PathPlain object, conforming to the IPath interface, looks after paths. The IPath interface is shown in Figure 10.4. Path objects contain both a path of a Wiener process and a path of the underlying constructed from them. There are getters for both the array of Wiener values and for the sample path itself. The setting is done by the setters, SetWiener() and SetPath(). The ApplyPayoff() method takes a payoff object as an argument. It allows the payoff to evaluate itself against the path the object contains.

PathPlain is shown in Figure 10.5. wien_ and path_ are the paths of the Wiener process and the underlying respectively. The path object assumes that values along the paths are at equally spaced points in time, starting from time 0, index 0, up to time Tmax_, index N_. The Private method IndexOfT() is provided to convert times into index values. Payoff objects can access indices through the ApplyPayoff() method.

PathPlain uses a standard technique to limit access to its data members. The Wiener generator, for instance, gains access to wien_ only by being explicitly passed it by PathPlain itself. We discuss this a little further below.

The path object is the natural home for methods that compute path statistics, such as average values, maximums, et cetera, but we do not add these in yet.

```plaintext
 Public Property Let SetValues(fact As Factory): End Property
 Public Sub SetWiener(wie As IWienerGenerator): End Sub
 Public Sub GetWiener(ByRef wie() As Double): End Sub
 Public Sub SetPath(gen As IPathGenerator): End Sub
 Public Sub GetPath(ByRef pth() As Double): End Sub
 Public Function ApplyPayoff(pay As IPayoff, T As Double) As Double: End Function
```

Figure 10.4 The IPath interface
Implementing Models of Financial Derivatives

Evolving sample paths

The IEvolver object is very similar to that seen in Chapter 7. Figure 10.6 shows the new version of the interface, and also changes to the single conforming object, WienerPlain. Essentially the same objects are involved – a Wiener generating object Wie_ and a path generating object gen_ – but their functionality has been extended to include path as well as slice evolution. The interface contains a single new method, GetPath(). (The only other change is due to revisions in the Factory object. We mention these in passing below.) GetPath() asks the Wiener object for a Wiener path which it then passes over to the path generator to construct a path of the underlying in situ.

The IWiener and IGenerator objects

The IWiener interface has a two new procedures, GetWiener() and GetWienerVec(). Their implementations in WienerPlain are shown in Figure 10.7. When WienerPlain is asked for a Wiener sample path, its GetWiener() method is called with a path object as an argument. GetWiener() calls the path’s SetWieners() method passing Me as an argument. Inside SetWieners() (Figure 10.5) the path sends
Figure 10.6 The IEvolver interface and changes to EvolverW

Figure 10.7 WienerPlain: additional methods

back to the Wiener generator an array for it to fill with a Wiener sample path. The GetWienerVec() method in WienerPlain finally does the standard boring evolution. In this way the path object controls what may or may not set its data members, limiting access to IPath objects.

Similarly the IGenerator interface has two additional methods: GetPath() and GetPathVec(). Their implementations in GeneratorGBM are shown in Figure 10.8.

The pattern is the same as for WienerPlain. When a GBM sample path is requested, the GetPath() method calls the path’s SetPath() method which just calls the GBM generator straight back, passing it a Wiener sample path and an array to fill with the GBM sample path. As before the purpose is to enable the path object to control who does what to its Private data members, here restricting access to IPath objects.

The options book and its valuation

In Figure 10.2, in AppMC::run(), the line

Call evl_.GetPath(pth_)  

(10.1)
asks the evolver to construct a path in the path object \( \text{pth}_- \). The next line,

\[
\text{Call specs_.ReceivePath(pth_)}
\]

(10.2)

hands the path over to the option specifications object where the second half of the Monte Carlo valuation takes place.

The IOptionSpecs interface is an addition in this version of the application. Before, when there was just a single option, it was reasonable to let the option look after all of its administration. Now that there is a set of options they need to be marshalled, hence the IOptionSpecs object.

\( \text{specs}_- \) contains arrays of option objects and their accumulators. Every time it is given a path it hands it over to each of its option objects, receiving back an option value. This it passes over to the corresponding accumulator object.

The \( \text{specs}_- \) object conforms to the IOptionSpecs interface (Figure 10.9). It has two getters, OValues() and SEValues(), that return arrays of option values and their standard errors, and a Sub, ReceivePath(), that accepts a path and hands it on to each option object it looks after.

There is just a single object of this type, OptionSpecsPlain, shown in Figure 10.10. It maintains two arrays of composited objects. \( \text{opt}_- \) is an array of IOptionWrapper conforming objects that contain the options to be valued; \( \text{acc}_- \) is an array of IAccumulator objects that hold the corresponding accumulators. Each IOptionWrapper object wraps a single option object. It can also contain additional functionality that facilitates the use of speed-ups.

Every time a path is received by ReceivePath() it is passed to the ComputePayoff() method of each IOptionWrapper object in turn. This computes the present value of its option payoff along the particular path. The value returned by it is immediately handed over to the corresponding accumulator object.
Valuing a Book of Options

'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
'OptionSpecsPlain
'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

Implements IOptionSpecs

Private opt_() As IOptionWrapper 'Array of option wrappers
Private acc_() As IAccumulator 'accumulators
Private Nprobs_ As Long 'number of options

'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

Private Sub Class_Terminate()
Dim i As Long
For i = 1 To Nprobs_
    Set opt_(i) = Nothing
    Set acc_(i) = Nothing
Next i
End Sub

'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

Friend Property Let IOptionSpecs_SetValues(fact As Factory)
    Nprobs_ = fact.Nprobs 'a
    ReDim opt_(1 To Nprobs_) As IOptionWrapper 'options
    ReDim acc_(1 To Nprobs_) As IAccumulator 'accumulators
    Dim i As Long
    For i = 1 To Nprobs_
        Dim spec As OptSpec: spec = fact.spec(i) 'b
        Set opt_(i) = fact.OptionWrapper(spec) 'c
        Set acc_(i) = fact.Accumulator(opt_(i).acctype) 'd
        acc_(i).SetUp = opt_(i)
    Next i
End Property

'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

Friend Sub IOptionSpecs_OValues(ByRef ovals() As Double)
    ReDim ovals(1 To Nprobs_) As Double
    Dim i As Long
    For i = 1 To Nprobs_
        ovals(i) = acc_(i).optvalue
    Next i
End Sub

Friend Sub IOptionSpecs_SEValues(ByRef sevals() As Double)
    ReDim sevals(1 To Nprobs_) As Double
    Dim i As Long
    For i = 1 To Nprobs_
        sevals(i) = acc_(i).se
    Next i
End Sub

'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

Friend Sub IOptionSpecs_ReceivePath(ByRef pth As IPath)
    Dim i As Long
    For i = 1 To Nprobs_
        acc_(i).update = opt_(i).ComputePayoff(pth)
    Next i
End Sub

'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

Figure 10.10 The option specifications object, OptionSpecsPlain
The process of option instantiation

opt_ and acc_ are constructed in OptionSpecsPlain::SetValues(). The Factory is asked for the number of options in the book (line 10.10a), and then, one by one, the Factory is asked for their specifications. On line 10.10c the Factory is passed the specification and is asked to construct a IOptionWrapper object for the option.

The accumulator type is allowed to vary polymorphically (and does so from a choice of one). The IOptionWrapper object is asked for the type of accumulator it needs. It returns a type specifier that is given to the Factory to construct and return the required accumulator (line 10.10d).

The ability to select an accumulator type is not exploited in this version of the application. Later on, when control variates are employed, an alternative accumulator object will be needed. At that stage the ability to select the appropriate accumulator object will be needed and a structure similar to this can be employed.

The request to create an option triggers a chain of activity. This is sketched in Figure 10.11. When the Factory is asked for an option spec it forwards the requests to the InputManager object that passes it on in turn to the Inputter object which has previously read in the entire set of options specs. It returns a spec back up the chain to OptionSpecsPlain.

The statement

\[
\text{Set } \text{opt(}_i\text{)} = \text{fact}.\text{OptionWrapper}(\text{spec}) \quad (10.3)
\]

asks the Factory to construct an IOptionWrapper object. The chain of calls it initiates is shown in lines (10.4).
and illustrated in Figure 10.11. At each step newly created objects are passed the Factory (and the spec) to allow them to set parameter values and to create any further composite objects that might be required. This can be followed through as we later examine some of the objects involved.

The accumulator is standard and is not displayed. It has an additional method, SetUp(), taking an IOptionWrapper as an argument, but this is a stub and is not used for the moment.

**The IOptionWrapper object**

The IOptionWrapper object, whose interface is shown in Figure 10.12, wraps options with additional functionality. As used here, it is a placeholder; it does almost nothing. Later when we look at speed-up methods we discover a need to extend the basic design. At that stage an IOptionWrapper could be used to contain, in addition to the option object, a method object that manages the speed-ups particular to that option; it will then have some real work to do.

The single conforming object is OptionWrapperPlain (Figure 10.13). The only thing that we allow the IOptionWrapper object to do here is to return a specifier for the accumulator object. Since there is currently only one accumulator object – AccumulatorPlain – this is at the moment a redundant function; the return value is hard-wired in. When later in the book we implement control variates a special accumulator is needed. The wrapper interrogates the method object to find out which sort of accumulator is required and passes this on.

### 10.1.2 The input and output side

There are changes not only to the input and output objects, but also, since in this design all input is mediated through the Factory, some small changes to that as well. We look first at input, then at output, then finally at the Factory.
Option specifications are stored on file or on a spreadsheet. The InputManager object, kept by the factory as a composite data member, inputs both parameter values and object specifications codes. It is shown in Figure 10.14. It has a compositied IInputOption object that is responsible for inputting option specifications; all other input comes from the front-end.

InputManager is directly responsible for inputting all non-option parameter values and object specifications. These are the process specification, the Monte Carlo parameters $N_{\text{probs}}$, $M_{\text{probs}}$, and $T_{\text{max}}$, and the Monitor output interval $N_{\text{c}}$. These are read in as usual when InputManager is instantiated.

Since a separate object is responsible for inputting option specifications, why not also have separate objects to look after the input of Monte Carlo and process specifications? This is certainly possible, and possibly sensible. We return to this idea later.

get_p_spec() reads in a process specification, of Enum type ProcType, from the front-end, from a hard-wired in location. get_p_type() and get_w_type() read in ID characters from the front-end and return Enums for the desired objects. They use the helper functions in the TypesModule code module to convert input Strings into Enums. This seems unnecessary; future designs will just pass the character on to the Factory and do away with these anti-OOP Enums.

The polymorphic inputter object, odat_, is set in the SetValues() method as usual. Only after SetValues() is run can items from odat_ be retrieved. In the factory, when the InputManager is instantiated, its SetValues() method is called on the same line so that odat_ is immediately instantiated.

The IInputOption objects read in option specifications. They construct an array of option specifications, specs(), with entries of type OptSpec. Their Property Get spec(i) returns the i-th specification (for i in a valid range). The two inputters described here each input every specification all at once when they are instantiated, but there is no reason why option specifications cannot be obtained by other means, perhaps from a form, or even from a console, only when requested.

The IInputOption interface is shown in Figure 10.15. It has three Properties. As well as spec() there is Nprobs() that returns the number of option specifications that have been read in, and SetValues() that enables an Environment object (see below) to pass a file path specification to the inputter if required.
Private opts_() As OptSpec 'Option specification
Private N_specs_ As Long 'Number of option specs
Private proc_ As ProcSpec 'Specification of the model
Private w_type_ As WieType 'specification of Wiener type
Private N_ As Long 'Number of time steps
Private M_ As Long 'Number of sample paths
Private Tmax_ As Double 'Maximum option maturity time
Private NC_ As Long 'output interval for Monitor
Private odat_ As IInputOption 'The inputter object
Private sht_ As String 'Name of front end input sheet

Private Sub Class_Initialize()
    sht_ = "FrontEnd" 'Hard-wired in
    Call Read_in_data
End Sub

Private Sub Class_Terminate()
    Set odat_ = Nothing
End Sub

Friend Property Get proc() As ProcSpec: proc = proc_: End Property
Friend Property Get wtype() As WieType: wtype = w_type_: End Property
Friend Property Get Nprobs() As Long: Nprobs = odat_.Nprobs: End Property
Friend Property Get N() As Long: N = N_: End Property
Friend Property Get M() As Long: M = M_: End Property
Friend Property Get Tmax() As Double: Tmax = Tmax_: End Property
Friend Property Get NC() As Long: NC = NC_: End Property
Friend Property Get spec(i As Long) As OptSpec
    spec = odat_.spec(i)
End Property

Private Sub Read_in_data()
    proc_ = get_p_spec(sht_, 13, 7) 'Process type
    w_type_ = Get_w_type(sht_, 19, 7) 'Wiener type
    N_ = Check_strictly_positive(get_long_wk(sht_, 13, 4), sht_ & ": N")
    M_ = Check_strictly_positive(get_long_wk(sht_, 14, 4), sht_ & ": M")
    Tmax_ = Check_strictly_positive(get_double_wk(sht_, 18, 4), sht_ & ": Tmax")
    NC_ = Check_strictly_positive(get_long_wk(sht_, 15, 4), sht_ & ": NC")
End Sub

Private Function get_p_spec(wk As String, X As Long, y As Long) As ProcSpec
    get_p_spec.p_type = Get_p_type(wk, X, y)
    get_p_spec.S0 = Check_strictly_positive(get_double_wk(wk, X + 1, y), wk & ": S0")
    get_p_spec.r = Check_strictly_positive(get_double_wk(wk, X + 2, y), wk & ": r")
    get_p_spec.sig = Check_strictly_positive(get_double_wk(wk, X + 3, y), wk & ": sig")
End Function

Private Function Get_p_type(wk As String, X As Long, y As Long) As ProcType
    Dim inp As String: inp = Worksheets(wk).Cells(X, y).value
    Get_p_type = LibTypes.StrgToProcType(inp)
End Function

Private Function Get_w_type(wk As String, X As Long, y As Long) As WieType
    Dim inp As String: inp = Worksheets(wk).Cells(X, y).value
    Get_w_type = LibTypes.StrgToWieType(inp)
End Function

Figure 10.14 The InputManager object
The sources of input for option specifications implemented here are from file or from spreadsheet:

*From file*

This is handled by InputOptionFile, Figure 10.16. The main Sub is Read_in_data(). It reads in comma-delimited data from file using VB intrinsic file functions (see Chapter 9).

The file location is obtained from the Environment object in SetValues(), setting the value of the path variable, o_path_. The Sub GetRecord() gets each record in turn from file. The name of the option payoff object is read in as a String. A TypesModule helper function converts the name to a PayType Enum. Telepathy guarantees that the file contains the same names as the helper function expects.

*From spreadsheet*

Input from a spreadsheet is done by InputOptionFE, shown in Figure 10.17. InputOptionFE expects to find specifications in a block on the sheet sht_ of ThisWorkbook, whose top left-hand corner is at location (x_, y_). If the data is not there, or not formatted as it expects, the object throws.

The procedure EndOfInput() tests if there is any more input. It peeks at the next input row. If this starts with a blank cell it assumes that it has reached the end of the input block.

Both InputOptionFile and InputOptionFE use the standard ReDim Preserve idiom to construct the array specs_() of option specifications. If there are no specifications to be input, and one of the methods EOF() or EndOfInput() immediately encounters no-more-data, N_specs_ has the value 0. Then the line

\[ \text{ReDim Preserve specs_(1 To N_specs_)} \]

(10.5)

throws an error. We do nothing about this, but really the error should be trapped and a useful message sent to the client as a reminder to let the application have some data, in the right location.

What happens if a new option is introduced, with a new payoff function? If no additional parameters are involved then just three things have to be done. First, the class module for the new payoff has to be created and added into the project; second, the payoff Enum and the helper function have to be extended; and third, the factory has to have its Payoff() method extended. That is all; in particular the input objects do not have to be altered. These changes might not seem too much but the last two jobs are unnecessary and with a polymorphic factory can be got rid of.

It is a different matter if the new payoff requires additional parameters. Then the OptSpec UDT has to be altered to include the new parameter, and every inputter also has to be altered to enable it to input the new values.

*Outputting results*

A chain of requests is initiated by the AppObWrapper object with the line

\[ \text{Call out_.FinalResults(app_, wch_.Elapsed_Time).} \]

(10.6)
Implements IInputOption
Private specs_() As OptSpec 'Option and method specification
Private N_specs_ As Long 'Number of option specs
Private o_path_ As String 'Path to option spec file

Friend Property Let IInputOption_SetValues(env As Environment)
o_path_ = env.i_path 'get input path
Call Read_in_data
End Property

Private specs_() As OptSpec 'Option and method specification
Private N_specs_ As Long 'Number of option specs
Private o_path_ As String 'Path to option spec file

Friend Property Get IInputOption_Nprobs() As Long
IInputOption_Nprobs = N_specs_
End Property

Friend Property Get IInputOption_spec(i As Long) As OptSpec
If Not check_in_range(CDbl(i), 1, N_specs_) Then
Call RaiseError(25, "Inputmanager.spec", "i out of range")
End If
IInputOption_spec = specs_(i)
End Property

Private Sub Read_in_data()
On Error GoTo error_label
Dim fnum As Long: fnum = FreeFile() 'Returns the next free file number
Open o_path_ For Input As #fnum
ReDim specs_(1 To 1) As OptSpec
N_specs_ = 0
Do Until EOF(fnum)
Dim p_spec As OptSpec: Call GetRecord(fnum, p_spec)
N_specs_ = N_specs_ + 1
specs_(N_specs_) = p_spec 'Captures the input
ReDim Preserve specs_(1 To N_specs_ + 1) 'Creates extra place
Loop
ReDim Preserve specs_(1 To N_specs_) 'Breaks if there are no specs
Close fnum
End Sub

Private Sub GetRecord(fnum As Long, ByRef o_spec As OptSpec)
Dim o_strg As String
Input #fnum, o_strg, o_spec.X, o_spec.T
o_spec.o_type = LibTypes.StrgToPayType(o_strg)
Close fnum
End Sub

Private Sub ReadInFromFile()
On Error GoTo error_label
Dim fnum As Long: fnum = FreeFile() 'Returns the next free file number
Open o_path_ For Input As #fnum
ReDim specs_(1 To 1) As OptSpec
N_specs_ = 0
Do Until EOF(fnum)
Dim p_spec As OptSpec: Call GetRecord(fnum, p_spec)
N_specs_ = N_specs_ + 1
specs_(N_specs_) = p_spec 'Captures the input
ReDim Preserve specs_(1 To N_specs_ + 1) 'Creates extra place
Loop
ReDim Preserve specs_(1 To N_specs_) 'Breaks if there are no specs
Close fnum
End Sub

Private Sub GetRecord(fnum As Long, ByRef o_spec As OptSpec)
Dim o_strg As String
Input #fnum, o_strg, o_spec.X, o_spec.T
o_spec.o_type = LibTypes.StrgToPayType(o_strg)
Close fnum
End Sub

Figure 10.16 The InputOptionFile object
Implements IInputOption

Private specs_() As OptSpec 'Option specification
Private N_specs_ As Long 'Number of option specs
Private x_ As Long 'row number of first data row
Private y_ As Long 'col number of first data col
Private sht_ As String 'Name of sheet with option specs

Private Sub Class_Initialize()
    sht_ = "OptionSpecs" 'Hard-wired in
    x_ = 6: y_ = 3 'coord of top left hand corner of input matrix
    Call Read_in_data
End Sub

Friend Property Let IInputOption_SetValues(env As Environment) 'Does nothing
End Property

Friend Property Get IInputOption_Nprobs() As Long
    IInputOption_Nprobs = N_specs_
End Property

Friend Property Get IInputOption_spec(i As Long) As OptSpec
    If Not check_in_range(CDbl(i), 1, N_specs_) Then
        Call RaiseError(25, "Inputmanager.spec", "i out of range")
    End If
    IInputOption_spec = specs_(i)
End Property

Private Sub Read_in_data()
    N_specs_ = 0
    ReDim specs_(1 To 1) As OptSpec
    Do Until EndOfInput(N_specs_)
        Dim spec As OptSpec: Call ReadInASpec(N_specs_, spec)
        N_specs_ = N_specs_ + 1 'increment number of entries
        specs_(N_specs_) = spec 'Captures the input
        ReDim Preserve specs_(1 To N_specs_ + 1) 'Creates extra place
    Loop
    ReDim Preserve specs_(1 To N_specs_) 'Breaks if there are no specs
End Sub

Private Function EndOfInput(i As Long) As Boolean
    EndOfInput = Worksheets(sht_).Cells(x_ + i, y_).value = ""
End Function

Private Sub ReadInASpec(i As Long, ByRef spec As OptSpec)
    spec = get_o_spec(x_ + i, y_)
End Sub

Private Function get_o_spec(x As Long, y As Long) As OptSpec
    get_o_spec.o_type = Get_o_type(x, y)
    get_o_spec.X = Check_positive(get_double_wk(sht_, x, y + 1), CStr(x) & ": X")
    get_o_spec.T = Check_positive(get_double_wk(sht_, x, y + 2), CStr(x) & ": T")
End Function

Private Function Get_o_type(x As Long, y As Long) As PayType
    Dim inp As String: inp = Worksheets(sht_).Cells(x, y).value
    Get_o_type = LibTypes.StrgToPayType(inp)
End Function

Friend Property Get InputOptionFE() As OptSpec
    InputOptionFE = specs_(1)
End Property
The OutputManager object (Figure 10.18) wraps a composite IOutputOption object, referred to as the outputter, that does the actual outputting of valuation results. OutputManager is directly responsible only for outputting the timer to the front-end. The name of the sheet it outputs to is hard-wired in. This is probably not too objectionable, until the client renames the sheet.

As usual, output takes place when the outputter object is destroyed, in its destructor. A better design might call the Write_out_data() method explicitly.

IOutputOption, Figure 10.19, has two interface methods; a SetValues() Property to enable the output file path to be set from an Environment object (see below), and a RegisterValues() Property to pass output values to the outputter. The OutputManager::FinalResults() method calls RegisterValues(), passing the application object to the outputter.

There are two conforming objects. Outputting results to file is performed by the OutputOptionFile object (Figure 10.20). Outputting to a spreadsheet front-end is performed by OutputOptionFE, shown in Figure 10.21.

---

```
Private elapsed_time_ As Double
Private sht_ As String 'Name of front end output sheet
Private o_out_ As IOutputOption

Private Sub Class_Initialize()
    sht_ = "FrontEnd" 'Hard-wired in
End Sub

Private Sub Class_Terminate()
    Call Write_out_data
    Set o_out_ = Nothing
End Sub

Friend Property Let SetValues(fact As Factory)
    Set o_out_ = fact.OutputOption()
End Property

Friend Sub StartRun()
    Worksheets(sht_).Cells(7, 5).value = ""
End Sub

Friend Sub FinalResults(ByRef app As IApp, T As Double)
    o_out_.RegisterValues = app
    elapsed_time_ = T
End Sub

Private Sub Write_out_data()
    Worksheets(sht_).Cells(7, 5).value = elapsed_time_
End Sub
```

Figure 10.18  The OutputManager object

```
Public Property Let RegisterValues(ByRef the_app As IApp): End Property
Public Property Let SetValues(env As Environment): End Property
```

Figure 10.19  The IOutputOption interface

---
Figure 10.20 The OutputOptionFile object

Figure 10.22 sketches what happens when FinalResults() is executed. The outputer asks the application object, AppMC, for its option values and its standard errors, passing arrays c_value_ and s_value_ to it. These are then passed down to the IOptionSpecs object. This asks each accumulator object in turn for its value or standard error. The returned values are passed back to the outputer.

What happens now depends on the IOutputOption conformance object. For OutputOptionFile the main output method is Write_out_data(). This creates and outputs tab-delimited data to a TextStream file (see Chapter 9). Each option has its results output to a separate line in the output file.

WriteLine outputs a String. A sequence of numerical values have to be converted to Strings, with CStr(), and then concatenated before being fed to WriteLine.

When output is to a spreadsheet a similar path is followed. OutputOptionFE sends output to a location on the sheet sht_ whose top-left corner is cell (x_, y_). This is a simple object. As usual the name of the sheet of ThisWorkbook it writes to, and the coordinates of the destination block, are hard-wired in. The code will crash if the name, sht_, becomes invalid.2

2 It would be possible to test in the constructor whether the worksheet sht_ exists, and to throw if it did not. We do not do this here.
Private c_value_() As Double
Private s_error_() As Double
Private x_ As Long 'row number of first output row
Private y_ As Long 'col number of first output col
Private sht_ As String 'Name of sheet for option output

Private Sub Class_Initialize()
    sht_ = "FrontEnd" 'Hard-wired in
    x_ = 14: y_ = 9
    Call ClearColumn(sht_, x_, y_)
    Call ClearColumn(sht_, x_, y_ + 1)
End Sub

Private Sub Class_Terminate()
    Write_out_data
End Sub

Friend Property Let IOutputOption_SetValues(env As Environment) 'Does nothing
End Property

Friend Property Let IOutputOption_RegisterValues(ByRef the_app As IApp)
    Call the_app.OValues(c_value_)
    Call the_app.SEValues(s_error_)
End Property

Private Sub Write_out_data()
    Call WriteColumn(sht_, x_, y_, c_value_)
    Call WriteColumn(sht_, x_, y_ + 1, s_error_)
End Sub

Figure 10.21 The OutputOptionFE object

Figure 10.22 Outputting results: program flow
The Factory object is shown in Figures 10.23 and 10.24. This has developed slightly since Chapter 9 but not much. In any case it will soon be made redundant by the polymorphic Factory coming up in Part IV.

Like the factory in MC_example_v5 it keeps references to input and output objects so that it can hand them out to clients that request them. Now there are five I/O objects. The Factory itself is the sole client of the InputManager, only the AppObWrapper requires the OutputManager, and only AppMC needs the Monitor. (Also, the InputManager and OutputManager are the only objects that need IInputOption and IOutputOption objects.)

An innovation is the Environment object (Figure 10.25). Since I/O is now polymorphic the specifications for I/O channels have to be read in and the appropriate objects created. This responsibility, at least for the IInputOption and IOutputOption objects, is given to a separate object, the Environment object.
Friend Property Get Application() As IApp
  Set Application = New AppMC: Application.SetValues = Me
End Property

Friend Property Get OptionSpecs() As IOptionSpecs
  Set OptionSpecs = New OptionSpecsPlain: OptionSpecs.SetValues = Me
End Property

Friend Property Get OptionWrapper(spec As OptSpec) As IOptionWrapper
  Set OptionWrapper = New OptionWrapperPlain: OptionWrapper.SetValues(spec) = Me
End Property

Friend Property Get Path() As IPath
  Set Path = New PathPlain: Path.SetValues = Me
End Property

Friend Property Get Evolver() As IEvolver
  Set Evolver = New EvolverW: Evolver.SetValues = Me
End Property

Friend Property Get Generator(spec As ProcSpec) As IGenerator
  Select Case spec.p_type
    Case gbm: Set Generator = New GeneratorGBM
    Case Else: Call RaiseError(26, "Factory.Generator", "Bad p_type")
  End Select
  Generator.SetValues(spec) = Me
End Property

Friend Property Get Wiener() As IWiener
  Select Case wtype
    Case plain_Wiener: Set Wiener = New WienerPlain
    Case Else: Call RaiseError(26, "Factory.Wiener", "Bad WieType")
  End Select
  Wiener.SetValues = Me
End Property

Friend Property Get Accumulator(atype As acctype) As IAccumulator
  Select Case atype
    Case plain_acc: Set Accumulator = New AccumulatorPlain
    Case Else: Call RaiseError(26, "Factory.Accumulator", "Bad atype")
  End Select
  Accumulator.SetValues = Me
End Property

Friend Property Get OptionObj(spec As OptSpec) As IOption
  Set OptionObj = New OptionEuro
  OptionObj.SetValues(spec) = Me
End Property

Friend Property Get Payoff(spec As OptSpec) As IPayoff
  Select Case spec.o_type
    Case calltype: Set Payoff = New PayoffCall
    Case puttype: Set Payoff = New PayoffPut
    Case stocktype: Set Payoff = New PayoffStock
    Case Else: Call RaiseError(26, "Factory.Payoff", "Bad o_type")
  End Select
  Payoff.SetValues = spec
End Property

'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
'XX Factory: Factory methods
'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

Figure 10.24 The Factory object: Factory methods
The `Environment` object merely reads in the I/O ID characters and file paths from the front-end. It produces the corresponding I/O objects from the ID characters, calling their `SetValues()` Property to enable file paths to be passed over. It is used by the `Factory` object to create the `IInputOption` and `IOutputOption` objects that are later passed on to the `InputManager` and `OutputManager` objects.

It would be possible to keep I/O object creation in the `Factory` itself, rather than parcelling it out to a separate object. However, making the separation leads to greater decoupling. The I/O channels can change without affecting the application.

### 10.2 TIMINGS

The application has had two features added in. The first is ability to value more than one option at a time, the second is the ability to output and input from file. In this section we examine the costs and the savings of these features.

#### Benefits of scale

Only a single set of random numbers are generated to value every option in the book. Since we anticipate that the cost of random number generation will dominate the cost of a Monte Carlo method (an anticipation confirmed in Chapter 15), we expect that there will be considerable savings to scale in valuing more than one option at a time.

Table 10.1 confirms this. It gives computation times for valuing various numbers of options.\(^3\) The relationship between time and number of options is almost affine. There is a fixed cost of about 8 seconds, largely taken up with generating sample paths; thereafter every additional batch of 10 options takes about 0.85 seconds to value. Starkly, valuing a hundred options takes only double the time of valuing one.

It is clearly extremely advantageous to be able to value options in parallel.

#### Cost of input and output channels

The major part the cost of the Monte Carlo application is the numerics, but I/O costs are not trivial. Table 10.2 displays costs for the four combinations of I/O channels that are currently possible. The times are the total execution times, including I/O, for valuing 100 options.

Inputting from file is slow. Its cost is proportional to the amount of input. For 100 options, file input adds about 1.7 seconds compared to spreadsheet input. Output to file is, by contrast, cheap. Output to a spreadsheet adds about 0.1 seconds to the cost compared to outputting to file.

One cause for the cost of file input is the presence of the `ReDim Preserve` statement. This, as we have seen, is used to increase the size of an array as new data arrives to be added to it. However, the `ReDim Preserve` statement works by creating a new array of the desired size, and then copying the contents of the old array across to it. Doing this for every option specification becomes expensive. A cheaper mechanism would be to construct a memory manager that expanded the array in chunks, and not one row at a time. We do not pursue this idea here.

The cost of inputting from file should be seen as part of the cost of valuing many options simultaneously. File input is really the only practical input mechanism for large numbers of options. The cost of inputting option specifications from file (perhaps 0.02 seconds per option) should be seen alongside

---

\(^3\) Options are valued with \( N = 100 \) time steps and \( M = 50000 \) sample paths. Input is from the front-end. Output cost is excluded. Calculations are done with the spreadsheet `MC_example_v5b_times.xls`. 
Private sht_ As String 'Location sheet of input
Private i_type_ As String 'file or spreadsheet input
Private o_type_ As String 'file or spreadsheet output
Private i_path_ As String 'Path to input file
Private o_path_ As String 'Path to output file

Private Sub Class_Initialize()
    sht_ = "FrontEnd"
    i_type_ = get_char_wk(sht_, 24, 4, "fs")
    o_type_ = get_char_wk(sht_, 25, 4, "fs")
    i_path_ = get_path_wk(sht_, 24, 5)
    o_path_ = get_path_wk(sht_, 25, 5)
End Sub

Friend Property Get i_path() As String: i_path = i_path_: End Property
Friend Property Get o_path() As String: o_path = o_path_: End Property

Friend Function InputOption() As IInputOption
    Select Case i_type
        Case "f": Set InputOption = New InputOptionFile
        Case "s": Set InputOption = New InputOptionFE
        Case Else: Call RaiseError(26, "Environment.InputOption", "Bad i_type")
    End Select
    InputOption.SetValues = Me
End Function

Friend Function OutputOption() As IOutputOption
    Select Case o_type
        Case "f": Set OutputOption = New OutputOptionFile
        Case "s": Set OutputOption = New OutputOptionFE
        Case Else: Call RaiseError(26, "Environment.OutputOption", "Bad o_type")
    End Select
    OutputOption.SetValues = Me
End Function

Table 10.1  Time (in seconds) against number of options

<table>
<thead>
<tr>
<th># options</th>
<th>1</th>
<th>2</th>
<th>5</th>
<th>10</th>
<th>20</th>
<th>50</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>8.02</td>
<td>8.08</td>
<td>8.35</td>
<td>8.78</td>
<td>9.95</td>
<td>12.29</td>
<td>16.51</td>
</tr>
</tbody>
</table>

Table 10.2  Comparison of costs (in seconds) of I/O channels

<table>
<thead>
<tr>
<th>Input</th>
<th>Output file</th>
<th>Output sheet</th>
</tr>
</thead>
<tbody>
<tr>
<td>file</td>
<td>18.17</td>
<td>18.30</td>
</tr>
<tr>
<td>sheet</td>
<td>16.49</td>
<td>16.55</td>
</tr>
</tbody>
</table>
the cost of valuing the option as a one off: maybe 6 seconds. In this context we regard file input as only relatively expensive – not absolutely expensive – and cheap as a mechanism to enable multiple option valuations.

### 10.3 SUMMARY

It seems easy to extend the Monte Carlo method to value multiple options, simply by holding options (and accumulators) in arrays. Armed with a knowledge of VBA file handling it is also simple to read in the specifications from file. The idioms used in the chapter have very wide applicability.

Various extensions are possible. Other I/O channels could be devised: perhaps input from forms could be added, or from database files. Output could usefully be extended to write out not only the option value but also the option specification itself, so that the value is strongly associated with a particular option.

The path object deserves to be considerably extended. When interest rates are stochastic it would hold sample paths for both the underlying asset and for the interest rate. In a general multi-factor model it would contain a sample path of the vector of state variables. Also, there is no need for time steps to be uniformly spaced; the path object could associate with a calendar object to keep track of times.

The main drawback of the application as it stands is the narrowness of the OptSpec UDT. This is a bit of a straw man since OptSpec can be extended easily to allow many more categories of options to be valued, as in previous versions of the application.

Part IV takes a great leap forwards. Fed up with Enums? Then welcome to the polymorphic factory.

### 10.4 EXERCISES

The application stream exercises for the PDE and lattice methods ask you to place them into the framework described in this chapter. For these methods how do run times go up as the number of options in the book increases?

The objective is to value all options in the book simultaneously, if possible, rather than make a separate run for each option. Read in every specification in one pass through the file.

1. **Implied volatility stream.** Read in option specifications from a book held on file where the specification has been expanded to include a market price. The specification of each record should be the same as in exercise 5, Chapter 9.

2. **PDE stream.** The specification of each record on file should be in the format

   \[ S, r, \sigma, X, T, \text{TypeCode, ExCode}, \]

   where TypeCode and ExCode have the same meaning as in exercise 6, Chapter 9. Devise a means of specifying a single set of common values for \( N, M, S_{\text{max}} \) and \( S_{\text{min}} \). Allow the user to specify EvolverCode globally, perhaps from the front-end.

3. **Lattice stream.** Each record should be specified the same way as exercise 2. Allow the user to specify a single set of common values for \( N \), LatticeCode and SliceCode (defined as in exercise 7, Chapter 9) perhaps from the front-end.
We have seen that an extremely important principle in any programming methodology is polymorphism. Previous parts have investigated how run-time polymorphism is achieved in VBA using interface classes. This is wonderful as far as it goes. There is still the problem that object creation is non-polymorphic. When a request is received for a new object, conforming to some interface, the factory constructs it from a list of conforming objects in a Select statement. Should a new conforming object be required, a new Case has to be supplied by the programmer to the Select statement. In a previous example there was even an Enum with an item for each conforming type: a new object required a change to the Enum.

What is needed is a fully polymorphic factory, one where new conforming objects do not need to be listed by the programmer in a Select statement.

This part shows how this can be done.

Chapter 11 presents the VBE object library and shows how a factory can be constructed in a simple application. Chapter 12 extends the basic factory to one that is suitable for our numerical applications. Finally, Chapter 13 presents an alternative, semi-polymorphic, factory that could be used if the VBE object library is not available. It introduces classes to hold meta-data in a somewhat complex structure.
We have been able to achieve a fair amount of decoupling in our application but there is one glaring omission: the factory in Chapter 8 is still coupled to everything else in the entire world. When it manufactures objects it has to know about the entire universe of objects it is able to construct, and a knowledge of these has to be hard-wired in by the programmer.

This is highly inconvenient. Adding in a new polymorphic type requires the programmer to make changes to the factory. Much better is to have some kind of registration facility that performs this task automatically without programmer intervention.

A fully polymorphic factory can be constructed in C++. The usual mechanism is to declare within each derived class a special variable with file scope. The default constructor for that variable executes code which registers the object with, for instance, a factory. Unfortunately it is not possible from within a VBA 6.X project to execute any code in that project that does not exist within a procedure.¹ This means that objects cannot register themselves with other objects without first being instantiated. Somewhere, by some means, another object has to have prior knowledge of their existence.

In VBA we are able to obtain this knowledge using the VBIDE object. This gives access, under program control, to code in a project. It gives us a knowledge of the objects in a project and enables us to exploit this knowledge to set up automatically a factory to manufacture whatever objects we need.

In this chapter we discuss the construction of such a polymorphic factory. We start by describing the facilities available in the VBE object library and then move on to give an example of a simple factory. Finally we construct a more sophisticated factory that can cope with the needs of the applications in the subsequent parts of this book.

### 11.1 USING THE VBE OBJECT LIBRARY

The Visual Basic Editor (VBE) has a set of objects that embody the entities manipulated by the programmer when using the editor. These objects can be accessed directly by an application under program control. This enables the application to do anything with the code that a programmer could do. The facility seems to have been provided mainly to enable the creation of add-ins, but we exploit it to enable us to build a polymorphic factory.

**Accessing the VBE object library**

The library is not automatically available. To activate it one must:

1. Make available a reference to it. In the Tools | References dialogue locate the “Microsoft Visual Basic for Applications Extensibility 5.3” item and tick its check-box.

---

¹ Strictly, only Const expressions.
2. Give permission to VBA to use it. In Excel 2007 go to the Developer tab in the Excel ribbon. In the Macro Security item check the “Trust access to the VBA project object model” item. In Excel 2003 go to the “Trusted Publishers” tab of the Tools | Macro | Security dialogue and tick the “Trust access to the Visual Basic Project” check-box.

This section does not describe anything like the full capabilities of the VBE object library. Instead we describe only some features that we use in the factory. For a fuller review of its capabilities see, for instance, Kimmel et al. (2004).

We do not use the VBIDE object itself directly, only the objects it exposes. The objects of chief interest to us are:

1. The VBProject object. Each workbook has a project associated with it.
2. The VBComponents collection object. This is just a collection of VBComponent objects.
3. VBComponent objects. These represent code modules (and their associated superstructure).
4. CodeModule objects. These contain the code attached to a VBComponent object.

We look at these in turn, focusing on the CodeModule object. This is our primary interest. For us the other objects are just a means to access the CodeModule objects within a project.

**The VBProject object**

Excel workbooks each have a VBProject object associated with them. We do not use the VBProject object associated with ThisWorkbook except to enable us to access its CodeModules. Table 11.1 lists three of the dozen or so properties of the VBProject object. We use only the VBComponents Property. This returns the collection of VBComponent objects associated with a project with lines (11.1).

```
Dim comps As VBIDE.VBComponents
Set comps = ThisWorkbook.VBProject.VBComponents '.
```

**The VBComponent object**

Three of the more important (to us) properties of the VBComponent object are displayed in Table 11.2. We use only these three.

<table>
<thead>
<tr>
<th>Property name</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>FileName</td>
<td>The name of the associated workbook</td>
</tr>
<tr>
<td>Name</td>
<td>The project name</td>
</tr>
<tr>
<td>VBComponents</td>
<td>The VB components collection associated with the project</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Property name</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>CodeModule</td>
<td>The CodeModule object associated with the component</td>
</tr>
<tr>
<td>Name</td>
<td>The name of the component</td>
</tr>
<tr>
<td>Type</td>
<td>The component type: eg whether it is a class module or code module</td>
</tr>
</tbody>
</table>

---

2 The Developer tab should be set. See Appendix A.
Modules can be of various different types. Our interest focuses on class modules. We use the Type Property to determine whether or not a module is a class module. The module type is returned as a predefined vb constant. The possible values and their meanings are given in Table 11.3.

### The CodeModule object

The CodeModule object represents an individual code module. Applications can edit code within modules and it is this aspect that concerns us most. Other aspects, such as the manipulation of windows onto the module with the CodePane object, we ignore.

Methods and properties enable chunks of code to be identified and deleted, and other code, provided as Strings, to be added into code modules. Tables 11.4 and 11.5 summarize some of the available methods and properties of concern to us.

Some methods refer to procedures by name. Since several Properties can have the same name a type needs to be supplied along with the name to disambiguate it. type takes the predefined values given in Table 11.6. In the ProcOfLine() Property the argument type is passed ByRef. Its value is altered to give the type of the procedure.

Find() searches for text in the range line l1, column c1, to line l2, column c2 returning true if text is found. It can take additional arguments specifying matching criteria. l1 and c1 are passed ByRef. If text is found their values are set to the location in the range of the first character of text.

### Table 11.3 Values of the vbext_ComponentType enumerative variable

<table>
<thead>
<tr>
<th>Type value</th>
<th>Module type</th>
</tr>
</thead>
<tbody>
<tr>
<td>vbext_ct_StdModule</td>
<td>A standard module</td>
</tr>
<tr>
<td>vbext_ct_ClassModule</td>
<td>A class module</td>
</tr>
<tr>
<td>vbext_ct_MSForm</td>
<td>A user form</td>
</tr>
<tr>
<td>vbext_ct_Document</td>
<td>An Excel object</td>
</tr>
<tr>
<td>vbext_ct_ActiveXDesigner</td>
<td>Anything else</td>
</tr>
</tbody>
</table>

### Table 11.4 Some CodeModule object properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>CountOfDeclarationLines</td>
<td>Number of declaration lines at the start of the module</td>
</tr>
<tr>
<td>CountOfLines</td>
<td>Total number of lines in the module</td>
</tr>
<tr>
<td>ProcCountLines(name,type)</td>
<td>Number of lines in procedure name, including preceding comments</td>
</tr>
<tr>
<td>ProcBodyLine(name,type)</td>
<td>Line number of declaration line of procedure name</td>
</tr>
<tr>
<td>ProcStartLine(name,type)</td>
<td>Line number of start line of comments preceding procedure name</td>
</tr>
<tr>
<td>ProcOfLine(num,type)</td>
<td>String containing the name of the procedure that line num lies in</td>
</tr>
<tr>
<td>Lines(start,num)</td>
<td>String containing the text of lines [start, start + num - 1]</td>
</tr>
</tbody>
</table>

### Table 11.5 Some CodeModule object methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>DeleteLines(start,num)</td>
<td>Deletes lines in the range [start, start + num - 1]</td>
</tr>
<tr>
<td>AddFromString(text)</td>
<td>Inserts text into module at the end of the declaration section</td>
</tr>
<tr>
<td>AddFromFile(fname)</td>
<td>Inserts text in file fname at the end of the declaration section</td>
</tr>
<tr>
<td>InsertLines(text,num)</td>
<td>Inserts text immediately before line number num</td>
</tr>
<tr>
<td>ReplaceLine(text,num)</td>
<td>Inserts text, overwriting line number num</td>
</tr>
<tr>
<td>Find(text,l1,c1,l2,c2)</td>
<td>Searches for text in a range. Returns a Boolean</td>
</tr>
</tbody>
</table>
Table 11.6 Values of the vbext_ProcKind enumerative variable

<table>
<thead>
<tr>
<th>Type value</th>
<th>Procedure type</th>
</tr>
</thead>
<tbody>
<tr>
<td>vbext_pk_Proc</td>
<td>Function or Sub</td>
</tr>
<tr>
<td>vbext_pk_Get</td>
<td>Property Get</td>
</tr>
<tr>
<td>vbext_pk_Set</td>
<td>Property Set</td>
</tr>
<tr>
<td>vbext_pk_Let</td>
<td>Property Let</td>
</tr>
</tbody>
</table>

Table 11.7 Some VBProject methods and properties

<table>
<thead>
<tr>
<th>Value required for:</th>
<th>Found as:</th>
<th>Values for:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>SendMessage</td>
</tr>
<tr>
<td>Start line of procedure block</td>
<td>ProcStartLine,s</td>
<td>6</td>
</tr>
<tr>
<td>Start line of procedure body</td>
<td>ProcBodyLine,b</td>
<td>11</td>
</tr>
<tr>
<td>Number of lines in block</td>
<td>ProcCountLines,N</td>
<td>8</td>
</tr>
<tr>
<td>Lines in procedure body</td>
<td>s + N - b</td>
<td>3</td>
</tr>
<tr>
<td>Number of comment lines</td>
<td>b - s</td>
<td>5</td>
</tr>
<tr>
<td>Line number of end line</td>
<td>s + N - 1</td>
<td>13</td>
</tr>
</tbody>
</table>

Figure 11.1 Locations of procedure start and body lines

The assumption underlying the line count and location Property is that procedures are preceded by blocks of comment lines or blank lines, and that when a procedure is changed it may be necessary also to change the comment block. We call the procedure code plus the preceding comment lines the procedure block. Figure 11.1 illustrates. It shows a simple class module containing two procedures, SendMessage() and Dummy(). Line numbers are added down the left-hand side. Return values for ProcStartLine(), ProcBodyLine() and ProcCountLines() are given in Table 11.7. The values returned by ProcStartLine() and ProcBodyLine() are clear; the procedure block of SendMessage(),
the span covering lines 6 to 13, is returned by `ProcCountLines("SendMessage", vbext_pk_Proc)`, and `ProcCountLines("Dummy", vbext_pk_Proc)` returns the span covering lines 14 to 25, the procedure block of `Dummy()`.

### 11.2 A SIMPLE FACTORY ILLUSTRATION

This section discusses a simple polymorphic factory. The application here has a single interface and a single layer of factory. Chapter 12 extends the application to one with multiple interfaces and a hierarchy of factories.

Building a polymorphic factory is not easy. To do so you have to find out which objects exist within an application, and then find some way of instantiating them from a knowledge of, for instance, their name presented in the form of a `String`. In C++ this is achieved by a registration mechanism that exploits the ability in C++ of instantiating objects at namespace level within a translation unit. Code can be executed while this is happening and this ability can be used by the registration process. The equivalent facility (non-existent) in VBA would be to enable objects in a declaration section not only to be declared but also to be instantiated, and to permit the instantiation to have side effects.

But this is not possible in VBA. Instead we rely on the lower level ability of VBA to alter its own code down to the level of individual characters.

The procedure is to:

1. Obtain the collection of objects associated with the project in `ThisWorkbook`.
2. Find their names. For each admissible object create the code that will later register and create it.
3. Run the registration code.
4. Subsequently, upon demand, run the creation code.

The implementation screens out objects that should not or cannot be created in the factory. This includes objects in the invocation chain (explicitly removed during code creation), interface objects and other objects that conform to no interface (caught during the execution of the registration code). Objects that can and should be created by the factory are called admissible.

The registration and creation code is constructed by the `Workbook_Open()` procedure when the workbook is opened. The factory object runs the registration code when it is constructed. The creation code is run by the factory when it is asked to manufacture an object.

The registration process causes two procedures to be created in the code module of the `FactoryCreator` object: `RegisterObjs()` and `CreateObjs()`. `RegisterObjs()` is called by `FactoryCreator`’s constructor. It runs code that registers each object with the factory. `CreateObjs()` is run whenever a request is received to create an object. It is given a `String` as an argument holding the name of the required object; it returns an object of the named type. We come back later to see how these are constructed.

#### 11.2.1 The Pi application

Our example is an application to compute the value of Pi by series expansion. Four different series expansions are given, due to Leibniz, Euler, Schellbach, and Bailey, Borwein and Plouffe (1997) (BBP).
The exact nature of these expansions does not concern us here, but each requires the client to specify the number of terms summed. The larger the number of terms, the better the approximation the series provide.

The application is in 13_factory.xls; the front-end is shown in Figure 11.2. The client supplies an ID letter, to choose the method, and the number of terms to be summed, and clicks the button. The application computes the series approximation as specified, and prints out that value, along with the true value of $\pi$ (to 13 or so decimal places).

There are four Pi application objects, one for each Pi method: $\text{Pi}_\text{Euler}$, $\text{Pi}_\text{Leibniz}$, $\text{Pi}_\text{Schellbach}$ and $\text{Pi}_\text{BBP}$. They conform to an interface, IPi. Each Pi class has one main procedure: a Private function that computes the series approximation. There are getters for this value, and for the analytical value of $\pi$.

The code module for a typical Pi method, $\text{Pi}_\text{Euler}$, is given in Figure 11.3. In addition to the application code each Pi object has a name and an ID character. These are set in their constructors and accessed by Property Gets. The name must be set to the name of the object type; the ID character is the character given by the client to specify the particular Pi method.

### 11.2.2 Registration and creation

The RegisterObjs() and CreateObjs() procedures for this example are given in Figure 11.4. They each have one entry for each non-invocation chain object. From the String given as an argument, the CreateObjs() procedure looks up and creates the object with the same name. The RegisterObjs() procedure creates a temporary object of each type (so creating each object should have no side effects) and passes it over to its Private Sub, Reg().

RegisterObjs() and CreateObjs() have lines for IPi even though this is not created in the factory. The reason why there are entries for IPi is that (i) they do no harm and (ii) at the moment we have not described how they could be removed polymorphically. At a later stage these entries are screened out.

Although RegisterObjs() and CreateObjs() know about every Pi object, and CreateObjs() even has a Select statement, these procedures are created automatically by the program and not by hand by the programmer. It is this that makes the factory polymorphic.

Reg() is the procedure that performs the actual registration. It does so by adding entries to a Dictionary object, $\text{regs_{-}}$, which has as items the names of the objects registered with it, keyed by their ID characters. In the calling statement,

\[
\text{Call Reg(New Pi_BBP, "Pi_BBP")}
\]
The VBE Object Library and a Simple Polymorphic Factory 185

Implements IPi

Private AnalyticPi_ As Double
Private NumericalPi_ As Double
Private terms_ As Long
Private id_char_ As String 'Used by the factory
Private name_ As String 'Used by the factory

Public Property Get IPi_IDchar() As String: IPi_IDchar = id_char_: End Property
Public Property Get IPi_name() As String: IPi_name = name_: End Property
Public Property Let IPi_SetValues(fact As Factory)
    terms_ = fact.terms
End Property

Private Sub Class_Initialize()
    id_char_ = "E" 'Sets the ID character
    name_ = "Pi_Euler" 'Sets the name. Must be the type name
    AnalyticPi_ = Application.Pi()
End Sub

Property Get IPi_NumericalPi() As Double
    Static IsComputed As Boolean
    If Not IsComputed Then
        NumericalPi_ = ComputePi
        IsComputed = True
    End If
    IPi_NumericalPi = NumericalPi_
End Property

Property Get IPi_AnalyticPi() As Double
    IPi_AnalyticPi = AnalyticPi_
End Property

Private Function ComputePi() As Double
    Dim sum As Double: sum = 0#
    Dim i As Long
    For i = 1 To terms_
        sum = sum + (1# / i) / i
    Next i
    ComputePi = Sqr(6# * sum)
End Function

Figure 11.3 The Pi_Euler object

Reg() is passed an unnamed object instantiated on the hoof. Inside Reg() the corresponding argument is Set to a new instantiation of the object. Unless it is passed on elsewhere, its lifetime is just the lifetime of the function.

When it is called with an object obj, say, as its argument Reg() asks obj for its name and ID. Reg() performs four checks:

1. If the code module name (passed to Reg() as an argument) is the interface name, IPi, then the procedure exits without registering the object.
2. If obj cannot be cast as an IPi then Reg() skips the registration and exits.
3. If the name supplied by the object does not correspond to the code module name, then it throws.
4. If the ID character supplied as an argument already exists as a key in the dictionary, then it throws.
Private regs_ As Object

Private Sub RegisterObjs()
    Call Reg(New IPi, "IPi")
    Call Reg(New Pi_BBP, "Pi_BBP")
    Call Reg(New Pi_Euler, "Pi_Euler")
    Call Reg(New Pi_Leibniz, "Pi_Leibniz")
    Call Reg(New Pi_Schellbach, "Pi_Schellbach")
End Sub

Private Sub Class_Initialize()
    Set regs_ = CreateObject("Scripting.Dictionary")
    Call RegisterObjs
End Sub

Private Sub Class_Terminate()
    Set regs_ = Nothing
End Sub

Private Sub Reg(o As Object, o_name As String)
    If o_name = "IPi" Then Exit Sub
    On Error GoTo EndSub 'Turn on error trapping
    Set o = CastIPi(o)
    On Error GoTo 0 'Turn off error trapping
    Dim id As String: id = o.idchar
    Dim nm As String: nm = o.name
    If nm <> o_name Then
        Call RaiseError(26, "FactoryCreator.Reg", nm & " " & o_name & " differ")
        End If
        If regs_.exists(id) Then
            Call RaiseError(26, "FactoryCreator.Reg", id & " already exists")
        End If
        regs_.Item(id) = nm
    EndSub:
End Sub

Private Function GetID(id As String) As String
    If Not regs_.exists(id) Then
        Call RaiseError(26, "FactoryCreator.CheckID", "ID " & id & " does not exist")
    End If
    GetID = regs_.Item(id)
End Function

Private Function CastIPi(obj As Object) As IPi
    Set CastIPi = obj
End Function

Figure 11.4 13_factory.xls, FactoryCreator CodeModule
If all the checks are fine then the name is added, keyed by the ID character.

The first check is not polymorphic: it relies upon a knowledge of the particular interface name. The second check uses a utility function `CastIPi()`. This throws if `obj` does not conform to `IPi`.

After `RegisterObjs()` has run, every admissible object in the project has an entry in the dictionary, and only the admissible objects.

The `FactoryCreator` object is a helper object of the factory object proper, `Factory`. Its code is given in Figure 11.5. It has two procedures: a Property Get, `terms()` that returns the value of the `terms` parameter, and a Function, `Create()` that returns newly created objects upon request.

`Factory` is used in the invoker (Figure 11.6). The client enters on the front-end a character to identify the Pi object that is wanted. Helper Functions, `GetTerms()` and `GetMethodType()`, do the inputting. The method identification character is passed to the `Factory::Create()` method along with the value of the `terms` parameter. `Create()` passes the ID character over to the `CreateObjs()` method of its composite `FactoryCreator`. This looks up, using the helper Function `GetID()`, the ID character in the
Implementing Models of Financial Derivatives

If the character is present, GetID() returns the object name that it keys, otherwise it throws. The object with that name is then created and returned to Factory::Create(), which calls the new object’s SetValues() method to set the value of the terms parameter.

Simple.

All that Factory::Create() does is to dispatch the create request on to the FactoryCreator object. It would be possible to combine the roles of Factory and FactoryCreator into a single object. We prefer to separate the process of object creation from that of supplying values to parameters. Factory does the latter; the former is delegated to FactoryCreator.

11.2.3 Automating registration

The difficulty is to create the RegisterObjs() and CreateObjs() procedures. Fortunately we have available the facilities provided by the VBIDE object. The create_registrations() method is called by Workbook_Open() to perform the necessary work. The full listing of the code in ThisWorkBook is given in Figures 11.7 and 11.8.

Code to perform the registration is created automatically without programmer intervention. If new Pi methods are added to the application they need only define a valid name and ID character to be automatically registered and made available to the client. The workbook must be saved, closed, and re-opened for the registration to take effect.

The create_registrations() procedure has two parts. The first part constructs Strings, reg_text and cre_text, that contain the code for RegisterObjs() and CreateObjs(). The guts of these procedures is put into the Strings g1 and g2 by MakeStrings(). These Strings are passed to MakeRegisterObjs() and MakeCreateObjs() to be wrapped into proper procedures.

MakeStrings() loops around each code module in ThisWorkbook. If it is a class module, and if its name is not inadmissible (that is, does not appear in a list held by CheckName()), then its name is passed over to MakeLines() to be incorporated into the procedure gut Strings, g1 and g2. For an object with the name name, a String containing line (11.3a) is appended to g1 and one containing (11.3b) to g2.

\[
\begin{align*}
\text{Call Reg(New name, "name")} & \quad \text{'a. appended to g1} \\
\text{Case "name": Set CreateObjs = New name} & \quad \text{'b. appended to g2} \\
\end{align*}
\] (11.3)

We saw previously how these lines are used: when RegisterObjs() is run, line (11.3a) causes object name to be registered; line (11.3b) forms part of the body of a Select statement in CreateObjs() that creates the name object when a String containing the characters in name is presented to it.

The second part does the textual manipulation of the code in the FactoryCreator code module. It

(1) deletes existing versions of the RegisterObjs() and CreateObjs() procedures from the FactoryCreator code module;
(2) inserts the new versions, the Strings reg_text and cre_text, into the FactoryCreator code module.

The deletion is undertaken by DeleteProc(). It first checks to see if the procedure name, either "RegisterObjs" or "CreateObjs", already appears in the code module. It then identifies the lines to delete and deletes them. The check is necessary since the code throws if the procedure is not present.\(^5\)

Adding the new versions of the procedures is straightforward.

\(^5\) It should always be present from the last time the code was run – but one day it may not be. The manipulation of FactoryCreator is not exception safe. If something goes wrong, code may be deleted but not replaced.
Private Sub Workbook_Open()
    Call create_registrations
End Sub

Private Sub create_registrations()
    Dim gl As String, g2 As String: Call MakeStrings(gl, g2)
    Dim reg_text As String: reg_text = MakeRegisterObjs(gl)
    Dim cre_text As String: cre_text = MakeCreateObjs(g2)
    Dim obj As VBIDE.CodeModule
    Set obj = ThisWorkbook.VBProject.VBComponents("FactoryCreator").CodeModule
    Call DeleteProc(obj, "RegisterObjs")
    Call DeleteProc(obj, "Create")
    Call AddProc(obj, MakeRegisterObjs(reg_text))
    Call AddProc(obj, MakeCreateObjs(cre_text))
    Set obj = Nothing
End Sub

Private Sub MakeStrings(ByRef g1 As String, ByRef g2 As String)
    g1 = ": g2 = "
    Dim comps As VBIDE.VBComponents
    Set comps = ThisWorkbook.VBProject.VBComponents
    For Each c In comps
        If c.Type = vbext_ct_ClassModule Then
            Dim cn As String: cn = c.name
            If CheckName(cn) Then Call MakeLines(g1, g2, cn)
        End If
    Next c
    Set comps = Nothing
End Sub

Private Function CheckName(cn As String) As Boolean
    CheckName = cn <> "FactoryRegistrator" And cn <> "Factory" And cn <> "FactoryCreator" And cn <> "ErrorHandler"
End Function

Private Sub MakeLines(ByRef g1 As String, ByRef g2 As String, name As String)
    g1 = g1 + " Call Reg(New " + name + ", ", + name + ")" + vbCrLf
    g2 = g2 + " Case ", + name + ": Set CreateObjs = New ", + name + vbCrLf
End Sub

Private Function MakeRegisterObjs(guts As String) As String
    MakeRegisterObjs = 
    "'XXXXXXXXXXXXX RegisterObjs XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX" + vbCrLf + _
    "Private Sub RegisterObjs()" + vbCrLf + _
    "guts +_" + vbCrLf + _
    "End Sub" + vbCrLf + _
    "End Function"
End Function

Private Function MakeCreateObjs(guts As String) As String
    MakeCreateObjs = 
    "'XXXXXXXXX CreateObjs XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX" + vbCrLf + _
    "Public Function CreateObjs(idchar As String) As IPi" + vbCrLf + _
    " Dim id As String: id = GetID(idchar)" + vbCrLf + _
    " Select Case id" + vbCrLf + _
    " guts +_" + vbCrLf + _
    " End Select" + vbCrLf + _
    "End Function" + vbCrLf + _
    "End Function"
End Function

Figure 11.7 13_factory.xls, ThisWorkbook CodeModule, create_registrations()
11.3 SUMMARY

It is possible to build a fully polymorphic factory in VBA. It is simple and effective. A potential problem is that the mechanism chosen to implement the factory – exploiting the VBIDE object – is the same mechanism used by virus builders. Our use is benign and pro-functional, but a virus checker may not know this.

In the next chapter we see how the concepts introduced here can be expanded to give a factory that works for the Monte Carlo application.

11.4 EXERCISES

These exercises concern the VBIDE object. The first set of exercises give some practice with using the VBIDE object to write various useful utility procedures. The second set asks you to devise bespoke factories for the application exercise streams.

1. Write a workbook utility, VBAstats(), that executes when a workbook is opened and produces a report on the modules within the workbook.

   It constructs a list of the modules and code modules in the workbook, and for each module lists the procedures it contains. In addition, for code modules, it lists the interfaces to which each code module conforms. The report should be written to a worksheet named VBAstatsSheet. If this does not exist it should be created.

2. Appendix A describes the notion of a clean recompile. Write a utility that accomplishes this. It should perform the following steps:

   (a) Find the names of every code module in the workbook.

   (b) For each code module, create a text file with a similar name to the code module. The contents of the code module should be written out in its entirety to the text file, and the code module should then be deleted.
(c) Each code module should then be re-created. A code module of the same name should be inserted into project and code from the text file copied back into it.

(d) Once all the code has been copied back successfully the text files must be deleted.

Does this work? If not how should it be altered to make it work?

**Warning.** Please test this and run it only on copies of your workbooks, not on the only copy of your live code. Please do not say you lost valuable code on my account.

3. **Pi stream.** In Chapter 8 you developed a level 5 pi application, pi_app_v5.xls.

(a) Adapt this to endow it with the simple pi factory described in the chapter. Name it pi_app_v5a.xls.

(b) The following series compute the value of π

\[
1 + \frac{1}{3} - \frac{1}{5} + \frac{1}{7} - \cdots = \frac{\sqrt{2}}{4}\pi, \quad (11.4)
\]

\[
\sum_{k=1}^{\infty} \frac{(-1)^{k+1}}{2k(2k+1)(2k+2)} = \frac{1}{4}(\pi-3), \quad (11.5)
\]

\[
\sum_{k=0}^{\infty} \frac{(k!)^2}{\phi^{2k+1}(2k+1)!} = \frac{2\pi}{5\sqrt{\phi+2}}, \quad \text{(Gosper)} \quad (11.6)
\]

where \(\phi = (1 + \sqrt{5})/2\) is the golden ratio.

Add objects to both pi_app_v5.xls and pi_app_v5a.xls to enable them to execute polymorphically the new methods. Assess for each spreadsheet the advantages and disadvantages of the procedures you have to go through to add in new pi methods.

4. The Pi factory has to create objects belonging only to the IPi hierarchy. If you have more than one hierarchy you will need to extend the factory in some way to cope with this. Chapter 12 presents a general-purpose factory. For the moment practise constructing a bespoke factories tailored to each particular application.

For each exercise stream build a simple factory, perhaps adapted from the Pi factory example, to enable it to construct objects from their application hierarchy, or hierarchies, polymorphically.

(a) **Implied volatility stream.** This has two main hierarchies, the root-finder hierarchy and the hierarchy of functors whose roots are to be found.

(b) **PDE stream.** This is likely to have an evolver hierarchy, and an option hierarchy.

(c) **Lattice stream.** This may have an evolver hierarchy, a solver hierarchy, a slice hierarchy and an option hierarchy.
Chapter 11 described a factory that could create polymorphically only instances of IPi objects. We need a rather more powerful factory capable of constructing objects from a range of interfaces. It is also desirable to be able to define a set of factories, one for each conceptual layer in a complete application, to separate out the services they provide. Needed are a factory for I/O, another for the application objects, and a third for the data objects on which the application operates.

In this chapter we use the techniques developed in Chapter 11 to build a polymorphic factory for the Monte Carlo application of Chapter 8. As in Chapter 11, the Workbook_Open() procedure creates two procedures in the class module FactoryCreator: RegisterObjs(), which registers objects and is run when a FactoryCreator is instantiated, and CreateObjs(), which runs whenever the factory is asked to produce a new object.

Other than making changes to the factory the Monte Carlo application itself is more or less unchanged. We start by looking at some new conceptual material needed for the factory, and then look at the factory proper. Finally we see how the factory works with the Monte Carlo application. We find that it fits in quite seamlessly.

### 12.1 CONCEPTUAL FEATURES

The new factory makes use of several facilities absent so far from our applications. Here we introduce and discuss the notion of a base class, parameter and synonym classes, and the mix-in interface classes, ICreatable and IReusable.

**The base class**

In principle there is no reason why objects should not implement several interfaces, conforming to each. Indeed, as well as conforming to their primary application interface objects may now also conform to mix-in interfaces like ICreatable and IReusable (described below).

By a mix-in interface is meant a utility interface providing particular but generally useful services not necessarily specific to a particular application. For instance, ICreatable is of this type. Conforming objects are endowed with the ability to be created by the factory, whatever application they may happen to belong to.

By an application interface is meant an interface specific to a particular application. For instance, the IAccumulator and IPayoff interfaces are application interfaces specific to a particular option valuation application.

Usually objects will conform to a single application interface – an accumulator object is not a payoff object – but this is not a requirement of VBA. However the factory we build will assume that objects may be created polymorphically as instances of only one prespecified interface. Objects may conform to more than one interface, but only one is distinguished by its use in the factory. The distinguished interface is called the object’s base type or base class. For example, payoff objects have the IPayoff application interface as their base class; AppMC has IApp as its base class.
Parameter classes

It is convenient to give names to parameters in the application. The name can be passed around the application as a String uniquely specifying the parameter type. To avoid telepathic use of the same String literal in different parts of an application, we define parameter classes whose purpose is to give names to a parameter type. This helps to ensure that problems can be picked up at compile time rather than at run-time.

The parameter interface, IPara, and an example of a parameter class, ParaSigma, are given in Figure 12.1. All that ParaSigma does is to define a name. Note that it would certainly be possible to allow the parameter class to hold also the value of the parameter, but we do not adopt such a design approach here.

Once defined the parameter names can be used to refer unambiguously (one hopes) to the parameters used by the application. One use is to employ the names as keys in a Dictionary object whose items are the parameter values.

Later on we see exactly how these classes are used. As an illustration, just for now, see Figure 12.16, (page 207). The AppMC::SetValues() method sets the value of the parameter M with the line

\[ M = \text{fact}\cdot\text{Value(New ParaNpaths)} \]  \hspace{1cm} (12.1)

Specifying the parameter using the class ParaNpaths ensures unambiguously that it is assigned a value that an inputter has explicitly associated with another ParaNpaths object. The procedure removes entirely the need for telepathy.

The ICreatable interface

It is convenient to define a special interface, ICreatable, shown in Figure 12.2, for objects that are to be created in the factory. ICreatable defines three Property Gets and a Property Let. The Property Gets are:

1. name(): returns the object name as a String;
2. base(): returns the name of the object’s base class as a String;
3. id(): returns a identification character for the object, unique among objects conforming to the particular base class.
The Property `Let SetValues(fact As Factory)` takes the factory as an argument, enabling it to be passed to an object so that composited objects can be created, or parameter values set. `SetValues()` (with `Me` as an argument) is called by the factory so its presence is required in every creatable object.

The name returned by `name()` must be the name of the object type. The returned name is used during registration to create code that returns an object with that name.

**The `IReusable` interface**

A number of objects need to be re-used, and in between cycles of use they need to be set up afresh. The `IReusable` interface, given in Figure 12.3, is intended to be used by this category of object. It defines two Subs: `Initialise()` and `Reset()`. These can be defined in conforming classes to do whatever is required.

In order to use `IReusable` it is necessary to cast an object from its base type to the `IReusable` type. The function `CastReusable()` (Figure 12.4) does just this. It is called in lines like (12.2)

```
CastReusable(und_).Initialise '.
```

`CastReusable(und_)` attempts to set its argument to an object of type `IReusable`, which it then returns. If its argument does not conform to `IReusable` there will be a run-time error.

**Synonym classes**

These classes perform for objects an analogous function to that performed for parameters by the parameter classes. The need for them arises from the need of applications to have instantiated simultaneously more than one object from the same base class. For instance, the Monte Carlo application has an outer and an inner application object, each conforming to the `IApp` interface, and an outer and an inner accumulator object, each conforming to the `IAccumulator` interface. The factory needs to be able to identify whether it is being asked for the outer or the inner instantiation of this object.

```
Public Sub Initialise(): End Sub
Public Sub Reset(): End Sub
```

Figure 12.2 The `ICreatable` interface

Figure 12.3 The `IReusable` interface
The application uses several very useful utility functions that we exploit routinely from now on. They enable us to cast from one type to another, and to allow a certain amount of syntactic sweetening of the code. The first type of utility function is epitomized by `CastReusable()`:

```vbnet
Public Function CastReusable(obj As Object) As IReusable
    Set CastReusable = obj
End Function
```

When an object has been declared to be of one interface type and you want to call one of its methods belonging to a different interface, then you have to cast to the new interface.

```vbnet
Dim evl_ As IEvolver 'a,
Set evl_ = New EvolverW 'b,
CastReusable(evl_).Initialise 'c.
```

Lines (12.4a) and (12.4b) declare and set `evl_` as an object conforming to the `IEvolver` interface. Assuming that `evl_` also conforms to `IReusable`, line (12.4c) calls the method `IReusable::Initialise`. An attempt to call directly a method belonging to the `IReusable` interface will fail: lines (12.5a) and (12.5b) both fail.

```vbnet
evl_.Initialise 'a, fails,
evl_.IReusable_Initialise 'b, fails.
```

A cast is required and `CastReusable()` performs the cast. The temporary reference created by `CastReusable()` exists only as long as the statement runs. `CastReusable()`, and similar casts like `CastCreatable()`, are found in the `LibFactory` library module.

The second useful utility is the function `anon()`, defined in lines (12.6).

```vbnet
Public Function anon(obj As Object) As Object
    Set anon = obj
End Function
```

All it does is to return its argument. Its usefulness is that it enables objects to be instantiated on the fly: an unnamed object can be passed on to `anon()` and its functionality popped. The three lines of code

```vbnet
Set obj = New SomeObj
obj.SomeMethod(97)
Set obj = Nothing
```

can be replaced by the single line

```vbnet
anon(New SomeObj).SomeMethod(97)'.
```

The scope of the popped object is restricted to the statement in which it appears.

---

**Figure 12.4** Casts

Our solution is to give a name to each slot to be filled by the factory. These names, along with the name of the base class of the object needed to fill the slot, are given by synonym objects.

The synonym interface, `ISyn`, is given in Figure 12.5, along with two examples of synonym classes: `SynInnerAcc` and `SynIWiener`. These classes are very simple. Their only function is to associate with their type the name of the slot and the name of the base class for that slot. They are called synonym classes because effectively they create synonyms for the names of base classes. `SynInnerAcc` associates the name "InnerAcc" with the base class `IAccumulator`; `SynIWiener` tautologically associates the name "IWiener" with the `IWiener` base class.

Now whenever the application requires an object it can pass to the factory not a `String` containing the name of the object required but a synonym object for that slot.
12.2 THE POLYMORPHIC FACTORY

The factory is a fairly straightforward extension of the factory described in Chapter 11. First we look at some changes to the ThisWorkBook code module, then we examine first the factory objects proper, and then the associated inputter objects.

12.2.1 The ThisWorkBook code module

The ThisWorkBook code module is very similar to that in Chapter 11. The only real change is that we are able to exploit the presence of the ICreatable interface to determine which objects should register with and be created by the factory. The changes occur in the MakeStrings() method, displayed in Figure 12.6. The previous version of MakeStrings() used a function CheckName() to check object names against those in a list, throwing away objects whose names it found. This version checks to see if the String "Implements ICreatable" occurs in the declarations section of each class module. The corresponding object is included only if this text is found. This ensures that no non-factory creatable object is registered.
12.2.2 The factory objects

There are three factory objects

1. Factory: This is the main factory object. It acts mainly as a façade, dispatching requests on to composite FactoryLine objects, although it sets up these objects initially. It also gives the rest of the application access to I/O objects.

2. FactoryLine: These are the main production units. Each has its own IInputter object. It is responsible for creating objects associated with that IInputter.

3. FactoryCreator: There is a single FactoryCreator object in the application, used by every FactoryLine object. It does the actual object creation.

We look at each object in turn.

The Factory object

The factory, shown in Figure 12.7, has three layers within it, with a FactoryLine object (and its associated IInputter object) for each layer. Each layer is responsible for constructing the input and output channels for the layer beneath it.

The top layer is the environmental layer. This inputs using an InputterEnv object. It is created on line 12.7a in the object constructor and has its input source set on line 12.7d. The middle layer is the application layer, created on line 12.7b. Its inputter object is constructed by the environmental layer FactoryLine. Its outputter is the monitor object. The application layer is responsible for constructing the application object.

The bottom layer is the data layer. Its inputter is passed to it by the application layer FactoryLine. It constructs the objects on which the application operates.

The factory has four categories of Public methods. The first category is a set of getters for the I/O objects in the application. These are the Monitor object, the Outputter object and the two externally accessible inputter objects (the environmental inputter stays local to the factory).
Private crea_ As FactoryCreator
Private envt_ As IInputter 'Inputter for environment data
Private fenv_ As FactoryLine 'FactoryEnv
Private fapp_ As FactoryLine 'FactoryApp
Private fdat_ As FactoryLine 'FactoryData
Private data_ As IInputter 'Inputter for the application data
Private moni_ As IMonitor 'The monitor
Private outp_ As IOutputter 'The outputter
Private spec_ As IInputter 'Inputter for the application specs

Friend Property Get Factory() As FactoryCreator: Set Factory = crea_: End Property
Friend Property Get outputter() As IOutputter: Set outputter = outp_: End Property
Friend Property Get monitor() As IMonitor: Set monitor = moni_: End Property
Friend Property Get data() As IInputter: Set data = data_: End Property
Friend Property Get spec() As IInputter: Set spec = spec_: End Property

Friend Function Value(val As IPara) As Double
If fenv_.ExistsVal(val) Then Value = fenv_.Value(val): Exit Function
If fapp_.ExistsVal(val) Then Value = fapp_.Value(val): Exit Function
If fdat_.ExistsVal(val) Then Value = fdat_.Value(val): Exit Function
Call RaiseError(26, 'Factory.Value', val.name & " not found")
End Function

Friend Function Name(val As IPara) As String
If fenv_.ExistsName(val) Then Name = fenv_.Name(val): Exit Function
If fapp_.ExistsName(val) Then Name = fapp_.Name(val): Exit Function
If fdat_.ExistsName(val) Then Name = fdat_.Name(val): Exit Function
Call RaiseError(26, 'Factory.Name', val.Name & " not found")
End Function

Friend Function Create(syn As ISyn) As ICreatable
If fenv_.ExistsObj(syn) Then Set Create = fenv_.Create(syn): Exit Function
If fapp_.ExistsObj(syn) Then Set Create = fapp_.Create(syn): Exit Function
If fdat_.ExistsObj(syn) Then Set Create = fdat_.Create(syn): Exit Function
Call RaiseError(26, 'Factory.Create', syn.name & " not found")
End Function

Private Sub Class_Initialize()
Set crea_ = New FactoryCreator: crea_.SetValues = Me
Set fenv_ = New FactoryLine: fenv_.SetValues = Me 'a. Environmental layer
Set fapp_ = New FactoryLine: fapp_.SetValues = Me 'b. Application layer
Set fdat_ = New FactoryLine: fdat_.SetValues = Me 'c. Data layer
Set envt_ = New InputterEnv: fenv_.SetSource = envt_ 'd
Set spec_ = Create(New SynAppInputter): fapp_.SetSource = spec_ 'e
Set data_ = Create(New SynDataInputter): fdat_.SetSource = data_ 'f
Set moni_ = Create(New SynIMonitor) 'g
Set outp_ = Create(New SynIOutputter) 'h
End Sub

Private Sub Class_Terminate()
Call Destroy
Set crea_ = Nothing
Set envt_ = Nothing
Set fenv_ = Nothing
Set fapp_ = Nothing
Set fdat_ = Nothing
Set outp_ = Nothing
Set moni_ = Nothing
Set spec_ = Nothing
Set data_ = Nothing
End Sub

Friend Sub Destroy()
If Not crea_ Is Nothing Then Call crea_.Destroy
End Sub

Figure 12.7 The Factory object
The second category has a single method in it. It is a getter, `Factory()`, for the composite `FactoryCreator` object within the factory that does the actual object creation. This getter enables the factory to pass the `FactoryCreator` on to the composite `FactoryLine` objects that need it.

The third category are the functions that return objects and parameter values: `Create()` for objects, `Value()` for `Double` s and `Name()` for `String`s. These take as an argument a synonym or parameter class for the type of object or parameter whose instance or value is required.

The fourth category also has a single method in it, `Destroy()`. This is present to enable links set up between the factory objects to be destroyed correctly. `Destroy()` must be called just before the `Factory` is set to `Nothing`.

The factory itself instantiates the top-layer inputter object, `envt_` of type `InputterEnv`, in its constructor (line 12.7d), at the same time as the composite `FactoryCreator` object, `crea_`, is instantiated. The factory then instances the three layers of `FactoryLine` objects. Each `FactoryLine` object in turn delivers an `Inputter` object that is used to set up the next `FactoryLine` object down the chain. The top layer `FactoryLine` object, `fenv_`, is set up with `envt_`. It returns the application inputter object, `spec_`, which sets up the middle `FactoryLine` object, `fapp_` (line 12.7e). Finally `fapp_` returns the data inputter object, `data_`, that sets up the bottom layer `FactoryLine` object, `fdat_` (line 12.7f).

The `FactoryCreator` object is shown in Figures 12.8 and 12.9. Figure 12.8 gives the `RegisterObjs()` and `CreateObjs()` methods, and Figure 12.9 gives the supporting methods.

The form of `RegisterObjs()` and `CreateObjs()` is identical to those in Chapter 11. However, the registration process, undertaken by the `Private` method `Reg()`, is a little different, reflecting the needs of this more complex factory.

As before, the registration process constructs a `Dictionary` object, `regs_`. The structure of `regs_` is illustrated in Figure 12.10. Its keys are the names of base classes, its items are themselves `Dictionary`s. These individual base-class `Dictionary`s hold ID characters and the corresponding object names for every object that conforms to the base class that keys it.

The `RegisterObjs()` procedure calls `Reg()` on every object it knows about. `Reg()` gets from the object its name, ID and base class. If `regs_` already has an item keyed by the base-class name then this is obtained, otherwise a new base-class `Dictionary` is created and inserted into `regs_`. `Reg()` then checks to see if the ID already exists. If is does, `Reg()` throws. Otherwise the object name is added to the base-class `Dictionary`, keyed by the ID.

Note that since `RegisterObjs()` creates a temporary instance of every object it registers it is important that doing so has no side effects. In particular the constructor and destructor of any `Factory` creatable object should do nothing substantive.¹

¹ Previous versions of some objects break this prohibition. They are modified in this version to reflect the new principle.
The ThisWorkbook.MakeStrings() procedure ensures that only those code modules that conform to the ICreatable interface\(^2\) appear in RegisterObjs() and CreateObjs(). In particular objects that do not conform to ICreatable, such as parameter classes, will not appear there. Nevertheless Reg() checks if the objects it is passed conform to ICreatable. First, if the object is ICreatable itself (it should never be) Reg() exits. Second, it tries to cast the object as ICreatable (using the utility CastCreatable()). This can succeed only if the object conforms, otherwise it throws. If it throws the error is trapped, Reg() exits, and the object is not registered.

**The FactoryLine object**

The FactoryLine object, Figure 12.11, looks after object creation involving data read in from a particular Inputter object, kept as a composited object within it. It maintains three Dictionaries, vals_, nams_ and syns_, that contain data read in by the composited Inputter object. vals_ contains parameter values in the form of Doubles, keyed by parameter names; nams_ contains values in the form of Strings; syns_ contains object ID characters keyed by names from the synonym objects that identify them.

Before it can be used a FactoryLine object must have its source set. SetSource() sets the value of the composite Inputter object and asks it to fill up vals_, nams_ and syns_.

\(^2\) Or at least contain the text "Implements ICreatable" in their declaration section.
Friend Function Create(base As String, id As String) As ICreatable
    Call CheckBase(base)
    Dim regs As Object: Set regs = regs_.Item(base)
    Dim valid_ids As String: valid_ids = GetKeyString(regs)
    Dim idchar As String: idchar = check_char(id, valid_ids, "")
    Dim objid As String: objid = regs.Item(idchar)
    Set Create = CreateObjs(objid): Create.SetValues = fact_
    Set regs = Nothing
End Function

Private Sub Reg(o As Object, o_name As String)
    If o_name = 'ICreatable' Then Exit Sub
    Dim dict As Object
    On Error GoTo EndSub
    Set o = CastCreatable(o)
    On Error GoTo 0
    Dim name As String: name = o.name
    Dim base As String: base = o.base
    Dim id As String: id = o.id
    If name <> o_name Then Call RaiseError(26, "Reg", name & " <> " & o_name)
    If Not regs_.exists(base) Then
        Set dict = CreateObject("Scripting.Dictionary")
        Set regs_.Item(base) = dict
    End If
    Set dict = regs_.Item(base)
    If dict.exists(id) Then Call RaiseError(26, "Reg", id & " exists")
    dict.Item(id) = name
EndSub:
    Set dict = Nothing
End Sub

Private Function GetKeyString(regs As Object) As String
    Dim ks As Variant: ks = regs.keys
    Dim lb As Long, ub As Long
    lb = LBound(ks): ub = UBound(ks)
    GetKeyString = ""
    Dim i As Long
    For i = lb To ub: Call AddChar(CStr(ks(i)), GetKeyString): Next i
End Function

Private Function CheckBase(base As String)
    Dim fn As String: fn = "FactoryCreator.Create"
    Dim ml As String: ml = " does not exist"
    If Not regs_.exists(base) Then Call RaiseError(26, fn, "Base " & & base & ml)
End Function

Private Sub BadName()
    Call RaiseError(26, "CreateObjs()", "Bad name")
End Sub

Private Function CastCreatable(obj As Object) As ICreatable
    Set CastCreatable = obj
End Function

Figure 12.9 The FactoryCreator object: supporting methods
A FactoryLine object is asked to return a parameter value by calling its `Value()` or `Name()` methods. For instance `Value()` takes a parameter class object as an argument. It gets the name of the parameter, looks it up in the `vals_ Dictionary`, and returns the item it finds there. There is a companion method, `ExistsVal()`, that tests to see whether `vals_` contains a particular parameter. It is assumed that `Value()` is to be used in conjunction with `ExistsVal()`.

The procedure for obtaining Strings from `name_` with `Name()` and `ExistsName()` is identical.

The procedure for returning a newly created object is similar. The `FactoryLine::Create()` method takes a synonym class as its argument and returns an object corresponding to the synonym, if it exists. `Create()` first looks to see if the object name exists in the `syns_ Dictionary`. If it does then the ID it keys is passed to the composite FactoryCreator object, along with the synonym base class. This returns the desired object.

### 12.2.3 Inputting parameter values and object specifications

The role of the inputter objects is to input parameter values and object specifications from (in this case) the front-end, filling Dictionaries held by the FactoryLine objects.

Inputter objects conform to the `IInputter` interface, shown in Figure 12.12, which has only two interface functions: `GetValues()` fills the parameter values Dictionary and `GetIDs()` fills the object ID Dictionary.

Inputter objects know which parameter and object specifications they are responsible for and where to find them. We discuss only the `InputterApp` object, shown in Figure 12.13. The other two inputter objects, `InputterDataFrontEnd` (Figure 12.14) and `InputterEnv` (Figure 12.15), are very similar. Only the code that differs from that in `InputterApp` is shown.³

`InputterApp::GetValues()` fills up its argument, `vals`, which it assumes is a Dictionary. The work is done by `GetLValue()`. This is passed a parameter class and a location specification (a cell coordinate). It reads in the cell value, does minor validation and inserts the value into the dictionary, keyed by the parameter name. `InputterApp::GetIDs()` is similar except that it is passed a synonym class.

`GetIDs()` and `InputterDataFrontEnd::GetValues()` use two other validation functions, `GetID()` and `GetDValue()`. These are very similar to `GetLValue()` except that it validates only for a Double, not a Long.

---

³ Only the constructor, the `GetValues()`, and `GetIDs()` code is different.
Private crea_ As FactoryCreator
Private data_ As IInputter "Inputter for the factory
Private vals_ As Object
Private nams_ As Object
Private syns_ As Object

Private Sub Class_Initialize()
Set vals_ = CreateObject("Scripting.Dictionary")
Set nams_ = CreateObject("Scripting.Dictionary")
Set syns_ = CreateObject("Scripting.Dictionary")
End Sub

Private Sub Class_Terminate()
Set crea_ = Nothing
Set data_ = Nothing
Set vals_ = Nothing
Set nams_ = Nothing
Set syns_ = Nothing
End Sub

Friend Property Let SetSource(data As IInputter)
Set data_ = data
Call data_.GetIDs(syns_)
Call data_.GetValues(vals_)
Call data_.GetNames(nams_)
End Property

Friend Property Let SetValues(fact As Factory)
Set crea_ = fact.Factory
End Property

Friend Function ExistsObj(syn As ISyn) As Boolean
Dim nm As String: nm = syn.name: ExistsObj = syns_.exists(nm)
End Function

Friend Function Create(syn As ISyn) As ICreatable
Dim name As String: name = syn.name
If Not syns_.exists(name) Then Call BadName(name)
Dim base As String: base = syns_.Item(name)
Dim id As String: id = syns_.Item(name)
Set Create = crea_.Create(base, id)
End Function

Friend Function ExistsVal(val As IPara) As Boolean
Dim nm As String: nm = val.name: ExistsVal = vals_.exists(nm)
End Function

Friend Function Value(val As IPara) As Double
Dim name As String: name = val.name
If Not vals_.exists(name) Then Call BadName(name)
Value = vals_.Item(name)
End Function

Friend Function ExistsName(val As IPara) As Boolean
Dim aName As String: aName = val.Name
If Not nams_.exists(aName) Then Call BadName(aName)
Name = nams_.Item(aName)
End Function

Private Sub BadName(nm As String)
Call RaiseError(1117, "FactoryLine", "Invalid name: " & nm)
End Sub

'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Figure 12.11 The FactoryLine object
A Fully Polymorphic Factory: Level 6

Figure 12.12  The IInputter interface

Figure 12.13  The InputterApp object
12.3 USING THE Factory OBJECT

The AppMC object (Figures 12.16 and 12.17) gives an example of how the factory is used. The methods implementing the ICreatable interface are shown in Figure 12.16, the remaining methods in Figure 12.17.

The SetValues() method concerns us. AppMC needs both the results outputter, out_, and the monitor object, mon_. It requests references for these from the factory. Then it asks for the application objects it needs: an IAccumulator object, acc_, and an IEvolver object, evl_. The line used to Set the acc_ object is

\[
\text{Set acc_ = fact.Create(New SynInnerAcc) '}. \quad (12.9)
\]
The Factory::Create() method is passed a SynInnerAcc object created on the fly. The name Property of this synonym object uniquely specifies the object required to the factory. Similarly the line

\[ M = \text{fact.Value(New ParaNpaths)} \]  

(12.10)

asks the factory to return the number-of-paths parameter, M.
AppMC implements the IReusable interface. It uses CastReusable() at several points to ensure that methods from this interface can be called. For instance, the main application method is run() (Figure 12.17). The line

\[
\text{CastReusable(Me).Reset} \quad (12.11)
\]

calls AppMC’s own IReusable::Reset() method.

### 12.4 SUMMARY

It is worth reflecting upon what we have achieved since Chapter 2. Even though, as a procedure, the level 1 design was quite clear it was also extremely limited. Everything was hard-wired in. The process, the numerical method, the option; everything was coupled, nothing could be generalized without pain. In the level 6 design anything can change, and relatively easily.

Why do any of this? Why not write a whole new level 1 procedure every time a new option or configuration comes along?
Because it is unmanageable. If you have 17 different programs, one for each option type say, and then you are told to value each option under a different process (perhaps with jumps or stochastic volatility) you have a big problem. Each of the 17 programs has to be rewritten.

Perhaps you had been sensible and had encapsulated the functionality that needs changing into a single function. You still have to go into every one of the 17 programs and change it.

The factory is quick and non-obtrusive. It works very effectively within its comfort zone. The worst problem is only that when adding further polymorphic objects to the application it needs to be saved, closed and then re-opened before the registrations take effect and the new object can be recognized.

What this factory is going to be poor at is handling applications where the underlying data is complex. The mechanism of synonym and parameter classes can work very well in medium scale applications where there is a limited number of objects to be instantiated and parameters to be read in. Once the numbers increase, or the numbers of simultaneously existent objects and parameters increase, then this style of factory could be stretched to breaking point. Fortunately we do not intend to stretch the bounds of the factory’s sphere of competence.

12.5 EXERCISES

For each application exercise stream endow the application with a fully polymorphic factory. Create additional polymorphic objects for the indicated functionality in each stream. How easy is it to add in these objects compared to earlier versions of the factory?

1. **Pi stream.** The following further series compute \( \pi \).

\[
\sum_{i=0}^{\infty} \binom{2i}{i} \frac{1}{(2i+1)2^{4i}} = \frac{\pi}{3},
\]

\[
\frac{2\sqrt{2}}{9801} \sum_{i=0}^{\infty} \frac{(4i)!(1103+26390i)}{(i!)^4396^{4i}} = \frac{1}{\pi}, \text{ (Ramanujan)}
\]

\[
\frac{1}{1^4} + \frac{1}{2^4} + \frac{1}{3^4} + \ldots = \frac{\pi^4}{90},
\]

\[
\frac{1}{1^3} - \frac{1}{3^3} + \frac{1}{5^3} - \frac{1}{7^3} + \ldots = \frac{\pi^3}{32}.
\]

Create and include into the Pi project objects that code up these series.

2. **Implied volatility stream.** Implement the Newton–Raphson root-finding method. The Newton–Raphson method requires the computation not only of the function whose root is to be found but also its derivative. How can this fit in with the requirements of the IFunct interface?

3. **PDE stream.** Let \( v_{T,H}(t, S_t) \) denote the value at time \( t \leq T \) of a European option written on \( S = (S_t)_{t \geq 0} \) with final maturity date \( T \) and payoff function \( H(S_T) \). Let \( T = \{T_k\}_{k=1,\ldots,K} \) be a set of reset dates. A Bermudan compound call option with strike \( Y \) and final maturity date \( T_K \leq T \) gives the holder the right to buy, at each time \( t \in T \), the option \( v_{T,H} \) for a price \( Y \) (and similarly for a Bermudan compound put option).

(a) Extend the option object to include a Bermudan compound option. Are any changes required on the structural side to enable you to do this?
(b) Is the application capable of constructing compound options that exercise into other compound options?

(c) Is the PDE application as it stands able to value these options? Amend it so that it is capable of doing so.

4. Lattice stream. Is the lattice application able to value the compound options you constructed in exercise 3? What changes are required to it, or to the option object, so that it can?

5. Lattice stream. The binomial lattice method is sometimes used to price options. In the binomial branching scheme a value \( \hat{S} \) for the discrete lattice process at time \( t_i \) branches to two values \( \hat{S}^j \), \( j = -1, 1 \), at time \( t_{i+1} \),

\[
\hat{S}^j = \hat{S} \exp \left( \left( r - \frac{1}{2} \sigma^2 \right) \Delta t + \sigma z^j \right), \quad j = -1, 1,
\]

where \( z^j = j \sqrt{\Delta t} \). Branching probabilities are \( p^1 = p^{-1} = \frac{1}{2} \).

Implement the binomial lattice as a conforming object in the fully polymorphic lattice application. What particular issues did you find in implementing the binomial method?
We have seen in Chapter 12 how a fully polymorphic factory can be constructed. In this chapter we consider what could be done if it were not possible to use the VBE object. This could happen if, for instance, an organization forbade its programmers from using it, or if it enforced the use of a virus checker that removed any module that contained code that used the VBE object.

As long as a factory can be informed of the existence of the objects in an application, then it can be made fully polymorphic. Objects could register with the factory, and the factory would be able to create objects of that type. Unfortunately without the use of the VBE object this does not seem to be possible in VBA 6.X.

Although this means that without the VBE object a fully polymorphic factory cannot be built, we can nevertheless come a little closer to one than we did in Chapter 8. Only a partial solution is presented here: we shall allow the factory to know about interfaces and for each interface to instantiate a special object that has a knowledge of all objects of that interface type. The factory itself is decoupled from a knowledge of any derived class.

The most obvious special object is the interface object itself. Unfortunately putting code into an interface class is rather fragile, and may not be supported. However one can still define a meta-class object whose purpose is to proxy the role that an interface class could be playing. A meta-class holds information common to a group of objects as a whole, rather than data specific to a single object. In C++ this is accomplished through the use of Static data members, unavailable to VBA. We have to mimic that facility.

The application described in this chapter is significantly more complex than those of previous chapters. The introduction of meta-class data and the more complicated factory (but less functional) make this chapter challenging, and indeed it can be skipped on a first reading; however, the benefit is well worth the effort. Not only does it reveal quite a bit about using VBA, and about design patterns in VBA, it also fortunately helps to support the underlying functionality of the Monte Carlo application.

We first look at the structure of the application as a whole, then at the meta-class objects and finally the factory. We refer to it as the level 6a application.

### 13.1 THE STRUCTURE OF THE APPLICATION

This chapter’s version of the application, MC_example_v6a.xls, has the structure shown in Figure 13.1 (we have subsumed the decorated application into the AppMC box). Referring to Figure 3.15 (page 36) it is clear that we have come a long way indeed. The level 6a application has all the layers shown in Figure 2.1 (page 20) – although, perhaps, in a slightly different configuration.\(^1\) Compared to Figure 3.15 we are beginning to get quite powerful. But this is as far as it goes. In succeeding chapters we focus on the numerical side and leave the system architecture alone.

The individual components are discussed in more detail below. In outline, a Factory object is used by AppObWrapper. It has an Environment object that reads in from a hard-wired location a specification of where the numerics configuration can be found, and creates an object to read it in. It passes the Monte Carlo

---

\(^1\)All errors are caught by the error handler; Figure 13.1 shows only the link to the error handling from main().
configuration and the specification of the data input and results output streams over to the factory in the factory constructor. The factory reads in the numerics configuration and creates both the Monte Carlo application and the input/output channels for data and results.

The input side of the level 6a front-end is shown in Figure 13.2. This makes explicit the division of inputs between the environment side, the numerics configuration side and the option and process data side.

The factory is declared and used in the AppObWrapper object (Figure 13.3). As in Chapter 6, the factory exists only in the AppObWrapper’s constructor. The factory (i) creates the Monte Carlo application object according the specification it reads in and (ii) sets up the data input and results output channels for the Monte Carlo method.

The factory creates a direct link between the AppObWrapper object and the IApp object, so that AppObWrapper::run() calls IApp::run() directly.

### 13.2 META-CLASS OBJECTS

An essential component of the design is the construction of a meta-class for each interface type. Every interface class will have associated with it a meta-class object. By convention, the name of the meta-class will be prefixed by the letter ‘M’ so, for instance, the meta-class associated with the interface class IOption is MOption.

The meta-class has two distinct roles. First, it can provide services to the interface conforming objects, such as supplying unique ID numbers, or reference counting. Second, it can provide services to external clients, such as creating objects of its interface type. Conceptually the role played by the meta-class is analogous to that played by the interface: the interface specifies the signatures of procedures common

---

2 Which should perhaps be separated out into distinct objects.
A Semi-Polymorphic Factory: Meta-Classes

Figure 13.2  The level 6a front-end (input side)

```
Private Sub Class_Initialize()
    Set wch_ = New StopWatch
    Dim fct_ As Factory: Set fct_ = New Factory
    Set App_ = fct_.ObjectRef("OuterApp")
    Set Out_ = fct_.StreamRef("Outputter")
    Set fct_ = Nothing
End Sub
```

Figure 13.3  AppObWrapper: Declaring and setting up the Factory
to conforming objects, the meta-class specifies the common data. The meta-class effectively determines additional components of the object interface not supplied by the interface object itself.

This is illustrated in Figure 13.4. Clients communicate with objects via their interface. Part of the interface is now there because of the meta-object. Clients of conforming objects cannot communicate directly with the meta-object, only via the conforming object. The meta-object does not (in this design) have direct access to each conforming object (by having references to them).

In our design there are only two clients of the meta-class objects: the factory and the conforming objects. Figures 13.5 and 13.6 give the meta-class definition of $M_{\text{Option}}$ for the option interface $I_{\text{Option}}$. Corresponding to the two clients it has two categories of material within it.

**Material used by the factory**

There are four items used by the factory to create conforming objects.

1. Information about the meta-class. Its name is held in $\text{name}_\text{}$ (line 13.5g).
2. Instantiations of each derived class (lines 13.5a and 13.5b).
3. A $\text{Create}()$ Function (Figure 13.6) that manufactures a $\text{New}$ object on request.
4. Information about the derived classes as a group: a $\text{String}$, $\text{type_string}_\text{}$ holding identification characters for each type (line 13.5c).

The meta-class holds instances of every object derived from its interface. These are used in two ways. First, to supply information about themselves; and, second, whenever a new object of some type is required the meta-class asks its tame instance to supply it.

**Material used by the conforming objects**

These are items used by conforming objects to help form part of their state, or to provide some other service. The example here implements tracking (other interfaces may want to use other sorts of non-factory meta-data). It has:

1. Support data for each derived class, such as an instance counter (lines 13.5e and 13.5f).
2. A counter $\text{ID}_\text{}$, to enable each instantiation to have a unique identifier (line 13.5d).

The constructor for the meta-class is given in Figure 13.5. It does the obvious setting up. Prototype conforming objects are instantiated (but do not have their $\text{SetValues}()$ method called), $\text{name}_\text{}$ and $\text{type_string}_\text{}$ are set-up, and the counters are initialized.
Figure 13.5  The option meta-class, MOption (a)

Figure 13.6  The option meta-class, MOption (b)
Each conforming class has an identification character. When the user specifies in the front-end what sort of object is required conforming to some interface, this is the character that identifies the particular conforming class. The type_string_ is set up on lines 13.5h, 13.5i, and 13.5j to contain the concatenation of the identification characters from each conforming class. type_string_ is used here to validate the user’s choice of derived class during input.

The MOption::Create() method is displayed in Figure 13.6. It takes the factory as an argument so that it has access to its input streams. It passes to the factory its name. The factory goes away and comes back with a character, type_char, that the user has supplied, denoting the choice of conforming object the user wants. A check is made that type_char is valid, that is, it is a character in type_string_ (using the validation utility function check_char()), then type_char is fed to the Select statement that calls upon the relevant prototype conforming object to return a reference to a new instantiation of itself.

If more than one object of the same type is required – for instance, the outer and inner application objects – then the ObjectRef() procedure in the relevant meta-class must have a separate line for each one.3

13.3 THE SEMI-POLYMORPHIC FACTORY

Semi-polymorphic object instantiation is a bit more interesting than what we have seen previously. In Chapters 6 and 8 the factory created all (conforming) objects. Here, as we have seen, it is the meta-class that intermediates object creation. The meta-class maintains an instantiation of each of its conforming classes and the factory instantiates an object of each meta-class type. It uses its knowledge of them to instantiate objects on request.

The factory is responsible for the management of a complete set of numerics meta-classes. It delegates this responsibility to a MetaManager object, a meta-class of meta-classes.

The meta-classes should be singletons and implement the singleton idiom. The MetaManager could supply the necessary functionality but for brevity, and to avoid further complexity, we do not do so here, preferring to use the OnceOnly object instead (see Figure 13.15, page 223). Implementing the singleton idiom would be very reasonable but it would perhaps obscure yet further our underlying objective: writing fast and maintainable Monte Carlo applications.

13.3.1 Semi-polymorphic instantiation

The structure of the semi-polymorphic instantiation process is shown in Figure 13.7. When the factory receives a request for an object it passes it on to the meta-class manager which sends it on to the relevant composited meta-class object. In turn this dispatches the request to the relevant composited conforming prototype object. This instantiates a new version of itself and passes a reference back through to the client. The factory and meta-class act as switching stations to forward the request on to the right prototype object.

The prototype class has to create a brand-new squeaky-clean instantiation; unlike C++ it cannot clone itself. In C++ objects routinely implement a clone() function that instantiates a copy of themselves. This is extremely useful. In VBA 6.X an object’s Private procedures and data are not visible even to an object inside its own class definition. This makes it impossible to clone an object without breaking encapsulation. Figure 13.8 illustrates some code that does not work in VBA. The attempt to access the Private data_, fails, even from inside AnObject’s definition. In C++ the analogous code does not fail.

In our context the create function is used only to instantiate a New object of the same type, call SetValues() on it, and return a reference to it.

---

3 This is a limitation of the particular design but can be overcome with greater care.
13.3.2 A swap function

A standard utility in C++ is a `swap()` function. It is an important component of exception-safe coding in C++. This does not translate exactly into an issue for VBA. Nevertheless, being able to interchange two references (to objects of the same conforming type) is useful. Figure 13.9 illustrates. Initially reference A points to object 1, and reference B to object 2 (solid lines); we want to end up with reference A pointing to object 2 and reference B to object 1 (dashed lines). The utility `Sub` in Figure 13.10 exchanges two
Implementing Models of Financial Derivatives

Friend Sub swap(Aref As IOption, Bref As IOption)
    Dim temp As IOption
    Set temp = Aref
    Set Aref = Bref
    Set Bref = temp
    Set temp = Nothing
End Sub

Figure 13.10 A swap utility

IOption references. Instead of having a single library utility taking Objects as its arguments we prefer to have a separate swap() Sub for each interface. It is natural to put its definition into the meta-class object MOption.

13.3.3 The Factory object

The role of the factory is to create objects. Conceptually objects come in two sorts: I/O objects and application objects. The factory reflects this division by containing two composite objects, one to look after each category of object. The NumericsStreams object manages the I/O streams and the MetaManager object the application objects.

The Factory is shown in Figure 13.11. It should be a singleton. This could be enforced in a separate meta-class but we do not do so here.

The factory is a switching station. Almost the only thing the factory does is to dispatch requests on to its composite sub-objects. Clients come in two sorts: those that request objects and those that request parameter and character values. The factory creates objects on demand, delegated to the MetaManager, and returns values from the input stream via the NumericsStreams object.

The whole process of object creation is a cascade. When an object is created the factory is passed to it as an argument. It then requests the factory for its parameter values and uses the factory to create its own composited objects.

The Environment object

The Environment object (Figure 13.12) is created in the factory constructor. It reads in the location of the Monte Carlo specification (which will for us here always be the front-end). It creates an object of type IInputterApp to read in the specifications, and an IMonitor object. It passes references to these over to the factory in the factory constructor.

Summary of the Factory interface

The Property Gets that provide the factory’s functionality are summarized in Table 13.1. ObjectRef() is the main factory method. It returns references to newly created objects. ObjectMetaRef() returns references to the meta-objects the factory contains and StreamRef() returns references to the I/O stream objects. StreamMetaRef() is included for completeness but is not used.

Getters for data – numeric parameter values and character-valued object type specifications – are summarized in Table 13.2. Value() and TypeChar() are used to return parameter values and characters to configure objects that the application operates upon. AppValue() and AppTypeChar() return the same for the application objects.

Further divisions would be sensible. Application objects divide into Monte Carlo method objects, process objects and option objects. Separating out the latter in particular would be a very good idea if a group of options were being valued together or in sequence.
Managing meta-classes

The object meta-class manager, MetaManager (Figures 13.13 and 13.14), is easy. It instantiates one of every application meta-class and configures them (a facility left as stubs in our application). These are the only instances of the meta-classes that exist in the application. It then has two responsibilities. The first is to return references to individual meta-classes (in MetaRef()); the second is to return references to new
Private name_ As String
Private Loc_char_ As String 'Location of numerics specs
Private Mon_char_ As String 'Monitor channel type
Private mmon_ As MMonitor 'Meta-object for Monitor
Private miap_ As MInputterApp 'Meta-object for Inputter
Private NC_ As Long

Private Sub Class_Initialize()
    name_ = "Environment"
    Call Read_in_data
    Set mmon_ = New MMonitor
    Set miap_ = New MInputterApp
End Sub

Private Sub Class_Terminate()
    Set mmon_ = Nothing
    Set miap_ = Nothing
End Sub

Public Property Get name() As String
    name = name_
End Property

Friend Property Get ObjectSpec(Name As String) As Object
    Select Case Name
        Case "InputterApp": Set ObjectSpec = miap_.Create(Me)
        Case "Monitor": Set ObjectSpec = mmon_.Create(Me)
        Case Else: Call RaiseError(1117, "EnvironmentObj", "Invalid name")
    End Select
End Property

Friend Property Get Value(Name As String) As Double
    Select Case Name
        Case "CounterInterval": Value = NC_
        Case Else: Call RaiseError(1117, "EnvObj", "Invalid name")
    End Select
End Property

Friend Property Get TypeChar(Name As String) As String
    Select Case Name
        Case "InputterApp": TypeChar = Loc_char_
        Case "Monitor": TypeChar = Mon_char_
        Case Else: Call RaiseError(1117, "EnvironmentObj", "Invalid name")
    End Select
End Property

Private Sub Read_in_data()
    Loc_char_ = get_char(14, 4, "") 'Application specs location
    Mon_char_ = get_char(17, 4, "") 'Monitor channel type
    NC_ = Check_strictly_positive(get_long(18, 4), "counter interval")
End Sub

Figure 13.12 The Environment object

Table 13.1 Getters for object references

<table>
<thead>
<tr>
<th>Return object type</th>
<th>Return object category type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>I/O</td>
</tr>
<tr>
<td>Standard object</td>
<td>ObjectRef()</td>
</tr>
<tr>
<td>MetaObject</td>
<td>ObjectMetaRef()</td>
</tr>
</tbody>
</table>
instances of a conforming type when asked to do so (in \texttt{ObjectRef()}). It just forwards these requests on to each meta-class.

The \texttt{MetaManager} should be configured only once. The \texttt{OnceOnly} object is a utility object that facilitates the enforcement of a once only execution condition on a procedure. If an attempt is made to run a protected procedure a second time \texttt{OnceOnly} causes an error to be raised. It uses the \texttt{Dictionary} object\footnote{See chapter 16.}.
Implementing Models of Financial Derivatives

but not so much as an associative container, more as a set. It exploits the speed of the Dictionary in determining whether a key exists.

OnceOnly is shown in Figure 13.15. The Dictionary object is used to register separate once-only instances. An example of using it is given by the Sub, MetaManager::configure(). It contains the line

\begin{center}
\texttt{Call once\_check("configure", 1117, "MetaManager.configure()", "Run twice")}.
\end{center}

This calls once\_’s check() method. Inside check(), once\_ looks to see if the key, "configure", is already present in its Dictionary, dict\_. If it is it throws; if it is not it adds the key to the Dictionary. The item keyed by the key is a dummy, unused in this implementation.

The numeric streams manager is slightly more complicated (Figure 13.16). It has meta-classes minp\_ and mout\_ for the input and output interfaces, and instantiates objects optI\_ and optO\_ for input and output. It has getters for these objects, and two other getters – one for parameter values and one for type characters – that just forward requests on to the input object.

\textbf{Object instantiation}

When a client requests an application object from the factory it calls the Factory::ObjectRef() Property Get, passing it a String argument containing the name of the type it needs. ObjectRef() dispatches the request on to the MetaManager, which dispatches on to the right meta-class.
In the Factory
Object creation is one of the very few places that we put up with Objects. Because it can return objects of any type the Factory::ObjectRef() method returns a reference to a New Object. In other circumstances, where many objects are being shuttled to and fro, returning an Object could be expensive. In our numerical applications it is only a slight hit incurred while setting up the application since the vast majority of time is expended on computation. If many options at a time are being valued, then creating a fresh option object every time could become more serious. In this case a prototype pattern could be used. This creates a small number of objects and just hands out references to them, avoiding the need for a potentially expensive creation step. We do not explore this any further.

In the MetaManager
This contains a meta-class object for every interface type that it has to return. In VBA 6.X, if access to the VBE object is not possible, there is no real choice but to have a nasty non-polymorphic Select statement in the MetaManager (Figure 13.14) to dispatch the request to the correct meta-class object.

The argument must be identical to the name of a meta-class, via telepathy, or ObjectRef() throws. The client has to hard-wire in the name. The dead give-away that all is not well, apart from the Select statement itself, is the presence of a Case Else line. This is a confession that (i) there is no object registration in the application and (ii) the application cannot detect errors at compile-time and can catch them only at run-time.

ObjectRef() calls the Create() method of the corresponding the meta-class. The function MOption::Create, for instance (Figure 13.6, page 215), then returns a reference to an object instantiated by one of the prototype objects it contains. This is passed back to the factory so that object creation can propagate onwards. The Create() function has the usual distasteful Select construction but again it is hard to see how this could be removed under the conditions we have imposed upon ourselves.

In the conforming class
Classes are now expected to satisfy an extended contract. We look in detail at the option object classes to see how the new architecture works its way through. The new IOption interface is shown in Figure 13.17. There are three vanilla getters for the object name, its identifier character and its ID. The setter,
SetValues(), will now have an additional task of setting a reference in the conforming class to its meta-class. Conforming objects thus have access to their meta-class. There is also now a Create() method in the interface.

The option functionality is provided by the interface methods ReceiveNextSlice() and FetchSlice(). Initialise() is used to reset the option.
An example of a conforming option class, OptionEuro, is given in Figures 13.18 and 13.19. It contains a reference to its meta-class and, for the purpose of this example, has been endowed with tracking.\(^6\)

It is the Create() function (Figure 13.19) which concerns us here. It instantiates a New OptionEuro and then calls its SetValues() Property. This is called OnceOnly() to set the object’s meta-class and its ID, to create its composited object, and to set its parameter values. It sees nothing of the complexities hidden inside the factory, thank heavens.

### The IInputter objects

There are two objects in this group. InputterApp reads in the Monte Carlo specifications and InputterDataFrontEnd reads in option specifications, each from the front-end. Since the group is polymorphic further objects could be added to enable each set of specifications to be read in from anywhere.

The IInputters have much the same functionality as the InputManager object of previous chapters. Here, to make the code more compact and much more legible, single functions return parameter values and characters. They each take a String, the name of the item being requested, as their argument. There is more or less the same degree of telepathy here as when there is a separate function for each data item.\(^7\)

The main procedures are Value() and TypeChar(). These return numerical parameter values and type characters respectively. Figure 13.20 shows these procedures for the InputterApp object. The InputterDataFrontEnd object is similar and is not shown.

The TypeChar() Property returns a character denoting the object the user has selected. The String passed as an argument is the name of the corresponding meta-class. There is a psychic connection between the meta-class and the TypeChar() method.

---

\(^6\)A numerical method is likely to be applied in turn to successive option objects so it is not unreasonable to suppose that option objects may want to use tracking. Other objects are either too insignificant (for instance, the payoff objects) or too fixed (for instance, the Monte Carlo application objects) to warrant tracking.

\(^7\)A more elegant implementation might involve storing values in a Dictionary object.
Figure 13.18  The OptionEuro option: non-structural

Extending the functionality

What does it take to make changes of various sorts? We look at additional input and output channels, adding objects to an existing interface, adding additional parameters, and adding in new interfaces.

Adding objects to an existing interface

Create a new conforming class. Add it into its meta-class: (i) add in a prototype instantiation, (ii) add its identification character to the type_string_, (iii) add a Case to the Select statement in the Create() method, (iv) update the front-end.

Additional input and output channels

Create a new conforming class, as above. If the inputter reads in the numerical configuration then the Environment object has to be amended. Otherwise the changes are confined either to the MInputter meta-class or to the MOutputter meta-class.
Private Sub Class_Initialize()
    name_ = "European option"
    type_char_ = "e"
    ID_number_ = -1
    done_ = False
    Set once_ = New OnceOnly
End Sub

Private Sub Class_Terminate()
    If ID_number_ <> -1 Then Call meta_.OptionDies(ID_number_, type_char_)
    Set meta_ = Nothing
    Set pay_ = Nothing
    Set once_ = Nothing
End Sub

Friend Property Get IOption_Create(ByRef fact As Factory) As IOption
    Set IOption_Create = New OptionEuro
    IOption_Create.SetValues = fact
End Property

Friend Property Let IOption_SetValues(fact As Factory)
    Call once_.Check("SetValues", 1117, "OptionEuro", "Already created")
    Set meta_ = fact.ObjectMetaRef("Option"): ID_number_ = meta_.ID
    Set pay_ = fact.ObjectRef("Payoff")
    T_ = fact.Value("T")
    M_ = fact.AppValue("NPaths")
    ReDim XValues_(1 To M_) As Double
End Property

Friend Property Get IInputterApp_Value(name As String) As Double
    Select Case name
        Case "NSteps": IInputterApp_Value = Nsteps_
        Case "NPaths": IInputterApp_Value = Npaths_
        Case "NRuns": IInputterApp_Value = Nruns_
        Case Else: Call RaiseError(1117, "InputterApp", "Invalid name")
    End Select
End Property

Friend Property Get IInputterApp_TypeChar(name As String) As String
    Select Case name
        Case "OuterApp": IInputterApp_TypeChar = OuterApp_type_
        Case "InnerApp": IInputterApp_TypeChar = InnerApp_type_
        Case "OuterAcc": IInputterApp_TypeChar = OuterAcc_type_
        Case "InnerAcc": IInputterApp_TypeChar = InnerAcc_type_
        Case "Evolver": IInputterApp_TypeChar = Evolver_type_
        Case "Wiener": IInputterApp_TypeChar = Wiener_type_
        Case "Inputter": IInputterApp_TypeChar = In_type_
        Case "Outputter": IInputterApp_TypeChar = Out_type_
        Case Else: Call RaiseError(1117, "InputterApp", "Invalid name")
    End Select
End Property

Figure 13.19 The OptionEuro option: structural

Figure 13.20 The InputterApp object Value() and TypeChar()
Adding additional parameters
Adding additional parameters is straightforward. (We suppose that input is from the front-end but if input is from elsewhere the procedure is just as easy.) Find a place for them to be read from on the front-end; give them names, and adapt the `InputterDataFrontEnd` object; amend client objects to request them by name from the factory.

Adding new (application) interfaces
This could be necessary if new options or processes came along that do not fit into the simple framework presented here. The procedure is:

1. Create the new I-class and M-class objects for the new interface, and amend the `MetaManager`.
2. Create a selection box for the new conforming objects on the front-end.
3. Adapt the `InputterDataFrontEnd` object to read in the new type character.
4. Finally change the application code to request the factory for instantiations of them.

Taken together the procedure should be significantly easier than repeatedly adapting the level 1 application.

13.4 SUMMARY

This chapter has covered a great deal of ground. The level 6a Monte Carlo is significantly more complex than the level 5 version, or indeed the level 6 version. To make the factory semi-polymorphic, avoiding the use of the `VBE` object library, we introduced the notion of meta-classes and implemented these to make object manufacture as polymorphic as possible within the constraints imposed upon us.

We decoupled from one another data input, results output, and the specification of the numerical configuration. Although we continue to value one option at a time, and all input is from the front-end, the design is sufficiently flexible to accept input from elsewhere, in particular, from file.

It is true that the application is certainly not fully polymorphic – there is a fair amount of psychic coupling and few changes are completely local – nevertheless it is flexible and adaptable to changing requirements. At level 6a the procedure to make changes may indeed seem complex, but you have only to make them once, and without too much thought. You are at less risk of introducing additional mistakes that may show up only in special circumstances. Nonetheless, the contrast with the fully polymorphic factory of level 6 is salutary; there no changes of any type were needed. Polymorphism meant that everything was dealt with automatically.

At this stage we feel that the pursuit of an object-oriented VBA platform may have gone far enough for the time being, and gratefully turn to some numerics.

13.5 EXERCISES

1. Add in reference counting to the fully polymorphic factory of Chapter 12. You may wish to create a separate factory environment file that enables the client to separately configure the factory. For instance the factory environment file could specify whether or not the factory was to use object tracking.

2. A standard module may be regarded and used as a singleton object (see Appendix E). How might you exploit this conceptual re-alignment to develop an alternative mechanism for mimicking meta-class data?
3. We saw in Chapter 6 that it is possible, though perhaps fragile and not advisable, to provide bodies for procedures in interface classes. Is it possible to exploit this feature to develop a further route towards implementing a meta-class data mechanism?

4. Meta-class data must be stored somewhere. Exercises 2 and 3 suggested storing it either in standard modules or in interface classes.

   (a) Is it possible to devise a method of holding meta-class data in files?

   (b) Are there any other possible storage locations that a meta-class data implementation mechanism could exploit?

5. Can the provision of meta-class services be automated?

   Is it feasible to create an IMetaClassData interface – which objects requiring meta-class data would have to conform to – that could be detected by a VBIDE factory which could then, somehow, ensure that the interface was correctly (and transparently) implemented?
This page intentionally left blank
Part V  Performance Issues in VBA

This part gives a discussion of performance and data representation issues surrounding the Monte Carlo application. Detailing costings are established and comparisons made at a variety of different levels.

**Chapter 14** is concerned with a low-level VBA programming technique. It looks at the costs of arithmetic operations and alternative programming mechanisms for low-level constructions.

**Chapter 15** compares the performance of the various versions of the Monte Carlo application developed so far in this book. It turns out that going polymorphic and using factories is not too expensive: there is no need on cost grounds to worry about introducing them.

Finally **Chapter 16** looks at the way that a Monte Carlo application can represent slices and states in VBA. The chapter is an excuse to discuss **Collections** and **Dictionaries**, although the timing tests reject their use in numerical code in favour of ordinary arrays and user-defined types.

A number of spreadsheets were used to produce the results in this part. An index of the spreadsheets used in this book is included after the appendices. It indicates which spreadsheets contributed to each table in this part.

For no particular reason the plain **StopWatch** object was used to obtain the results presented in this part. The reader may prefer to use instead one of the other stopwatches presented in Chapter 6.
This page intentionally left blank
Performance here means speed. Speed is important, especially for numerical applications, but not for everything. Some items of code have a clear encapsulated atomic functionary, for instance an implementation of an algorithm to compute the inverse of the beta distribution function. Libraries for procedures like these can be optimized as much as you like, the more the better, and fittingly so, but for components of a larger application considerably more care needs to be taken.

There is a saying about optimizing code: The first rule is don’t do it; the second is don’t do it anyway. It is certainly true that premature optimization is a big mistake, but *ceteris paribus* there is no reason not to choose the faster of two otherwise identical ways of doing something. Some features of VBA are known to be slow and there may be alternatives that are faster and just as clear. Where they exist there is no reason not use these cheap speed-ups.

In practice things are rarely equal; when the slower way is clearer and cleaner you may well prefer to use it instead of the faster way. In any case it is clearly bad to wring out the last millisecond of speed if it makes the code difficult to follow.

There are four factors contributing to the performance of a Monte Carlo method. The first is what we have referred to in this book as its level; the second is the way that the data being evolved in the Monte Carlo method is represented in the language; the third is coding nitty-gritty; and the fourth is the Monte Carlo method itself. Examples of each factor are given in Table 14.1.

A simple coding rule is never to use $u^2$. It is far better to write $u * u$ instead. The power operator is heavy-duty and general, and in VBA is not optimal for small integral powers; repeated multiplication is much cheaper.

The focus of speed-up methods is often directed at the Monte Carlo method itself, with techniques like stratified sampling. Of course we explore these later on; however, as our time-trials show, coding and design decisions play a significant, and sometimes crucial, role.

Simply doing a bit of precomputing (for instance, of a drift coefficient) can result in huge speed-ups. At a higher level, evolving $\ln(S_t)$ instead of $S_t$ (to avoid a repeated $\exp()$ operation), can help tremendously.

Later we compare several different data-structure representations. It will not be a surprise when we find that evolving data in an array is both fast and clear.

The level appropriate for an application depends heavily on its ultimate purpose. Fully OOP applications are likely to be slowest, but not by much once you get beyond level 3, and their extensibility outweights their complexity in many non-stand-alone situations.

Features that are specific to the VBA implementation, unconnected to the Monte Carlo method, are the level of the application and, at the other extreme, the coding style. Monte Carlo speed-ups are purely method dependent. The choice of data type sits in between method and coding, but can make a considerable difference to performance.

In this chapter we look in detail at the cost, and the possible savings, in using various VBA coding nitty-gritty features. In Chapter 15 we look at the effect of level, and in Chapter 16 we look at data types. We run performance checks on the Monte Carlo methods themselves in Part VI.

The timings in the tables in this chapter are given in seconds.
Table 14.1 Determinants of an application’s performance

<table>
<thead>
<tr>
<th>Determinant</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level</td>
<td>OOP polymorphic code instead of procedural code.</td>
</tr>
<tr>
<td>Data representation</td>
<td>Evolving a vector instead of a scalar.</td>
</tr>
<tr>
<td>Coding</td>
<td>Using $u * u$ instead of $u^2$.</td>
</tr>
<tr>
<td>MC speed-up</td>
<td>Using stratified sampling or antithetic variates.</td>
</tr>
</tbody>
</table>

A cautionary note

One thing should be stressed. The results produced in this part are specific to the software and hardware used to compute them. Not only absolute times but also relative times vary from one implementation to another. The results should therefore be regarded as illustrative rather than authoritative. Although the broad conclusions are likely to remain valid, readers should nevertheless duplicate the tests on their own platforms before committing one way or another in a sensitive application.

Categories of coding issues

Detailed coding issues can be grouped loosely into four categories.

1. Arithmetic operations
2. Logical flow
3. Procedure calls
4. Data typing and casting.

We look at arithmetic and logical flow in section 14.1, procedures in section 14.2, and finally data typing and casting issues in section 14.3.

If code needs to be maintained, and what does not, some care must to be taken not to pursue the nitty-gritty too far since really fast code can be ‘tricky’. Precomputing is an obvious speed-up, but how much time can be saved by doing even simple things? Category 1, arithmetic operations, is well and truly part of coding nitty-gritty. Logical flow is another aspect of coding; how much does a control structure cost?

The biggest set of issues surrounds data representation. Here we include typing and the relative cost of using Types. We find that we must heed these issues; not to do so incurs expense, but just being sensible and typing everything as tightly as possible gets you most of the way there.

The timing harness

To conduct our timing test we use the stub `main()` shown in Figure 14.1. It loops around $N$ times, with $N = 10^8$ unless otherwise stated, timing the execution of the test line. Times are recorded using the `StopWatch` object we saw in section 5.2.

When both $i$ and $N$ are `Long` the loop itself, with empty content, takes around 1.12 seconds for $10^8$ iterations (see the discussion around Table 14.12, page 241). Unless otherwise stated, all times are adjusted by having the loop time removed.

---

1 Timings here were made on a Dell D610 Latitude laptop with a 2.00 GHz processor running Windows XP 2002. VBA was 6.5 version 1024 with Excel 2007.
Sub main()
    Dim N As Long: N = Cells(7, 5).Value 'Usually N = 10^8
    Dim clock As StopWatch: Set clock = New StopWatch
    Dim a As Double: a = 2.5: Dim b As Double: b = 3.5 'Or whatever
    Dim c As Double: c = 7.5: Dim x As Double: x = 7.5 'Or whatever
    Dim f As Double 'Or whatever
    Call clock.Start_Timer
    Dim i As Long
    For i = 1 To N
        f = a + b + c + x 'The test line to be timed
    Next i
    Call clock.Stop_Timer
    Cells(8, 5).Value = clock.Elapsed_Time
End Sub

Figure 14.1  Stub for timing

Unlike some C++ compilers VBA does not appear to perform any optimization on loop contents. The test line really is executed N times. In the current version of VBA the Timer returns times in 64ths of a second. Reported times are the averages of a number (normally three) separate timing runs.

Although times are reported to 2 decimal places ordinary variation on individual runs can be ±0.05 seconds or so. However, there can be much greater, surprisingly wide, variation in timings depending on the context and environment of the test code, and on which side of bed it got out of that morning. The times reported below are the fastest times consistently achievable for each test. Some effort has been made to make them as reliable as possible but some anomalies may still exist.

Before getting involved in detailed timings we discuss the performance of the StopWatch object itself. We use the StopWatch object extensively in our timing runs so it makes sense to establish at an early stage how expensive, or cheap, it is to use.

Using the StopWatch object

To gauge the effectiveness of the StopWatch object we use two instantiations in the stub main(); an outer StopWatch to record the time of an entire run, and an inner StopWatch inside the For loop to switch itself on and off 10^8 times (see Figure 14.2). The outer StopWatch measures the cycle time of the inner StopWatch. We find that the outer StopWatch records ~99 seconds and the inner StopWatch records ~48 seconds. Subtracting away the loop time, this means that the cycle time for stopping and starting the StopWatch is ~98 seconds. Roughly 48 seconds are spent in between exiting Start_Timer() after recording the time, up to the moment that elapsed_time_ is computed in Stop_Timer(). It takes about 50 seconds to go from setting elapsed_time_ back to setting current_begin_time_ in Start_Timer() (refer to Figure 5.5, page 64).

These times dwarf the time taken to execute a line of arithmetical computation, but that does not matter too much. The StopWatch object should not be used to profile at the level of individual lines of code, but

Figure 14.2  Outer and inner stop watches

2 For instance, the order of declaration of variables entering into test expressions can make a difference to execution time.
used at scales next to which its cycle time is insignificant. Fortunately a nice chunky piece of numerical code, such as a loop taking \(\sim 5\) seconds to run, operates at scales where it is perfectly reasonable to use \texttt{StopWatch}. A time of 100 seconds for \(10^8\) replications is \(10^{-6}\) seconds for a single cycle, which is negligible compared to the time taken by the test code.

Reassured, we proceed to use the \texttt{StopWatch} object in the rest of this chapter.

## 14.1 ARITHMETIC OPERATIONS

In this section we look at some basic operations and the potential for speed-ups or, at least, the avoidance of slow-downs. The timings in the tables in this chapter are in seconds.

### Comparisons of arithmetic operations

Table 14.2 gives costs for arithmetic operations, and for some compositions. All variables have been declared \texttt{Double} so the results are unaffected by type conversions.\(^3\) Times include the cost of an assignment of the result of an expression into a variable on the left-hand side (this costs 1.07 seconds; see Table 14.17, page 244).

We see from Table 14.2 that addition and multiplication cost much the same (despite \(a \times b \times c\) appearing to be cheaper than \(a + b + c\)). If there is a difference it is surprisingly slight. Subtraction is about the same cost as addition but division is much more expensive than the other three operations.

Taking away the cost of the assignment, the first operation in a statement costs about 0.60; the second costs an additional 0.30 or so, and the third an additional 0.23 or so.

Casting makes a difference (Table 14.3). Executing \(f = 2 \times a\) costs 1.99, whereas \(f = 2\# \times a\) costs only 1.71. Multiplying by a \texttt{Const Double} (denoted by \texttt{Two} in the table) is a little more expensive than multiplying by a non-\texttt{Const Double} when it prefixes \(a\), but is about the same when it suffixes \(a\).

Division is significantly more expensive than multiplication and addition. Executing a general \(f = a / b\) costs 3.26, but this varies considerably. Table 14.4 has execution times for various divisions. Each entry in the table has two parts: the first part is the division that is timed, the part in brackets is

### Table 14.2 Comparative costs of arithmetic operators

<table>
<thead>
<tr>
<th>Expression</th>
<th>(\times)</th>
<th>(+)</th>
<th>(/)</th>
<th>(-)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(f = a \circ b)</td>
<td>1.68</td>
<td>1.69</td>
<td>3.26</td>
<td>1.67</td>
</tr>
<tr>
<td>(f = a \circ b \circ c)</td>
<td>2.04</td>
<td>2.31</td>
<td>5.33</td>
<td>1.97</td>
</tr>
<tr>
<td>(f = a \circ b \circ c \circ d)</td>
<td>2.27</td>
<td>2.19</td>
<td>7.22</td>
<td>2.21</td>
</tr>
</tbody>
</table>

### Table 14.3 Special cases: multiplication and literals, \(f = a \times b\)

<table>
<thead>
<tr>
<th>Expression</th>
<th>(\times) (0.25)</th>
<th>(\times) (0.5)</th>
<th>(\times) (2#)</th>
<th>(\times) (a)</th>
<th>(\times) (a)</th>
<th>(\times) (\text{Two})</th>
<th>(\times) (a)</th>
<th>(\times) (\text{Two})</th>
<th>(\times) (2)</th>
<th>(\times) (a)</th>
<th>(\times) (2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>1.77</td>
<td>1.70</td>
<td>1.71</td>
<td>1.64</td>
<td>2.11</td>
<td>1.63</td>
<td>1.99</td>
<td>2.07</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\(^3\)We note that computation times vary depending on the size of the \texttt{Double} (although relative times seem to remain roughly the same). Our comparisons use small values, typically between 1 and 10, although there can be significant variation even within that range.
the time taken. Divisions are separated into four empirically determined categories, based on speed of execution: fast, intermediate, and two levels of lethargy. On this platform execution time is reduced for some divisors (0.5, 1#, 2# – perhaps with a concordant internal representation) but is greater if casting is needed. It is cheaper to divide by 2# and 4# than it is to multiply by 0.5 and 0.25. Here, surprisingly, dividing by 0.5 (at 1.49) is cheaper than multiplying by 2# (at 1.64).

**The cost of an extra multiplication**

Consider two ways of computing the value of a quadratic polynomial, \( f(x) = ax^2 + bx + c \),

\[
\begin{align*}
f &= a \times x \times x + b \times x + c \quad \text{'a} \\
f &= c + x \times (b + a \times x) \quad \text{'b}.
\end{align*}
\]  

(14.1)

Version (14.1a) uses three multiplications and two additions; version (14.1b) uses only two multiplications and two additions (indicated in brackets alongside the results in Table 14.5). Logic dictates that (14.1b) should be faster, but the difference turns out to be insignificant. Version (14.1a) costs around 2.85 while (14.1b) is 2.89, equivalent to within computational noise.

For a cubic polynomial the speed-up is worth talking about. The direct calculation uses six multiplications and three additions, while the nested form uses just three multiplications. The nested form is over 10% faster.

**The power operator, ^**

The power operator requires a separate section to itself. It is slow. Table 14.6 illustrates. When \( a \) and \( b \) are both Doubles, \( f = a^b \) costs 24.61 seconds; when \( b \) is a literal Long the cost is slightly greater. In these tests it is cheaper to use \( \text{Sqr}(a) \), at 19.03 seconds, than to use \( a^{0.5} \) at 24.61, and for low integer powers of \( a \), Table 14.2 shows that it is much, much, faster to use repeated multiplication.

**Table 14.4** Special cases: division. Variations in computation times, \( f = a/b \)

<table>
<thead>
<tr>
<th></th>
<th>( a / 0.5 ) (1.46)</th>
<th>( a / 1# ) (1.46)</th>
<th>( a / 2# ) (1.51)</th>
<th>( a / 4# ) (1.51)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fast:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intermediate:</td>
<td>( a / 1 ) (2.28)</td>
<td>( a / 2 ) (2.27)</td>
<td>( a / 4 ) (2.28)</td>
<td></td>
</tr>
<tr>
<td>Slow (1):</td>
<td>( a / 0.501 ) (2.99)</td>
<td>( a / 1.5 ) (2.97)</td>
<td>( a / 3.5 ) (2.98)</td>
<td></td>
</tr>
<tr>
<td>Slow (2):</td>
<td>( 0.5 / a ) (3.16)</td>
<td>( 1 / a ) (3.57)</td>
<td>( 1# / a ) (3.46)</td>
<td></td>
</tr>
</tbody>
</table>

**Table 14.5** Alternative computation of low-order polynomials

<table>
<thead>
<tr>
<th>Polynomial</th>
<th>Direct</th>
<th>Nested</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quadratic</td>
<td>2.85, (3, 2)</td>
<td>2.89, (2, 2)</td>
</tr>
<tr>
<td>Cubic</td>
<td>4.24, (6, 3)</td>
<td>3.70, (3, 3)</td>
</tr>
</tbody>
</table>

**Table 14.6** Computing powers

<table>
<thead>
<tr>
<th>Expression</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a^b )</td>
<td>24.61</td>
</tr>
<tr>
<td>( a^2 )</td>
<td>24.98</td>
</tr>
<tr>
<td>( a^3 )</td>
<td>24.98</td>
</tr>
<tr>
<td>( \exp(b \times \log(a)) )</td>
<td>22.11</td>
</tr>
<tr>
<td>( \text{pow}(a, b) )</td>
<td>36.19</td>
</tr>
<tr>
<td>( a^{0.5} )</td>
<td>24.61</td>
</tr>
<tr>
<td>( \text{Sqr}(a) )</td>
<td>19.03</td>
</tr>
</tbody>
</table>
It seems that $^\wedge$ is implemented internally in the long-hand form as $a^b = \exp(b \ln(a))$. In fact executing $f = \exp(b \times \log(a))$ is about 2.5 seconds cheaper than $f = a^b$. If you are tempted to parcel this expression into a function, $\text{pow}(a, b)$, don’t do it. Unfortunately the overhead of a function call puts the time up by 50%.

As an illustration, the polar rejection method computes $w = u^2 + v^2$ to see if $(u, v)$ lies in the unit circle. This could be computed either as (14.2a) or as (14.2b),

\[
\begin{align*}
w &= u \times u + v \times v \quad 'a \\
w &= u \wedge 2 + v \wedge 2 \quad 'b.
\end{align*}
\]

Table 14.20 in section 14.3 compares costs of generating normal variates by various methods. It computes the polar rejection method using (14.2a). If instead (14.2b) is used – in the case polar + ran0() + global for instance (see below) – the time increases by about 40%. This cost is intolerable in what needs to be a tight lean function.

The \texttt{Int()} function

The built-in function \texttt{Int(a)} returns the integer part of the \texttt{Double a} (as a \texttt{Double}), but if you want to round (down) a \texttt{Double} there are a number of other possibilities. Some of these are listed in Table 14.7. In every case the result is assigned to a \texttt{Long}.

The first column shows the cost just of an assignment of a \texttt{Double} to a \texttt{Long} (see Table 14.17, page 244). The next two columns show the effect of casting into a \texttt{Double}. The final five columns compare different functions that do the conversion in various ways. \texttt{Int()} truncates towards $-\infty$, \texttt{Fix()} towards 0; and \texttt{CLng()} rounds to the nearest integer.\footnote{If the fractional part is exactly 0.5, \texttt{CLng()} rounds towards the nearest even number.} Implicit conversion (assigning a \texttt{Double} to a \texttt{Long}) follows the same rules as \texttt{CLng()}. \texttt{Round()} is a more general function that rounds to a specified number of decimal places (rounding to the nearest valid value).

\texttt{CLng()} costs the same as direct assignment and costs less than half as much as \texttt{Int()}. \texttt{Int()} should be used only where its particular direction of rounding is required. \texttt{Round()} is too expensive to be used just to round down to an integer. For cheapness, safety and clarity, use \texttt{CLng()}. If the fractional part is exactly 0.5, \texttt{CLng()} rounds towards the nearest even number.

Table 14.8 compares several ways of returning a \texttt{Long} from the division of a \texttt{Double a} by a \texttt{Double b}. As before, the result is assigned to a \texttt{Long}. The results are consistent with those in Table 14.7.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|c|c|}
\hline
Expression & $f = a$ & $a \downarrow 1$ & $a \downarrow 1#$ & Int(a) & CLng(a) & CLng(a - 0.5) & Fix(a) & Round() \\
\hline
Time      & 2.42 & 3.00 & 5.02 & 5.87 & 2.42 & 2.74 & 5.89 & 11.15 \\
\hline
\end{tabular}
\caption{Rounding down a \texttt{Double}, $l = fn(a)$}
\end{table}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|c|}
\hline
$a \div b$ & Int($a \div b$) & CLng($a \div b$) & CLng($a \div b - 0.5$) & Fix($a \div b$) & Round($a \div b$) \\
\hline
5.72 & 5.04 & 8.83 & 5.43 & 5.69 & 8.80 & 14.04 \\
\hline
\end{tabular}
\caption{Rounding a division, $l = fn(a/b)$}
\end{table}
A little care is necessary to implement an If statement in a critical piece of code. The five lines (14.3a) to (14.3e) each compute the maximum of \( a \) and \( b \) in an in-line calculation:

\[
\begin{align*}
\text{If } a &> b \text{ Then } f = a \text{ Else } f = b & \quad \text{'}a \\
\text{If } a &\geq b \text{ Then } f = a \text{ Else } f = b & \quad \text{'}b \\
\text{If } a &< b \text{ Then } f = b \text{ Else } f = a & \quad \text{'}c \\
\text{If } a &\leq b \text{ Then } f = b \text{ Else } f = a & \quad \text{'}d \\
f = a; \text{ If } a < b \text{ Then } f = b & \quad \text{'}e
\end{align*}
\]  

(14.3)

For lines (14.3a) to (14.3d), instead of computing in-line, a block If-Else-End-If could be used. This gives a total of nine possible forms of If-statement. Table 14.9 gives times when \( a \), \( b \) and \( f \) are Doubles and the numerical value of \( a \) is less than that of \( b \).

Block-If should be used when several statements are toggled but on speed grounds it is a little poorer to use a Block-If than to use an in-line-If.

Which form of in-line-If should be used? If you expect a particular variable, \( a \) say, usually to be less than the other, \( b \), then the in-line-If (14.3a) or (14.3b) should be used. If you do not know which of the two variables is usually greater, then you are still probably better off using (14.3a).

Comparable tests for Longs are also possible. Results are given in Table 14.10 for the in-line case only. Perhaps not surprisingly it is cheaper (in this implementation) to compare Longs than Doubles. There is no difference between the strict and non-strict comparators; the difference between the case when \( a \) is less than \( b \) and when it is greater is the same as for Doubles, but here the \( a < b \) form is preferred to the \( a > b \) form.

Of course, you should not use IIF() in cost-sensitive code. This function can be useful, and is a bright idea, but is seemingly not well implemented. A function call

\[
f = \text{IIF}(a > b, a, b)
\]

(14.4)

took 21.23 seconds. Note that if \( a \) and \( b \) are expressions, IIF always evaluates them both, regardless of the value of the Boolean expression.

<table>
<thead>
<tr>
<th>Table 14.9</th>
<th>Cost of If statements: Doubles, when ( a &lt; b )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Form:</td>
<td>( a &gt; b )</td>
</tr>
<tr>
<td>In-line If</td>
<td>2.94</td>
</tr>
<tr>
<td>Block If</td>
<td>3.23</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 14.10</th>
<th>Cost of If statements: Longs, when ( a &lt; b )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Form:</td>
<td>( a &gt; b )</td>
</tr>
<tr>
<td>In-line If</td>
<td>2.01</td>
</tr>
</tbody>
</table>
The Select statement

The Select statement

```
Select Case a_char
    Case "a": Call Do_a
    Case "b": Call Do_b
    Case Else: Call Do_Else
End Select
```

(14.5)

is equivalent to the compound If statement

```
If a_char = "a" Then
    Call Do_a
Else If a_char = "b" Then
    Call Do_b
Else
    Call Do_Else
End If
```

(14.6)

Lines (14.5) are far more elegant than lines (14.6) but this elegance comes at a cost.

Table 14.11 presents cost comparisons between (14.5) and (14.6) depending on which case is selected. Do_a(), Do_b() and Do_Else() are empty stubs and the Select and compound If statements are wrapped inside a function call. Results are based on $10^7$ replications.

Running the Select statement costs at least around 40% more. This difference is very big. If this really makes no difference then prefer the Select statement but in cost-sensitive code you cannot afford to use Select.

Loops

We include tests of looping here. Normally the calculations that take place inside a loop are likely to dominate the cost of the loop itself. Nevertheless in tight loops the iteration time will make a difference.

There are two main types of loop: Do loops and For loops. For loops should always be used when the number of iterations is known in advance and early exit from the loop is not possible (or possible only exceptionally).

There are five basic variations on the Do loop: Do-While and Do-Until – each of which can test their condition at either the top or the bottom of the loop – provide four of the variations, and the Do-Exit-Loop, where a Do loop contains an If-Exit-Do statement, provides the fifth.\(^5\)

In our tests there is a loop counter, incremented inside the loop; the loop is otherwise empty. The loop exits when the required number of iterations has been made. We compare the time it takes for each loop

<table>
<thead>
<tr>
<th>Table 14.11</th>
<th>Select and compound If</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statement type</td>
<td>Position of selected case</td>
</tr>
<tr>
<td></td>
<td>First</td>
</tr>
<tr>
<td>Select</td>
<td>4.65</td>
</tr>
<tr>
<td>Compound If</td>
<td>2.83</td>
</tr>
</tbody>
</table>

\(^5\) There is also a While-Wend loop construction, but its behaviour and cost are equivalent to a Do-While loop with the condition at the top. We ignore this construction.
to cycle $10^8$ times. Results for the four main `Do` loops are given in Table 14.12. `Do-While` and `Do-Until` loops behave similarly; we see that it is slightly faster to test the loop condition at the bottom rather than at the top of the loop.

As might be expected, the `Do-Exit-Loop` performs worse (at 2.75 seconds). For a fixed number of iterations the `For` loop at 1.12 seconds is much better than any of the others. In fact the `Step` version,

$$\text{For } i = 1 \text{ To } N \text{ Step } 1 \text{ ',}$$

is even faster at 0.95 seconds.

In a conditional loop where the number of iterations is not fixed beforehand, the `Do` loop forms with the condition at the bottom should be preferred *ceteris paribus*.

### Arithmetic functions

We give the cost of several important VBA arithmetic functions (taking `Double` as arguments and return values). Table 14.13 gives the costs of $f = \text{fn}(a)$ for various `fn()`s. The results are not too unexpected.

`Abs()` is very cheap, considering that the cost of 1.62 includes an assignment costing 1.07 all by itself. The time shown for `Sgn()` is for taking the sign of a positive `Double` (taking the sign of a negative `Double` costs 2.37). `Exp()` is expensive. Later we find that the cost of exponentiation is a major component of a method that naively evolves a GBM. However, note that `Exp()` costs only about eight times as much as a single multiplication. Compare this to the situation in C++. For DevCpp, a freeware C++ compiler, the library `std::exp()` function takes about 300 times longer than a single C++ multiplication.\(^6\)

### String concatenation

Although it is not an arithmetic operation, we lump this in here. There are two ways to concatenate a pair of `Strings`: using `&` or using `+`, and it makes little difference which is used. The test here concatenates two `Strings` each composed of half a dozen or so characters. Table 14.14 presents the comparison; there is no difference in times.

<table>
<thead>
<tr>
<th>Function</th>
<th>Exp()</th>
<th>Abs()</th>
<th>Sgn()</th>
<th>Log()</th>
<th>Sin()</th>
<th>Cos()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>13.29</td>
<td>1.62</td>
<td>2.48</td>
<td>8.65</td>
<td>7.72</td>
<td>8.12</td>
</tr>
</tbody>
</table>

\(^6\)But a single VBA multiplication takes about 300 times longer than a C++ multiplication, so the cost of exponentiation across platforms is not too different.
14.2 PROCEDURE CALLS

Functions encapsulate functionality. They have been around almost as long as programming. If you write a procedure, the code it contains can be re-used; you do not have to paste in a whole block of code every time you want the functionality it contains. There is however a cost, and it can be great. We look at five aspects of procedure calls: the cost of a call itself, the cost of passing arguments (and whether they are ByRef or ByVal), the cost of returning values, and the cost of calls to the Application object.

The cost of a function call

Adding in function calls incurs a cost. Replacing a line like

\[ \text{path}_S = \text{path}_S \times \text{Exp}(\text{drift} + s_{\text{root}_t} \times \text{cndev(\text{global}\_\text{ran0})}) \]  

(14.8)

with a function call, as in the level 2 application (section 4.3), increases the computation time. In the polar + ran0() + global case the time increases by about 15% just because of a function call.

We investigate more systematically the cost of calling procedures. In our first example we give times for procedures with signatures of the form shown in lines (14.9),

\[
\begin{align*}
\text{Private Sub TestSub}(a & \text{ As Double}, \ldots) & \quad \text{a} \\
\text{Private Function TestFn}(a & \text{ As Double}, \ldots) \text{ As Double} & \quad \text{b},
\end{align*}
\]  

(14.9)

where there are zero or more arguments, and the procedures are strongly typed. TestFn() is called by assigning its return value to a Double. Neither TestSub() nor TestFn() have a body; they are merely stubs. Execution times, given in Table 14.15, represent just the cost of calling the procedures. Using Call to run a Sub adds no time to its execution.

Times are shown for each procedure when the arguments are each qualified ByVal, or ByRef, or not qualified at all (as in lines (14.9)). In addition, the last row of the table shows times for TestFn() when it is called as, for example, Call TestFn(), with its return value not assigned.

There is a great deal of noise in these numbers, but the trend is increasing, faultingly, in the number of arguments. There is a step change going from zero arguments to one argument, and running with three arguments seems to be anomalously expensive, but otherwise the performance hit in going from no arguments to six is only around 10%. This is minor compared to the cost of incurring a procedure call in the first place. A Function call is significantly more expensive than a Sub call, by about 15%.

<table>
<thead>
<tr>
<th>Number of arguments:</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>TestSub()</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ByVal</td>
<td>–</td>
<td>9.55</td>
<td>9.64</td>
<td>10.08</td>
<td>10.15</td>
<td>10.53</td>
<td>10.84</td>
</tr>
<tr>
<td>ByRef</td>
<td>–</td>
<td>9.51</td>
<td>9.55</td>
<td>10.09</td>
<td>9.80</td>
<td>10.16</td>
<td>10.27</td>
</tr>
<tr>
<td>No qual.</td>
<td>9.13</td>
<td>9.57</td>
<td>9.54</td>
<td>9.96</td>
<td>9.77</td>
<td>10.17</td>
<td>10.26</td>
</tr>
<tr>
<td>TestFn()</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ByVal</td>
<td>–</td>
<td>11.12</td>
<td>11.28</td>
<td>11.98</td>
<td>11.79</td>
<td>11.94</td>
<td>12.32</td>
</tr>
<tr>
<td>ByRef</td>
<td>–</td>
<td>11.14</td>
<td>11.21</td>
<td>12.10</td>
<td>11.31</td>
<td>11.71</td>
<td>11.64</td>
</tr>
<tr>
<td>No qual.</td>
<td>10.61</td>
<td>11.11</td>
<td>11.19</td>
<td>11.85</td>
<td>11.31</td>
<td>11.74</td>
<td>11.72</td>
</tr>
<tr>
<td>No ret.</td>
<td>10.15</td>
<td>10.74</td>
<td>10.80</td>
<td>10.91</td>
<td>10.93</td>
<td>11.29</td>
<td>11.72</td>
</tr>
</tbody>
</table>

7 VBA does not optimize away the cost of the procedure call in such cases.
When fewer than two or three arguments are passed there is very little discernible difference in cost between the different ways of passing them. With four or more arguments passing ByVal seems to become more expensive. (An unqualified argument is by default passed ByRef so it is not surprising that the times for unqualified and ByRef are similar.)

**Typing arguments**

It is needlessly expensive (as well as bad) to give a procedure untyped (and therefore Variant) arguments or return type. Table 14.16 shows the cost of calling TestFn() when the (single) argument and the return type are declared to be either Double, or Variant, or not declared at all.

The cost of calling TestFn() with an untyped (that is, Variant) argument is 11.50 compared to 11.11 when it is typed. Explicitly passing a Variant is just as bad.

Much worse is failing to type the return value. This almost doubles the cost of the function call to 19.56.

**Returning values**

The cost of returning values to arguments passed ByRef is slight. We compare the cost of returning a Double either explicitly from a Function or from a Sub via an argument passed ByRef. Figure 14.3 shows the two test procedures. The cost of calling TestSub() is 10.52 and of calling TestFn() is 11.65. This is consistent with the cost of calling a Sub with one argument or a Function with no arguments, plus the cost of an assignment of a Double to a Double.

For returning a Double it is cheaper to use an implicit ByRef return value from a Sub.

**Calls to the Application object**

In case anyone was wondering, using Application.NormSInv() (instead of cndev()) is a trifle slow. Used with ran0() + global it takes over 6 seconds to run a plain Monte Carlo method with \( N = 100 \) and \( M = 500 \), about 100 times slower than using cndev(). Application.Max() also should never be used; writing your own little `max()` is hugely superior. For computing the maximum of two Doubles, \( 10^6 \) replications of Application.Max() took 20.16 seconds whereas a function call to an If-Then-Else statement took 13.61 seconds for \( 10^8 \) replications, faster by a factor of about 150.

<table>
<thead>
<tr>
<th>Return type</th>
<th>Argument</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Double</td>
<td>Variant</td>
<td>None</td>
</tr>
<tr>
<td>Double</td>
<td>11.11</td>
<td>11.43</td>
<td>11.50</td>
</tr>
<tr>
<td>Variant</td>
<td>19.53</td>
<td>20.00</td>
<td>20.37</td>
</tr>
<tr>
<td>None</td>
<td>19.56</td>
<td>20.39</td>
<td>20.16</td>
</tr>
</tbody>
</table>

**Table 14.16** TestFn(): cost of typing arguments and returns

Private Sub TestSub(ByRef a As Double)
    a = 2#
End Sub

Private Function TestFn() As Double
    TestFn = 2#
End Function

**Figure 14.3** Returning a Double
14.3 DATA TYPING ISSUES

This is a fundamental issue; strong typing is a long-standing programming paradigm. If you know what your types are you can sort out problems at compile-time, rather than at run-time. You can pass objects to your clients and they know what they have got from you.

VBA is not particularly strongly typed and for what there is it offers specific mechanisms to circumvent it. The Variant data type can contain anything, and the Object type can stand in for any object reference. This is not good. It might help if the late-bound Object type permitted and enabled polymorphism, but it does not. Variant may be “flexible”, but as far as non-polymorphic typing goes, flexible means bad.

We look at a number of data-representation issues, starting with casting and working through to the uses of Statics and globals and the NOT_FIRST_TIME idiom. Here we specifically do not look at objects. We look at VBA containers in Chapter 16.

Implicit typing

VBA implicitly casts numeric data from one type to another. In the assignment \( a = b \), where both \( a \) and \( b \) are numeric, VBA casts \( b \) into the type of \( a \). Table 14.17 gives times for various combinations of types of \( a \) and \( b \). Letters stand for Integer, Long, Single, Double, and Variant.

Variants come in sub-types. The table has two numbers in each entry of the Variant column. The first is the case where \( b \) is Variant-Single, the second where it is Variant-Integer.

It is clear from the table that Integers and Longs behave similarly, as do Singles and Doubles. In fact the only difference between Singles and Doubles, from this perspective, is in the cost of assigning Variants to them. Integers and Longs differ only in the cost of assigning one to another, and an assignment of a float to an Integer is more expensive than to a Long.

Using Types compared to non-Types; the cost of “.”

Assigning into Types and extracting from them is relatively cheap. Assigning into a Double slot of a type containing only Doubles costs 1.50; extracting from the Type costs 1.08, the same as a Double to Double assignment. In arithmetical operations there seems to be little or no difference between using a Double directly and using it from within a Type. Further, it costs the same to pass a whole Type (of Doubles) to a procedure as it does to pass a single Double.

The advantage of using Types is that you can bundle up a whole lot of (primitive) data together and pass it around an application. This can reduce both the number procedure calls and the number of arguments passed to procedures. This reduces the overhead of procedure calls and, more importantly, makes code cleaner and safer.

From our tests there is little or no cost disadvantage in defining and using Types. Used appropriately they reduce cost. There are plenty of other reasons to want to use them: clearer code, encapsulation, better typing. Use Types.

| Table 14.17 Costs of assignment |
|-----------------------------|---|---|---|---|---|
| \( a = b \) | I | L | S | D | V |
| \[ I \] | 0.56 | 0.74 | 2.68 | 2.67 | 3.23 / 1.30 |
| \[ L \] | 0.73 | 0.55 | 2.42 | 2.42 | 3.18 / 1.45 |
| \[ S \] | 1.85 | 1.85 | 1.07 | 1.11 | 2.01 / 2.11 |
| \[ D \] | 1.85 | 1.84 | 1.08 | 1.07 | 1.85 / 2.01 |
| \[ V \] | 1.48 | 1.49 | 2.19 | 2.23 | 1.02 / 1.03 |
**Using Const for constant variables**

If the value of a variable is fixed then it should be declared as `Const`, even if there is a minor cost hit. It is honte. Using `Const` may have a small cost affect but this is likely to be tolerable when compared to the increase in safety.

**Higher dimensional arrays**

Higher dimensional arrays are expensive. It is much more expensive to index an array of two or more dimensions than it is to index a 1-dimensional array. We compare the cost of iterating through three arrays of `Double`,

```vba
ReDim Arr1(1 To 16 ^ 6) As Double 'a
ReDim Arr2(1 To 16 ^ 3, 1 To 16 ^ 3) As Double 'b
ReDim Arr3(1 To 16 ^ 2, 1 To 16 ^ 2, 1 To 16 ^ 2) As Double 'c,
```

using `For` loops. Note that each array has the same number of elements.

The cost of the `For` loops is the same. It does not matter if you use one loop with $16^6$ iterations or three nested loops each with $16^2$ iterations, the cost of the loops by themselves is constant at 0.19.

We look at the costs of (i) assigning to each array member, and (ii) reading back the value of each array member. Costs are shown in Table 14.18 (times exclude loop-times). The 1-dimensional array is very significantly faster than the 2- and 3-dimensional arrays. The 2- and 3-dimensional arrays are similar. The 2-dimensional array is slower at writing than the 3-dimensional array but roughly the same at reading.

Since the cost difference going from 1-dimension to several is so large, it should be kept in mind when designing the data structures in a numerical application. If it is possible to structure the data as a set of 1-dimensional arrays instead of one 2-dimensional array, then perhaps one should do so.

**Does the type of the counter in a For loop matter?**

Yes it does. Table 14.19 shows times for executing empty loops with `i` and `N` either `Long` or `Double`. Not unexpectedly, it is cheaper for `i` to be `Long`. Declaring `N` to be `Const` appears to make an insignificant difference. This is not too unexpected since it seems that the value of `N` is computed just once at the start of the loop.8

**Statics or globals?**

`Static` variables exist to encapsulate data within the bodies of procedures, so that globals need not be used. In pre-OOP days they were the best available tentative groping towards objects with full encapsulation. The spirit was right but the mechanism remained underdeveloped until objects were introduced.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Arr1</th>
<th>Arr2</th>
<th>Arr3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assignment:</td>
<td>0.40</td>
<td>2.27</td>
<td>2.06</td>
</tr>
<tr>
<td>Extraction:</td>
<td>0.43</td>
<td>3.00</td>
<td>3.07</td>
</tr>
</tbody>
</table>

---

8 Altering its value within the loop makes no difference to the number of iterations performed, but please do not alter it in your code.
Table 14.19  Bound and loop counter types: loop times

<table>
<thead>
<tr>
<th></th>
<th>L</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>1.12</td>
<td>2.22</td>
</tr>
<tr>
<td>D</td>
<td>1.12</td>
<td>1.73</td>
</tr>
</tbody>
</table>

Table 14.20  Comparison of random number generators: using Statics and globals

<table>
<thead>
<tr>
<th>Method</th>
<th>ran0()</th>
<th>ran2()</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Statics</td>
<td>Globals</td>
</tr>
<tr>
<td>Polar: Globals</td>
<td>9.39</td>
<td>8.76</td>
</tr>
<tr>
<td>Inv. T: (arrays) Statics</td>
<td>9.35</td>
<td>9.49</td>
</tr>
<tr>
<td>Inv. T: (arrays) Globals</td>
<td>9.32</td>
<td>9.64</td>
</tr>
<tr>
<td>Inv. T: (Doubles) Statics</td>
<td>9.60</td>
<td>8.61</td>
</tr>
<tr>
<td>NormSInv() (10^5 variates)</td>
<td>7.55</td>
<td>7.51</td>
</tr>
</tbody>
</table>

The availability of objects reduces the need for Statics but there are still occasions when they can, and should, be used. In this section we evaluate their cost in a practical application: generating random variates.

Table 14.20 gives times in seconds for various combinations of methods for generating uniform variates and obtaining normal variates from uniforms. Times are for generating 10^7 normal variates, except for the final row where times are for generating only 10^5 normal variates. Uniforms are generated either with ran0() or with ran2(). Normal variates are generated either by polar rejection or by inverse transform. These functions use seed variables that need to keep their values between function calls, and constants that do not need to be initialized on every function call. These are stored either as Statics within the function or as globals at module level. Where required they are initialized in every case using the NOT_FIRST_TIME idiom; the reported costs are the costs of accessing the variable, not of repeated initialization.

The inverse transform function uses a large number of constants. In Table 14.20 in the rows labelled ‘Inv. T: (arrays)’ these are stored for convenience in three arrays. In the row labelled ‘Inv. T: (Doubles)’ they are kept as individual Static Doubles.

For comparison, and as a warning, the table also includes results for inverse transform using Application.NormSInv(). These results are really bad. Calling the Excel NormSInv() function is around 100 times more costly than using a hand-made function like cndev().

Unfortunately using Statics gives a performance hit, which is a shame since they are designed specifically to encapsulate data inside functions in order to avoid having globals. How much of a hit is incurred depends on the functions. For polar + ran0() the hit in going from fully global to fully Static is over 10%, and for polar + ran2() it is over 20%. For the array version of inverse transform things are less clear cut. It appears to be as expensive to have global arrays as it does to have them as Statics. Here, using globals with ran0() even increases the cost. Much clearer is the case of individual Doubles where arrays are avoided. Replacing Statics by globals reduces the cost by 10–15%.

---

9 ran2() is a more sophisticated uniform generator than ran0(), also adapted from Press et al. (2002), *Numerical Recipes in C++* (2nd edition).

10 There are too many individual constants to seriously consider having them as globals.
Should one avoid Statics? Certainly not. In a library function one can go all out for speed, or if you are certain that you completely understand your application and that it will never change, then perhaps wearing the equivalent of go-faster stripes is for you. Otherwise, encapsulate everything you can, decouple as much as possible, with whatever tools you have. The level 2 application, which uses polar \( + \text{ran0}() + \text{Static} + \) function calls, is almost 15% slower than the corresponding yukky level 0 version; in all but the most time-critical application this is time well spent.

On the other hand, why not go the whole hog and create an object to take responsibility for the functionality? The object NormalGenerator (Figure 5.13 on page 74), using polar \( + \text{ran0}() \), costs 8.15 using \texttt{Const} data members (increasing to 9.08 when these are declared non-\texttt{Const}), over 10% cheaper than the non-member function form. Use objects.

**Initialization and the NOT_FIRST_TIME idiom**

The \texttt{NOT_FIRST_TIME} idiom is used in \texttt{ran0()}, in \texttt{cndev()}, and in various other non-member procedures throughout this book. Since testing the value of a \texttt{Boolean} should be cheaper than assigning a value, we expect that it gives significant cost savings. We give just one example. In \texttt{cndev()} there are four coefficient arrays whose values do not change and are set using the idiom. If their values are set each time the function is called, then the computation time goes up by a factor of 3 from around 9 seconds to around 27 for \texttt{ran0()} and 30 for \texttt{ran2(())}.

In non-member procedures there is no excuse not to set the values of constants once and for all using the \texttt{NOT_FIRST_TIME} idiom.

### 14.4 SUMMARY

Most of the results of this chapter are common sense, but some are counter intuitive.

Table 14.21 summarizes some easy speed-ups. Other rules, like

(1) precompute as much as possible outside loops;
(2) prefer multiplication to division; and
(3) avoid unnecessary casting

are just common sense.

<table>
<thead>
<tr>
<th>Table 14.21</th>
<th>Some cost saving advice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type integer literals as Doubles when used in floating point arithmetic.</td>
<td></td>
</tr>
<tr>
<td>Use ( a * a ) and ( a * a * a ) not ( a ^ 2 ) or ( a ^ 3 ).</td>
<td></td>
</tr>
<tr>
<td>Use \texttt{CLng}(a ,-, 0.5) not \texttt{Int}(x).</td>
<td></td>
</tr>
<tr>
<td>Use in-line-\texttt{If} in preference to block-\texttt{If}.</td>
<td></td>
</tr>
<tr>
<td>Use \texttt{For} loops (with \texttt{Step}) when the number of iterations is known.</td>
<td></td>
</tr>
<tr>
<td>Avoid unnecessary use of VBA functions</td>
<td></td>
</tr>
<tr>
<td>Avoid any use of Application functions, if possible.</td>
<td></td>
</tr>
<tr>
<td>Explicitly qualify all procedure arguments (with \texttt{ByRef} or \texttt{ByVal}).</td>
<td></td>
</tr>
<tr>
<td>Explicitly type all procedure arguments and return values.</td>
<td></td>
</tr>
<tr>
<td>Use Types.</td>
<td></td>
</tr>
<tr>
<td>Use \texttt{Const}.</td>
<td></td>
</tr>
<tr>
<td>Design to avoid, if possible, higher dimensional arrays.</td>
<td></td>
</tr>
<tr>
<td>Be prepared to use Statics in non-member functions, despite their cost.</td>
<td></td>
</tr>
<tr>
<td>Use the Static initialization idiom to avoid re-initializing constants.</td>
<td></td>
</tr>
<tr>
<td>Divide by 0.5 instead of multiplying by 2# , divide by 2# instead of multiplying by 0.5. } Check on your platform!</td>
<td></td>
</tr>
</tbody>
</table>
In numerical applications speed matters, but so does clarity. Despite everything written here it is usually best to program for clarity rather than speed. If you have to write tricky code to get it to execute fast enough, please comment it thoroughly.

14.5 EXERCISES

I refer to the combination of your computer hardware and your version of Excel and VBA as your installation. This set of exercises asks you to perform timing runs with your installation to compare to those displayed here. Not only will the absolute times be different but it is likely also that the relative times will be different. You may find that some anomalies reported here do not appear with your installation, but some new ones do.

A result of having thought through the consequences of these exercises is that you gain a better appreciation of those programming techniques you can trust and those you cannot. A tweak that works on one installation, because of some feature of the hardware, or the version of VBA, may not work on another, and may even give worse performance. As a rule, prefer to code simply. Even if a tricky bit of programming works well on one installation the improvement in performance may not port to another.


   (a) Is division half the speed of multiplication?

   (b) Is multiplication the same speed as addition?

   (c) Do the same special cases occur for multiplication and division (Tables 14.3 and 14.4)?

   (d) Are built-in functions as slow on your installation (Table 14.13)?

2. Control structures. Tables 14.9–14.11 and 14.12 give timings for control structures. How do they compare with times on your installation?

3. Procedure calls, arguments and return types. Are relative costs of these the same on your installation as those on the installation used to produce Tables 14.15 and 14.16?

4. Assignment and casting. Are the relative costs of assignment and casting the same for you as Table 14.17?

5. Assess the advice given in Table 14.21.

   (a) Does it hold on your installation?

   (b) Based on your experience, can you offer any further general advice?

6. The Sub OutputCounter() (Figure 4.9, page 48) can be speeded-up. How much faster can you make it run?
This chapter compares the performance of Monte Carlo applications at the different levels. The main comparisons are in the costs of valuing European options (section 15.2.1) and valuing barrier options (section 15.2.2), for polymorphic and non-polymorphic versions of the application.

First we look at the way different components of the very simplest Monte Carlo application contribute to the overall cost. This enables us to identify where improvements might be made. Variations to the basic code are experimented with and the effect assessed.

We find that OOP polymorphism in VBA is very cheap. There is no reason, on cost or design grounds, not to go OOP.

The times given in the tables in this chapter are in seconds.

15.1 VARIATIONS OF THE LEVEL 0 APPLICATION

It is important to understand where the cost of an application lies. There is no built-in profiler in VBA so we use the stopwatch to time the execution of component parts of the Monte Carlo application.

The base case Monte Carlo application is shown in Figure 15.1. This level 0 version is based on Figure 3.14 in Chapter 3 but with some function calls removed. With \( N = 100 \) time steps and \( M = 50\,000 \) sample paths its cost is 5.77 seconds.

Level 0 is the most basic of the versions of the Monte Carlo application that we have seen, but even here there is some scope for variation. We investigate the costs of using different code to compute payoffs, to evolve the underlying asset processes from one step to the next, and to evolve \( R_t = \ln(S_t) \) instead of evolving \( S_t \).

Results were computed using the spreadsheet MC_example_v1a.xls. Lines in the code module MC_application were commented in and out to establish the costs of different combinations of code. Before looking at this we first investigate how the different parts of the level 0 application contribute to the total cost. This gives a better perspective on the effect of changes we make to the code.

**Breakdown of costs**

Results are given in Table 15.1. It gives costs for two ways of generating normal variates: polar rejection and inverse transform.

The total base case cost is broken down into seven components.

(1) ‘End stuff’ is the cost of computing the option values and standard errors from accumulated sums of values and their squares, and discounting.

(2) ‘Loop counter’ is the cost of computing and outputting the loop counter

\[
\text{If } j \div 5000 = j \backslash 5000 \text{ Then Cells}(8, 7).\text{Value} = j ';
\]

(15.1)

(3) ‘Uniforms’ is the cost of obtaining uniform variates.

(4) ‘Uniforms → Normals’ is the additional cost of computing normals from uniforms.

(5) ‘Uniforms’ is the cost of obtaining uniform variates.

(6) ‘Uniforms → Normals’ is the additional cost of computing normals from uniforms.

(7) ‘Uniforms’ is the cost of obtaining uniform variates.

(8) ‘Uniforms → Normals’ is the additional cost of computing normals from uniforms.
Sub main()
    Cells(19, 9).Value = ""
    Dim S_0 As Double: S_0 = Cells(12, 6).Value
    Dim rr As Double: rr = Cells(13, 6).Value
    Dim sigma As Double: sigma = Cells(14, 6).Value
    Dim X As Double: X = Cells(17, 6).Value
    Dim T As Double: T = Cells(18, 6).Value
    Dim N As Long: N = Cells(12, 9).Value
    Dim M As Long: M = Cells(13, 9).Value
    Dim dt As Double: dt = T / N
    Dim drift As Double: drift = (rr - 0.5 * sigma * sigma) * dt
    Dim s_root_t As Double: s_root_t = sigma * Sqr(dt)
    Dim acc_vals As Double: acc_vals = 0
    Dim acc_squs As Double: acc_squs = 0
    Dim e_time As Double: e_time = Timer
    Dim j As Long
    For j = 1 To M 'for each sample path
        If j / 1000 = j \ 1000 Then Cells(8, 7).Value = j
        Dim path_S As Double: path_S = S_0
        Dim i As Long
        For i = 1 To N 'for each time step
            path_S = path_S * Exp(drift + s_root_t * GetNormal)
        Next i
        Dim payoff As Double: payoff = my_max(0#, path_S - X)
        acc_vals = acc_vals + payoff
        acc_squs = acc_squs + payoff * payoff
    Next j
    Dim val As Double: val = acc_vals / M
    Dim se As Double: se = Sqr(acc_squs - acc_vals * acc_vals / M) / M
    Dim dis As Double: dis = Exp(-rr * T)
    val = dis * val
    se = dis * se
    e_time = Timer - e_time
    Cells(17, 9).Value = val
    Cells(18, 9).Value = se
    Cells(19, 7).Value = e_time
End Sub

Figure 15.1 The level 0 main()

Table 15.1 Breakdown into component costs

<table>
<thead>
<tr>
<th>Component</th>
<th>Polar rejection</th>
<th>Inverse transform</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) End stuff:</td>
<td>0.05</td>
<td>0.05</td>
</tr>
<tr>
<td>(2) Loop counter:</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>(3) Uniforms:</td>
<td>2.36</td>
<td>1.86</td>
</tr>
<tr>
<td>(4) Uniforms → Normals:</td>
<td>2.39</td>
<td>2.87</td>
</tr>
<tr>
<td>(5) Normals → ln(S):</td>
<td>0.25</td>
<td>0.25</td>
</tr>
<tr>
<td>(6) exp()</td>
<td>0.7</td>
<td>0.7</td>
</tr>
<tr>
<td>(7) Payoffs:</td>
<td>0.02</td>
<td>0.02</td>
</tr>
<tr>
<td><strong>Total:</strong></td>
<td><strong>5.78</strong></td>
<td><strong>5.76</strong></td>
</tr>
</tbody>
</table>

Percentage means given as a percent.
(5) ‘Normals → ln(S)’ is the cost of the line

\[ \text{path}_S = \text{path}_S + \text{drift} + s_{\sqrt{t}} \times \text{GetNormal} \]  

(15.2)

less the cost of the code for generating the normal variate (denoted here by GetNormal).

(6) ‘exp( )’ is just the cost of applying exp().

(7) ‘Payoffs’ is the cost of the payoff calculation.

Uniform variates are generated using \texttt{ran0()}. There are any number of better ways of generating uniforms; see Press et al. (2007) for a discussion. Popular in the finance world is a generator called the Mersenne twister (Matsumoto and Nishimura (2002)). Efficient implementations of this, and of quite a few other modern generators, uses bit-shift operators. These are not directly supported by VBA\(^1\) so here we stick to older style generators like \texttt{ran0()}.  

Inverse transform generates a normal variate \(n\) from a uniform variate \(u\) by setting

\[ n = N^{-1}(u), \]  

(15.3)

where \(N\) is the standard normal distribution function. It is easy to verify that if \(u \sim U(0,1)\) is uniform then \(n\) is indeed standard normal. In fact for any distribution function \(F\), the variate \(F^{-1}(u)\) has the distribution \(F\).

The function \texttt{cndev()} (Moro (1995)) computes \(N^{-1}\). In Table 15.1 the ‘Uniforms → Normals’ entry for inverse transform is exactly the cost of executing \texttt{cndev()}. However, for polar rejection the figure for the cost of uniforms requires a little explanation.

The polar rejection method works by generating pairs of uniforms from the range \([-1,1]\). If they give the coordinate of a point lying within the unit circle in \(\mathbb{R}^2\) then they are accepted and used to generate two normals. Otherwise they are rejected and another pair of uniforms is generated.

The cost of generating two uniforms, all by themselves, is 3.71. In the polar rejection method an average of \(4/\pi\) pairs of uniforms have to be drawn before a pair is accepted; these then give two normals. Hence the average cost of generating uniforms per normal variate, the number shown in the table, is \(\frac{1}{2} \times 3.71 \times \frac{4}{\pi} = 2.36\). The total cost of generating normal variates by polar rejection is 4.75, so ‘Uniforms → Normals’ is set to be 4.75 \(- 2.36 = 2.39\).

The major component of the cost of the method is the cost of generating normal variates. Inverse transform is slightly faster than polar rejection (but not by as much as expected from the results of Chapter 14). Generating normals accounts for over 80% of the cost of the application for both methods. Half the cost of the polar rejection method is generating uniforms; for inverse transform the cost of generating uniforms accounts for around 40% of the total.

\texttt{cndev()} is just by itself half the cost of the inverse transform version of the application. A faster algorithm to compute \(N^{-1}\) would be of direct benefit.

The majority of the remaining cost is the cost of computing exponentials, at 12%, with residual calculations taking only around 6% of the time. Most of this is incrementing the value of the underlying asset from the normal variate. Outputting a counter is effectively free (at the frequency chosen here), and computing payoffs and getting option values and standard errors from them is only 1–2% of the total cost.  

Using the \texttt{NormalGenerator} object (page 74) to compute normals is slightly cheaper, and brings the overall cost down by 0.05 seconds.

\(^{1}\) Although bit-shift can be mimicked using arithmetic and type conversions. See the implementation of the library utility \texttt{sobseq()} in \texttt{LibSobol}.
**Scope for improvement**

Since the code used in the example is reasonably efficient, it is clear that if you want to make the method go faster you have to go beyond VBA *per se* and examine the application itself. Can random numbers be generated by faster algorithms? How can the Monte Carlo method be improved to get better results? The second question is the most fruitful and is addressed in this book; one should routinely use the fastest, decent, random number generator available in what ever library you have.

Getting it wrong makes a potentially large difference. Having computed the base case we now examine the effect of changing the details of the calculations. We start with the payoff calculation, then look at evolving $S_t$, and finally at evolving $R_t$.

**Computing the payoff**

To compute the payoff the following variations were tried:

\[
\begin{align*}
\text{If } & \text{ path}_S > X \text{ Then payoff} = \text{ path}_S - X \text{ Else payoff} = 0 & \text{ 'a} \\
\text{payoff} = \text{my}_\text{max}(0#, \text{ path}_S - X) & \text{ 'b (base case)} \\
\text{payoff} = \text{Application.Max}(0#, \text{ path}_S - X) & \text{ 'c} \\
\text{payoff} = \text{Compute}_\text{PO}(\text{path}_S, X) & \text{ 'd}.
\end{align*}
\]

Table 15.2 gives the comparison. As perhaps anticipated, since the payoff calculation contributes only a little to the overall cost, minor variations in the payoff statement make correspondingly little difference; cases (15.4a), (15.4b) and (15.4c) are essentially indistinguishable within noise. The function $\text{Compute}_\text{PO}()$ encapsulates the payoff calculation away from the Monte Carlo loops and (at this level) is the preferred code. A major mistake would be to use $\text{Application.Max}()$ (line (15.4c)). All by itself this needlessly increases the cost of the Monte Carlo application by a sixth.

**Evolving $S_t$**

Even worse is to fail to precompute the drift and volatility coefficients. Lines (15.5),

\[
\begin{align*}
\text{path}_S = & \text{path}_S * \text{Exp}((\text{rr} - 0.5 * \text{sigma} * \text{sigma}) * \text{dt} _ \\
& + \text{sigma} * \text{Sqr(dt)} * \text{GetNormal}) & \text{ 'a} \\
\text{path}_S = & \text{path}_S * \text{Exp}(\text{drift} + \text{s_root}_t * \text{GetNormal}) & \text{ 'b} \\
\text{path}_S = & \text{next}_S(\text{path}_S, \text{drift}, \text{s_root}_t) & \text{ 'c},
\end{align*}
\]

give three versions of code to evolve $S_t$. Line (15.5b) is the base case. Line (15.5c) uses a function and is preferred to the base case for that reason.

Evolving with line (15.5a) would be a mistake; Table 15.3 shows just how big. Failing to precompute the drift and volatility costs a fifth extra on the computation time. There is really no excuse for this; it is an expense with no redeeming benefits.

<table>
<thead>
<tr>
<th>Case</th>
<th>Line (15.4a)</th>
<th>Line (15.4b)</th>
<th>Line (15.4c)</th>
<th>Line (15.4d)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time:</td>
<td>5.74</td>
<td>5.77</td>
<td>6.74</td>
<td>5.77</td>
</tr>
<tr>
<td>Variation (%):</td>
<td>-1</td>
<td>(base case)</td>
<td>+17</td>
<td>0</td>
</tr>
</tbody>
</table>
Table 15.3 Effect of variation of $S_t$ evolution

<table>
<thead>
<tr>
<th>Case</th>
<th>Line (15.5a)</th>
<th>Line (15.5b)</th>
<th>Line (15.5c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time:</td>
<td>6.97</td>
<td>5.77</td>
<td>6.42</td>
</tr>
<tr>
<td>Variation (%)</td>
<td>+21</td>
<td>(base case)</td>
<td>+11</td>
</tr>
</tbody>
</table>

Using a function call adds on 10%, but this extra cost can be tolerated because it increases safety and helps to decouple the application from a specific type of evolution.

Evolving $R_t$ instead of $S_t$

Taking exponentials contributes 12% of the cost of the level 0 application. However most of the time it is redundant to actually compute the value of $S_t$ itself. Its value is required only when the payoff is being computed. Instead, one may evolve $R_t = \ln(S_t)$, computing $S_t = \exp(R_t)$ only when needed at the final time. Code to evolve $R_t$, instead of $S_t$, is shown in lines (15.6):

```plaintext
Dim path_R As Double: path_R = ln_S0 'a
    ...
    path_R = path_R + drift + s_root_t * GetNormal 'b
    ...
Dim fin_S As Double: fin_S = Exp(path_R) 'c
    payoff = my_max(0#, fin_S - X) 'd.
```

The variable `path_R` contains the current value of $R_t$, initialized in line (15.6a) with $\ln(S_0)$. It follows the process

$$dR_t = \left( r - \frac{1}{2} \sigma^2 \right) dt + \sigma dz_t, \quad (15.7)$$

operationalized in line (15.6b). At the final time an exponentiation has to be calculated to recover the asset value against which the payoff is made (lines (15.6c) and (15.6d)). Table 15.4 compares the cost of base case evolution with that of evolving $R_t$. The reduction of 12% is exactly as expected.

In general it may not be possible to make a cheap saving just by evolving $\ln(S_t)$. It certainly works for GBM, but even then only in cases where it is not necessary to know the asset value on every step. An option such as a continuously reset barrier option requires a barrier condition to be tested on every step and would appear to require the value of $S_t$ to be known on each step. Even so, it may be possible to test the barrier condition against $\ln(S_t)$ instead of $S_t$ itself. However when you start doing things like this you get the feeling that maintenance is going to get tricky and error prone.

Table 15.4 Evolving $S_t$ and evolving $R_t$

<table>
<thead>
<tr>
<th>Case</th>
<th>Evolving $S_t$</th>
<th>Evolving $R_t$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time:</td>
<td>5.77</td>
<td>5.10</td>
</tr>
<tr>
<td>Variation (%)</td>
<td>(base case)</td>
<td>-12</td>
</tr>
</tbody>
</table>
15.2 EFFECT OF LEVEL ON TIMES

In the first four parts of this book we described various styles of Monte Carlo implementations, characterized with levels. Does the superstructure and overhead that comes with these varying levels of programming sophistication cause a time hit? We investigate this now.

15.2.1 Valuing European call options

The first comparison is on the cost of valuing a European call option. Since the option is a vanilla call with a payoff only at the final time and we are assuming the asset follows a GBM it could be valued using long-step Monte Carlo. Nevertheless for comparison purposes we use short-step. The methods use \( N = 100 \) time steps and \( M = 50000 \) sample paths.

Table 15.5 gives timings in seconds for each level, 1 to 6a. The final column gives the percentage increase in execution time over the level 0 version. Variation between individual runs can be as high as 1% or 2%, so the reported times are the average of five individual runs. For levels 0 to 4, normal variates are generated by polar rejection and uniform generator \( \text{ran0()} \) implemented with \( \text{Static} \) variables; for levels 4b to 6a inverse transform is used.

Becoming procedural, going from level 0 to level 1, increases the cost by 12%. This is not insignificant but should be thought of as a necessary overhead. In a complex application, trying to save 12% by turning your nicely structured procedural code into a incomprehensible monolithic block would be inconceivable.

More serious is the cost in moving to level 2. The reason for the extra 5% cost is not due to validation, or using a \( \text{Type} \), or using separate modules, but due to the use of \( \text{Static} \)s in the evolution function, \( \text{next}_S_{v2}() \). We have seen that \( \text{Static} \)s are expensive. They are a pre-OOP mechanism to encapsulate data within procedures. Their use is a strong signal that perhaps the affected procedure should be a member function in an object.

Introducing objects, at level 3a, reduces the cost of the application by a little. This is largely because one can avoid using \( \text{Static} \) variables in the random number generators (and incidentally get rid of the flawed

<table>
<thead>
<tr>
<th>Level</th>
<th>Evolves</th>
<th>Description</th>
<th>Time</th>
<th>Increase (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Doubles</td>
<td>Yukky</td>
<td>5.77</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>Doubles</td>
<td>Procedural</td>
<td>6.44</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>Doubles</td>
<td>Validation</td>
<td>6.74</td>
<td>17</td>
</tr>
<tr>
<td>3a</td>
<td>Path-wise</td>
<td>Objects</td>
<td>6.68</td>
<td>16</td>
</tr>
<tr>
<td>3b</td>
<td>Path-wise</td>
<td>Wrapper</td>
<td>6.66</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>Doubles</td>
<td>Polymorphic</td>
<td>10.74</td>
<td>86</td>
</tr>
<tr>
<td>4b</td>
<td>Slice-wise</td>
<td>Slices</td>
<td>6.10</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>Slice-wise</td>
<td>Factory</td>
<td>6.20</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>Slice-wise</td>
<td>Poly-factory</td>
<td>6.15</td>
<td>7</td>
</tr>
<tr>
<td>6a</td>
<td>Slice-wise</td>
<td>Meta-objects</td>
<td>6.14</td>
<td>6</td>
</tr>
</tbody>
</table>

2 Black–Scholes call with \( T = 1, X = 100, S_0 = 100, \sigma = 0.2, r = 0.05. \) Its value is \( \sim 10.45. \)
3 For levels 5 and 6 this is the time taken for five replications. There is a only a minuscule hit in running five replications rather than five individual runs.
4 We repeat the warning that \( \text{Static} \)s should not be used to set the values of non-\( \text{Const} \) variables in the \( \text{Static} \) initialization idiom. \( \text{next}_S_{v2}() \) breaks this injunction and should not be used in this form.
next_S_v2() procedure). Adding in the application wrapper object in level 3b makes no difference. The cost of 3b is the same, modulo noise, as level 3a.

The polymorphic level 4 version takes 50% longer than the non-polymorphic level 3b, and getting on for double the cost of level 0, but the extra cost is not because of polymorphism. Rather it is because the level 4 application employs element-wise evolution. It generates values of \( S_t \) one by one along each sample path and then at every step passes them over immediately to the option object. This incurs a very large number of unnecessary function calls, putting up the cost considerably, and is not appropriate in higher level applications.

The problem is overcome with the slice-based level 4b application. At each time step, instead of making 50,000 function calls, it hands over to the option object a single slice in a single function call. Evolving, returning, and processing an array as a single entity is far cheaper than operating with functions on its separate elements.

With level 4b the cost falls markedly, down to less than the cost of the procedural level 1 application, but this is not due solely to becoming slice-based. The method of computing normal variates also changes at level 4b. At lower levels normal variates have been generated by polar rejection. At level 4b we switch over to using inverse transform. As expected the cost is reduced. Using polar rejection the cost is 6.82 (18% slower than level 0). With inverse transform this goes down to 6.10, shown in the table, a decrease of over 10% from polar rejection.

The non-polymorphic factory of level 5 is cheap at 6.20. This is not unexpected since the level 5 factory is only a little more sophisticated than the previous object creation mechanisms. The cost of the fully polymorphic factory in level 6 and the semi-polymorphic factory at level 6a are the same within noise.

Going from level 0 to level 6 increases the cost by only 7%. This is nothing. In return for the full power of polymorphism the additional cost is comparable to just having ordinary procedure calls.

### 15.2.2 Valuing barrier options

We move on to the second comparison: the cost of valuing knock-in and knock-out barrier options.

Continuously reset barrier options are only a little harder to value than European call options. We have already discussed, in Chapters 6 and 7, how barrier options can be valued at levels 4 and 4b, and the form taken by the option objects. We briefly show how they can be valued at lower levels and then present timing comparisons.

The phrase ‘only a little harder to value’ is disingenuous. True, one can easily test a barrier condition at each time step and arrive at a Monte Carlo value for a barrier option. Unfortunately, as is very well known, and as we demonstrated in Chapter 6, the value will be heavily biased: knock-in options are undervalued and knock-out options are overvalued. This is because a Monte Carlo method effectively performs only discrete monitoring of the barrier condition at each time step. The barrier is hit less frequently than it would be in continuous time leading to the biases as described. We discuss this problem, and contributions to its resolution, in later chapters.

Results are given in Table 15.6. This gives times for knock-in and knock-out double barrier options, at three combinations of barriers, computed by various versions of the application. Barrier values are given in brackets. The first entry is the value of the lower barrier, the second that of the upper barrier. (90, 110) is a very tight set of barriers, (70, 130) is relatively loose and (80, 120) is intermediate.

For levels 0, 1 and 2 the workbooks used to value the barrier options are modifications of those used to produce the results in section 15.2.1. The changes are discussed below. The polymorphic cases have objects to value double barrier options, as discussed in Chapters 6 and 7.

---

5 These are the workbooks MC_v0_barrier.xls, MC_v1_barrier.xls, and MC_v2_barrier.xls.
Table 15.6 Comparison of timings: barrier options

<table>
<thead>
<tr>
<th>Level</th>
<th>Double barrier, KO</th>
<th>Double barrier, KI</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(90, 110)</td>
<td>(80, 120)</td>
</tr>
<tr>
<td>0</td>
<td>1.86</td>
<td>4.39</td>
</tr>
<tr>
<td>1</td>
<td>2.31</td>
<td>5.46</td>
</tr>
<tr>
<td>4</td>
<td>3.99</td>
<td>9.26</td>
</tr>
<tr>
<td>4b</td>
<td>6.78</td>
<td>7.40</td>
</tr>
<tr>
<td>5</td>
<td>6.69</td>
<td>7.30</td>
</tr>
<tr>
<td>6</td>
<td>6.73</td>
<td>7.32</td>
</tr>
<tr>
<td>6a</td>
<td>6.79</td>
<td>7.41</td>
</tr>
</tbody>
</table>

Levels 3 and 3b are omitted. The level 3 applications ask the generator object to return a value of the underlying only for the final time. They are the end of a line of development aimed at valuing European style options and are not really designed to cope with path-dependent options. Making the necessary modifications for them to be able to value barrier options would give us something like the level 4 application, but without the polymorphism. Rather than get sidetracked we go straight from level 2 to level 4.

It is easy to alter the level 0 code to value barrier options. Figure 15.2 shows the main loop in a variation that values double barrier options. \( U \) and \( D \) are the upper and lower barrier levels, whose values have been read in earlier on in \texttt{main()}. A Boolean variable, \texttt{hit}, keeps track of whether the barriers have been hit or not.\(^6\)

Different things happen depending on whether the option is knock-in or knock-out. The code shows the knock-out version. Commented out is the code to value a knock-in option; to value a knock-in option the knock-out code must be commented out and the knock-in code commented in. It should be clear how the code would be altered to value up-barrier and down-barrier options.

For the knock-out option the \texttt{For}-loop is exited when the barrier is hit. As Table 15.6 shows, this reduces the computation times considerably. With the tightest barrier range the cost is only 1.86

---

\(^6\) No check is made to confirm that at step zero the barrier condition is not already met. This is a fault that should be corrected.
compared to the European call base case cost of 5.77 (Table 15.5). The reduction is solely because the tightness of the barriers means that they are hit often and early, so that the number of sample paths that have to be generated in full is relatively small. As the barrier range widens the computation times increase.

The knock-in option takes a little longer to run than the European call. This is because the barrier condition has to be checked at every step (up to the time the barrier is hit). Here also widening the barrier range increases the average time before a barrier is hit and so increases the computation time.

It would be possible to make the knock-in code run faster. For instance, once the barrier has been hit, and the option knocked-in, it would be possible to evolve straight to the final time in a single long-step. This would bring the computation times for the knock-in option much closer to those of the knock-out option.

The level 1 application replaces statements in level 0 with procedures. The main loop is shown in Figure 15.3 (compare this with Figure 3.14, page 35). UpdateHit() just tests the barrier condition, and Compute_PO() computes the option payoff conditional on the barrier condition.

At this stage one is beginning to feel the need for an array to hold an entire path; nevertheless we persist with element-wise evolution.

At level 2 the UpdateHit() procedure (Figure 15.4) is altered to use Statics with the Static initialization idiom. There again this is highly dangerous since the Statics in question are non-Const. If the value of either U or D is changed the project must be Reset to force the Statics to be recalculated. Level 2 brings a number of advantages, but using Statics this way is not one of them. Apart from this somewhat backwards step the structure of the main loop in this version is much the same as the level 1 version.

As in the case of the European call the level 4 application is very expensive, and for the same reason: it has too many function calls. This is a consequence of its element-wise evolution design. At these levels, for path-dependent options, it is not sensible to be element-wise.
Levels 4b to 6a all cost much the same. The underlying objects for path generations and for the options are similar; the main differences lie on the structural side with variations to the factory creation procedures.

For knock-in options these levels have similar costs to the level 1 application (although admittedly they use inverse transform and not polar rejection). They compute every sample path in full – there is no early exit for these versions of the application. Even so it is cheaper to value options with narrower barrier ranges: the options do less work once a barrier is hit.

In a practical application there is really no disadvantage in having to compute every sample path in full. You will have a book of options. Not only would it be costly and complicated for the generating object to keep track of which options have or have not knocked out (as opposed to the individual options themselves), in any case it would not be effective. As long as a single option requires the full set of paths, the full set would have to be generated. You lose very little by deciding in advance to generate everything, willy-nilly, as efficiently as possible. Our decoupled designs give options slices for every time step and lets them decide what to do with them. This is simple and tidy.

At around 7.60 for wide-range knock-out options, the polymorphic applications are around 30% slower than level 0. This cost is entirely acceptable given the power and flexibility of the polymorphic applications.

15.3 SUMMARY

We have seen that polymorphism is relatively cheap. Even accounting for the faster method of generating normal variates, the polymorphic level 6 application is only 15% or so slower that the level 0 for European calls, and maybe 30% slower for knock-in options.7

The flexibility gained from going polymorphic is cheap. There is no reason whatsoever in not going polymorphic in any but the most basic application, or where time really is super-critical.

The main cost of a Monte Carlo application in VBA is the cost of generating random increments in the driving process. Here we have looked only at increments to a Wiener process driving a GBM – possibly the fastest vaguely realistic case – other processes are likely to take an even greater proportion of the cost. Employing a faster algorithm translates directly and proportionately into a faster Monte Carlo.

There is little scope to make a Monte Carlo method run faster by programming devices alone. One can hope – and expect – to be able to avoid making costly mistakes but there is no untapped magic VBA resource at hand to make things run faster.

Instead one looks to the Monte Carlo method itself. Regarding the generation of a set of sample paths as a fixed overhead, how can they be used as efficiently and effectively as possible to get out an option value with the least possible standard error? This is the topic of Part VI.

---

7 We assume that, in general, it is not possible to benefit from the theoretical possibility of terminating sample paths early.
15.4 EXERCISES

1. Table 15.1 analyses component costs for a level 0 Monte Carlo application. For the PDE and lattice exercise applications:

(a) What component costs should the total execution time be broken down into?

(b) What are the relative costs of the components you have identified?

2. Table 15.5 compares the effect of level on performance. Repeat the analysis (as appropriate) for the PDE and lattice applications.

(a) How large a performance hit does going OOP have?

(b) How much difference to performance does it make for code to be polymorphic?

3. Table 15.6 compares execution times for valuing barrier options. Construct analogous tables for the PDE and lattice applications. What features contribute most to the variation (or lack of variation) in computing times?
This page intentionally left blank
Data structures are partial answers to the question: How do you actually go about implementing a method? If a method evolves a slice from one step to the next just what is it that reifies the slice? The data structure chosen to do this affects both the clarity and the speed of the application. Choosing a data structure is about a balance between (i) playing to the strengths of a language while avoiding its weaknesses and (ii) aligning the program structure with the underlying logic of the application.

For instance with a single state variable a Brownian bridge Monte Carlo method (see Chapter 18) can be implemented either by path-wise evolution or by slice-wise evolution. Slice-wise evolution calls a time-stepping procedure $N$ times; path-wise evolution calls a space-stepping procedure $M$ times. Since normally $M \gg N$ from the programming perspective a slice implementation may be preferred: you have a much smaller number of steps (although operating on a much larger structure). Since the Monte Carlo method itself is usually agnostic about the choice\(^1\) the slice-wise approach is much better.

It could be natural in other settings to evolve a different data structure. For instance, in a model with several state variables one might want to evolve a higher dimensional array. Unfortunately, as we have seen, VBA appears to be disproportionately slow when there is more than one dimension in an array, and it gets worse as the number of dimensions increases. In this case the natural data structure would probably give way to a less problematic data structure.

As in Chapter 1 we suppose we are evolving a stochastic process $X_t \in \mathbb{R}^Q$ of dimension $Q$. We focus on slice-wise evolution with $N$ time steps on a slice of size $M$. $M$ is part of the model, and $N$ can depend on both the method and the option being valued;\(^2\) in any case $M$ and $N$ are not $\texttt{Const}$ and neither strictly is $Q$. However it is very convenient to suppose that the model is predetermined so that $Q$ is fixed; if so then the value of a state can fit into a static array, or into a UDT.

At each time $t_i$ one has a slice $\hat{X}_i = \{\hat{X}_{i,q}\}_{q=1}^{M}$, a 2-dimensional array. There is plenty of choice in how to structure the set $\hat{X}_i$ in a VBA implementation. We briefly review the data structures available to VBA, and how they can be used in a Monte Carlo application. Then we present some timings comparisons of their use in variations of the level 0 Monte Carlo application.

The times given in the tables in this chapter are in seconds.

### 16.1 DATA STRUCTURES IN VBA

Abstract data containers come in many varieties: sets, vectors and arrays, lists, queues, hash tables, trees and graphs, \textit{et cetera}. In principle, VBA can construct any container type you could wish for; in practice expense is a big issue. We return to this in section 16.2.

VBA has a relatively small number of components that can be combined to make containers. These are arrays, UDTs, \texttt{Variant} arrays, \texttt{Dictiornary}s and \texttt{Collections}, and POD objects. We look briefly at the last four.

\(^1\) Some methods are not. To avoid storing every sample path Longstaff and Schwartz (2001) least squares Monte Carlo requires slice-wise evolution (backwards from the final time). This is possible only if a bridge method is available.

\(^2\) If long-step Monte Carlo can be used, $N$ will equal the number of reset dates.
**Variant arrays**

A **Variant array** is a **Variant** containing an array. For most purposes it is the same as an array of **Variants**. It can be indexed and each position can be get/set just like an ordinary array. The difference is that a **Variant** array can hold anything in each slot, including other arrays.\(^3\)

Despite its convenience we are justifiably suspicious of this structure. It is anti-type, can be dangerous to use, and is likely to be expensive. However there are situations where a **Variant** has to be used, or where its use is convenient because alternatives are clumsy. In the technical programming sense, **Variants** are evil.

**The Dictionary object**

This is an associative container with keys and items. Keys are **Strings** or numeric types, items can be quite general. There are few methods and fewer **Properties**.

Declaring and **Setting** **Dictionaries** is a little different to other objects. The construction in equation (16.1) must be used.

\[
\begin{align*}
\text{Dim } \text{dict } & \text{ As Object} \quad \text{'}a \\
\text{Set } \text{dict } & \text{ = CreateObject("Scripting.Dictionary") } \quad \text{'}b.
\end{align*}
\]

This sets **dict** to be a reference to a **Dictionary**.\(^4\) You can add or remove item-key pairs, change items or keys, and test if a key exists. You can copy all keys and all items from a **Dictionary** into a **Variant** array, and clear the **Dictionary**. A **Dictionary** can take **Strings** as keys, as well as numeric types. Items can be varied, but, like **Variant** arrays and **Collections**, it is for our purposes effectively not possible to use **UDTs** as items.

**Dictionaries** are a useful data structure but not so good for computation. It is not possible to iterate directly through a **Dictionary** using **For-Each-Next**. If the keys are first written out as a **Variant** array (using the **Keys()** method) then looping through this array gives indirect access to the **Dictionary** items, but access is still very slow.

Since it is not possible to loop directly through entries in a **Dictionary** – only indirectly via their keys, which is slow – this structure is inappropriate for numerical work. Our trials confirm this.

**The Collection object**

A **Collection** object holds an indexed list of other objects. **Collections** are returned by, for example, the **FileSystemObject** **Files** method. They are used extensively in the application object model.

The **Collection** object is an older form of container, predating the **Dictionary** object. Its properties (and purpose) differ from the **Dictionary** in several important ways. A **Collection** is not really an effective associative container.

Each item in a **Collection** has an index number, and can be referred to by that index, but these indexes are volatile. Removing or inserting items from the middle of a **Collection** causes the following indexes to be renumbered. Indexes run from 1 (**Collections** are one-based) to \(M\), where \(M\) is the number of items in the **Collection** (the value returned by the **Count()** Property).

---

\(^3\) But not, in practice, **UDTs**. In principle a **Variant** can hold a **UDT** but arranging for this to be able to happen is beyond the scope of this book. The same problem arises for **Collections** and **Dictionaries**. See the discussion for the **Collection** object.

\(^4\) This is late-bound. An alternative, if a reference to Microsoft Scripting Runtime is available (settable in the Tools | References window), is to write **Dim dict As Scripting.Dictionary** directly.
When items are added to a Collection they are added by default onto the end of the Collection. (It is possible to specify a relative position to insert at, but we do not discuss this here.)

Optionally, when items are added to a Collection they can be given a key (which must be a String). The corresponding item can then be referred to by its key, just like in a Dictionary. If a key is not supplied when the item is added then it can be accessed only through a For-Each-Next statement (or with its index, if this is known, or iterating with For-Next on the index through the entire Collection). There is no exists() method for keys in a Collection. The only way to test whether a key exists is to try to use it (and to trap the error that results if the key does not after all exist).

Once an item has been added to a Collection it may not be modified, only read or removed. However a Collection can contain references to objects. Although the reference cannot be altered, so that the reference cannot be Set to point at a different object, the referenced objects can be modified. For-Each-Next can be very effective in this circumstance.

If a Collection does not contain (references to) objects, and you want to modify the items it contains, it turns out to be feasible to remove items at the start of the Collection and then add on the modified items at the end. In this way a Collection behaves more like a queue than a set. Of course this is much slower than iterating through its members using For-Each-Next. It can be worthwhile constructing a wrapper object to hold non-object data just so that For-Each-Next can be used.

Like Variants and Dictionaries there is a problem if you want a Collection to hold a UDT. If you attempt to add your own (Publicly declared) UDT to a Collection, Variant, or Dictionary then you will get a notorious error message: ‘Only user defined types defined in public object modules can be coerced to or from a variant or passed to late bound functions.’ What this helpful message is telling you is that VBA does not allow you to use a UDT in this way. For the purposes of this book we shall assume that a UDT cannot be added to these containers.

In practice, there is in any case no real need to use UDTs in a Collection or a Dictionary. You may as well use a POD object. A POD can be modified in a For-Next loop without removal and reattachment, so it is very significantly faster to use, and has much greater power and flexibility than a UDT.

**POD objects**

Strictly, a POD (plain old data) object is one holding only Public data members, with no member Functions, Subs or Properties. It is functionally similar to a UDT, but a POD is more general. A POD object can be used conveniently where a UTD may not be, such as in Collections or Dictionaries. The same syntax to access data members applies.

Fixed length Strings cannot be declared as Public in a class module but are allowed in UDTs. Although not strictly POD, one may define a POD-like object with Private data, accessed through Properties, with a constructor, to circumvent this restriction. Having a constructor means that a POD-like object is able to initialize the values of the data it holds. We may also refer to this more complicated type of object as semi-POD or even as POD, but we are stretching the definition a little.

Figure 16.1 shows three basic POD objects: StatePOD, wrapping three Public Doubles; StatePODudt, wrapping a Private UDT (UDTs cannot be Public data members in a class module); and StatePODarray, wrapping an array of Doubles. StatePOD has no Properties, no constructor, no interface, no nothing, but it can at least be used in a Collection. StatePODudt has getter and setter Properties for its UDT, st_ of type state (see section 16.2.1). StatePODarray, hardly a strict POD, has several sets of getter and setter Properties, and its array size can be set dynamically. It is discussed further below.

Each of these objects is used later to wrap a state composed of three Doubles.

---

5 But take care if Add()ing or Remove()ing items while iterating on the index.
A 2-dimensional array can be variously represented other than as a 2-dimensional VBA array. We look at four categories of representation: sliced (a set of 1-dimensional arrays), as a 1-dimensional array with striding, as a Variant array, or as a container structure.

We use $\hat{X}_i$ as our motivating example. For fixed $i$ and $j$ the set of values $\{\hat{X}_{i,q}^j\}_{q=1,\ldots,Q}$ is a state of the model. A slice, $\hat{X}_i = \{\hat{X}_{i,q}^j\}_{q=1,\ldots,Q}$, is a 2-dimensional array. It can be regarded as a set of $M$
states. It could also be regarded as a set of $Q$ separate 1-dimensional sample paths, but this may be less useful.

An important consideration is that having generated a slice (or a path) it will be handed over to the part of the valuation system that values options from the set of slices (or paths). The thing that is passed over had better be (i) self-contained, (ii) cheap and, if at all possible, (iii) not depend on telepathy. Self-contained means that a single object is passed, or a fixed, small, number of objects. Cheap here means not having to copy anything; you need to be able to pass a reference or a structure ByRef. Avoiding telepathy means that the receiving object knows what it has received, without having to guess. An obvious idea is therefore to make a slice an object, and this should indeed be done, but what is the data structure wrapped by the slice object?

First we discuss how states may be represented and then at how they can be assembled into slices.

### 16.2.1 States

A state should be a small lightweight entity. There is not too much functionality that could be attached to it. For instance, a state need not know how to evolve itself.6

There are three obvious ways to represent a state: as an array, a UDT, or as a set of $Q$ separate Doubles.

1. **Representation with Doubles**: Using three Doubles for the three state variables is minimalist and structure-free: a state-less state representation. It is likely to be very cheap, but also the least maintainable. This is a level 0 approach.
2. **Representation with a UDT**: One defines a simple UDT, `state`, holding three Doubles to represent the values of the three state variables. Figure 16.2 shows the Type definition. The value of $Q$ is implicitly hard-wired in. Using a UDT is level 2.
3. **Representation with an array**: Line (16.2) sets up an array, `st`, to represent a state:

   ```
   Dim st() As Double: ReDim st(1 To Q) As Double'.
   ```

   An advantage of using an array is that it is polymorphic in the sense that the number of state variables, $Q$, need not be hard-wired in.

   Each of the three state representations can be either naked or wrapped in a POD object, giving us six different possibilities, listed in Table 16.1.

   The table summarizes a view about the merits of each possibility. ‘✓’ means possibly sensible; ‘∼’ means potential misgivings; ‘×’ means potential problems. We examine the possibilities. For concreteness set $Q = 3$.

   Encapsulating a state within an object is a good design principle, and in some slice representations is highly desirable for language reasons (for instance, to enable the state to be modifiable in a Collection). It is convenient to tidy away a set of Doubles into an object, a semi-POD object can provide services to

   ```
   Private Type state
   q1 As Double
   q2 As Double
   q3 As Double
   End Type
   ```

   Figure 16.2 The UDT, state

---

6 Conceivably it might know its geometric relationship to adjacent states but we do not pursue this here.
a dynamic array, and any repetitive code used for a UDT could also in principle be tidied away inside a non-POD object.

A wrapped UDT seems redundant. If you need a POD then wrapping a set of Double is likely to be cheaper than wrapping a UDT. The latter just adds an unnecessary layer of indirection between the client and the data.

The full advantage of an array state can be achieved only if the array is wrapped in a semi-POD object. Recall the semi-POD StatePODarray object (Figure 16.1) which wraps an array. It has a SetValues() method to set the value of $Q$ and to ReDim the array $st_\cdot$ to the right size, and Properties to get and set array elements. There are three pairs of getters and setters. The first pair is for the array as a whole, the second for the (three) individual elements, and the third for array elements specified by an index. Only the first and third forms are polymorphic; of course it turns out that the second form is the cheapest.

### 16.2.2 Timing comparisons

We run timing comparisons with different reifications of the state. For instance, the code fragment

```plaintext
clock1.Reset_Timer: clock1.Start_Timer 'a
rv = StateEvolvePODarray(s, drft, sgrt, q, N) 'b
clock1.Stop_Timer 'c
```

times the execution of the procedure StateEvolvePODarray(), shown in Figure 16.3. This iterates the state, here expressed as a StatePODarray object, $N$ times. Code for other representations is analogous.
Each state is evolved either with a Function or a Sub, or plain, without using a procedure. In addition, we give times when the state is wrapped in a POD (StatePOD, StatePODudt or StatePODarray), but only for plain evolution. Figure 16.3 illustrates plain evolution with a wrapped array, using a loop.

There is no slice involved in this evolution, so there is no need to extract states from a slice before evolution or to insert them back in afterwards. The times are purely for state evolution. Section 16.3.2 gives cost comparisons when states are contained in a variety of different types of slice.

Evolution times for different state representations are given in Table 16.2 (produced with the workbook Timing_state_evolution.xls). The table shows evolution costs for \( N = 2 \times 10^6 \) iterations of the three state types – arrays, UDTs and plain Doubles – evolved in four ways: plain (wrapped and unwrapped), with a Function, and with a Sub.

The table gives times for using an array both looping through the array and unwinding the loop by having a separate statement for each index. Looping with an array is 5% or so more expensive than unwinding the loop, but looping allows greater flexibility since \( Q \) does not have to be hard-wired in. Executing a Function is always more expensive, by a few percent, than calling a Sub. Wrapping the data in a (semi-) POD is expensive when data has to be accessed with Properties. Table 16.2 shows costs for two ways of accessing wrapped array elements. 9.44 is the cost of accessing individual elements (using the non-indexed Properties \( q_1() \), \( q_2() \) and \( q_3() \) of StatePODarray in Figure 16.1); 9.67 uses the indexed Property \( q() \). The final Property (\( s_t() \) in Figure 16.1) costs a staggering 12.39.\(^7\) Wrapping Doubles is cheap because the implementation accesses them directly, without an intermediating Property call.

From the table one is tempted to use a naked UDT. This is one of the cheapest mechanisms and it conveniently locks away a state in a single variable. However there is a problem, already noted, when one tries to assemble UDTs into a slice. UDTs can be used conveniently only as items of an array. To be used in other containers it is necessary for us to wrap a UDT in an object, or, even simpler, just replace the UDT with a POD.

From the table, wrapped Doubles are cheaper than a wrapped UDT. Perhaps surprisingly a wrapped UDT is (slightly) cheaper than a wrapped array, but of course a UDT has to have \( Q \) hard-wired in.

Table 16.3 shows costs when state evolution is managed by a single procedure call. Internally, the procedures perform plain evolution. The state is passed either as a single argument to a Function or a Sub, or to a Sub as multiple arguments in the form of three separate Doubles. It is slightly cheaper to pass three Doubles than a single UDT, but for arrays the cost is about the same. It is expensive to return an array from a Function. Passing a single entity representing a state is much better design than passing separate Doubles.

\(^7\) Not reported in Table 16.2.
Table 16.3  State evolution, single calls: 2 000 000 replications

<table>
<thead>
<tr>
<th>State representation</th>
<th>Sub</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Array (loop)</td>
<td>7.22</td>
<td>7.22</td>
</tr>
<tr>
<td>UDT</td>
<td>7.18</td>
<td>6.98</td>
</tr>
<tr>
<td>Doubles</td>
<td>—</td>
<td>7.01</td>
</tr>
</tbody>
</table>

As expected it is cheaper to have a single procedure call than three (Table 16.2), and is much better design. Here the three 1-dimensional processes are assumed to be independent. If they were correlated it would be hard, certainly sub-optimal if not impossible, to make three individual procedure calls.

16.2.3 Slices

We look at constructing a slice either as separate arrays, a single 1-dimensional array with striding, or as a VBA container. We also discuss the advantages of wrapping the slice in an object. Here we stick to level 0/1 programming style. Later the slice container can be wrapped in an object.

Separate arrays

Figure 16.4 illustrates three ways to structure a slice as a 1- or 2-dimensional array. Figure 16.4a is a set of $Q$ arrays each of length $M$, 16.4b is a set of $M$ arrays of length $Q$, and 16.4c is a 2-dimensional array, which slots in here for completeness.

In our applications $Q$ is likely to be in single figures (maybe a little larger in some implementations of Libor market models), and may be as small as one or two, but $M$ is likely to be of the order of thousands or greater. This makes 16.4a attractive and rules out 16.4b. It is far easier to manage two or three arrays, each of length 50,000 say (with one entry for each sample path), than it is to manage 50,000 arrays each of length 2 or 3.

Striding

Striding refers to decomposing a higher dimensional array by rearranging it into a 1-dimensional array. For instance, the $M \times Q$ array slice $\hat{X}_i = \{\hat{X}_{i,q}\}_{q=1,...,Q}$ can be unwrapped and laid out into a 1-dimensional array.
array in two ways, illustrated in Figure 16.5. The idea is to concatenate together successive columns, 16.5a, or rows, 16.5b, into a single vector. $\hat{X}_j$ is mapped onto a 1-dimensional array $Y$ of length $MQ$. The array 16.5a maps $\hat{X}_{i,q}^j$ to $Y_{j+(q-1)M}$; 16.5b maps $\hat{X}_{i,q}^j$ to $Y_{q+(j-1)Q}$.

To extract row elements from 16.5a or columns from 16.5b an offset has to be added in, striding through the vector in leaps. The coordinates of the $j$th row in 16.5a are $(j, j + M, \ldots, j + (Q - 1)M)$ and in 16.5b the coordinates of the $q$th column are $(q, q + Q, \ldots, q + (M - 1)Q)$. When $Q = 1$ of course 16.5a amounts to the same thing as 16.5b.

In our applications it is likely to be more efficient to group contiguously the coordinates of the state vector in $\mathbb{R}^Q$, so a priori 16.5b might be preferred. In practice one would want a striding representation to be wrapped inside an object. Figure 16.6 shows how this could be done with the object SliceStride. The getter and setter Property, $q()$, implements $j$-striding on the array st_. Unfortunately this implementation runs slowly (see Table 16.7, page 274).

```
Private M_ As Long
Private q_ As Long
Private N_ As Long
Private sl_() As Double

Friend Sub SetValues(M As Long, q As Long)
    If q <> 3 Then Call Err.Raise(111, "StatePODarray", "q wrong")
    M_ = M
    q_ = q
    N_ = q_ * M_
    ReDim sl_(1 To N_) As Double
End Sub

Friend Sub Initialize(s As Double)
    Dim i As Long
    For i = 1 To N_
        sl_(i) = s
    Next i
End Sub

Property Get q(j As Long, i As Long) As Double
    q = sl_(j + (i - 1) * M_) 'computed
End Property

Property Let q(j As Long, i As Long, v As Double)
    sl_(j + (i - 1) * M_) = v 'computed
End Property
```

Figure 16.6 Wrapping a striding array: the SliceStride object
The fact that one cannot cheaply hide away the arithmetic associated with striding makes it impractical for use in a robust application.

**Container structures**

The final category is the most interesting, if not the fastest. The idea is to try to exploit the data structures natural to the language (although one might be hard pressed to beat arrays; these should be regarded as the container of default). There will be an outer container and an inner container. Figure 16.7a represents an outer container with $M$ slots, holding $Q$ items of data in each slot, and 16.7b is an outer container of size $Q$ and inner containers, each of size $M$.

The outer container could be one of a Dictionary, a Collection, Variant array, or just a plain array. The inner container could be either an array, a UDT, or a POD object containing an array, a UDT, or a set of Doubles. Some combinations do not work or are dominated by other combinations. For instance, Variant arrays, Collections and Dictionaries cannot (we declare) contain UDTs. Arrays can contain UDTs and PODs, but not other arrays. We test only the possibilities indicated in Table 16.4. In particular wrapped arrays are tested only with array outer containers, and wrapped UDTs only with arrays and Variant arrays.

We use the notation (array, UDT), for instance, to denote the structure with an array slice containing UDT states.

**Wrapped slices**

Whatever the structure representing the slice it may well be advantageous for it to be wrapped inside a façade object. This would enable it to have a suitable interface with the outside world, and to endow it

![Diagram of container structures](image)

**Figure 16.7** Container structures

**Table 16.4** Slices: possible representations

<table>
<thead>
<tr>
<th>Outer container</th>
<th>Inner container</th>
<th>Unwrapped</th>
<th>Wrapped</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Array</td>
<td>UDT</td>
</tr>
<tr>
<td>Dictionary:</td>
<td>✓</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Collection:</td>
<td>✓</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Variant array:</td>
<td>✓</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Array:</td>
<td>x</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>
with useful functionality. It would accommodate transparency to clients and polymorphism in the choice of container; the structure can then change without affecting its clients.

Mediating access to a complicated underlying structure (as in the striding example) can be useful (if expensive). In a Monte Carlo application the states form an ordered set, but in other numerical methods a slice may have a geometric structure. For instance, in a lattice method a slice organizes states into a lattice structure in one or more dimensions; in such a case a slice wrapper object could provide a vital service.

Of course providing a wrapper object has the undesirable feature of adding an extra layer of indirection in a critical part of the application. Despite this in advanced applications a wrapped slice will be usual.

16.3 NUMERICAL COMPARISONS

We test a number of likely data structures in the context of a level 0 Monte Carlo. This level was chosen to bring out as clearly as possible the effect of the scheme on performance. We consider two aspects of the implementation. The first is the choice of container; the second is the detail of how the container is evolved. It is possible that some evolution methods are more appropriate for some containers than for others.

First we look at the 1-factor case, and then at a 3-factor example. Figure 16.8 shows an example of a harness used to perform the timing test in the 1-factor case. data is a UDT that holds data read in

```vBnet
Private Sub InitializeSlice(ByRef slice() As Double, S As Double)
    Dim lb As Long: lb = LBound(slice)
    Dim ub As Long: ub = UBound(slice)
    Dim j As Long
    For j = lb To ub
        slice(j) = S
    Next j
End Sub

Private Sub EvolveSlice(ByRef slice() As Double, drift As Double, sgrt As Double)
    Dim lb As Long: lb = LBound(slice)
    Dim ub As Long: ub = UBound(slice)
    Dim S As Double: S = slice(lb)
    Dim j As Long
    For j = lb To ub
        slice(j) = slice(j) * Exp(drift + sgrt * GetNormal)
    Next j
End Sub
```

Figure 16.8 The harness
from the front-end, as in a level 2 application. `InitializeSlice()` sets the initial values in the slice, declared in line 16.8a of `main()`. `main()` iterates forwards one step at a time calling the procedure `EvolveSlice()` to perform the evolution. Line 16.8b in `EvolveSlice()` increments the `j`th entry in slice. It is altered to time the various cases presented below. The timing harness in the 3-factor case is similar. Line 16.8a in `main()` and `InitializeSlice()` both have to be altered to reflect the choice of outer container.

Times are given for \(N = 100\) times steps with \(M = 50\,000\) sample paths, so that slices are of size 50,000.

Note that the clock times the initialization of the `slice`, but not the ReDim of it. In some cases ReDiming an existing structure can be expensive; we prefer to avoid adding in this cost, if present, as part of the total.

### 16.3.1 1-factor: Evolving an array

In the 1-factor case a state is represented by a single `Double`. It is fairly clear that the best choice of outer container is a plain old array holding plain old `Doubles`. We test no alternatives; for \(M\) sample paths our container is an array of length \(M\).

There is some choice as to how the array should be evolved from one step to the next. Of course a `For`-loop should be used to iterate through the elements of the array but how should individual elements be operated upon?

At each step the `j`th element of the array is randomly incremented and put back into the array. Lines (16.4) give some examples of how this could be done for a GBM. The 1-dimensional array representing the array is named `slice`, indexed by a `Long`, `j`. `drift` and `sig_rt` represent \(\mu/\Delta t = (r - \frac{1}{2}\sigma^2) \Delta t\) and \(\sigma/\Delta t = \sigma \sqrt{\Delta t}\) respectively.

```plaintext
slice(j) = slice(j) * Exp(drift + sig_rt * GetNormal) 'a
s = slice(j): slice(j) = s * Exp(drift + sig_rt * GetNormal) 'b
slice(j) = NextSfn(slice(j), drift, sig_rt) 'c
s = slice(j): slice(j) = NextSfn(s, drift, sig_rt) 'd
Call NextSsub(slice(j), drift, sig_rt) 'e
s = slice(j): Call NextSsub(s, data): slice(j) = s 'f.
```

Line (16.4a) is the plain case: no function call. It is quite clear, and possibly the first thing to come to mind (although that might be the function call 16.4c). Line (16.4c) substitutes a function call to `NextSfn()` for the direct calculation. This is probably just as clear as (16.4a), and has more of an encapsulated spirit, but the additional two argument to the call, `drift` and `sig_rt`, look clumsy and add expense. Line (16.4e) tries to improve upon (16.4c) by replacing a function call by an argument passed `ByRef` to a `Sub`.

Lines (16.4b), (16.4d) and (16.4f) attempt to speed-up (16.4a), (16.4c) and (16.4e) by replacing an index operation by an assignment. We call this pre-extraction.

Table 16.5 gives for the possibilities in line (16.4) the cost in seconds of evolving from time \(t_0\) to time \(t_N\) with \(N = 100\) time steps and \(M = 50\,000\) sample paths. Pre- and post-evolution costs are ignored, so the comparison is between the evolution methods alone.

Times are dominated by the cost of generating random variates, so small differences in Table 16.5 reflect large differences in the performance of the evolution method.

The fastest times are for the plain evolution. Lines (16.4a) and (16.4b) are about the same, so (16.4a) is preferred because it is less tricky. The function calls are the slowest. This is not unexpected; from Table 15.5, page 254, going from level 0 to level 1 costs about the same. `Subs` are intermediate.
### Table 16.5 1-factor, comparison of evolution methods

<table>
<thead>
<tr>
<th></th>
<th>Time</th>
<th>Percent over plain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plain</td>
<td>ab</td>
<td>c d</td>
</tr>
<tr>
<td>6.22</td>
<td>6.13</td>
<td>7.27 7.29</td>
</tr>
<tr>
<td>0</td>
<td>−1</td>
<td>17 17</td>
</tr>
<tr>
<td>6.72</td>
<td>6.93</td>
<td>8 11</td>
</tr>
</tbody>
</table>

It is not clear whether avoiding an index operation by pre-extraction leads to an appreciable speed-up, if any. Since it is artificial and tricky-looking programming, one would hesitate to use it even if it were marginally faster; we happily dismiss its use.

The relative costs of a plain statement, executing a `Function`, or calling a `Sub`, are much as expected. It is around 10% slower to call a `Sub` than to use plain evolution, but using a `Sub` is much safer and clearer programming. `Functions` are about 15% more expensive than plain evolution, but no clearer than `Subs`. If there is only a small difference in performance then why not go for the clearest? We would agree with this sentiment. 10% is not too small but we would still use a `Sub` or, better yet, replace a call to a `Sub` by a request to an object.

### 16.3.2 Multi-factor state evolution

In the multi-factor case there is now some choice over the data structure. As before we restrict consideration to the 3-factor case, \( Q = 3 \). To evolve a slice one loops through it, state by state. The loop can either itself evolve each state, or it can pass them over to a `Function` or `Sub` (or an object) to take responsibility for the task.

The results here should be seen in the light of those of section 16.2.2. In that section states were considered in isolation; here they are contained in slice entities. If they cannot be accessed by reference, states need to be extracted from the slice before being modified, and then put back.

We divide the investigation into two parts. First we look at costs when the state is structureless, that is, just a set of three `Doubles`, and then at cases where the state is an array or UDT, or a POD.

#### Structureless state

We examine the cases where the underlying state is represented as, or extracted as, three `Doubles`. These involve array representations.

Table 16.6 gives times for array-style containers. These are three 1-dimensional arrays, a 2-dimensional array and two versions of striding using a 1-dimensional representation. The first three columns are for evolution either using a `Function` or a `Sub`, or plain with no procedure call. Shown are times for both looping and non-looping versions, and times when pre-extraction into a `Double` is performed.

The first entry in each bracketed pair is where there is no pre-extraction, the second is with pre-extraction. As before, the evidence is mixed as to whether pre-extraction should be used. On the whole it seems faster not to pre-extract. In any case it makes for trickier programming. Without the existence of a clear cost or design advantage to pre-extraction, we prefer to avoid it.

Looping forms are slower than non-looping forms by about 1–2%. This is not considerable; the added clarity and convenience makes up for the slight time hit.

Function calls are always slowest, often by a large margin. `Subs` are up to about 10% slower than plain evolution; `Functions` are up to around 15% slower in the worst case. With a `Function` you can return only a single entity. It seems better to use a `Sub`, with argument or arguments qualified `ByRef`, than to use a `Function`. 
### Table 16.6 Comparison of evolution times: three Doubles

<table>
<thead>
<tr>
<th>Container type</th>
<th>Evolution type</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Plain</td>
<td>Function</td>
<td>Sub</td>
<td>Single call</td>
<td></td>
</tr>
<tr>
<td>Three 1-d arrays</td>
<td>(18.70, 18.86)</td>
<td>(21.65, 21.09)</td>
<td>(20.60, 20.65)</td>
<td>19.79</td>
<td></td>
</tr>
<tr>
<td>2-d array</td>
<td>loop: (19.19, 19.46)</td>
<td>(21.95, 22.34)</td>
<td>(20.98, 22.22)</td>
<td>19.65</td>
<td></td>
</tr>
<tr>
<td></td>
<td>no-loop: (18.97, 18.84)</td>
<td>(21.64, 21.59)</td>
<td>(20.37, 21.13)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1-d array, stride by path</td>
<td>loop: (19.27, 20.00)</td>
<td>(21.49, 21.87)</td>
<td>(21.01, – )</td>
<td>19.64</td>
<td></td>
</tr>
<tr>
<td></td>
<td>no-loop: (18.86, 19.61)</td>
<td>(20.89, 21.61)</td>
<td>(20.55, – )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1-d array, stride by state</td>
<td>loop: (19.48, 20.43)</td>
<td>(21.94, 22.30)</td>
<td>(21.27, – )</td>
<td>19.82</td>
<td></td>
</tr>
</tbody>
</table>

### Table 16.7 Comparison of containers and evolution times

<table>
<thead>
<tr>
<th>Container type</th>
<th>Evolution type</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Outer</td>
<td>Inner</td>
<td>Plain</td>
<td>Function</td>
</tr>
<tr>
<td>3 × 1-d</td>
<td>n/a</td>
<td>18.70</td>
<td>21.65</td>
</tr>
<tr>
<td>2-d (loop)</td>
<td>n/a</td>
<td>19.19</td>
<td>21.95</td>
</tr>
<tr>
<td>Stride (loop)</td>
<td>n/a</td>
<td>19.27</td>
<td>21.49</td>
</tr>
<tr>
<td>Stride, POD</td>
<td>n/a</td>
<td>28.06</td>
<td>30.75</td>
</tr>
<tr>
<td>Variant</td>
<td>Array (loop)</td>
<td>21.71</td>
<td>23.43</td>
</tr>
<tr>
<td></td>
<td>POD</td>
<td>22.03</td>
<td>24.30</td>
</tr>
<tr>
<td></td>
<td>POD-UDT</td>
<td>23.52</td>
<td>25.59</td>
</tr>
<tr>
<td>Array</td>
<td>UDT</td>
<td>18.63</td>
<td>21.17</td>
</tr>
<tr>
<td></td>
<td>POD</td>
<td>20.49</td>
<td>22.67</td>
</tr>
<tr>
<td></td>
<td>POD-array</td>
<td>26.56</td>
<td>28.93</td>
</tr>
<tr>
<td></td>
<td>POD-UDT</td>
<td>21.63</td>
<td>23.72</td>
</tr>
<tr>
<td>Collection</td>
<td>Array (loop)</td>
<td>53.79</td>
<td>55.67</td>
</tr>
<tr>
<td></td>
<td>POD</td>
<td>22.19</td>
<td>25.05</td>
</tr>
<tr>
<td></td>
<td>POD, keyed</td>
<td>43.25</td>
<td>45.71</td>
</tr>
<tr>
<td>Dictionary</td>
<td>Array</td>
<td>63.81</td>
<td>65.70</td>
</tr>
<tr>
<td></td>
<td>POD</td>
<td>53.19</td>
<td>55.55</td>
</tr>
</tbody>
</table>

Striding by path \((j\text{-striding})\) is about the same as striding by state \((q\text{-striding})\) although there is variation. If anything striding by path may be a little faster. Striding is slow, and would be even slower if access to the underlying 1-dimensional array were mediated by a Function. From a design perspective wrapping a striding array in an object would be very desirable. Unfortunately the cost of using SliceStride (Figure 16.6) is getting on for half as much again as using plain striding (Table 16.7). This is far too great a cost to bear.

The fastest form is the three 1-dimensional arrays, the next fastest the 2-dimensional array, then striding. The speed difference between the fastest and the slowest is maybe only 5%, so every method is similar from the cost perspective.

The 2-dimensional array is faster than striding. This may be surprising considering how slow we found 2-dimensional arrays to be compared to 1-dimensional arrays. However the extra arithmetic involved in striding outweighs the gain in less expensive array access costs.

Since a 2-dimensional array is fairly intuitive to use, is a single entity, and is not too expensive, it is preferred to the other possibilities in Table 16.6.
The table also shows costs when evolution is by a single procedure call, in this case to a Sub taking three Doubles as arguments, ByRef. Not surprisingly there is no clear difference in cost.

**Structured state**

Costs for different outer and inner container structures are given in Table 16.7. There is no pre-extraction and, where relevant, times are shown for forms with looping. The table includes summary results from Table 16.6. ‘POD’, ‘POD-UDT’ and ‘POD-array’ inner types stand for StatePOD, StatePODudt and StatePODarray respectively (Figure 16.1). For each slice and state representation three different evolution methods are used: plain, where there are no procedure calls, and with either a Function or a Sub call.

An array slice with POD states is 10% slower than an array with UDT states. A POD wrapping a UDT is surprisingly fast, not too much slower than the plain POD. StatePODarray is very slow. Access costs here are relatively greater than those of Table 16.2.

A Collection of arrays can be looped through by index looping with removal and re-attachment. This is very expensive at over 50 irrespective of the evolution method.

With PODs, one can loop through a Collection in a For-Each loop. This is much cheaper, costing much the same as ordinary looping through arrays. By comparison, index looping through PODs with removal and reattachment costs 46.75, more than double the cost of For-Each looping. When keys are supplied to a Collection it is possible to loop on the keys. Again, this is twice as expensive as For-Each looping but still less than index looping on an array.

Some runs were attempted to index loop though a Collection of PODs (without removal and re-attachment). This works, in the sense that the PODs are modifiable, the code compiles and runs, and the right sort of answers are output. Unfortunately it appears that the cost of this form of looping rises exponentially in the number of items in the Collection. Table 16.8 summarizes. I bowed out of running this on \( M = 50000 \) sample paths.\(^8\)

**Dictionarys** are far too expensive for computational work. This is not going to surprise anyone: they are designed for a different purpose. Perhaps one should be surprised that they are only about three times the cost of the array container. A Dictionary with PODs is cheaper than with arrays. Keys here are Longs, so it is possible in this special case to loop on the key. The reported times were obtained with this mechanism; if instead the keys are first extracted into a Variant array, which is then looped through, the cost of looping goes up by about 1%.

As in Table 16.2, Function evolution is about 10% slower than plain evolution. Sub evolution is about the same.

Table 16.9 shows times where evolution is accomplished in a single procedure call. This is intended to parallel the use of an object to evolve the state. Three versions are given. Executing a Sub comes in two forms, the first where a single entity representing a state is passed (ByRef), the second where three Doubles are passed (ByRef). Executing a Function comes in a single form, where a state entity is returned.

Passing three Doubles is slower than passing a single state, and worse design. Executing a Function is usually slower than calling a Sub. Fastest, and best design, is passing a state to a Sub.

<table>
<thead>
<tr>
<th>Table 16.8</th>
<th>Index looping through a Collection of PODs</th>
</tr>
</thead>
<tbody>
<tr>
<td>( M )</td>
<td>2000</td>
</tr>
<tr>
<td>Time:</td>
<td>2.67</td>
</tr>
</tbody>
</table>

\(^8\)If the relationship really is, and continues to, exponential, then \( M = 50000 \) would take over 800 hours to run. I write this instead of waiting.
Table 16.9  Comparison of containers and evolution times: single call

<table>
<thead>
<tr>
<th>Container type</th>
<th>Sub Function</th>
<th>State</th>
<th>Doubles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variant</td>
<td>Array</td>
<td>21.30</td>
<td>22.65</td>
</tr>
<tr>
<td></td>
<td>POD</td>
<td>23.25</td>
<td>23.53</td>
</tr>
<tr>
<td></td>
<td>POD-UDT</td>
<td>24.88</td>
<td>24.61</td>
</tr>
<tr>
<td>Array</td>
<td>UDT</td>
<td>19.06</td>
<td>19.48</td>
</tr>
<tr>
<td></td>
<td>POD</td>
<td>20.70</td>
<td>22.11</td>
</tr>
<tr>
<td></td>
<td>POD-array</td>
<td>27.31</td>
<td>26.97</td>
</tr>
<tr>
<td></td>
<td>POD-UDT</td>
<td>22.91</td>
<td>22.91</td>
</tr>
<tr>
<td>Collection</td>
<td>Array (loop)</td>
<td>54.15</td>
<td>54.00</td>
</tr>
<tr>
<td></td>
<td>POD</td>
<td>23.53</td>
<td>23.26</td>
</tr>
<tr>
<td></td>
<td>POD, keyed</td>
<td>44.35</td>
<td>44.63</td>
</tr>
<tr>
<td>Dictionary</td>
<td>Array</td>
<td>63.83</td>
<td>64.14</td>
</tr>
<tr>
<td></td>
<td>POD</td>
<td>53.42</td>
<td>53.60</td>
</tr>
</tbody>
</table>

The results are in line with Table 16.7. A Collection with PODs and For-Each looping is comparable in cost to the simpler structures. Dictionaries are too expensive and cannot be made cheaper.

There are just four serious candidates for the structure of choice: a 2-dimensional array, an array slice with UDT states, an array slice with POD states, and a Collection with POD states. The 2-dimensional array is dominated by (array, UDT) which also dominates (Collection, POD). The Collection container, despite heroic effort, is the slowest of the four. It has no compensating advantages and can be dismissed from further consideration. (array, POD) might have advantages over (array, UDT), for instance if it facilitated a polymorphic design, but otherwise the 10% additional expense it incurs seems unnecessary.

We conclude that a natural, inexpensive, slice structure is an array with UDT states.

It is possible to go one step further. If one requires polymorphism, and the flexibility that comes from hiding an implementation behind an interface, we can get this by creating an object to put our slice into. Figure 16.9 shows the object SliceArrayUDT and Table 16.10 give cost comparisons with the

```
Private sl_() As state  'The slice: an array of states
Private M_ As Long

Friend Sub SetValues(s As Double, M As Long)
    M_ = M: ReDim sl_(1 To M) As state
    Dim st As state: st.q1 = s: st.q2 = s: st.q3 = s
    Dim i As Long
    For i = 1 To M_  
        sl_(i) = st
    Next i
End Sub

Friend Property Get st(j As Long) As state:  st = sl_(j): End Property
Friend Property Let st(j As Long, st As state): sl_(j) = st: End Property
```

Figure 16.9  Wrapping a slice: SliceArrayUDT
unwrapped (array, UDT) representation. The additional cost is great: about 20%. Is it worth it? It all depends on where the application is going. For basic Monte Carlo the expense is too great, but when you come on to use effective speed-up methods you may find you need the services that a slice object can provide.

### 16.4 SUMMARY

This has been an extensive chapter covering a large number of issues. We have seen that the way that data is represented and structured has a noticeable effect on the run-times of a Monte Carlo method.

At this level an array slice with UDT states is cheap to implement, clear and uncomplicated. It is the recommended structure. VBA data structures like Collections and Dictionaries have their place, but not for numerical work (although we have seen that Collections with PODs made the short list of contenders).

Wrapping states and slices in objects was investigated. With a plain Monte Carlo we were able to benchmark the cost of doing this and found it to be excessive. However, for more sophisticated implementations the additional cost will be found to be worthwhile.

### 16.5 EXERCISES

1. The PDE and lattice applications are slice based. Backwards evolution applies a set of weights to values extracted from the slice at the previous time. The entity that holds the weights is what concerns us here. The weights could be passed around, or held, as set of Doubles, or as a UDT, or be an array, or use some other structure.

   Construct versions of the PDE and lattice applications in which weights are held as either

   (a) a set of Doubles;

   (b) a UDT;

   (c) an array.

   What effect does the choice of structure have upon performance? Does the choice of one form or another lead to greater or lesser clarity, or make the application more or less flexible?

2. Consider the following three 2-factor models.

   (a) The Heston model (Heston (1993), and see also Gatheral (2006)). The Heston model is a stochastic volatility version of geometric Brownian motion. The two factors are the asset value \( S_t \) and its

---

**Table 16.10  Encapsulating a slice**

<table>
<thead>
<tr>
<th>Container type</th>
<th>Evolution type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Plain</td>
</tr>
<tr>
<td>(array, UDT)</td>
<td>18.63</td>
</tr>
<tr>
<td>SliceArrayUDT</td>
<td>22.00</td>
</tr>
</tbody>
</table>
variance, \( v_t = \sigma_t^2 \),

\[
\begin{align*}
    dS_t &= r S_t \, dt + \sqrt{v_t} S_t \, dz_t^S, \\
    dv_t &= \alpha (\mu - v_t) \, dt + \phi \sqrt{v_t} \, dz_t^v,
\end{align*}
\]

(16.5) (16.6)

with \( dz_t^S \, dz_t^v = \rho \, dt \).

Option payoffs are computed on the values of \( S_t \) in the usual way, but the evolution of \( S_t \) depends on the process \( v_t \). Prices of vanilla call exhibit a volatility smile.

(b) The structural credit model of Shimko, Tejima and Van Deventer (1993), (see also Cossin and Pirotte (2001)). The two factors are the firm value \( V_t \) and the short riskless interest rate \( r_t \),

\[
\begin{align*}
    dV_t &= r_t V_t \, dt + \sigma V_t \, dz_t^V, \\
    dr_t &= \alpha (\mu - r_t) \, dt + \phi \, dz_t^r,
\end{align*}
\]

(16.7) (16.8)

with \( dz_t^V \, dz_t^r = \rho \, dt \).

In Merton-style structural credit models the payoff \( H(V_T) \) to a pure discount bond issued by the firm with face value \( F \) maturing at time \( T \) is \( H(V_T) = \min(V_T, F) \). A Monte Carlo method generates \( M \) sample paths over \( N \) time steps obtaining a set of pairs \( \{(V_j^i, r_j^i)\}_{j=0}^{M} \). A plain Monte Carlo value \( \hat{d} \) of a risky pure discount bond (a bond making a payout only at maturity) written on the firm is

\[
\hat{d} = \frac{1}{M} \sum_{j=1}^{M} b_j \min \left( V_N^j, F \right)
\]

(16.9)

where

\[
    b_j = \exp \left( - \sum_{i=0}^{N-1} r_j^i \Delta t \right)
\]

(16.10)

is the riskless discount factor computed along the \( j \)th path.

(c) The extended 2-factor Gaussian model of interest rates, described in Brigo and Mercurio (2001).

In this interest rate model the short rate \( r_t \) is the sum of a deterministic function \( \phi(t) \) and two Gaussian processes, \( x_t \) and \( y_t \), that mean-revert to zero,

\[
\begin{align*}
    r_t &= \phi(t) + x_t + y_t, \\
    dx_t &= -\alpha x_t \, dt + \sigma_x \, dz_t^x, \\
    dy_t &= -\alpha y_t \, dt + \sigma_y \, dz_t^y,
\end{align*}
\]

(16.11) (16.12) (16.13)

where \( dz_t^x \, dz_t^y = \rho \, dt \).

A plain Monte Carlo method generates \( M \) sample paths over \( N \) time steps obtaining a set of pairs \( \{(x_t^i, y_t^i)\}_{i=0}^{N} \). A Monte Carlo estimate of the value \( \hat{b} \) of a riskless pure discount bond in
this model is
\[
\hat{b} = \frac{1}{M} \sum_{j=1}^{M} \exp \left( - \sum_{i=0}^{N-1} r_i^j \Delta t \right) \tag{16.14}
\]

where \( r_i^j = \phi(t_i) + x_i^j + y_i^j \).

For each of these three models implement a Monte Carlo valuation method to compute the values of the indicated instruments. The emphasis here is on the program design, rather than accuracy or efficiency of computation. A crude Euler discretization is acceptable. This could take the form

(a) The Heston model (Heston (1993)):
\[
S_{i+1} = S_i + r S_i \Delta t + \sqrt{v_i} S_i \sqrt{\Delta t} \varepsilon_i^1, \tag{16.15}
\]
\[
v_{i+1}' = v_i + \alpha (\mu - v_i) \Delta t + \phi \sqrt{v_i} \sqrt{\Delta t} \varepsilon_i^2, \tag{16.16}
\]
\[
v_{i+1} = (v_{i+1}')^+. \tag{16.17}
\]

Use parameter values \( S_0 = 100, r = 0.05, v_0 = 0.045, \alpha = 0.04, \phi = 0.5, \rho = 0.5 \), to value European call options with \( T = 1 \) and \( X = 90, 95, 100, 105, 110 \).

(b) Shimko, Tejima and Van Deventer (1993):
\[
V_{i+1} = V_i + r_i V_i \Delta t + \sigma V_i \sqrt{\Delta t} \varepsilon_i^1, \tag{16.18}
\]
\[
r_{i+1} = r_i + \alpha (\mu - r_i) \Delta t + \phi \sqrt{v_i} \sqrt{\Delta t} \varepsilon_i^2. \tag{16.19}
\]

Use parameter values \( V_0 = 100, \sigma = 0.2, r_0 = 0.05, \alpha = 0.2, \mu = 0.04, \phi(t_i) \equiv \phi = 0.02, \rho = -0.5 \), to value PDBs written on the firm with \( T = 1 \) and \( F = 50, 60, 70, 80, 90 \).

(c) Extended 2-factor Gaussian.
\[
x_{i+1} = x_i + \alpha_x x_i \Delta t + \sigma_x \sqrt{\Delta t} \varepsilon_i^1, \tag{16.20}
\]
\[
y_{i+1} = y_i + \alpha_y y_i \Delta t + \sigma_y \sqrt{\Delta t} \varepsilon_i^2. \tag{16.21}
\]

Use parameter values \( x_0 = 0.01, \alpha_x = 0.2, \sigma_x = 0.1, y_0 = -0.01, \alpha_y = 0.5, \sigma_y = 0.2, \rho = 0.1 \), with \( \phi(t) \equiv 0.04 \), to value riskless PDBs with \( T = 0.25, 0.5, 1, 2, 5, 10 \).

In each case \( \varepsilon_i^1 \) and \( \varepsilon_i^2 \) are normal variates with mean 0, variance 1, and correlation \( \rho \). \( (\varepsilon_i^1, \varepsilon_i^2) \) can be generated from independent standard normal variates \( (z_i^1, z_i^2) \) by setting
\[
\varepsilon_i^1 = z_i^1, \tag{16.22}
\]
\[
\varepsilon_i^2 = \rho z_i^1 + \sqrt{1 - \rho^2} z_i^2. \tag{16.23}
\]
This page intentionally left blank
Parts I to V looked at using advanced methods in VBA to implement a basic Monte Carlo method. The emphasis in those parts was on the VBA. Now we move on to look at Monte Carlo itself.

In this part we investigate how Monte Carlo can be made to run more quickly; in the next part we look at the equally important topic of making a Monte Carlo application produce unbiased results.

We cover in this part several key techniques. The first set of three methods concerns improving the construction of a set of Wiener sample paths: using antithetic variates, stratified sampling and low discrepancy sampling. These are investigated in Chapters 17, 18 and 19.

The fourth technique, the topic of Chapters 20 and 21, involves correcting the Monte Carlo estimate of the unknown value of an option by valuing simultaneously a benchmark instrument. This is the method of control variates.

The final method, importance sampling, values an option under an alternative process, chosen to reduce the standard error of the Monte Carlo estimate. The value found under the alternative process is rotated back to yield a value under the pricing process. Chapter 22 discusses this technique.

Chapter 23 ties everything together. The techniques can be used simultaneously, combining to augment one another; the chapter quantifies this for some benchmark instruments.

The methods described and implemented in this part largely represent generic examples of more general, and more powerful, modes of application. Glasserman (2004) and Jäckel (2002) and others, describe techniques that take the methods further, tailoring them to particular models and options, and achieving, in some cases, much greater efficiency gains than shown here. The reader is directed to those sources for further information. Glasserman in particular is a key reference.

Meanwhile, even for the generic examples described here, very large efficiency gains are certainly possible. Note also that we are concerned not with the speedy valuation of individual options, in isolation, important though this is, but with options in a book of options. Here generic methods, while not precluding tailored methods for individual cases, are simpler and cheaper to implement.

**Benchmark options**

The results in this part refer mainly to the valuation of a benchmark arithmetic average rate option, with strike $X = 100$ and time to maturity $T = 1$. The initial asset value is $S_0 = 100$, with riskless rate $r = 0.05$ and volatility $\sigma = 0.2$. The options have $N$ reset dates, where $N$ ranges from 4 to 64. Reset dates are at times $t_i = iT/N$, $i = 1, \ldots, N$. (Sometimes an additional date at $t_0 = 0$ may be included.)

We also price the standard European call with the same strike, final maturity time and process, and in some chapters the pricing of an exotic option with a quadratic payoff is also investigated.

Results are often presented as triplets. In the triplet

$$\left( \hat{\theta}, se, \tau \right) = \left( 6.37, (0.014), [2.33] \right)$$

the top entry is the Monte Carlo estimate of the option value, the middle entry in round brackets is the standard error and the bottom entry in square brackets is the computation time.
Efficiency gains

Given two Monte Carlo procedures, the first giving a standard error $se_1$ in time $\tau_1$, the second a standard error $se_2$ in time $\tau_2$, the relative efficiency $E = E_{1,2}$ of the second method compared to the first is

$$E = \frac{se_2^2\tau_2}{se_1^2\tau_1}. \quad \text{(VI.2)}$$

$E$ is the multiple of the time taken by the first method, that the second method needs to take to achieve the same standard error. The efficiency gain may be presented in bold alongside, or under, the triple.

OTM options

Options that are far out of the money (or far in the money) present particular problems for a Monte Carlo method. Relatively few sample paths are in the money (out of the money), and consequently the standard error of the Monte Carlo estimate is relatively large. Other difficult options are barrier options that are either very close to their barrier or very far away.

For extreme options of this sort, standard speed-up techniques can still be applied, but special techniques are available that work especially well for them. We apply these techniques to the benchmark average rate option and European call with the same specifications as above, but now with strike $X = 150$. An OTM version of the exotic option is also introduced.
This chapter focuses on a technique that affects only the generation of sample paths of the Wiener process: antithetic variates. From this perspective it is related to the methods looked at in Chapter 18: stratified sampling and low discrepancy sampling.

The emphasis in the exposition is on path-based Monte Carlo, for simplicity, but it is extremely easy to implement antithetic variates with a slice-based Monte Carlo.

Techniques that improve the sampling of an underlying Wiener process apply irrespective of the model being used or the option being valued. Simply by producing a better sample of paths for the Wiener process significant savings – sometimes very significant savings – can be made.

Often, though, the savings achieved are not all that great compared to those obtainable using other methods, but in isolation they are automatic and easy. In such circumstances there is usually no reason not to use these methods.

In this chapter we first give a little background on where Wiener path generation fits in to a Monte Carlo method, then the antithetic variate method is described, an implementation is given, and numerical results discussed.

17.1 GENERATING WIENER SAMPLE PATHS

Simulated paths of Ito processes are constructed from sample paths of Wiener processes (perhaps transformed in some way). For instance let \( S = (S_t)_{t \geq 0} \) be a geometric Brownian motion, then a sample path \( \hat{w} = (\hat{w}_t)_{t \geq 0}, \hat{w}_0 = 0 \), of a Wiener process generates immediately a path \( \hat{S} \) for \( S \),

\[
\hat{S}_t = \hat{S}_0 \exp \left( \left( r - \frac{1}{2} \sigma^2 \right) t + \sigma \hat{w}_t \right).
\] (17.1)

In discrete time one has \( \hat{S}_{t_{i+1}} = \hat{S}_{t_i} \exp ((r - \frac{1}{2} \sigma^2) \Delta t_i + \sigma \Delta \hat{w}_{t_i}) \) where \( \Delta t_i = t_{i+1} - t_i \) and \( \Delta \hat{w}_{t_i} = \hat{w}_{t_{i+1}} - \hat{w}_{t_i} \). Preparing a good set of discrete sample paths for \( w \), and investigating the resulting efficiency gains, is the topic of this chapter and the next two.

We shall be concerned with generating a set \( W = \{w^j\}_{j=1,\ldots,M} \), of \( M \) discrete sample paths \( w^j = \{w^j_i\}_{i=0,\ldots,N} \). Fully sampling a single path needs a set of \( N \) normal variates, \( \varepsilon^j = \{\varepsilon^j_i\}_{i=1,\ldots,N} \). As we see below, these are usually obtained by inverse transform from a sample of \( N \) uniform variates, \( u^j = \{u^j_i\}_{i=1,\ldots,N} \).

The vector \( u^j \in [0, 1]^N \) is a sample from the unit hypercube of dimension \( N \). Antithetic variates are a way of obtaining cheaply an improved sample \( U = \{u^j\}_{j=1,\ldots,M} \subset [0, 1]^N \). More advanced, but more effective, techniques are presented in Chapter 18.

17.2 ANTITHETIC VARIATES

An easy way to achieve speed-ups, although these might be slight, is to use antithetic variates. The principle is to generate two sample paths at the same time. The second is obtained cheaply from the first so that, effectively, you get two samples paths for the price of one. One expects a saving of \( \sim \sqrt{2} \). We see that in both theory and practice the saving can sometimes be much larger than this.
17.2.1 The antithetic variate method

Let \( w = (w_0, \ldots, w_N) \) be a discrete sample from a Wiener sample path, with \( w_0 = 0 \), and let \( S(w) = (S_0, S_1, \ldots, S_N) \) be an asset sample path generated from it (for instance, using equation (17.1)). Write \(-w = (-w_0, \ldots, -w_N)\) for the sign-reversed path obtained from \( w \) and set \( S'(w) = S(-w) \). \( S' \) and \( S \) are generated from the same underlying Wiener process sample path, but \( S' \) uses the negatives of the values used by \( S \).

The valuation procedure is:

1. Generate successively \( M \) Wiener sample paths, \( w^j = (w^j_0, \ldots, w^j_N), j = 1, \ldots, M \).
2. For each \( j \) let \( V_j^+ \) be the option value computed from \( S(w^j) \) and \( V_j^- \) the value computed from \( S'(w^j) \).
   
   Set \( V_j = \frac{1}{2}(V_j^+ + V_j^-) \).
3. Set \( V = \frac{1}{M} \sum_{j=1}^{M} V_j = \frac{1}{2M} \left( \sum_{j=1}^{M} V_j^+ + \sum_{j=1}^{M} V_j^- \right) \) as usual.

\( V \) is the option value generated with antithetic sampling. Without antithetics the option value would be just \( V^+ = \frac{1}{M} \sum_{j=1}^{M} V_j^+ \) so \( \text{var}(V^+) \) is the variance of the option value without antithetics. If \( V_j^+ \) and \( V_j^- \) are negatively correlated then \( \text{var}(V) < \text{var}(V^+) \). In fact

\[
\text{var}(V) = \frac{1}{4} \text{var}(V^-) + \frac{1}{4} \text{var}(V^+) + \frac{1}{2} \text{cov}(V^+, V^-),
\]

but \( \text{var}(V^+) = \text{var}(V^-) \) and \( \text{cov}(V^+, V^-) = \text{var}(V^+) \rho^{+-} \) where \( \rho^{+-} \) is the correlation between \( V^+ \) and \( V^- \). Hence

\[
\text{var}(V) = \frac{1}{2} \text{var}(V^+)(1 + \rho^{+-}) \leq \text{var}(V^+),
\]

and \( \rho^{+-} < 0 \Rightarrow \text{var}(V) < \frac{1}{2} \text{var}(V^+) \).

The extent to which this represents a speed-up depends on how much extra work is involved in getting the antithetic sample. There are two extreme cases. Suppose first that there is no extra work involved. This would effectively be the case if the main computational burden of a method was in getting a good sample of vectors of uniforms, \( u \). In this case there is always a speed-up as long as \( \rho^{+-} < 1 \).

Suppose, on the other hand, that generating the antithetic case takes as much computational effort as generating the thetic case. This would be the situation if the main cost of the method was in getting \( S(w) \) from \( w \). Then there would be a speed-up only if \( \text{var}(V) < \frac{1}{2} \text{var}(V^+) \), that is, only if \( \rho^{+-} < 0 \).

More generally, if the total amount of work is a factor \( k \) of the ordinary case, \( 1 \leq k \leq 2 \), then there is a speed-up only if \( \text{var}(V) < \text{var}(V^+)/k \), that is, from equation (17.3), \( \rho^{+-} < (2-k)/k \).

In any case, whatever the computational cost, the closer \( \rho^{+-} \) is to \(-1\) the better the speed-up. In practice one would expect \( \rho^{+-} \) to be negative. It would not be if the payoff was symmetric around the expected future value of asset. When the payoff is non-decreasing (or non-increasing), in the asset value the correlation is likely to be negative.

Antithetic variates can be related to samples from the unit hypercube. Suppose that \( w \) is computed, via a sample of normal variates, from an \( N \)-dimensional uniform variate \( u \in [0, 1]^N \). Then, if nothing strange is going on, \(-w\) would have been generated from the point \( 1-u = (1-u_1, \ldots, 1-u_N) \in [0, 1]^N \); whenever \( u \) is in the sample so, effectively, is the point \( 1-u \). The effect of this is to:
1. Generate cheaply a second point in \([0, 1]^N\) from each original point.
2. Ensure that the first moment of the augmented set of sample points is zero.
3. Ensure the set of sample points is symmetric around the centre of \([0, 1]^N\).

Hence the method is faster, may have less bias, and, as a consequence of 3, benefits from correlation effects if present.

### 17.2.2 Implementing antithetic sampling

Incorporating antithetic sampling is very easy. A path-based Monte Carlo generates a Wiener process path and from that a path of the underlying asset (from which an option value along that path is computed). An antithetic sampling method affects only the Wiener path generator. Having obtained a path it is used twice, once as it arrives, and again with the sign changed on every element.

Similarly a slice-based Monte Carlo evolves two slices simultaneously: the normal slice and an antithetic slice. The antithetic slice uses the same increments as the normal slice but with their signs reversed. (In fact in practice a single slice of twice the length may be evolved; the concatenation of the normal and antithetic slices.)

Here we illustrate only a path-based Monte Carlo. It is a level 3b non-polymorphic implementation.

Figure 17.1 shows the object `App_MC_anti` from `MC_AverageRate Anti timely.xls`. It is instantiated as a composited application object inside `AppObWrapper`. Input and output is managed by separate objects.

`App_MC_anti` is the object that implements the antithetic method. It computes and outputs both a plain and an antithetic estimate. (To compute only the antithetic estimate line 17.1d should be removed along with all references to the `acc_pl_` object and the corresponding properties on lines 17.1l and 17.1m.)

The main loop is very straightforward. Line 17.1a asks the `WienerGeneratorPlain` object to fill up a Wiener sample. This is put byRef into `WieVec`. 17.1b passes `WieVec` over to the `PathGenerator` which puts a GBM path into `path`. The option object, here just `PayoffCall`, computes a payoff from `path` on line 17.1c which on 17.1d is passed to the `AccumulatorPlain` object.

So far this is just plain Monte Carlo; the antithetic part comes on lines 17.1e to 17.1i. The accumulator for the antithetic method is `acc_at_`. It is updated with the thetic payoff value on 17.1e. Line 17.1f changes the sign of the elements in `WieVec`, which is then passed over to the `PathGenerator` to return another path, to get another payoff value, to accumulate into `acc_at_`.

This works but is clumsy; it is much better to delegate the detail of fetching an array of Wiener increments to a Wiener object. This is what happens in `MC_AverageRate anti.xls`. The object `App_MC_anti_better` is displayed in Figure 17.2. Instead of itself negating the Wiener vector it leaves everything to a Wiener generator object of type `WienerGeneratorAnti`. `App_MC_anti_better` is decoupled from a knowledge of just what it is that the Wiener generator is doing.

`WienerGeneratorAnti` is shown in Figure 17.3. It decorates a plain Wiener generator object, `inner_wie_`. When a client requests a Wiener sample path, by calling the `GetWienerVec()` method, the object first checks the parity of the `Private` data member `thetic_`. If this is `True` a fresh Wiener path is needed; this is supplied by `inner_wie_`. The new sample path is put in the array `WienVec_`, and then returned, and `thetic_` is set to `False`. If `thetic_` is `False`, when `GetWienerVec()` is called the previous `WienVec_` is negated (using the library procedure `LibVec::NegateVec()` and returned, and `thetic_` is set to `True`.

The procedure is elegant and simple. The returned sample paths are supplied in pairs, first an entirely new path, and then its negative.
Private out_ As OutputCounter
Private wie_ As WienerGeneratorPlain
Private acc_pl_ As AccumulatorPlain 'accumulator for plain MC
Private acc_at_ As AccumulatorPlain 'accumulator for antithetic MC
Private gen_ As PathGenerator
Private pay_ As PayoffCall
Private M_ As Long 'number of paths
Private N_ As Long 'number of time steps

Private Sub Class_Initialize()
Set out_ = New OutputCounter
Set acc_pl_ = New AccumulatorPlain
Set acc_at_ = New AccumulatorPlain
Set gen_ = New PathGenerator
Set pay_ = New Payoff
Set wie_ = New WienerGeneratorPlain
End Sub

Private Sub Class_Terminate()
Set acc_pl_ = Nothing
Set acc_at_ = Nothing
Set pay_ = Nothing
Set wie_ = Nothing
Set gen_ = Nothing
Set out_ = Nothing
End Sub

Friend Sub SetValues(ByRef data As InputManager)
Call out_.SetValues(data)
Call wie_.SetValues(data)
Call pay_.SetValues(data)
Call acc_pl_.SetValues(data)
Call acc_at_.SetValues(data)
Call gen_.SetValues(data)
M_ = data.M
N_ = data.N
End Sub

Friend Property Get pl_val() As Double: pl_val = acc_pl_.val: End Property 'j
Friend Property Get pl_se() As Double: pl_se = acc_pl_.se: End Property 'k
Friend Property Get at_val() As Double: at_val = acc_at_.val: End Property 'l
Friend Property Get at_se() As Double: at_se = acc_at_.se: End Property 'm

Friend Sub run()
Dim path() As Double: ReDim path(0 To N_) As Double 'Stock values
Dim WieVec() As Double: ReDim WieVec(0 To N_) As Double 'Wiener increments
Dim Op_value As Double 'Option value
Dim i As Long
For i = 1 To M_ 'For each sample path
Call out_.OutputCounter(i)
'XXXXXXXXXXXXXXXXXXXXXXX thetic path XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Call wie_.GetWienerVec(WieVec) 'a
Call gen_.Get_Wpath(WieVec, path) 'b
Op_value = pay_.pay_off(path) 'c
Call acc_pl_.update(Op_value) 'd
'XXXXXXXXXXXXXXXXXXXXXXX antithetic path XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Call acc_at_.update(Op_value) 'e
Call NegateVec(WieVec) 'f
Call gen_.Get_Wpath(WieVec, path) 'g
Op_value = pay_.pay_off(path) 'h
Call acc_at_.update(Op_value) 'i
Next i
End Sub
Private out_ As OutputCounter
Private wie_ As WienerGeneratorAnti
Private acc_ As AccumulatorPlain
Private gen_ As PathGenerator
Private pay_ As PayoffCall
Private M_ As Long 'number of paths
Private N_ As Long 'number of time steps

Private Sub Class_Initialize()
Set out_ = New OutputCounter
Set acc_ = New AccumulatorPlain
Set gen_ = New PathGenerator
Set pay_ = New Payoff
Set wie_ = New WienerGeneratorAnti
End Sub

Private Sub Class_Terminate()
Set acc_ = Nothing
Set pay_ = Nothing
Set wie_ = Nothing
Set gen_ = Nothing
Set out_ = Nothing
End Sub

Friend Sub SetValues(ByVal data As InputManager)
Call out_.SetValues(data)
Call wie_.SetValues(data)
Call pay_.SetValues(data)
Call acc_.SetValues(data)
Call gen_.SetValues(data)
M_ = data.M
N_ = data.N
End Sub

Friend Property Get val() As Double: val = acc_.val: End Property 'a
Friend Property Get se() As Double: se = acc_.se: End Property 'b

Friend Sub run()
Dim path() As Double: ReDim path(0 To N_) As Double 'Stock values
Dim WieVec() As Double: ReDim WieVec(0 To N_) As Double 'Wiener increments
Dim i As Long
For i = 1 To 2 * M_ 'For each sample path
    Call out_.OutputCounter(i)
    Call wie_.GetWienerVec(WieVec) 'c
    Call gen_.Get_Wpath(WieVec, path) 'd
    Dim o_val As Double: o_val = pay_.pay_off(path) 'e
    Call acc_.update(o_val) 'f
Next i
End Sub

Figure 17.2  Antithetic variates: App_MC Anti Better
Implementing Models of Financial Derivatives

Private inner_wie_ As WienerGeneratorPlain 'The inner generator
Private thetic_ As Boolean
Private WienVec_() As Double

Private Sub Class_Initialize(): Set inner_wie_ = New WienerGeneratorPlain: End Sub
Private Sub Class_Terminate(): Set inner_wie_ = Nothing: End Sub
Friend Sub SetValues(ByRef data As InputManager)
    Dim N As Long: N = data.N
    ReDim WienVec_(0 To N) As Double
    Call inner_wie_.SetValues(data)
    thetic_ = True 'first path comes from the inner generator
End Sub

Friend Sub GetWienerVec(ByRef WienerVec() As Double)
    If thetic_ Then
        Call inner_wie_.GetWienerVec(WienVec_)
    Else
        Call NegateVec(WienVec_)
    End If
    WienerVec = WienVec_
    thetic_ = Not thetic_
End Sub

17.3 NUMERICAL ASSESSMENT

We illustrate the antithetic method by applying it to the benchmark average rate option. Table 17.1 gives results as the number of reset dates varies from $N = 4$ to $N = 64$. Entries in the table are the option value, its standard error in round brackets, the computation time in square brackets, and the efficiency gain in bold.

Using antithetics increases computation time by 50% but standard errors reduce by about two-thirds giving an overall efficiency gain of about 1.5. This is not large by any reckoning, but it is still worthwhile. It exists and is easily obtained.

<table>
<thead>
<tr>
<th>Method</th>
<th>$N = 4$</th>
<th>$N = 8$</th>
<th>$N = 16$</th>
<th>$N = 32$</th>
<th>$N = 64$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plain</td>
<td>7.01</td>
<td>6.34</td>
<td>6.09</td>
<td>5.92</td>
<td>5.84</td>
</tr>
<tr>
<td></td>
<td>(0.031)</td>
<td>(0.028)</td>
<td>(0.027)</td>
<td>(0.026)</td>
<td>(0.026)</td>
</tr>
<tr>
<td></td>
<td>[1.00]</td>
<td>[1.67]</td>
<td>[3.03]</td>
<td>[5.67]</td>
<td>[10.93]</td>
</tr>
<tr>
<td>Antithetic</td>
<td>6.96</td>
<td>6.35</td>
<td>6.05</td>
<td>5.92</td>
<td>5.85</td>
</tr>
<tr>
<td></td>
<td>(0.022)</td>
<td>(0.020)</td>
<td>(0.019)</td>
<td>(0.019)</td>
<td>(0.018)</td>
</tr>
<tr>
<td></td>
<td>[1.45]</td>
<td>[2.33]</td>
<td>[4.06]</td>
<td>[7.55]</td>
<td>[14.51]</td>
</tr>
<tr>
<td></td>
<td>1.4</td>
<td>1.4</td>
<td>1.5</td>
<td>1.5</td>
<td>1.5</td>
</tr>
</tbody>
</table>

Figure 17.3 Antithetic variates: the WienerGeneratorAnti object

Table 17.1 Antithetic variates: speed-ups with an ATM average rate option, $X = 100$. (Times in seconds)
Table 17.2  Antithetic variates: speed-ups with an OTM average rate option, $X = 150$. (Times in seconds)

<table>
<thead>
<tr>
<th>Method</th>
<th>$N = 4$</th>
<th>$N = 16$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plain</td>
<td>0.0194 0.0047</td>
<td>0.0016 0.0006</td>
</tr>
<tr>
<td></td>
<td>(0.0015)</td>
<td>(2.97)</td>
</tr>
<tr>
<td>Antithetic</td>
<td>0.0198 0.0049</td>
<td>0.0011 0.0005</td>
</tr>
<tr>
<td></td>
<td>(0.0011)</td>
<td>(3.16)</td>
</tr>
<tr>
<td></td>
<td>1.8 1.6</td>
<td>1.8 1.6</td>
</tr>
</tbody>
</table>

For the OTM average rate option the situation is similar. Table 17.2 illustrates efficiency gains for this case. Introducing antithetic sampling has a slightly greater effect here than in the at the money case, but the difference is very small. The plain Monte Carlo has a standard error of around 8% of the option value for $N = 4$ rising to 12% for $N = 16$. With antithetic sampling these percentages fall to around 6% and 10% respectively. While this is an improvement something more is needed.

In both cases efficiency gains are largely due to getting a second path cheaply, not because of correlation effects per se. Further examples are given in Chapter 23 with similar conclusions.

Is antithetic sampling worth the effort? Yes. There is a cost saving, even though it may be quite slight. There is no complication in implementing the method; the only time you would not do so is if you were already doing something that encompasses it or supersedes it – using low-discrepancy sequences for instance. You also have the comfort of knowing that at one level at least you have removed bias in your simulation: the sample of normals has mean zero.

### 17.4 SUMMARY

Antithetic sampling is cheap and easy to implement. It is a universal generic method. But you don’t get something for nothing; the speed-up you get is not great. Nevertheless it should be employed automatically, as a matter of course, unless you are doing something better; the bad news is, as we see later, that often you can do something better.

Antithetic sampling also has the advantage, if used sensibly, of automatically reducing the bias in the Monte Carlo estimate.

### 17.5 EXERCISES

Consider a stock price process following a geometric Brownian motion with drift $r = 0.05$, volatility $\sigma = 0.2$ and initial value $S_0 = 100$.

1. What are the correlations between the payoffs along the thetic and antithetic paths for

   (a) a European call option with $T = 1$ and $X = 100$;

   (b) a European call option with $T = 1$ and $X = 150$;
(c) an arithmetic average rate call option with $T = 1$ and $X = 100$;

(d) an arithmetic average rate call option with $T = 1$ and $X = 150$.

For the average rate option assume that there are either $N = 4$ or $N = 64$ reset dates per year. How much extra effort is required to generate the additional sample paths, as a proportion of total computation time, in these cases?

2. What are the efficiency gains available in applying the antithetic variate method to

(a) a European call option with $T = 1$ with strike $X$ varying between 80 and 120;

(b) a discretely reset knock-out barrier option with $T = 1$, 12 equally spaced reset dates, strike $X = 100$, and barrier level $B$ varying from 105 to 155 (in steps of 10).

3. In a 2-factor model such as Heston (see page 277) two Wiener increments are needed on each time steps requiring a uniform sample $(u_1, u_2) \in [0, 1]^2$. In this case antithetic sampling generates from each pair $(u_1, u_2)$ three additional points: $(u_1, -u_2)$, $(-u_1, u_2)$ and $(-u_1, -u_2)$. What would happen if, instead, only the single additional point, $(-u_1, -u_2)$, were to be generated?
In our examples using antithetic variates enabled minor efficiency gains to be made. The technique described here, stratified sampling, and low discrepancy sampling described in Chapter 19, can secure far greater speed-ups. The emphasis is again on path-based Monte Carlo but, with some additional work, the methods can be used with slice-based Monte Carlo.

Like antithetic variates these methods are about getting better sample paths for a Wiener process. To be effective they must be used in conjunction with bridge methods. In this book we describe only generic stratified sampling methods. Glasserman (2004) discusses tailored methods that can bring greater variance reduction.

The times given in the tables in this chapter are in seconds.

18.1 STRATIFIED SAMPLING

The closer that a simulated sample matches its underlying distribution, the smaller the variance of the Monte Carlo estimate; but what does 'closer' mean? One definition is that the sample has moments similar to their theoretical values. In Chapter 26, in the context of discretizing a continuous time process, we discuss moment-matching methods that explicitly match a few low-order moments. The antithetic variate method, as a by-product, ensures that the first moment of the normal variates generated by the method is zero.

Here we discuss methods that do not attempt to match any particular moment, but try to get close to the distribution as a whole. In doing so one expects that low-order moments will, as a group, be more closely matched. The example we focus on is stratified sampling.

A stratified sample is one in which the simulated distribution matches the theoretical distribution at a finite set of quantile levels. More precisely, let \( Q = \{q_i\}_{i=0, \ldots, N} \subset [0, 1] \), with \( 0 = q_0 < q_1 < \ldots < q_{N-1} < q_N = 1 \), be a set of quantile levels and for a distribution function \( F_X \) let \( f_i = F_X^{-1}(q_i) \), \( i = 1, \ldots, N - 1 \), be the corresponding quantiles.

Let \( S = \{x^j\}_{j=1, \ldots, M} \) be a sample drawn from the distribution \( F_X \). Then \( S \) is stratified at the levels \( Q \) if
\[
\frac{1}{M} \{ x \in S \mid x < f_i \} = q_i, i = 1, \ldots, N. \tag{18.1}
\]

The empirical distribution function of \( S \) is the same as \( F_X \) at the resolution of the levels \( f_i \); at the levels \( f_i \) the histograms of \( S \) and \( F_X \) coincide,
\[
\hat{h}[f_{i-1}, f_i] = \frac{1}{M} \{ f_{i-1} \leq x^j < f_i \} = q_i - q_{i-1} = F_X(f_i) - F_X(f_{i-1}) = h[f_{i-1}, f_i] \tag{18.2}
\]
where \( \hat{h} \) and \( h \) are the empirical and theoretical probability masses respectively.
Equation (18.1) requires \( q_iM \) to be integer with \( M \geq N \). Suppose that \( N = M \) then \( q_i = i/M \), \( i = 1, \ldots, M \), and there is a single sample point \( x \) in each quantile bucket \( [q_{i-1}, q_i), i = 1, \ldots, M \). Call this a fully stratified sample. It is clear that a fully stratified sample is very regular, with no clustering at the granularity of the mesh \( m_Q = \max |q_i - q_{i-1}| = 1/M \) of \( Q \).

It is not too hard, for common distributions, to construct a fully stratified sample. Given a distribution function \( F : \mathbb{R} \to [0, 1] \) suppose that the inverse function \( F^{-1} : [0, 1] \to \mathbb{R} \) is both well defined and computable, in the sense that a ‘sufficiently accurate’ numerical approximation exists.\(^1\) If \( U = \{u^j\}_{j=1,\ldots,M} \) is a fully stratified sample from the uniform distribution \( U[0, 1] \) set \( x^j = F^{-1}(u^j) \). \( \{x^j\}_{j=1,\ldots,M} \) is a fully stratified sample from \( F \). Indeed, \( u^j \in [q_{j-1}, q_j) \iff x^j \in (f_{j-1}, f_j) \). This is just the inverse transform method applied to \( U \).

It is rather easy to obtain a fully stratified sample of \( U[0, 1] \). Set \( q_j = j/M \), \( j = 1, \ldots, M \), and let \( U = \{u^j\}_{j=1,\ldots,M} \) be an ordinary sample of \( U[0, 1] \). Set \( v^j = (u^j + j - 1)/M \). \( V = \{v^j\}_{j=1,\ldots,M} \) is a fully stratified sample of \( U[0, 1] \).

Whenever a readily computable inverse transform function exists then it is easy to get a stratified sample. This is the case for the normal distribution and for other simple, and therefore commonly used, distributions.

### 18.1.1 Bridge methods

Suppose one is given an initial value \( S_0 = S_{t_0} \) of a stochastic process \( S = (S_t)_{t \geq 0} \) and that it is possible to construct a stratified sample \( \{S^j \}_{j=1,\ldots,M} \) from \( S_t \mid S_{t_0} \). Given the set of pairs \( \{(S_0, S^j_N)\}_{j=1,\ldots,M} \) one requires a set of entire sample paths \( \{(S_0, S^1_j, \ldots, S^j_N, S^j_N)\}_{j=1,\ldots,M} \). Constructing these is where the bridge distribution comes in. For \( t_0 < t_i < t_N \) the bridge distribution is the conditional distribution of \( S_i \mid S_{t_0}, S_{t_N} \). If this is known, and can be sampled from, an entire path can be constructed by filling in at every intermediate time.

Suppose that increments are time homogeneous and independent. Write \( F_i \) for the distribution \( S_t \mid S_{t_0} \) and suppose that \( x \sim F_i \) and \( y \sim F_{N-i} \) are random variables with densities \( f_i \) and \( f_{N-i} \) and joint density function \( f_{i,N-i} \). Set \( z = x + y \) so that \( z \) has density \( f_N \). We are interested in the conditional density \( f_{i|N} \) of \( x \mid z \). We have

\[
 f_{i|N}(x) f_N(z) = f_{i,N}(x, z) = f_{i,N-i}(x, z - x), \tag{18.3}
\]

with our assumptions, so

\[
 f_{i|N}(x) = \frac{f_{i,N-i}(x, z - x)}{f_N(z)} \tag{18.4}
\]

and since \( x \) and \( y \) are independent,

\[
 f_{i|N}(x) = \frac{f_i(x) f_{N-i}(z - x)}{f_N(z)} \tag{18.5}
\]

Given a value of \( z \) this gives the density of \( x \); it is the bridge distribution of \( x \mid z \).

We give the bridge distribution for a Wiener process. It is also possible to construct bridge distributions and sampling methods for other processes. For the gamma, variance gamma, inverse Gaussian and normal inverse Gaussian processes see Ribeiro and Webber (2002, 2003).

---

\(^1\) ‘Sufficiently accurate’ is here meant only informally but it can be made rigorous.
The Wiener Process and Stratified Sampling

The bridge distribution for a Wiener process

For a Wiener process all densities are normal,

\[ f_i(x) = \frac{1}{\sqrt{2\pi i}} \exp\left(-\frac{1}{2} \frac{x^2}{i}\right) \]
\[ \sim N(0, i), \]

and the bridge distributions can be calculated quite easily. It is convenient to define \( \tau_x = i_i - t_0 \), \( \tau_y = i_N - i_i \) and \( \tau_z = i_N - t_0 \). Substituting into equation (18.5) we obtain

\[ f_{i|N}(x) = \frac{1}{\sqrt{2\pi} b} \exp\left(-\frac{1}{2} \frac{(x - az)^2}{b^2}\right) \]
\[ \sim N(za, b^2). \]

where

\[ a = \frac{\tau_x}{\tau_z}, \]
\[ b = \sqrt{\frac{\tau_y \tau_x}{\tau_z}}. \]

If we are given \( w_0 \) and \( w_N \), then \( x = i_i - w_0 \) is normally distributed with

\[ x = w_i - w_0 \sim N(za, b^2) \sim N\left((w_N - w_0) \frac{\tau_x}{\tau_z}, \frac{\tau_y \tau_x}{\tau_z}\right). \]

This is the bridge distribution for a Wiener process, also known as a Brownian bridge. Operationalizing it (and given \( w_0 \) and \( w_N \)), a value \( w_i \) for time \( i_i \) is constructed as

\[ w_i = w_0 + (w_N - w_0) \frac{\tau_x}{\tau_z} + \sqrt{\frac{\tau_y \tau_x}{\tau_z}} e_i \]
\[ = \frac{i_N - i_i}{i_N - t_0} w_0 + \frac{i_i - t_0}{i_N - t_0} w_N + \sqrt{\frac{(i_N - i_i)(i_i - t_0)}{i_N - t_0}} e_i \]

with \( e_i \sim N(0, 1) \).

18.1.2 The direction of evolution

Once values \( w_N \) have been computed from \( w_0 \) the bridge is applied to generate values of the Wiener process at intermediate times, but what order should these values be generated in? This depends on the method, the option, and the physical limits of the computer.
Some methods, such as certain moment matching methods, require the entire sample \( \{ S^j_i \}_{j=0,\ldots,N} \) to be stored. Some methods require you to store a single sample path \( S^j \) at a time, and others require a single slice \( S_i \), or several slices, to be stored.

Memory constraints are no longer binding in a way they once were. It is usually quite feasible, no problem at all, to store a sample path or a slice even for very large simulations (to store an entire sample may still be too great a burden). Even with decreasing clock times the performance of a method is more likely to be limited by computation time than by memory.

**Choices for direction**

Bridge methods are often implemented, for convenience or necessity, as slice-based. Slices are usually evolved in one of four ways depending on the order in which they are computed. The four sequences are

1. **Forwards.** Computed in order \( t_0, t_N, t_1, t_2, \ldots, t_{N-1} \);
2. **Backwards.** Computed in order \( t_0, t_N, t_{N-1}, \ldots, t_1 \);
3. **Binary chop.** Computed in order \( t_0, t_N, t_{[N/2]}, t_{[N/4]}, t_{[3N/4]}, \ldots, t_1 \) (where \( [x] \) denotes here the integer nearest to \( x \));
4. **Partial binary.** Computing the first few slices by binary chop order; filling up using plain backwards evolution.

Forwards is most efficient in cases where the option payoff is likely to be decided early on; backwards is used in methods like the Longstaff and Schwartz least squares Monte Carlo for Bermudan (or American) options, or in cases where the option payoff is likely to be decided later. Binary chop is the most efficient way\(^2\) of sampling a Wiener process at the high variance times first, but may involve storing a relatively large number of slices.

Binary chop is often used with stratified sampling methods, and with low discrepancy sampling. If one can generate a stratified or LD sample only of limited dimension, then a partial binary chop method may be used; partial binary chop is appealing when the number of time steps is larger than the maximum feasible number of stratification times. We elaborate on this next.

**Partial binary chop evolution**

Suppose that \( N = 2^K \) for some \( K \geq 1 \). Let \( 0 \leq Q \leq K \) and let \( q_i = i2^{K-Q}, i = 1, \ldots, 2^Q \). The \( q_i \) are equally spaced in the set \( \{1, \ldots, 2^K\} \) at intervals \( 2^{K-Q} \).

A partial binary chop method of order \((K, Q)\) constructs stratified slices at times \( t_{q_i} \), by binary chop. At times \( j \in (q_i, q_{i+1}) \) slices are constructed by ordinary backwards evolution from time \( q_{i+1} \) back to time \( q_i + 1 \). \( Q \) is the binary degree of the partial binary method. When \( Q = 0 \) the method is pure backwards evolution; when \( Q = K \) the method is complete binary chop.

A method of order \((K, Q)\) can be constructed while storing at most \( Q + 1 \) slices. This is clear by induction. When \( Q = 0 \) construct a slice for time \( 2^K \); the non-stratified span is of length \( 2^K \); iterate using a backwards bridge, discarding the slice at time \( t_j \) once it has been used to construct the slice at time \( t_{j-1} \). Only a single slice ever need be stored. (We presume that the slice at time \( t_0 \) needs to be neither constructed nor stored).

Now suppose that a \((K, Q)\) scheme has been constructed, \( Q < K \), so that the largest non-stratified span is of length \( 2^{K-Q} \). We construct a \((K, Q + 1)\) scheme with the largest span \( 2^{K-Q-1} \) that stores at most

\(^2\) Spectral decomposition is a little better but considerably trickier in practice. See Glasserman (2004).
\( Q + 2 \) slices. Indeed, if the \((K, Q)\) scheme has a non-stratified span of length \(2^{K-Q}\) between indexes \(q\) and \(p\), then when a backwards iteration is required from \(p\) to \(q\) instead construct by Brownian bridge a slice at time \(v = \frac{1}{2}(q + p) = q + 2^{K-Q-1}\). Temporarily store this additional slice. Iterate with backwards evolution from \(p\) to \(v\) and then from \(v\) to \(q\). Now throw away the additional slice.

This is a \((K, Q + 1)\) algorithm that stores at most \(Q + 2\) slices in which the largest span is of length \(2^{K-Q-1}\).

Such an algorithm is easy enough to implement. The only slight complication that can arise is if different stratification methods are used for different times. For instance a scheme might fully stratify at times \(N\) and \(N/2\) and at other \((K, Q)\) slice times use low-discrepancy sampling.

One would like to rank slice times in order of importance, stratifying the more important times with a better method than the times of lesser importance. Normally the times of greatest importance are those of greatest (conditional) variance. For a Wiener process the ranking is in order \(N, \frac{1}{2}N, \frac{1}{4}N, \frac{3}{8}N, \frac{1}{8}N, \frac{3}{4}N, \frac{7}{8}N, \ldots\).

We use the fact that for \(i \in [1, 2^K - 1]\), \(i\) has a unique representation as \(i = 2^k(1 + 2l)\), where \(2^k\) is the greatest power of 2 that divides \(i\). Set \(p_i = 2^k - k + l\). \(p_i\) is the importance ranking of node \(i\).

Conversely, given an importance ranking \(p\), \(p\) can be expressed uniquely as \(p = 2^r + s\) where \(2^r \leq p < 2^{r+1}\) and \(0 \leq s < 2^r\). Set \(k = K - 1 - r\) and \(l = s\) and then set \(i = 2k(1 + 2l)\). \(i\) is the node number of importance \(p\).

Partial binary chop can be used with Longstaff and Schwartz least squares Monte Carlo, and is an efficient component of methods of variance reduction in this case. As usual with partial binary chop, only \(Q + 1\) slices need to be stored while generating slices back from time \(t_N\).

Partial binary chop is a compromise between memory and efficiency, but, depending on the option, this method can work well.

If memory really is a constraint then it is perfectly feasible to run a bridge method path-based, one sample path at a time, instead of slice-based. The method cannot then be used for least squares Monte Carlo, but for many other types of path-dependent option it can work just as well as slice-based evolution, and better if more stratification times are needed than memory allows with slice-based evolution.

### 18.1.3 Applying the bridge

From a pair \((S_0, S_N)\) it is possible to construct an entire sample path, at least for processes whose bridge distributions are known. For simplicity we consider here only path-based evolution.

To get a complete set of sample paths, first a stratified sample \(\{S^j_N\}_{j=1,\ldots,M}\) for time \(t_N\) is generated, then the bridge is used to fill up sample paths from \(S_0\) at time \(t_0\) up to each \(S^j_N\).

Figure 18.1 shows code to implement a bridge. The code is in three parts: first, the client code; second, code to evolve the bridge by either binary chop, forwards or backwards evolution; and, third, the Brownian bridge code itself.

The bridge sampling procedure is \texttt{Do_BB()}\%. It takes as arguments: an array of Doubles, \texttt{BB}, holding the partially constructed Brownian bridge path; three index positions, \(i, j \) and \(N\); the time step \(d_t\); and a normal variate \(w\). \texttt{Do_BB()} constructs a bridge value for time \(t_j\) from values previously computed for times \(t_i\) and \(t_N\), and puts the result into \texttt{BB}(\(j\)).

The forwards and backwards evolution code is simple. In the procedures \texttt{BB_forward()} and \texttt{BB_back()}, \texttt{Do_BB()} is called in a loop to increment either forwards or backwards through \texttt{BB} one step at a time.

More interesting is \texttt{BB_binary()}\%. It uses recursion to fill up the vector \texttt{BB} with values. When it is called, it first generates a bridge point at the intermediate time step determined by its arguments. It then
'In the client code

Dim BB() As Double: ReDim BB(0 To N) As Double
BB(0) = 0
BB(N) = FinalValue 'Gets a final value for the final time
Call BB_binary(BB, 0, Int(0.5 + 0.5 * N), N, d_t) 'Binary chop evolution
'or Call BB_forward(BB, N, d_t) 'Forwards evolution
'or Call BB_back(BB, N, d_t) 'Backwards evolution

Sub BB_binary(ByRef BB() As Double, i As Long, j As Long, N As Long, d_t As Double)
Dim w As Double: w = GetNormal()
Call Do_BB(BB, i, j, N, d_t, w)
If j > i + 1 Then Call BB_binary(BB, i, Int(0.5 + 0.5 * (i + j)), j, d_t)
If j < N - 1 Then Call BB_binary(BB, j, Int(0.5 + 0.5 * (j + N)), N, d_t)
End Sub

Sub BB_forward(ByRef BB() As Double, N As Long, d_t As Double)
Dim i As Long
For i = 1 To N - 1 Step 1
Dim w As Double: w = GetNormal()
Call Do_BB(BB, i - 1, i, N, d_t, w)
Next i
End Sub

Sub BB_back(ByRef BB() As Double, N As Long, d_t As Double)
Dim i As Long
For i = N - 1 To 1 Step -1
Dim w As Double: w = GetNormal()
Call Do_BB(BB, 0, i, i + 1, d_t, w)
Next i
End Sub

Sub Do_BB(ByRef BB() As Double, i As Long, j As Long, N As Long, d_t As Double, _
           w As Double)
Dim c_1 As Double: c_1 = (N - j) / (N - i)
Dim c_2 As Double: c_2 = (j - i) / (N - i)
Dim c_3 As Double: c_3 = Sqr(d_t * c_1 * c_2 * (N - i))
BB(j) = c_1 * BB(i) + c_2 * BB(N) + c_3 * w
End Sub

Figure 18.1  Constructing a sample path by Brownian bridge

calls itself twice, once to fill up the side of BB to the left of the new point, and a second time to fill up the right-hand side.

Recursion can often be expensive. In this case it is not. In time trials recursion compared not unfavourably with an inelegant alternative that keeps track by hand of the times done and the times yet to be done. Note that using recursion the stack is at most \(\lceil \ln_2(N) \rceil\) calls deep; not too bad.

The code implements path-based evolution. If instead a slice is evolved, in a slice-based evolution, there are no additional problems. If evolving forwards or backwards, at most three arrays of length \(M\) need to be stored as the evolution proceeds. If binary chop evolution is used then there may be a potential issue with storage. Of course it would be possible to store every array for every time in an \(M \times N\) array, but this would be an unnecessarily profligate holistic evolution. Not only does it eat memory but we saw in Part V that in VBA it is expensive to access a 2-dimensional array. In any case we saw earlier that it is possible to do binary chop evolution while storing at most \(1 + \lceil \ln_2(N) \rceil\) arrays of length \(M\).
Private Sub GetStrat(M As Long, M1 As Long, M2 As Long, ByRef Strat() As Double)
If M <> M1 * M2 Then Call RaiseError(1234, "GetStrat", "M <> M1 * M2")
ReDim Strat(M, 2) As Double
Dim j As Long
For j = 1 To M Step 1
Dim j1 As Long: j1 = (j - 1) \ M2 'a. j1 runs from 0 to M1 - 1
Dim j2 As Long: j2 = j - j1 * M2 'b. j2 runs from 1 to M2
Strat(j, 1) = (j1 + ran0) / M1 'c.
Strat(j, 2) = (j2 + ran0 - 1#) / M2 'd.
Next j
End Sub

Figure 18.2 Stratifying the unit square

One of the really big attractions of using a bridge is that it is possible to stratify not only at the final time but also at intermediate times. To construct an entire sample path, \(N\) uniform variates \((u_1, \ldots, u_N)\) are needed (for GBM). These form a sample \(u = (u_1, \ldots, u_N) \in [0, 1]^N\) from the \(N\)-dimensional unit hypercube. It is possible in theory to construct a stratified sample from \([0, 1]^N\).

Suppose that one wants to obtain a stratified sample from \([0, 1]^K\). Divide the \(k\)th dimension into \(M_k\) intervals, \(k = 1, \ldots, K\), and set \(M = \prod_{k=1}^{K} M_k\). A coordinate \(i = (i_1, \ldots, i_K)\), \(1 \leq i_k \leq M_k\), defines a sub-hyper-rectangle

\[
I^i = \left[ \frac{i_1 - 1}{M_1}, \frac{i_1}{M_1} \right] \times \cdots \times \left[ \frac{i_K - 1}{M_K}, \frac{i_K}{M_K} \right].
\]  

(18.14)

A fully stratified sample of type \((M_1, \ldots, M_K)\) of \([0, 1]^K\) samples a single point uniformly from each sub-hyper-rectangle \(I^i\).

As a special case a stratified sample from the unit square \([0, 1]^2\) of size \(M = M_1 M_2\) is a set of pairs

\[
\left\{ (u_1^{(j,k)}, u_2^{(j,k)}) \right\}_{(j,k) \in M_1 \times M_2}
\]

where
\[
(u_1^{(j,k)}, u_2^{(j,k)}) \in \left[ \frac{j - 1}{M_1}, \frac{j}{M_1} \right] \times \left[ \frac{k - 1}{M_2}, \frac{k}{M_2} \right].
\]  

(18.15)

There is one point in each of the \(M_1 M_2\) sub-rectangles of \([0, 1]^2\).

The code in Figure 18.2 constructs a set of stratified pairs from the unit square. It fills its argument, Strat of size \(M \times 2\), with a stratified set of pairs iterating through each of the \(M_1 M_2\) cells. Constructing a sample in a sub-rectangle (lines 18.2c and 18.2d) is easy; only slightly harder is the procedure for iterating through the sub-rectangle (lines 18.2a and 18.2b).

The 2-dimensional stratified sample is used to stratify at two times. Given a pair \((u_1, u_2)\), \(u_1\) is used to construct a stratified point at time \(T\) and \(u_2\) is used to construct the draw needed for the bridge at time \(\frac{1}{2}T\). As \((u_1, u_2)\) iterates through the set of stratified pairs so the bridge becomes stratified at both times \(T\) and \(\frac{1}{2}T\).

The effect is to generate a set of paths in \(\Omega\) with much less clustering than either a purely random set of paths or a set of paths stratified only at the final time.

Stratified Sampling in \(K\)-dimensions

To stratify at times \(t_N/4, t_N/2, t_N/4, \) and \(t_N\) four stratified uniforms \((u_1, u_2, u_3, u_4)\) are required. In general to stratify at \(K\) times, a stratified sample from \([0, 1]^K\) is required. In theory stratification can be from any number of dimensions; in practice, unfortunately, diminishing returns set in.
The objective is to sample, without clustering, from the unit hypercube \([0, 1]^K\) where \(K\) is large. A fully stratified sample cannot be made; it suffers from the curse of dimensionality. For instance, consider drawing \(10^6\) points from \([0, 1]^K\) where \(K\) is small. For \(K = 1\) there is no problem; a fully stratified sample of order \(10^6\) is straightforward. For \(K = 2\) the degree of stratification falls markedly. A sample can be made with \(10^3\) draws in each dimension so that the unit square is partitioned into \(10^3 \times 10^3\) sub-squares. However, projecting down into either dimension, each is stratified into only \(10^3\) cells with \(10^3\) points in each cell. Things are even worse when \(K = 3\). For an even sample each dimension must be partitioned into just \(10^2\) cells, giving \(10^2 \times 10^2 \times 10^2\) sub-cubes overall. Along each dimension each of its \(10^2\) cells contains \(10^4\) points. This is hardly stratified at all; clustering is as great as if there were no stratification.

The solution, to generate a set of points from \([0, 1]^K\) with low clustering, is to use low-discrepancy sequences, discussed in Chapter 19.

### 18.2 IMPLEMENTING STRATIFIED SAMPLING

Stratified sampling should always be implemented. Not every option benefits to the same extent but the additional cost is minor and the savings are reliable and easily obtained. In this chapter we illustrate with only two levels of full stratification.

Figure 18.3 displays the object `App_MC_strat` in spreadsheet `MC_AverageRate_Strat.xls`. This implements stratification at either one or two times, although the only hint of this in the code is in the name of the Wiener generator object type, `WienerGeneratorStrat`. Line 18.3d returns a Wiener path stratified either just once, at the final time, or twice, at the final time and the mid-time, as specified by the client.

The key object is the new Wiener generating object, `WienerGeneratorStrat`, shown in Figures 18.4 and 18.5. The `GetWienerVec()` method is non-polymorphic and crude. It toggles between three cases in a structural `If-ElseIf` statement of the sort deprecated elsewhere in this book and replaced by a proper polymorphic construction.

\(K\) is the number of stratification times, specified by the client. \(K = 0\) is the non-stratified case; the object supplies a plain sample path. For \(K = 1\) and \(K = 2\) the normal generator object, `NormalGenerator`, is asked to provide stratified samples of normals. The `NormalGenerator` object combines the roles of a uniform generator and an object that generates normals from uniforms. There are distinct functions and are separated out in the next illustration.

- `Get_1d_WienerVec()` calls `NormalGenerator::StratNormal_1d()`. This returns the \(j\)th stratified normal out of \(M\) (see Figure 18.6). `cndev()` computes the inverse normal distribution function as usual.
- `Get_2d_WienerVec()` calls `NormalGenerator::StratNormal_2d()`, also shown in Figure 18.6. This returns, `ByRef`, a pair of normals, the \(j\)th pair out of a stratified sample of length \(M\) of the unit square with \(M_1\) buckets in the first dimension and \(M_2\) in the second (with \(M_1 \times M_2 = M\)). It is very similar to the `Sub GetStrat()` shown in Figure 18.2. The first normal is used by `Get_2d_WienerVec()` to stratify at the final time, the second to stratify at the mid-time.

Both `Get_1d_WienerVec()` and `Get_2d_WienerVec()` use a binary chop Brownian bridge to fill up an entire sample path with `BB_binary()`, shown in Figure 18.5.

### 18.3 NUMERICAL ASSESSMENT

Stratified sampling can produce significant savings. Generally, up until the point where all the path dependency is being stratified, the greater the number of stratification times the greater the efficiency gain. A European option needs stratification only at the final time; an average rate option with four reset dates needs stratification only at those four times.
Table 18.1 shows efficiency gains achievable for the benchmark option with fully stratified sampling. We consider only one or two stratification times; the first stratification time is always time $t_N$, the second, if there is one, at the mid-time, $t_{N/2}$, when we set $(M_1, M_2) = (1000, 100)$. As we have discussed, adding in additional (fully) stratified times is problematical; for three or more stratification times low-discrepancy sampling is required.
Private nor_ As NormalGenerator 'Generates the normal variates
'................ further declarations omitted ..............
Private Sub Class_Initialize(): Set nor_ = New NormalGenerator: End Sub
Private Sub Class_Terminate(): Set nor_ = Nothing: End Sub
Friend Sub SetValues(ByRef data As InputManager)
  T_ = data.T 'Final time
  rtT_ = Sqr(T_) 'sqrt(T)
  dt_ = T_ / data.N 'T/N
  rtdt_ = Sqr(dt_) 'sqrt(dt)
  M_ = data.M 'Number of sample paths
  M1_ = data.M1 'Number of buckets at final time
  M2_ = data.M2 'Number of buckets at mid time
  j_ = 0 'Number of paths so far generated
  K_ = data.K 'Number of stratification times
  Call CheckK(K_, M_, M1_, M2_)
End Sub
Friend Sub Resetj(): j_ = 0: End Sub
Friend Sub GetWienerVec(ByRef WienerVec() As Double)
  j _=j _+1
  If K_ = 0 Then
    Call GetPlainWienerVec(WienerVec) 'Does no stratification
  ElseIf K_ = 1 Then
    Call Get_1d_WienerVec(WienerVec) 'Does stratification at the final time
  ElseIf K_ = 2 Then
    Call Get_2d_WienerVec(WienerVec) 'Does strat. at final and mid time
  End If
End Sub
Friend Sub GetPlainWienerVec(ByRef WienerVec() As Double)
  Dim lb As Long: lb = LBound(WienerVec)
  Dim ub As Long: ub = UBound(WienerVec)
  WienerVec(lb) = 0#
  Dim i As Long
  For i = lb + 1 To ub
    WienerVec(i) = WienerVec(i - 1) + rtdt_ * nor_.GetNormal
  Next i
End Sub
Friend Sub Get_1d_WienerVec(ByRef WienerVec() As Double)
  Dim lb As Long: lb = LBound(WienerVec)
  Dim ub As Long: ub = UBound(WienerVec)
  WienerVec(lb) = 0#
  WienerVec(ub) = rtT_ * nor_.StratNormal_1d(j_, M_) 'jth normal strat. from M
  Call BB_binary(WienerVec, lb, Int(0.5 + 0.5 * ub), ub)
End Sub
Friend Sub Get_2d_WienerVec(ByRef WienerVec() As Double)
  Dim lb As Long: lb = LBound(WienerVec)
  Dim ub As Long: ub = UBound(WienerVec)
  Dim n1 As Double, n2 As Double
  Call nor_.StratNormal_2d(j_, M1_, M2_, n1, n2) 'Pair of strat. normals
  WienerVec(lb) = 0#
  WienerVec(ub) = rtT_ * nor_.GetNormal
  'Use n1 to strat. at final time
  Dim j As Long: j = Int(0.5 + 0.5 * (lb + ub)) 'Index of mid time
  Call Do_BB(WienerVec, lb, j, ub, n2) 'Use n2 to strat. at mid time
  If j > lb + 1 Then Call BB_binary(WienerVec, lb, Int(0.5 + 0.5 * (lb + j)), j)
  If j < ub - 1 Then Call BB_binary(WienerVec, j, Int(0.5 + 0.5 * (j + ub)), ub)
End Sub

Figure 18.4  Stratified sampling: the WienerGeneratorStrat object, path constructors
For stratified sampling the monotonically sequential generation of sample paths prevents the internally computed standard error from accurately reflecting the true error in the method. The standard error shown in round brackets in the table was computed as the standard deviation of 12 separate Monte Carlo runs. This provides only a noisy estimate. Further results, computed from 100 replications, are presented in Chapter 23.

With a single stratification the savings are not great. They are around 6 to 7 for small \( N \), and less for options with more resets. The results are patchy; the noise in both the standard deviations and in the computation times accounts for the variation in speed-up over the plain case. However the efficiency gains are real.
Table 18.1  Stratified sampling: speed-ups with an ATM average rate option, $X = 100$

<table>
<thead>
<tr>
<th>Method</th>
<th>$N = 4$</th>
<th>$N = 8$</th>
<th>$N = 16$</th>
<th>$N = 32$</th>
<th>$N = 64$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[0.98]</td>
<td>[1.68]</td>
<td>[3.04]</td>
<td>[5.67]</td>
<td>[10.93]</td>
</tr>
<tr>
<td>Plain</td>
<td>6.99</td>
<td>6.41</td>
<td>6.02</td>
<td>5.89</td>
<td>5.87</td>
</tr>
<tr>
<td></td>
<td>(0.031)</td>
<td>(0.028)</td>
<td>(0.027)</td>
<td>(0.026)</td>
<td>(0.026)</td>
</tr>
<tr>
<td>Stratified, 1 level</td>
<td>6.97</td>
<td>6.37</td>
<td>6.04</td>
<td>5.91</td>
<td>5.83</td>
</tr>
<tr>
<td></td>
<td>(0.011)</td>
<td>(0.015)</td>
<td>(0.008)</td>
<td>(0.013)</td>
<td>(0.012)</td>
</tr>
<tr>
<td></td>
<td>[1.39]</td>
<td>[2.33]</td>
<td>[4.33]</td>
<td>[8.32]</td>
<td>[16.18]</td>
</tr>
<tr>
<td></td>
<td>5.8</td>
<td>2.6</td>
<td>7.0</td>
<td>2.5</td>
<td>2.9</td>
</tr>
<tr>
<td>Stratified, 2 levels</td>
<td>6.945</td>
<td>6.358</td>
<td>6.060</td>
<td>5.917</td>
<td>5.832</td>
</tr>
<tr>
<td></td>
<td>(0.0053)</td>
<td>(0.0071)</td>
<td>(0.0066)</td>
<td>(0.0053)</td>
<td>(0.0053)</td>
</tr>
<tr>
<td></td>
<td>[1.41]</td>
<td>[2.40]</td>
<td>[4.46]</td>
<td>[8.34]</td>
<td>[16.35]</td>
</tr>
<tr>
<td></td>
<td>23.3</td>
<td>10.9</td>
<td>11.1</td>
<td>16.1</td>
<td>15.5</td>
</tr>
</tbody>
</table>

Table 18.2  Fully stratified sampling: an OTM average rate call option, $X = 150$

<table>
<thead>
<tr>
<th>Method</th>
<th>$N = 4$</th>
<th>$N = 16$</th>
<th>$N = 64$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[1.4]</td>
<td>[3.1]</td>
<td>[9.9]</td>
</tr>
<tr>
<td>Plain</td>
<td>0.0238</td>
<td>0.0045</td>
<td>0.0040</td>
</tr>
<tr>
<td></td>
<td>(0.0017)</td>
<td>(0.00066)</td>
<td>(0.00068)</td>
</tr>
<tr>
<td>Stratified, 1</td>
<td>0.0185</td>
<td>0.0046</td>
<td>0.0038</td>
</tr>
<tr>
<td></td>
<td>(0.00099)</td>
<td>(0.00061)</td>
<td>(0.00060)</td>
</tr>
<tr>
<td></td>
<td>[1.8]</td>
<td>[4.6]</td>
<td>[15.8]</td>
</tr>
<tr>
<td></td>
<td>2.5</td>
<td>0.8</td>
<td>0.8</td>
</tr>
<tr>
<td>Stratified, 2</td>
<td>0.0181</td>
<td>0.0052</td>
<td>0.0034</td>
</tr>
<tr>
<td></td>
<td>(0.00078)</td>
<td>(0.00054)</td>
<td>(0.00042)</td>
</tr>
<tr>
<td></td>
<td>[1.9]</td>
<td>[4.7]</td>
<td>[15.8]</td>
</tr>
<tr>
<td></td>
<td>3.8</td>
<td>1.0</td>
<td>1.6</td>
</tr>
</tbody>
</table>

An additional stratification time increases the speed-up, maybe by a factor of 3 or 4. This is not unexpected. For a highly path-dependent option like an average rate option, dependent on asset values at reset times across the whole period it is alive, an additional stratification time in the middle of the period has very great benefit.

For options that are way out or way in the money, the affect of stratification is nothing like as great. Table 18.2 (with results produced from the spreadsheet MC_speedups_all.xls) illustrates efficiency gains for an OTM average rate call (with strike $X = 150$). When the number of stratification times is large compared to the number of reset dates (the case when $N = 4$), there are slight savings, but in the other two cases efficiency gains are very small. Indeed due to its additional cost, stratification may even be detrimental to performance. In Chapter 19 we see that only when further stratification times are introduced are substantial efficiency gains possible.

Where to stratify?

If one is stratifying at more than one time there is a choice of where the stratification is emphasized. Table 18.3 shows the affect of varying the degree of stratification between two stratification times. The
Table 18.3  Stratified sampling: effect of varying stratifications

<table>
<thead>
<tr>
<th>Stratification and average rate options</th>
<th>( M = 100,000 )</th>
<th>( N = 4 )</th>
<th>( N = 16 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T/2 )</td>
<td>( T )</td>
<td>( T/2 )</td>
<td>( T )</td>
</tr>
<tr>
<td>1</td>
<td>100,000</td>
<td>6.93</td>
<td>6.05</td>
</tr>
<tr>
<td>10</td>
<td>10,000</td>
<td>6.951</td>
<td>6.060</td>
</tr>
<tr>
<td>100</td>
<td>1,000</td>
<td>6.949</td>
<td>6.049</td>
</tr>
<tr>
<td>200</td>
<td>500</td>
<td>6.930</td>
<td>6.061</td>
</tr>
<tr>
<td>500</td>
<td>200</td>
<td>6.936</td>
<td>6.055</td>
</tr>
<tr>
<td>1000</td>
<td>100</td>
<td>6.936</td>
<td>6.055</td>
</tr>
<tr>
<td>10,000</td>
<td>10</td>
<td>6.943</td>
<td>6.039</td>
</tr>
<tr>
<td>100,000</td>
<td>1</td>
<td>6.95</td>
<td>6.03</td>
</tr>
</tbody>
</table>

Table shows results for two of our standard ATM average rate options; the first with \( N = 4 \) reset dates, the second with \( N = 16 \) reset dates. There are \( M = 100\,000 \) sample paths. The first column shows the number of stratification levels at the mid-time, the second column the number of stratification levels at the final time. The first option is more dependent on the pair of times \( \{T/2, T\} \) than the second; they contribute more to the payoff. This results in higher efficiency gains brought by stratification – twice the gains to the second option. The greatest gain for the first option is with levels \( (100, 1000) \) and for the second at levels \( (1000, 100) \), although it is clear that there is noise in the results, and efficiency gains are very similar.
Table 18.4  Speed-ups with stratified sampling and antithetic variates, average rate option $X = 100$

<table>
<thead>
<tr>
<th>Option</th>
<th>Plain</th>
<th>Antithetic</th>
<th>Stratified</th>
<th>Anti. + Strat.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Results</td>
<td>$E$</td>
<td>Results</td>
<td>$E$</td>
</tr>
<tr>
<td>$N = 4$</td>
<td>6.96</td>
<td>(0.030)</td>
<td>1.1</td>
<td>6.944</td>
</tr>
<tr>
<td></td>
<td>[1.3]</td>
<td></td>
<td>[1.1]</td>
<td></td>
</tr>
<tr>
<td>$N = 16$</td>
<td>6.04</td>
<td>(0.026)</td>
<td>1.3</td>
<td>6.063</td>
</tr>
<tr>
<td></td>
<td>[3.0]</td>
<td></td>
<td>[2.3]</td>
<td></td>
</tr>
<tr>
<td>$N = 64$</td>
<td>5.86</td>
<td>(0.026)</td>
<td>1.5</td>
<td>5.827</td>
</tr>
<tr>
<td></td>
<td>[9.8]</td>
<td></td>
<td>[6.4]</td>
<td></td>
</tr>
</tbody>
</table>

across a range of levels for each option. If anything the first option has more of its value explained by the value of the underlying at the final time than the second option.

For these options, where each time contributes with the same weight to the final payoff, there seems no reason not to split stratification fairly evenly between stratification times.

**Combining stratified sampling and antithetic sampling**

Stratified sampling and antithetic variates can be used together. To operationalize this simply replace in the antithetic method the decorated `WienerGeneratorPlain` object in `WienerGeneratorAnti` with a `WienerGeneratorStrat` object. Results are given in Table 18.4 (produced with MC_speedups_all.xls). Standard errors are obtained by computing the standard deviation of option values from 100 replications of the Monte Carlo method. Stratification is at two times. There is noise in the results but it is clear that the efficiency gain is greater when both methods are employed together. The effect here is approximately multiplicative. The effect of stratification in the ATM case decreases as the number of reset dates increases.

### 18.4 SUMMARY

For path-dependent options stratification should always be used whenever bridge methods are available. These depend heavily on a knowledge of how to sample conditional distributions of the underlying processes. Although these are known for Wiener processes, so that they can be used for SDEs with solutions that are time-changed Wiener processes, in general they are not known and in these circumstances bridge methods cannot be used.

Bridge methods make sense only if used in conjunction with stratified sampling or low discrepancy sequences. The latter work extremely well with any type of option. If the inverse distribution function can be computed to a good enough approximation, then stratified sampling can and should be used.
18.5 EXERCISES

Consider a stock price process following a geometric Brownian motion with drift $r = 0.05$, volatility $\sigma = 0.2$ and initial value $S_0 = 100$.

1. Add in a third level of stratification to MC_AverageRate_Strat.xls. Are the results really that bad?

2. Create an object to reify a discretely reset lookback call option. This has payoff

$$h(S_T) = \max(0, S_T - m_T)$$

at time $T$, where $m_T = \min_{i=1}^{N} S_{t_i}$ for a set $\mathcal{T} = \{t_i\}_{i=1}^{N}$ of reset dates. What stratification times are most effective for this option? Is the final time still the most important, followed by the mid-time?

3. Suppose that a (small) book contains just four options. They are all ATM European calls with times to maturity $T_1 = 0.5$, $T_2 = 1$, $T_3 = 1.5$ and $T_4 = 2$. You value all four options simultaneously from a single set of sample paths going out to time 2. If you stratify at only a single time, which time should this be? If you are able to stratify at two times, which should these be?
This page intentionally left blank
Low-Discrepancy Sampling

Low-discrepancy sampling, as described in Chapter 18, can go only so far before dimensional effects render it unworkable. To extend stratification to three or more dimensions a different type of sampling is required: low-discrepancy sampling.

In this chapter we sketch how these sampling methods are used and describe an implementation of one of the most commonly used LD sequences: the Sobol’ sequence. Its effectiveness and limitations are discussed.

19.1 LOW-DISCREPANCY SAMPLING

A low-discrepancy sequence is a deterministically generated sample from the unit hypercube that fills the hypercube evenly; this means that integrating with these points leads to estimates with reduced error. ‘Low discrepancy’ has a precise definition that translates informally as low clustering. There is an enormous literature on this topic. As usual I cite only the books of Glasserman (2004) and Jäckel (2002). We refer to sampling with low-discrepancy sequences as LD sampling, and to stratification with random draws as fully stratified sampling.

There are many types of low-discrepancy sequence. One of the most widely used in finance is the Sobol’ sequence. It is cheap and simple to generate and if used appropriately (and cautiously) gives good results. It is beyond the scope of this book to describe in detail the theory behind this or any other LD sequence; instead please refer to Glasserman, Jäckel, *Numerical Recipes*, or any of a variety of other sources. What I do here is to describe some of the features of Sobol’ sequences, give an implementation, and assess its practical effectiveness.

For a fixed dimension $K > 0$ a Sobol’ sequence is a deterministic sequence $\{s^K_j\}_{j \geq 0} \subseteq [0, 1]^K$. Each $s^K_j = (s^K_{j,1}, \ldots, s^K_{j,K})$ is a $K$-dimensional vector. Various projections of $\{s^K_j\}_{j=1,\ldots,N}$ for $N = 10,000$ are given in Figure 19.1. The panels in the figure show plots of $(s^K_{j,m}, s^K_{j,n})_{j=1,\ldots,N}$ for various pairs $(m, n)$. This is looking sideways through the unit hypercube through the $(m, n)$-face.

For low values of $(m, n)$, illustrated in the upper row of the figure, the unit square is covered fairly evenly with points; to the eye there is very little clustering. However for larger values of $(m, n)$, for instance in the examples of the lower row, clustering can become pronounced. All three projections are very blocky. This type of block structure, and other patterns of clustering, is bad. Integration in those dimensions will have large errors.

The Sobol’ sequence is actually a family of sequences. Sobol’ sequences depend upon a set of so-called direction numbers, and there is some freedom of choice over these. It turns out that the clustering quality of the generated Sobol’ sequence is sensitive to the choice of direction numbers. A poor choice leads to a sequence with unnecessary clustering. Figure 19.1 uses direction numbers due to Bratley and Fox (1988). These direction numbers work up to maybe 20 or so dimensions with acceptable levels of clustering. Beyond this, as Figure 19.1 illustrates, problems develop. Jäckel discusses the issue at length, describing in full the shortcomings of this choice, and other practical issues concerning the use in real life of LD sampling. He provides guidance on how better sequences can be implemented.

The Bratley and Fox direction numbers, used in *Numerical Recipes*, are public domain. There are commercial packages that generate Sobol’ sequence numbers that are effective into much higher dimensions.
Implementing Models of Financial Derivatives

Figure 19.1 Low-discrepancy samples: Projections onto facets

Projection onto dimensions 11 and 23

Projection onto dimensions 1 and 19

Projection onto dimensions 1 and 9

Projection onto dimensions 6 and 17

Projection onto dimensions 28 and 29

Projection onto dimensions 29 and 39

Projection onto dimensions 11 and 23

Projection onto dimensions 11 and 23

(m, n) = (6, 17)

(m, n) = (1, 19)

(m, n) = (1, 9)

(m, n) = (28, 29)

(m, n) = (29, 39)

(m, n) = (11, 23)
Private prim_poly_() As Long 'Specifies the primitive polynomials
Private degrees_() As Long 'Degree of the primitive polynomials
Private Const MAXBIT_ As Long = 30 'Word length
Private Const MAXDIM_ As Long = 39 'Maximum number of dimensions
Private x_big_() As Long 'Successive (unscaled) sobol sequence numbers
Private coeff_values_() As Long
Private factor_ As Double 'scaling factor_ to get double in range [0,1]
Private step_number_ As Long 'Incremented by 1 on each call of sobseq
Private MAXSTEPS_ As Long

Private Sub Class_Initialize()
    ReDim x_big_(1 To MAXDIM_) As Long 'Successive (unscaled) sobol numbers
    ReDim coeff_values_(1 To MAXBIT_, 1 To MAXDIM_) As Long
    ReDim prim_poly_(1 To MAXDIM_) As Long 'Specifier of the primitive polynomial
    ReDim degrees_(1 To MAXDIM_) As Long 'Degree of the primitive polynomials
    factor_ = 2 & ^ -MAXBIT_ 'scale by factor_ to create double in range [0,1]
    MAXSTEPS_ = 2 & ^ MAXBIT_

    'XXXXXXXXXXXXXXX Initialises. Uses Bratley and Fox numbers XXXXXXXXXXXXXXXXXXXXXXXXXXX
    degrees_(1) = 1: prim_poly_(1) = 0 'B&F: 3. NR = (BF - (1 + 2^degree))/2
    '....... 37 lines omitted .......
    degrees_(39) = 8: prim_poly_(39) = 21 'B&F: 299

    Call Reset
End Sub

Friend Sub Reset()
    coeff_values_(1, 1) = 1 'Initial values. Poly 1
    '....... Many lines lines omitted .......
    coeff_values_(8, 39) = 39
    Dim j As Long, k As Long, l As Long
    Dim powers() As Long: ReDim powers(0 To MAXBIT_) As Long: powers(0) = 1 &
    For k = 1 & To MAXDIM_
        powers(k) = 2 & * powers(k - 1)
    Next k
    For k = 1 & To MAXDIM_
        x_big_(k) = 0 &
    Next k
    For k = 1 & To MAXDIM_
        For j = 1 & To degrees_(k)
            'Just normalise the given values
            coeff_values_(j, k) = coeff_values_(j, k) * powers(MAXBIT_ - j)
        Next j
        For j = degrees_(k) + 1 & To MAXBIT_
            'Work out the rest by recursion
            Dim current_poly As Long: current_poly = prim_poly_(k)
            Dim coeff As Long: coeff = coeff_values_(j - degrees_(k), k)
            coeff = coeff Xor CLng(Int(coeff / powers(degrees_(k))))
            For l = degrees_(k) - 1 & To 1 & Step -1 &
                If current_poly And 1 & Then 'If 1 in l'th place
                    coeff = coeff Xor coeff_values_(j - l, k)
                End If
            Next l
            current_poly = CLng(Int(current_poly / 2 &))
        Next j
    Next k
    step_number_ = 0 & 'Incremented by 1 on each call of sobseq
End Sub

Figure 19.2 The SobolGenerator object: initialization
VBA is not the ideal language to write efficient code to generate Sobol’ sequence numbers. The best techniques rely on bit-arithmetic operations, not directly available in VBA. However an implementation is given in the SobolGenerator object (Figures 19.2 and 19.3). This is based on the Numerical Recipes implementation of Bratley and Fox, transcribed from C++ into VBA. It works surprisingly well, despite the absence in VBA of bit operators. Instead, one divides or multiplies by an appropriate power of 2. This looks odd, and is inefficient, but it works. The SobolGenerator object generates LD samples from the unit hypercube. It is used in the spreadsheet sobol_40.xls to generate the numbers that produced Figure 19.1.

SobolGenerator has two interface methods: sobseq() generates and returns successive Sobol’ numbers and Reset() resets the sequence to start over again. Its constructor initializes the sequence by setting up the initial direction numbers. Sobol’ sequence numbers in dimension \( K' \) are the projections of numbers in higher dimensions. If \( s^K_j = (s^K_{j,1}, \ldots, s^K_{j,K}) \) in \( K \) dimensions then for all \( K' \leq K \) we have \( s^K_{j,i} = s^K_{j,i} \) for all \( 1 \leq i \leq K' \).

### 19.2 IMPLEMENTING LD SAMPLING

Where possible the maximum feasible number of stratification dates should be used. This ensures not only that path-dependent options receive regularized paths but also that non-path-dependent options are stratified at their final maturity date. The illustrations go up to 32 levels of LD stratification. They stop at 32 (and should probably stop at 16) because the LD sequence used is a Sobol’ sequence based on the Bratley and Fox (1988) realization.

In the implementation described here the final time is always fully randomly stratified. Only the remaining stratification times are actually low discrepancy. This is to ensure that there is at least some noise in the Monte Carlo estimate, so that there is a chance that the standard deviation of the Monte Carlo remains a reasonable indicator of the error in the estimate. Whether this is effective or not is debatable but, without
Private nor_ As NormalGeneratorLD
Private M_ As Long
Private N_ As Long
Private K_ As Long 'Number of stratifications
Private Const Kmax_ As Long = 32 'maximum # of allowed sampling levels
Private T_ As Double 'Final time
Private dt_ As Double 'T/N
Private rtdt_ As Double 'sqrt(dt)
Private rtT_ As Double 'sqrt(T)
Private j_ As Long 'Number of samples generated so far

Private Sub Class_Initialize(): Set nor_ = New NormalGenerator: End Sub
Private Sub Class_Terminate(): Set nor_ = Nothing: End Sub
Friend Sub SetValues(ByRef data As InputManager)
    N_ = data.N
    M_ = data.M
    T_ = data.T 'Final time
    rtT_ = Sqr(T_) 'sqrt(T)
    dt_ = T_ / N_ 'T/N
    rtdt_ = Sqr(dt_) 'sqrt(dt)
    K_ = data.k '# stratifications
    If K_ > N_ Then Call Err.Raise(123, "WienerGeneratorLD", "K > N")
    If K_ > Kmax_ Then Call Err.Raise(123, "WienerGeneratorLD", "K > Kmax")
End Sub
Friend Sub Reset()
    j_ = 0
    Call nor_.Reset
End Sub

Friend Sub GetWienerVec(ByRef WienerVec() As Double)
    j_ = j_ + 1
    Dim lb As Long: lb = LBound(WienerVec)
    Dim ub As Long: ub = UBound(WienerVec)
    Dim Norms() As Double: ReDim Norms(lb + 1 To ub) As Double
    Call nor_.LD_NormVec(Norms, j_, M_, K_)
    WienerVec(lb) = 0#
    WienerVec(ub) = rtT_ * Norms(ub)
    Dim j As Long: j = Int(0.5 * (lb + ub + 1)) 'Index of mid time
    Call BB_binary(WienerVec, lb, j, ub, Norms)
End Sub

Private Sub BB_binary(ByRef BB() As Double, i As Long, j As Long, N As Long, Norms() As Double)
    Dim c_1 As Double: c_1 = (N - j) / (N - i)
    Dim c_2 As Double: c_2 = (j - i) / (N - i)
    Dim c_3 As Double: c_3 = Sqr(dt_ * c_1 * c_2 * (N - i))
    BB(j) = c_1 * BB(i) + c_2 * BB(N) + c_3 * Norms(j)
End Sub

Private Do_BB(BB, i, j, N, Norms() As Double)
    Dim c_1 As Double: c_1 = (N - j) / (N - i)
    Dim c_2 As Double: c_2 = (j - i) / (N - i)
    Dim c_3 As Double: c_3 = Sqr(dt_ * c_1 * c_2 * (N - i))
    BB(j) = c_1 * BB(i) + c_2 * BB(N) + c_3 * w
End Sub

Figure 19.4  Low-discrepancy sampling: the WienerGeneratorLD object
it, low-discrepancy methods return deterministic estimates with no noise. It is then hard to judge the size of the error, although below we try to give some idea of the error with the reference option. See Jäckel (2002) for further discussion.

LD sampling is implemented in the WienerGeneratorLD object, displayed in Figure 19.4. The object that instantiates it, App_MC_LD in spreadsheet MC_AverageRate_LD.xls, is almost identical to App_MC_strat in Figure 18.3. The only difference is that it instantiates a WienerGeneratorLD object instead of a WienerGeneratorStrat object. App_MC_LD is not shown.

WienerGeneratorLD has the same Public methods as WienerGeneratorStrat. The GetWienerVec() method returns a Wiener sample path constructed by binary chop where normal increments are provided by a NormalGeneratorLD object (Figure 19.5). This in turn constructs normal increments from uniform variates provided by a UniformGenerator object (Figure 19.6), and this calls on a composited SobolGenerator object to provide LD samples.

When it requires an array of normals the WienerGeneratorLD::GetWienerVec() method calls the NormalGeneratorLD::LD_NormVec() method. The arguments to this method are an array passed ByRef to hold the returned normals, the index number \( j \) of the sample out of the total sample size \( M \), and the degree \( K \) of the stratification. NormalGeneratorLD::LD_NormVec() just calls on the UniformGenerator::LD_UniVec() method to supply an array of uniforms and applies inverse transform to them.

All the real work is done by the UniformGenerator object. It is responsible for providing the required stratification. The LD_UniVec() method has the same arguments as NormalGeneratorLD::LD_NormVec().

```
'XX NormalGeneratorLD
'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX data section XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Private uni_ As UniformGenerator
'...................... Remaining declarations omitted ......................
'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Private Sub Class_Initialize()
    Set uni_ = New UniformGenerator
    '...................... Remaining statements omitted ......................
End Sub
Private Sub Class_Terminate()
    Set uni_ = Nothing
End Sub
Friend Sub Reset()
    Call uni_.Reset
    '...................... Remaining statements omitted ......................
End Sub
'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX interface XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Friend Sub LD_NormVec(ByRef Norms() As Double, j As Long, M As Long, k As Long)
    Dim lb As Long: lb = LBound(Norms)
    Dim ub As Long: ub = UBound(Norms)
    Dim Unis() As Double: ReDim Unis(lb To ub) As Double
    Call uni_.LD_UniVec(Unis, j, M, k)
    Dim i As Long
    For i = lb To ub Step 1
        Norms(i) = cndev(Unis(i)) 'inverse transform
    Next i
End Sub
'...................... Remaining procedures omitted ......................
'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
'XX end of file
'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

Figure 19.5 Low-discrepancy sampling: the NormalGeneratorLD object
If \( K \) is zero the array argument \( \text{Unis} \) is filled entirely with randomly generated uniforms. If \( K \) is 1, the final slot in \( \text{Unis} \) is filled with a fully stratified sample and the remaining elements are purely random as before.

The interesting case is when \( K \) is 2 or more. Then the \textit{UniformGenerator} asks the \textit{SobolGenerator} for an LD sample of length \( K - 1 \). These are inserted into regularly spaced locations in \( \text{Unis} \) with the remaining slots filled, as before, with plain uniforms. If \( K \) divides \( N \) exactly, \( N / K = L \), say, then the LD
numbers are put into slots $iL, i = 1 \text{ to } K - 1$. If $K$ does not divide $N$ then the numbers are put in the
slots with indexes $\text{Int}(i \times L)$. The $N$th slot is filled with a fully stratified sample as before.

The effect is that in the Wiener sample path returned by $\text{WienerGeneratorLD}$ the final time is stratified
by fully stratified sampling, and other times, those that the $\text{UniformGenerator}$ object puts LD numbers
into, are stratified by LD sampling. For instance if $N = 16$ and $K = 4$ then the Wiener sample path will
be of length 17, running from time 0 to time $N \Delta t$. The final time, index position 16, is fully stratified;
index positions 4, 8 and 12 are LD stratified.

19.3 NUMERICAL ASSESSMENT

We apply LD sampling to the reference average rate option. Table 19.1 shows some results. The final time
is always fully stratified, the remaining stratification times are LD sampled.

Results for LD sampling at two levels are much the same as those for fully stratified sampling at two
levels (Table 18.1, page 302). As further levels are introduced, LD sampling continues to experience
efficiency gains. Up until the final stratification level the standard error roughly halves for each doubling
of the number of stratification times while the computation time remains approximately constant. The
efficiency gain is quadrupling at each stage. This is a very worthwhile speed-up.

Care must be taken in interpreting the results for the last stratification level. The apparently enormous
speed-ups are not what they seem.

The problem is that LD samples are deterministic. A fully LD-sampled option value, with no randomness
anywhere, would have a standard error of zero, yet the option value would still have error in it. The error
would reduce as the number of sample points increases, but error there remains.

For instance, in Table 19.1 the value of 6.93950 for the option with $N = 4$ resets valued with 4 LD
stratification levels is supposed to have a standard error of $\sim 5 \times 10^{-5}$ on the fifth decimal place. This
accuracy is spurious. It reflects only the determinism of the LD sample.

Table 19.1 Low-discrepancy sampling: speed-ups with an average rate option, $X = 100$

<table>
<thead>
<tr>
<th>Method</th>
<th>$N = 4$</th>
<th>$N = 16$</th>
<th>$N = 64$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Results</td>
<td>$E$</td>
<td>Results</td>
</tr>
<tr>
<td>LD, plain</td>
<td>6.94</td>
<td>(0.031)</td>
<td>6.07</td>
</tr>
<tr>
<td></td>
<td>[1.63]</td>
<td></td>
<td>[4.69]</td>
</tr>
<tr>
<td>LD, 2 levels</td>
<td>6.941</td>
<td>(0.0056)</td>
<td>6.057</td>
</tr>
<tr>
<td></td>
<td>[1.89]</td>
<td></td>
<td>[5.19]</td>
</tr>
<tr>
<td>LD, 4 levels</td>
<td>6.93950</td>
<td>$(5.2 \times 10^{-5})$</td>
<td>6.058</td>
</tr>
<tr>
<td></td>
<td>[1.84]</td>
<td></td>
<td>[5.04]</td>
</tr>
<tr>
<td>LD, 8 levels</td>
<td>–</td>
<td></td>
<td>6.057</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>[5.02]</td>
</tr>
<tr>
<td>LD, 16 levels</td>
<td>–</td>
<td></td>
<td>6.05846</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>[4.83]</td>
</tr>
<tr>
<td>LD, 32 levels</td>
<td>–</td>
<td></td>
<td>–</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 19.2 illustrates. It gives results for LD pricing of a discrete geometrically averaged average rate option. In this case an explicit solution is known so the error can be quantified. In curly brackets under the efficiency gain the error in the Monte Carlo estimate is expressed as a multiple of the standard error. This is a measure of the potential bias of the method.

The option with 4 reset dates has a true value of 6.73349 (to 5 decimal places). The LD value with 4 stratification times has a standard error of $\sim 5 \times 10^{-5}$ but the value it gives is incorrect by a factor of about 7 times that standard error. Similarly, the $N = 16$ option with 16 stratification times is in error by a multiple of 25 times the standard error. With fewer stratification times the error is a reasonable multiple of the standard error; the error is acceptable until the option payoff is fully stratified. At that stage the determinism of LD sampling, by artificially lowering the standard error, causes very large apparent errors to appear.

One concludes that either a separate error analysis is required before LD sampled option values can be trusted, or else, in the absence of such an analysis, one should simply avoid computing option values by LD sampling at every reset date.

**LD sampling and extreme options**

OTM calls, and other extreme options, suffer from the problem that only a small number of sample paths are in the money. These rare events behave like a Poisson process. As a consequence, standard errors are very high. Can LD sampling help? Table 19.3 presents some results for the reference OTM average rate option. There is a huge amount of noise in the results. The efficiency gains are much lower than those found with the ATM option; even with full LD sampling the efficiency gains are lower, and the ratio of reported standard error to option value is still several percent.

<table>
<thead>
<tr>
<th>Method</th>
<th>$N = 4$</th>
<th></th>
<th>$N = 16$</th>
<th></th>
<th>$N = 64$</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Expl. value: 6.73349</td>
<td>Expl. value: 5.841672</td>
<td>Expl. value: 5.620434</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Results</td>
<td>$E/\text{error}$</td>
<td>Results</td>
<td>$E/\text{error}$</td>
<td>Results</td>
<td>$E/\text{error}$</td>
</tr>
<tr>
<td>Plain</td>
<td>6.74</td>
<td>(0.030) [1.27]</td>
<td>5.88</td>
<td>(0.026) [3.30]</td>
<td>5.63</td>
<td>(0.027) [16.6]</td>
</tr>
<tr>
<td>LD, 2 levels</td>
<td>6.727</td>
<td>(0.0056) [2.27]</td>
<td>5.833</td>
<td>(0.0065) [5.49]</td>
<td>5.620</td>
<td>(0.0068) [17.1]</td>
</tr>
<tr>
<td></td>
<td>6.73379</td>
<td>(4.5 \times 10^{-5}) [2.22]</td>
<td>5.849</td>
<td>(0.0033) [5.50]</td>
<td>5.622</td>
<td>(0.0035) [17.1]</td>
</tr>
<tr>
<td>LD, 8 levels</td>
<td>5.839</td>
<td>(0.0016) [5.45]</td>
<td>5.619</td>
<td>(0.0016) [17.0]</td>
<td>5.619</td>
<td>(0.0016) [17.0]</td>
</tr>
<tr>
<td>LD, 16 levels</td>
<td>5.84233</td>
<td>(3.1 \times 10^{-5}) [5.25]</td>
<td>5.6212</td>
<td>(0.00071) [17.4]</td>
<td>5.6212</td>
<td>(0.00071) [17.4]</td>
</tr>
<tr>
<td>LD, 32 levels</td>
<td>5.6203</td>
<td>(0.00041) [17.1]</td>
<td>5.6203</td>
<td>(0.00041) [17.1]</td>
<td>5.6203</td>
<td>(0.00041) [17.1]</td>
</tr>
</tbody>
</table>
Table 19.3  Low-discrepancy sampling: speed-ups with an OTM average rate option, $X = 150$

<table>
<thead>
<tr>
<th>Method</th>
<th>$N = 4$</th>
<th>$N = 16$</th>
<th>$N = 64$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$E$</td>
<td>$E$</td>
<td>$E$</td>
</tr>
<tr>
<td>LD, plain</td>
<td>0.021</td>
<td>(0.0017)</td>
<td>0.0049</td>
</tr>
<tr>
<td></td>
<td>[1.63]</td>
<td></td>
<td>[4.69]</td>
</tr>
<tr>
<td>LD, 2 levels</td>
<td>0.0190</td>
<td>9.0</td>
<td>0.0049</td>
</tr>
<tr>
<td></td>
<td>(0.00053)</td>
<td></td>
<td>(0.00038)</td>
</tr>
<tr>
<td></td>
<td>[1.89]</td>
<td></td>
<td>[5.03]</td>
</tr>
<tr>
<td>LD, 4 levels</td>
<td>0.01907</td>
<td>1100</td>
<td>0.0050</td>
</tr>
<tr>
<td></td>
<td>$(4.8 \times 10^{-5})$</td>
<td></td>
<td>(0.00019)</td>
</tr>
<tr>
<td></td>
<td>[1.85]</td>
<td></td>
<td>[5.04]</td>
</tr>
<tr>
<td>LD, 8 levels</td>
<td>–</td>
<td>62.1</td>
<td>0.00519</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$(9.4 \times 10^{-5})$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>[5.02]</td>
</tr>
<tr>
<td>LD, 16 levels</td>
<td>–</td>
<td>562</td>
<td>0.00493</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$(3.2 \times 10^{-5})$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>[4.83]</td>
</tr>
<tr>
<td>LD, 32 levels</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

19.4  SUMMARY

LD sampling lets you extend stratification from one or two times up to (with decent sequences) as many times as you like. A little stratification is a good thing; more is better, but stop before you stratify at every step, saturating the option payoff, and unquestioningly accept the result.

19.5  EXERCISES

1. Adapt the spreadsheet MC_AverageRate_LD.xls to modify the stratification procedure.
   (a) Adapt it so that it always fully stratifies at not one but two times – the final time and the mid-time – with further stratification by LD sampling.
   (b) Now adapt it so that there is no fully stratified time: all sampling is LD.

Do either of these adaptations improve the behaviour of the LD-sampled estimates of option values?

2. Add in objects to enable the LD application to price (discretely reset single) barrier options: knock-in and knock-out, up and down, calls and puts.
   (a) What order of efficiency gains are possible for these options?
   (b) How do the relative values of the strike, $X$, the barrier level, $B$, and the initial asset value, $S_0$, affect the efficiency gain?
   (c) Does the importance of a stratification times depend on the relative values of $X$, $B$ or $S_0$?

3. Recall the Heston stochastic volatility model (exercise 2(a), page 277). How could an LD-stratified sample be used to evolve effectively the two state variables in the Heston model?
Monte Carlo, and other numerical methods, are used when no explicit solution is available for the value of an option. Quite often, however, there may be some value, similar to or related to that of the unknown option value, that can be computed explicitly. For instance in some models the values of vanilla European calls, or other options with piece-wise linear payoffs, may be explicitly computable but values for options with more complex but superficially similar payoffs may not be.

A control variate (CV) method computes simultaneously both the unknown option value and the value of one or more quantities whose values are known explicitly. Errors in the known quantities are then used to correct values computed for the unknown option value.

The classical example of a control variate in option pricing, one that we explore in this chapter, is to use the explicitly known value of a discretely reset geometrically averaged average rate option as a control variate for a discretely reset arithmetically averaged average rate option. When the underlying asset follows a geometric Brownian motion there is an explicit solution for the former option. Its value is very highly correlated to that of the latter option and it serves as a highly effective control variate.

Strictly it is not necessary that the known value be computable analytically. It is enough that it is cheaply available through some other pricing mechanism. European option values are available in the Heston model only through a numerical integration procedure. Nevertheless if these can be computed quickly enough then their prices could form a basis for control variates.

The use of control variates can lead to very great speed-ups indeed. The more closely correlated the known value is to the unknown value, the greater the available efficiency gain. The difficulty is that an appropriate control variate may not be available.

This chapter is the first of a pair about control variates. Here we are mainly concerned with theory; in Chapter 21 we present implementations and numerical results.

### 20.1 CONTROL VARIATES

When they can be used, control variates are a very effective technique of variance reduction. Suppose that one wishes to compute an option value \( c \). The idea is to find a variable \( d^j \) with mean zero generated along with \( c^j \) on the \( j \)-th sample path, \( j = 1, \ldots, M \). \( d \) is the control variate. The greater the correlation of \( d \) to \( c \) the greater the ultimate efficiency gain.

The result of the simulation is a set of pairs \((c^j, d^j)\), \( j = 1, \ldots, M \). Set \( W^j = c^j - \beta d^j \), where \( \beta = \text{cov}(c, d)/\text{var}(d) \). Then \( W = (1/M) \sum_{j=1}^{M} W^j \) is an estimate for the option value, since \( \mathbb{E}[W] = \mathbb{E}[c] \), but \( \text{var}(W) \leq \text{var}(c) \). Indeed,

\[
\text{var}(W) = \text{var}(c - \beta d) = \text{var}(c) + \beta^2 \text{var}(d) - 2\beta \text{cov}(c, d) ,
\]

and \( \text{var}(W) \) is minimized precisely when \( \beta = \text{cov}(c, d)/\text{var}(d) \). With this value of \( \beta \)

\[
\text{var}(W) = \text{var}(c) \left(1 - \rho_{c,d}^2 \right) \leq \text{var}(c)
\]

(20.2)

where \( \rho_{c,d} \) is the correlation between \( c \) and \( d \). The closer \( |\rho_{c,d}| \) is to 1 the greater the potential speed-up.
Operationalizing a control variate is easy. Given the pairs \( \{(c^j, d^j)\}_{j=1,...,M} \) regress \( c^j \) against \( d^j \) to obtain
\[
c^j = \alpha + \beta d^j + \varepsilon_j
\]
where \( \beta = \text{cov}(c, d)/\text{var}(d) \) and \( \mathbb{E}[\varepsilon_j] = 0 \). This means that the value of \( \alpha = \mathbb{E}[c - \beta d] \) in the regression is precisely the control variate estimate of the option value.

Note that if more than one control variate is available, then the additional controls can just be added in to the regression. Suppose that \( d \) and \( e \) are control variates so that at each iteration a vector \( (c^j, d^j, e^j) \) of three values is obtained. Then the control variate estimate is the intercept \( \alpha \) in the regression
\[
c^j = \alpha + \beta_d d^j + \beta_e e^j + \varepsilon_j.
\]

The extent of the speed-up

Whether a control variate \( d \) yields a significant efficiency gain depends not only on the correlation but also on the amount of additional work required to generate it. The extra work may not be trivial. If the additional computing time over the case with no control variate is a factor of \( k \), then there is a speed-up if \( \text{var}(W) < (1/k)\text{var}(c) \), that is, if \( \rho_{c,d}^2 > (k - 1)/k \). For various values of \( k \), Table 20.1 shows the minimum (positive) value, \( \rho_{\text{min}} \), of \( \rho_{c,d} \) required before the effort involved is worthwhile. If no extra work is required \( (k = 1) \) then any value \( \rho > 0 \) gives a speed-up. If 5 times the work is needed then only when \( \rho > 0.89 \) is a speed-up achieved.

It is worth a bit of work to obtain a good control variate but there are diminishing returns as Table 20.2 illustrates. The speed-up factor is \( E = 1/[k(1 - \rho^2)] \). A speed-up of less than 1 indicates a slow-down.

Even when there is no additional work \( (k = 1) \), and with a correlation of 0.95, the speed-up is only just over 10. However as the correlation approaches 1 the speed-ups can become very large.

Approaches to obtaining control variates

There are two main sources of control variates. These are illustrated in Figure 20.1. A control variate is found either by valuing two instruments or quantities simultaneously while keeping the model fixed (an auxiliary instrument control variate, Figure 20.1(a)), or by varying the model while keeping the instrument fixed (an auxiliary model control variate, Figure 20.1(b)), or by varying both together. In each case one

<table>
<thead>
<tr>
<th>Table 20.1</th>
<th>Minimum correlations required for various levels of effort</th>
</tr>
</thead>
<tbody>
<tr>
<td>( k )</td>
<td>1</td>
</tr>
<tr>
<td>( \rho_{\text{min}} )</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 20.2</th>
<th>Speed-up ( E ) as a function of effort ( k ) and correlation ( \rho )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( k )</td>
<td>0.5</td>
</tr>
<tr>
<td>( \rho )</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>10</td>
</tr>
</tbody>
</table>
is supposed to know explicitly the value of an instrument, or some other useful computable quantity such as the delta, to use as a control for the unknown option value.

If the model is a perturbation of a tractable auxiliary model – for instance, a stochastic volatility extension of a geometric Brownian motion – then one may proceed in one of two standard ways:

1. Evolve simultaneously the asset value under the pricing model and the auxiliary model, using a known value in the auxiliary model as a control;
2. Use a delta control variate (section 20.2.2) where the delta is computed in the auxiliary model as an approximation to the unknown delta in the pricing model.

In the first method the set of innovations used in the pricing model to drive the asset process are also used for the asset process in the auxiliary model.

An example of combining an auxiliary instrument with an auxiliary model is in the valuation of a discretely reset arithmetic average rate option in the Heston stochastic volatility model. When the instrument is geometrically averaged and the asset process in GBM there is an explicit solution. One may evolve a GBM sample path alongside each Heston sample path and use the geometrically averaged option value as an auxiliary instrument control in the auxiliary model.

There are other examples. In jump-diffusion models some instruments may not have explicit solutions, whereas there may be when the process has no jumps. We are in case (b) in Figure 20.1 and a control variate may exist.

### 20.2 EXAMPLES OF CONTROL VARIATES

There are several related families of standard control variates. These are underlying asset control variates; delta control variates; payoff approximating control variates; and tailored control variates.
20.2.1 Underlying asset control variates

The underlying asset can be used as a control variate. Suppose that one has a set of times \( T = \{t_i\}_{i=0,\ldots,N} \) and the method generates a set of sample paths \( S = \{S^j\}_{j=1,\ldots,M} \). The \( j \)th sample path is \( S^j = (S^j_0, \ldots, S^j_N) \), where \( S^j_i \) is the asset value generated for time \( t_i \), and \( S^j_0 = S_0 \) is the initial asset value at time \( 0 = t_0 \).

For each time \( t_i \) there is available a model estimate \( \hat{S}^j_0 \) for \( S_0 \):

\[
\hat{S}^j_0 = e^{-r t_i} \sum_{j=1}^M S^j_i .
\]

This leads naturally to a set of control variates

\[
CV_{S,i}^j = e^{-r t_i} S^j_i - S_0, \quad i = 1, \ldots, N .
\]

Since \( \mathbb{E}[CV_{S,i}^j] = 0 \), for every \( i \), \( CV_{S,i}^j \) is a control variate.

The effectiveness of \( CV_{S,i}^j \) depends on its correlation with the unknown option payoff. In general this may be low. However it would be unusual for a model not to have available automatically – hence cheaply – the value of the underlying asset at each time step, so \( CV_{S,i}^j \), and its associated speed-ups, comes almost free.

Functions of the set \( \{CV_{S,i}^j\}_{i=1,\ldots,N} \) provide a powerful extension of the basic \( CV_{S,i}^j \) control variates. For instance let \( w = \{w_i\}_{i=1,\ldots,N} \) be a set of weights and set

\[
CV^j_w = \sum_{i=1}^N w_i CV_{S,i}^j ,
\]

then \( CV^j_w \) is a control variate.

A suitably chosen function may have higher correlation with an option value than any individual \( CV_{S,i}^j \). For example, consider the payoff to a discretely reset arithmetic average rate option. Along the path \( S^j \) this is

\[
H^a(S^j) = (a^j - X)^+
\]

where \( a^j = (1/N) \sum_{i=1}^N S^j_i \) (and for simplicity we assume that every time step is a reset date).

\( CV^j_a = a^j - \mathbb{E}[a^j] \) is a control variate whose value is likely to be better correlated to \( H^a \) than that of any individual \( CV_{S,i}^j \). But since \( \mathbb{E}[a^j] = (1/N) \sum_{i=1}^N e^{r t_i} S_0 \) we have \( CV^j_a = CV^j_w \) for \( w_i = e^{r t_i} / N \),

\[
CV^j_a = \sum_{i=1}^N \frac{1}{N} e^{r t_i} CV_{S,i}^j
\]

\[
= \frac{1}{N} \sum_{i=1}^N \left( S^j_i - e^{r t_i} S_0 \right)
\]

\[
= CV^j_a .
\]

The \( CV_a \) control variate is an example of a control variate based on the path statistic that the option pays off against, as is \( CV_{S,i}^j \) for a vanilla European option maturing at time \( t_i \).
An alternative to \( CV_{S,i}^j \) is the multiplicative form \( \overline{CV}_{S,i}^j = e^{-rt_i} S_i^j / S_0 - 1 \) (since \( \mathbb{E}[\overline{CV}_{S,i}^j] = 0 \)). The geometric average \( g^j = \left( \prod_{i=1}^{N} S_i^j \right)^{1/N} \) is a function of the \( \overline{CV}_{S,i}^j \),

\[
g^j = \left( \prod_{i=1}^{N} S_i^j \right)^{1/N} = S_0 \left( \prod_{i=1}^{N} \frac{S_i^j}{S_0} \right)^{1/N} \tag{20.12}
\]

\[
= S_0 \left( \prod_{i=1}^{N} e^{r t_i} \left( \overline{CV}_{S,i}^j \right) + 1 \right)^{1/N} \tag{20.13}
\]

\[
= S_0 \exp \left( r \frac{1}{N} \sum_{i=1}^{N} t_i \right) \left( \prod_{i=1}^{N} \left( \overline{CV}_{S,i}^j \right) + 1 \right)^{1/N} \tag{20.14}
\]

\[
= S_0 \exp \left( r \frac{1}{N} \sum_{i=1}^{N} t_i \right) \left( \prod_{i=1}^{N} \left( \overline{CV}_{S,i}^j \right) + 1 \right)^{1/N} \tag{20.15}
\]

### 20.2.2 Delta control variates

The weights vector \( w \) used in the control variate \( CV_w \) does not have to be a predetermined constant; \( w_i \) can be any quantity computable at time \( t_i \). An example of this is given by the delta control variate.

A delta hedge control variate simulates the value of a discretely rehedged hedging portfolio. Since it is a hedge

1. its expected value is an unbiased estimate of the option value, and
2. along each sample path its value is (should be) highly correlated with the simulated option value along that path.

Suppose that an option value \( c_t \) is contingent upon an underlying state variable \( S_t \), and consider a hedge portfolio formed by borrowing and taking a position in \( S_t \). For simplicity, suppose that \( t_{i+1} - t_i = \Delta t \) is constant. Write \( \delta_t = \partial c_t / \partial S_t \) for the option delta with respect to \( S_t \). Let the hedge ratio at time \( t_i \) on the \( j \)th sample path be \( \delta^j_i \equiv \delta_t (S_t^j) \) and suppose that the interest rate \( r \) is constant.

The hedge is put on at time \( t_0 \) with cost \( \delta^j_0 S_0^j \). At times \( t_i, i = 1, \ldots, N - 1 \), it is rehedged with net cost \( (\delta^j_i - \delta^j_{i-1}) S_i^j \). Finally at time \( t_N \) the hedge is taken off yielding \( \delta^j_{N-1} S_N^j \). The total discounted cost \( h^j \) is

\[
h^j = \delta^j_0 S_0^j + \sum_{i=1}^{N-1} e^{-r t_i} \left( \delta^j_i - \delta^j_{i-1} \right) S_i^j - e^{-r t_N} \delta^j_{N-1} S_N^j \tag{20.16}
\]

\[
= \sum_{i=0}^{N-1} e^{-r t_i} \delta^j_i \left( S_i^j - e^{-r \Delta t} S_{i+1}^j \right) \tag{20.17}
\]

\[
= \sum_{i=0}^{N-1} e^{-r t_i + \Delta t} \delta^j_i \left( e^{r \Delta t} S_i^j - S_{i+1}^j \right) \tag{20.18}
\]
Since \( h^j \) is the present value of the net cash inflow to a hedge portfolio we anticipate that \( \mathbb{E} [h^j] = 0 \), and indeed

\[
\mathbb{E} \left[ e^{r \Delta t} S_i^j - S_{i+1}^j \right] = e^{r t_{i+1}} S_0^j - e^{r t_{i+1}} S_0^j = 0
\]  

(20.19)

for every \( i \), so \( h^j \) is a control variate. In fact, since \( h^j \) is strongly related to the value of a hedge portfolio we expect it to be, potentially, a very good control variate.

We discuss later in this chapter using \( h^j \) as a control variate, but for the moment we give two other representations of \( h^j \).

The first relates \( h^j \) to the \( CV^j_{S,i} \) control variates. We have

\[
h^j = \sum_{i=0}^{N-1} e^{-r t_{i+1}} \delta_i^j \left( e^{r \Delta t} S_i^j - S_{i+1}^j \right) = \sum_{i=0}^{N-1} e^{-r t_{i+1}} \delta_i^j \left( e^{r t_{i+1}} S_0^j - S_{i+1}^j \right) + \sum_{i=0}^{N-1} e^{-r t_{i+1}} \delta_i^j \left( e^{r \Delta t} S_i^j - e^{r t_{i+1}} S_0^j \right)
\]

(20.20)

\[
= \sum_{i=0}^{N-1} \delta_i^j \left( CV^j_{S,i} - CV^j_{S,i+1} \right)
\]

(20.21)

\[
= \sum_{i=1}^{N-1} \left( \delta_i^j - \delta_{i-1}^j \right) CV^j_{S,i} - CV^j_{S,N-1} CV^j_{S,N}
\]

(20.22)

since \( CV^j_{S,0} \equiv 0 \). Set \( w_i = \delta_i^j - \delta_{i-1}^j, i = 1, \ldots, N-1 \), and \( w_N = -\delta_{N-1}^j \), then \( CV^j_w \) for \( w = \{w_i\}_{i=1,\ldots,N} \) is an underlying asset control variate with adapted weights.

The second representation is convenient when the state variable is not a traded asset. Note that \( e^{r \Delta t} S_i^j = \mathbb{E} \left[ S_{i+1}^j \mid S_i^j \right] \), so from equation (20.18)

\[
h^j = \sum_{i=0}^{N-1} e^{-r t_{i+1}} \delta_i^j \left( e^{r \Delta t} S_i^j - S_{i+1}^j \right) = \sum_{i=0}^{N-1} e^{-r t_{i+1}} \delta_i^j \left( e^{r \Delta t} S_i^j - S_i^j + S_i^j - S_{i+1}^j \right)
\]

(20.23)

\[
= \sum_{i=0}^{N-1} e^{-r t_{i+1}} \delta_i^j \left( \mathbb{E} \left[ \Delta S_i^j \mid S_i^j \right] - \Delta S_i^j \right)
\]

(20.24)

where \( \Delta S_i^j = S_{i+1}^j - S_i^j \).

Equation (20.26) can be applied to any state variable, even if it is not traded. \( \delta_i^j = \partial c_t / \partial S_t \) is just the relevant Greek rather than a hedge ratio per se.

To implement a delta control variate it is a requirement that it should be possible to compute \( \delta_i^j \), but this may seem problematical; if \( c_t \) is unknown how can \( \delta_t \) be known? Fortunately it is possible to use approximations for \( \delta_t \) arising, for instance, from an auxiliary model. See section 20.3.
20.2.3 Payoff approximating control variates

Consider the exotic European option $c^e$ whose payoff $H^e$ at time $t_N$ is

$$H^e(S_N) = (X_1 - (S_N - X_2)^2)^+.$$  \hfill (20.27)

This is a quadratic payoff with maximum value $X_1$ when $S_N = X_2$ and value 0 when $S_N \leq X_2 - \sqrt{X_1}$ or $S_N \geq X_2 + \sqrt{X_1}$. When $S$ follows a geometric Brownian motion this option has an explicitly computable value (see exercise 1); however, what this is may not be immediately apparent. In practice, if a value is needed rapidly one might go ahead and compute the option value by Monte Carlo and wait for the explicit solution to arrive in its own good time.

The payoff (20.27) is quadratic but it can be approximated by a piece-wise linear function. Let $a < b \leq X_2$ and set $a' = 2X_2 - a$, $b' = 2X_2 - b$, then the function $H_{a,b}$,

$$H_{a,b}(S) = \begin{cases} 
0, & S \leq a, \\
\frac{S - a}{b - a}X_1, & a < S \leq b, \\
X_1, & b < S \leq b', \\
\frac{a' - S}{a' - b'}X_1, & b' < S \leq a', \\
0, & a' < S, 
\end{cases}$$  \hfill (20.28)

is an approximation to $H^e$ for suitably chosen values of $a$ and $b$. Figure 20.2 plots the payoff $H^e$ to the exotic option when $(X_1, X_2) = (100, 110)$ and a matching payoff $H_{a,b}$ for the piece-wise linear option with $(a, b) = (99, 107)$. The tent-like payoff of the piece-wise linear option matches fairly closely the payoff of the exotic.

![Figure 20.2 Matching the payoff of the exotic option](image-url)
The option with payoff function \( H_{a,b} \) may be valued explicitly in a Black–Scholes world. Its value \( c_{a,b} \) is

\[
c_{a,b} = \frac{X}{b - a} \left( c(a) - c(b) - c(b') + c(a') \right)
\]  

(20.29)

where, in equation (20.29), \( c(X) \) is the Black–Scholes value of a European call with strike \( X \).

It is clear that, for well-chosen values of \( a \) and \( b \), \( c_{a,b} \) may be a good approximation to \( c^e \). Hence

\[
d^j_{a,b} = e^{-rT} H_{a,b}(S^j_N) - c_{a,b}
\]  

(20.30)

is a control variate for \( c^e \); depending on \( a \) and \( b \) it may even be a good control variate. Clearly a better piece-wise linear approximation leads to a better control variate.

\( d_{a,b} \) is a control variate that arises by approximating an unknown option with a known option devised by approximating the payoff function. A numerical assessment of its effectiveness is given in Chapter 21.

Equation (20.29) also provides us with a delta control variate. Just set

\[
h^j_{a,b} = \sum_{i=0}^{N-1} e^{-rT_i} \delta^j_i \left( \mathbb{E}\left[ \Delta S^j_i \mid S^j_i \right] - \Delta S^j_i \right)
\]  

(20.31)

where

\[
\delta^j_i = \frac{\partial c_{a,b}}{\partial S_i} = \frac{X}{b - a} \left( \frac{\partial c(a)}{\partial S_i} - \frac{\partial c(b)}{\partial S_i} - \frac{\partial c(b')}{\partial S_i} + \frac{\partial c(a')}{\partial S_i} \right).
\]  

(20.32)

\( h_{a,b} \) is likely to be a much less effective CV than \( d_{a,b} \). To be an effective hedge, a hedge portfolio needs to be reset quite frequently. This means that \( h_{a,b} \) is not likely to be an effective CV unless \( N \) is relatively large. By contrast \( d_{a,b} \) depends only on the value of the stock at the final time and its computation does not require a large number of steps. Because of the additional time steps \( h_{a,b} \) is likely to be much more expensive than \( d_{a,b} \) to compute and, in any case, its correlation with \( c^e \) is likely to be much lower than that of \( d_{a,b} \).

### 20.2.4 Tailored control variates and average rate options

Particular options may have available control variates tailored specifically for them. The exotic option with payoff (20.27) and tailored CV \( d_{a,b} \) is one case in point. Other non-generic, highly specific control variates that are targeted at specific instruments may exist in particular models. The classic example is to use a geometric average rate option as a control for an arithmetic average rate option. This is an example of control variates performing at their best.

Pricing geometric average rate options in the Black–Scholes framework is reviewed in Appendix B. Write \( a^j \) and \( g^j \) for the values along the \( j \)th sample path of the arithmetic and geometric averages,

\[
a^j = \frac{1}{N} \sum_{i=1}^N S^j_i,
\]  

(20.33)

\[
g^j = \left( \prod_{i=1}^N S^j_i \right)^{1/N},
\]  

(20.34)

respectively.
It is clear that $a^j$ and $g^j$ are likely to be very highly correlated, but not only is $E[g^j]$ known explicitly, so is the option value $v_t = e^{-rtN}E[(g^j - X)^+]$. This makes available a control variate

$$CV^j_{GA} = e^{-rtN} (g^j - X)^+ - v_t,$$

(20.35)

which is very highly correlated with $(a^j - X)^+$. The delta CV, $CV^j_{GA,\delta}$ derived from $CV^j_{GA}$, is

$$CV^j_{GA,\delta} = \sum_{i=0}^{N-1} e^{-rt_{i+1}} \delta^G,j_i (e^{r \Delta t} S^j_i - S^j_{i+1})$$

(20.36)

where $\delta^G,j_i$ is the delta at time $t_i$ of the part-way discretely reset geometric average rate option in the GBM model. This delta is known explicitly from equation (B.27) in Appendix B.

In Chapter 21 we explore the efficiency gains available with these CVs.

## 20.3 AUXILIARY MODEL CONTROL VARIATES

We give two examples of auxiliary model control variates. The first is the valuation of bond options in the Fong and Vasicek (1991) term structure model. The second is option pricing in the Heston model (Heston, 1993). In each case we investigate (i) evolving the auxiliary model and (ii) delta controls obtained from the auxiliary model.

### 20.3.1 Bond options in the Fong and Vasicek model

This example is described by Clewlow and Strickland (1997). Further material on interest rate modelling can be found in James and Webber (2000) and Brigo and Mercurio (2001).

The Fong and Vasicek model is a stochastic volatility version of the Vasicek interest rate model. The Vasicek model has a single state variable, the short rate $r_t$, with process

$$dr_t = \alpha(\mu - r_t) \, dt + \sigma \, dz_t.$$

(20.37)

This has constant volatility $\sigma$.

In the Fong and Vasicek model $\sigma$ is stochastic. The defining SDEs are

$$dr_t = \alpha(\mu - r_t) \, dt + \sqrt{v_t} \, dz'_t,$$

(20.38)

$$dv_t = \beta(\kappa - v_t) \, dt + \sigma \sqrt{v_t} \, dz''_t,$$

(20.39)

$$dz'_t \, dz''_t = \rho \, dt,$$

(20.40)

where $r_t$ is the short rate and $v_t$ is its squared volatility.

Let $B_t(T)$ be the value at time $t$ of a pure discount bond maturing at time $T > t$ with value 1. Under the accumulator account numeraire

$$B_t(T) = E\left[ \exp\left( - \int_t^T r_s \, ds \right) \right]$$

(20.41)
where \( r_t \) is the short rate. Write \( r_t(T) \) for the spot rate at time \( t \) on a PDB maturing at time \( T \) so that \( B_t(T) = \exp(-r_t(T)(T - t)) \). A European bond call option with strike \( X \) and maturity time \( T \) on an underlying pure discount bond with maturity \( T_1 > T \) has payoff

\[
H_T = (B_T(T_1) - X)^+
\]  

(20.42)
at time \( T \).

Suppose that you want to value a bond option in the Fong and Vasicek model. In this model there are explicit solutions for pure discount bond prices, so Fong and Vasicek PDB prices can be used as controls. We do not pursue this here; instead we describe how the Vasicek model can be used as an auxiliary model to value a bond option.

Note that since bond prices can be computed in the Fong and Vasicek model it is necessary to evolve the state variables only up to time \( T \), not all the way to time \( T_1 \). This is important. Not only is there a direct saving because of the shorter sample paths but, much more importantly, it means that it is not necessary to use an American Monte Carlo method (see Part VIII) to be able to make optimal exercise decisions at time \( T \).

**Evolving an auxiliary model**

In the Vasicek model there are explicitly computable values for bond option prices (Jamshidian (1989)). Suppose \( t < T < T_1 \) and let \( c_t^V(T, T_1 | X) \) be the value in the Vasicek model at time \( t \) of a bond call option, with maturity time \( T \) and strike \( X \), on an underlying PDB maturing at time \( T_1 \). The payoff \( H \) to this option at time \( T \) is

\[
H = (B_T^V(T_1) - X)^+
\]  

(20.43)
and its value at time \( t \) in the Vasicek model is

\[
c_t^V(T, T_1 | X) = B_t^V(T_1)N(d_1) - B_t^V(T)XN(d_2),
\]  

(20.44)
where

\[
d_1 = \frac{1}{\sigma^B} \ln \left( \frac{B_t^V(T_1)}{B_t^V(T)X} \right) + \frac{1}{2} \sigma^B,
\]  

(20.45)
\[
d_2 = d_1 - \sigma^B,
\]  

(20.46)
\[
\sigma^R = \sigma \left( 1 - e^{-2\alpha(T_1 - T)} \right) \sqrt{1 - e^{-2\alpha(T - t)}}
\]  

(20.47)
and \( B_t^V(T) \) are bond prices in the Vasicek model. Write \( r_t^V \) for Vasicek short rate (to distinguish it from the Fong and Vasicek short rate). Write \( r_t^V(T) \) for the Vasicek spot rates then

\[
B_t^V(T) \equiv B_t^V(T | r_t^V) = \exp \left( -r_t^V(T)(T - t) \right)
\]  

(20.48)
where

\[
r_t^V(T) = r^\infty + (r_t^V - r^\infty) \frac{1 - e^{-\alpha(T - t)}}{\alpha (T - t)} + \frac{\sigma^2 (T - t)^2}{4\alpha} \left( \frac{1 - e^{-\alpha(T - t)}}{\alpha (T - t)} \right)^2
\]  

(20.49)
for \( r^\infty = \mu - \sigma^2/2\alpha^2 \).
Since the Fong and Vasicek model is close to the simpler, more tractable, Vasicek model, and can be regarded as a perturbation of it, bond option values in the Vasicek model can be used as a control.

To generate a pair of sample paths for \( r_t \) and \( v_t \), a pair of Wiener sample paths, \( w^r = \{w^r_i\}_{i=0,\ldots,N} \) and \( w^v = \{w^v_i\}_{i=0,\ldots,N} \), are needed. The path \( w^r \) is used to generate a sample path for \( r^V_t \) under the Vasicek process, equation (20.37).

The distribution of \( r^V_t \mid r^V_0 \sim N[\mu(r^V_0, t), \sigma^2(r^V_0, t)] \) is normal with mean and variance

\[
\mu(r^V_0, t) = \mu(1 - e^{-\alpha t}) + r^V_0 e^{-\alpha t}, \\
\sigma^2(r^V_0, t) = \frac{1 - e^{-2\alpha t}}{2\alpha} \sigma^2,
\]

so the sequence \( \{r^V_i\}_{i=0,\ldots,N} \) can be generated as

\[
r^V_{i+1} = \mu(r^V_i, \Delta t) + \sigma \sqrt{\frac{1 - e^{-2\alpha \Delta t}}{2\alpha \Delta t}} (w^r_{i+1} - w^r_i)
\]

starting from \( r^V_0 = r_0 \), the current value of the short rate.

The Vasicek bond option payoff can be computed from \( r^V_N \) along each sample path, yielding a CV. Set

\[
d = \hat{B}_t(B^V_T(T_1 \mid r^V_N) - X)^+ - c^V_t(T, T_1 \mid X)
\]

where

\[
\hat{B}_t \equiv \hat{B}_t(\{r^V_i\}_{i=0,\ldots,N}) = \exp\left(-\Delta t \sum_{i=0}^{N-1} r^V_i\right)
\]

is an approximation to the discount factor along the generated sample path.

Because Fong and Vasicek is so close to Vasicek, \( d \) is likely to be a very effective auxiliary model CV.

**The delta control variate**

Another way of using the Vasicek model is to obtain a delta control variate. The control variate is

\[
h^i = \sum_{i=1}^{N-1} \delta_i^j \left( \Delta r^j_i - \mathbb{E}[\Delta r^j_i] \right)
\]

where \( \Delta r^j_i = r^j_{i+1} - r^j_i \) and \( \delta_i^j \) is the Fong and Vasicek bond option delta. The term \( \Delta r^j_i - \mathbb{E}[\Delta r^j_i] \) is automatically available to the Monte Carlo method and an approximation to \( \delta_i^j \) can be derived from the Vasicek model.

The exact form of \( \Delta r^j_i - \mathbb{E}[\Delta r^j_i] \) depends on the discretization method used to evolve \((r, v_t)\). For example, suppose that a modified Euler discretization has been used for \( v_t \) and a moment freezing method for \( r_t \). \( v_t \) has the discrete approximation

\[
u_{t+\Delta t} = v_t + \alpha (\mu - v_t) \Delta t + \eta \sqrt{v_t} (w^v_{t+\Delta t} - w^v_t),
\]

\[
u_{t+\Delta t} = \max(\nu_{t+\Delta t}, 0)
\]

It cannot be allowed to become negative or the square root blows up.
The Euler discretization produces biased results unless the time step is quite small. Since here we are mainly concerned with speed-ups, and these are largely unaffected by bias, we defer a discussion of superior methods until Chapter 26 where simulating the CIR process (20.39) is discussed at greater length. Meanwhile we note that option prices computed by this discretization are biased, particularly so for small $N$.

To evolve $r_t$, suppose that $v_t$ is constant (frozen) over each (small) time step. Then $r_t$ is approximated by

$$r_{t+\Delta t} = \mu (1 - e^{-\alpha \Delta t}) + r_t e^{-\alpha \Delta t} + \sqrt{v_t} \sqrt{1 - e^{-2\alpha \Delta t}} \left( w''_{t+\Delta t} - w''_t \right). \quad (20.58)$$

In this case

$$\Delta r^j_i - \mathbb{E}[\Delta r^j_i] \sim \sqrt{v^j_i} \sqrt{1 - e^{-2\alpha \Delta t}} \left( w''_{t+1} - w''_t \right). \quad (20.59)$$

If $r_t$ is evolved with an Euler discretization the expression is even simpler:

$$\Delta r^j_i - \mathbb{E}[\Delta r^j_i] \sim \left( w''_{t+1} - w''_t \right) \sqrt{v^j_i}. \quad (20.60)$$

In either case $\Delta r^j_i - \mathbb{E}[\Delta r^j_i]$ is a simple function of already computed random increments and values.

To obtain a value for $\delta^j_i$ there is an excellent ready-made approximation. Since Fong and Vasicek is a stochastic volatility variation of Vasicek, the Vasicek bond option delta, derived from equation (20.44), can used as a close approximation to $\delta^j_i$. Hence, expression (20.55) is numerically tractable, by approximation, and serves as a suitable control variate in this case. Clewlow and Strickland state that speed-ups of $\sim50$ were obtained with this CV.

### 20.3.2 Option pricing in the Heston model

The Heston model is GBM with stochastic volatility. The state variables are the asset price $S_t$ and its squared volatility $v_t$ obeying the SDEs

$$dS_t = r_t S_t \, dt + \sqrt{v_t} S_t \, dz^S_t, \quad (20.61)$$
$$dv_t = \alpha (\mu - v_t) \, dt + \eta \sqrt{v_t} \, dz^v_t, \quad (20.62)$$
$$dz^S_t dz^v_t = \rho \, dt. \quad (20.63)$$

It is a popular and attractive model. Explicit solutions are available for the value of European style instruments, such as vanilla calls and puts, although these involve a numerical integration. For instance, the value $c_t$ of a European call option with strike $X$ and time of maturity $T$ at time $t < T$ is

$$c_t = S_t F_1 - e^{-r(T-t)} X F_2 \quad (20.64)$$

where

$$F_1 = \int_X^{\infty} S f^H_T (S) \, dS, \quad (20.65)$$
$$F_2 = \int_X^{\infty} f^H_T (S) \, dS. \quad (20.66)$$
and $f_H^T(S) = f_H^T(S | S_t)$ is the (Heston) probability density of $S_T | S_t$. It is not hard to show (see, for instance, Gatheral (2006)) that in the Heston model $F_1 = F(1, \alpha - \rho \eta)$ and $F_2 = F(-1, \alpha)$ where

$$F(s, b) = \frac{1}{2} + \frac{1}{\pi} \int_0^\infty \Re \left[ \frac{1}{iu} \exp \left( A \mu + B v_t + i u \ln \left( \frac{S_t}{X e^{-r(T-t)}} \right) \right) \right] \, du,$$  \hspace{1cm} (20.67)

$$A = \frac{\alpha}{\eta^2} \left( (\beta - d) (T - t) - 2 \ln \left( \frac{1 - g e^{-d(T-t)}}{1 - g} \right) \right),$$  \hspace{1cm} (20.68)

$$B = \frac{\beta - d}{\eta^2} \left( \frac{1 - e^{-d(T-t)}}{1 - g e^{-d(T-t)}} \right),$$  \hspace{1cm} (20.69)

$$g = \frac{\beta - d}{\beta + d},$$  \hspace{1cm} (20.70)

$$\beta = b - i \rho \eta u,$$  \hspace{1cm} (20.71)

$$d^2 = \beta^2 - \eta^2 (isu - u^2).$$  \hspace{1cm} (20.72)

The integration in equation (20.67) can be performed reasonably efficiently. It is convenient to use a complex number class to compute values for the integrand although the integration itself is entirely real. A complex number class is described in Appendix C as is a set of quadrature classes. Its accuracy seems good to at least 3 or 4 decimal places at least in benchmark examples,\(^1\) but is less accurate in other cases.

The Heston model is a perturbation of GBM. Explicit formulae for option values are available for simpler options in the GBM model, with both constant and deterministic volatility. One expects that GBM model values will provide good control variates – either as an auxiliary model CV or to provide delta controls – for the Heston model.

As an application suppose that the value of a discretely reset arithmetic average rate option is required in the Heston model. The explicit value of a discretely reset geometric average rate option on an asset with GBM enables us to obtain a good control variate.

**Evolving the Heston model**

The literature on evolving the Heston model is extensive. It is difficult to construct discrete time sample paths in the Heston model both accurately and speedily; we discuss some of the issues in Chapter 27. A fast method is likely to be inaccurate in the sense that the distribution of the stock price generated for the final time will not exactly match its theoretical distribution. For instance it may be biased, with its expected mean value not equaling the theoretical mean. This has serious consequences for option pricing.

For the moment we adopt a very simple expedient. The volatility process is evolved with a floored Euler approximation, using equations (20.56) and (20.57), and for $S_t$ a moment freezing method is used. If one assumes that the value of $v_t$ is constant over the succeeding time step we can set

$$S_{t+\Delta t} = S_t \exp \left( \left( r - \frac{1}{2} v_t \right) \Delta t + \sqrt{v_t} \sqrt{\Delta t} \epsilon_t^S \right)$$  \hspace{1cm} (20.73)

\(^1\)This formulation avoids “the little Heston trap” (Albrecher et al. (2006)). The spreadsheet HestonExplicit.xls provides an implementation of equation (20.67).
for a normal variate $\epsilon_i^S$. Again, like the Fong and Vasicek example, this method produces biased sample paths unless $\Delta t$ is small.

**Evolving an auxiliary model**

To evolve the Heston model two (correlated) Wiener process paths are required, $w^S = \{w^S_i\}_{i=0,\ldots,N}$ for the stock and $w^v = \{w^v_i\}_{i=0,\ldots,N}$ for the variance, so that in equation (20.73) one sets

$$\sqrt{\Delta t} \epsilon^S_i = w^S_{i+1} - w^S_i. \quad (20.74)$$

Write $S^H = \{S^H_i\}_{i=0,\ldots,N}$ for a sample path of the asset value in the Heston model. The pair of increments $(w^S_{i+1} - w^S_i, w^v_{i+1} - w^v_i)$ evolves $(S^H_i, v_i)$ to $(S^H_{i+1}, v_{i+1})$.

Using $w^S$, a second sample path is evolved for the asset under plain GBM. Write $S^G = \{S^G_i\}_{i=0,\ldots,N}$, with $S^G_0 = S^H_0$, for a sample path of the asset in the GBM model. $w^S_{i+1} - w^S_i$ evolves $S^G_i$ to $S^G_{i+1}$ under GBM, with volatility constant at its initial value,

$$S^G_{i+1} = S^G_i \exp \left( \left( r - \frac{1}{2} v_0 \right) \Delta t + \sqrt{v_0} \left( w^S_{i+1} - w^S_i \right) \right). \quad (20.75)$$

The control variate is $CV_{GA}(S^G) = e^{-rT_N} (g(S^G) - X)^+ - c_i^G$, where $g(S^G)$ is the geometric average along the path $S^G$ and $c_i^G$ is the analytic value of the discretely reset geometric average rate option under GBM with stock value $S^G_0$ and (constant) volatility $\sqrt{v_0}$.

One anticipates a high correlation between $CV_{GA}$ and the discounted payoff $p(S^H) = e^{-rT_N} (g(S^H) - X)^+$ computed along $S^G$.

**The delta control variate in the Heston model**

The delta control variate $h$ is in the form given by equation (20.18),

$$h = \sum_{i=0}^{N-1} e^{-rT_i} \delta^G_i \left( e^{r\Delta t} S^H_i - S^H_{i+1} \right), \quad (20.76)$$

where $\delta^G_i$ is the delta at time $t_i$ of the part-way discretely reset geometric average rate option in the GBM model,

$$\delta^G_i = \frac{\partial v_i}{\partial S_i} \left( X, t_N, T, S_i, r, \sqrt{v_i} \mid G_i, i \right) \quad (20.77)$$

(where $v_i$ is given by equation (B.27), page 526). This CV is expensive to compute and in any case it seems likely that the correlation between $h$ and $p(S^H)$ would be less than that between $h$ and $p(S^G)$.

Implementations of both the auxiliary model CV and the delta CV are described in Chapter 27.
20.4 SUMMARY

Control variates, if you can find some good ones, give very good speed-ups. Unfortunately you don’t get something for nothing. They are harder and more expensive to implement and are very much a case by case affair. The closer the control variate is to your problem the better it will be. Generic control variates are available, such as asset values themselves, but these give poorer results.

There are explicit solutions in the GBM and Vasicek models. Many more practical models, some widely used, are elaborations of GBM or Vasicek, and can be regarded as perturbations of them. GBM and Vasicek can therefore be used appropriately as auxiliary models in these cases, supplying potentially very effective control variates.

20.5 EXERCISES

1. If \( x \sim \text{LN}[\mu, \sigma^2] \) is log-normal, with density

\[
f(x) = \frac{1}{x\sigma \sqrt{2\pi}} \exp \left( -\frac{1}{2} \left( \frac{\ln(x) - \mu}{\sigma} \right)^2 \right),
\]

then

\[
\int_{K}^{\infty} x^k f(x) \, dx = \exp \left( k\mu + \frac{1}{2} k^2 \sigma^2 \right) N \left( \frac{\mu + k\sigma^2 - \ln(K)}{\sigma} \right).
\]

(a) Prove this.

(b) Find an explicit formula for the value of the option with payoff given by equation (20.27) when the underlying asset follows a geometric Brownian motion.

2. Consider a discretely reset up-and-in barrier call option. The option receives a payoff of \((S_T - X)^+\) at time \( T = t_N \) only if at some reset date \( t_i \leq t_N \) the value \( S_i \) of the asset on that date is greater than a barrier level \( U \).

(a) What might constitute a good CV for this option?

(b) Now suppose that the option is an up-and-out call option so that a payoff of \((S_T - X)^+\) is always received unless at some reset date \( t_i \leq t_N \) the value \( S_i \) of the asset on that date is greater than the barrier level \( U \). Are the same CVs likely to be effective for this option as for the up-and-in barrier call?

3. One way of computing hedge ratios by Monte Carlo is by path-wise differentiation. Suppose that a European option has payoff \( h(S) \) at time \( T \). Given a sample path \( S^j = \{S^j_0, \ldots, S^j_N\} \) write \( h^j = e^{-rT} h(S^j_N) \) for the discounted payoff along this path, so that with \( M \) sample paths the plain Monte Carlo estimate of the option value at time zero is \( \hat{c} = (1/M) \sum_{j=1}^{M} h^j \). Differentiate to obtain an estimate \( \hat{\delta} \) of the option delta,

\[
\hat{\delta} = \frac{\partial \hat{c}}{\partial S_0} = \frac{1}{M} \sum_{j=1}^{M} \frac{\partial h^j}{\partial S_0} = \frac{1}{M} \sum_{j=1}^{M} \frac{\partial h^j}{\partial S_N^j} \frac{\partial S_N^j}{\partial S_0}.
\]
If the process for $S$ is geometric Brownian motion with drift $r$ and volatility $\sigma$, then

$$\frac{\partial S^j_N}{\partial S_0} = \exp\left( \left(r - \frac{1}{2}\sigma^2\right) T + \sigma (w^j_N - w^j_0) \right)$$ \hspace{1cm} (20.81)

for the Wiener sample path $w^j = \{w^j_0, \ldots, w^j_N\}$ used to generate $S^j$.

(a) Suppose the option is a European call with $h(S) = (S - X)^+$. What control variates might be used, and might be effective, if you wanted to compute $\hat{\delta}$ by equation (20.80)?

(b) Can this approach work if the option is either an up-and-in barrier call option or an up-and-out barrier call option?
How are control variates to be implemented in practice? Logically a knowledge of its control variates is owned by each individual option object; only the option itself knows which CVs are suitable for it. This does not mean, however, that individual options should themselves compute their control variates. This would be inefficient in practice. Different options could request the same CV leading to potentially severe duplication.

In practice it is not an individual option that is being valued: it is a book of many options, and it is far more sensible to decide upon a bank of CVs for the book as a whole. These are valued alongside the options in the book. Each option has only to specify which CVs in the bank should be used as its particular CVs.

In fact there may be few candidates for CVs (although these can have a variety of strikes and times to maturity). Somewhere there has to be an explicit formula, for the option value or its delta, under the pricing model or a simplification of it used as an auxiliary model. There is a strong case for arguing that alongside the options in the book a representative of every option with a formula should go into the portfolio’s bank of CVs.

The cost of doing this is slight. We saw in Chapter 10 that the marginal cost of an additional option in a book is small; the cost of generating a CV (in most cases) is likely to be similar to the cost of generating an option payoff. The marginal cost of a single CV is very small and in any case the cost savings are likely to more than repay the outlay.

There are exceptions. As we see, delta CVs can be very expensive to implement, and tailored CVs, useful only for a particular option, may also be expensive.

In section 21.1 we present a control variate application. The version described here maintains a bank of CVs that individual options can select from. Timings are then given in section 21.2. Two options are used as test cases: the average rate option and the exotic option from Chapter 20. We find that large efficiency gains are possible for tailored CVs.

### 21.1 A CONTROL VARIATE APPLICATION

Control variates are not too hard to implement. MC_AverageRate_bank_CV.xls gives an example of pricing either the average rate option or the exotic option under GBM. The spreadsheet allows you to choose to add in from one to three control variates, or more if you define them, or to have none at all.

A special object, the CVmanager, is responsible for managing a bank of CVs. For each sample path it fills an array with CV values. This is passed, together with a discounted payoff value, to the option’s accumulator object.

**The application object**

The application object, App_MC_CV, is displayed in Figure 21.1. Its structure is not too different from MC_AverageRate_LD.xls and other speed-up examples from previous chapters. Some objects are polymorphic, but it is the same basic level 3b design.
Private out_ As OutputCounter
Private wie_ As WienerGeneratorPlain
Private acc_ As IAccumulator 'accumulator for CV MC
Private cvm_ As CVmanager 'manages CV objects
Private pay_ As IPayoff
Private gen_ As PathGenerator

Private M_ As Long 'number of paths
Private N_ As Long 'number of time steps
Private K_ As Long 'number of CVs

Private Sub Class_Initialize()
Set out_ = New OutputCounter
Set wie_ = New WienerGeneratorPlain
Set cvm_ = New CVmanager
Set gen_ = New PathGenerator
End Sub

Private Sub Class_Terminate()
Set wie_ = Nothing
Set acc_ = Nothing
Set cvm_ = Nothing
Set pay_ = Nothing
Set gen_ = Nothing
Set out_ = Nothing
End Sub

Friend Sub SetValues(ByRef data As InputManager)
Call out_.SetValues(data)
Call wie_.SetValues(data)
Call cvm_.SetValues(data)
Call gen_.SetValues(data)
Set acc_ = data.GetAccumulator
Set pay_ = data.GetOption
Call cvm_.SetCVlink(pay_) 'f
Call acc_.SetCVinds(pay_) 'g
M_ = data.M
N_ = data.N
K_ = cvm_.K
End Sub

Friend Property Get val() As Double: val = acc_.val: End Property
Friend Property Get se() As Double: se = acc_.se: End Property

Friend Sub run()
Dim path() As Double: ReDim path(0 To N_) As Double 'Stock values
Dim WieVec() As Double: ReDim WieVec(0 To N_) As Double 'Wiener path
Dim cv_vals() As Double
Dim val As Double
Dim i As Long
For i = 1 To M_ 'For each sample path
    Call out_.OutputCounter(i)
    Call wie_.GetWienerVec(WieVec) 'a. Wiener path
    Call gen_.Get_Wpath(WieVec, path) 'b. Asset path
    Call cvm_.Get_cvs(path, cv_vals) 'c.
    val = pay_.payoff(path) 'd.
    Call acc_.update(val, cv_vals) 'e.
Next i
End Sub

Figure 21.1 Control variate Monte Carlo: the application object, App_MC.CV
For each sample path the Wiener generator object, $\text{wie}_\text{1}$, first constructs a Wiener sample path. This is passed over to the model path generator, $\text{gen}_\text{1}$, which returns a sample path for the underlying asset. This path is given to the $\text{CVmanager}$ object, $\text{cvm}_\text{1}$, which computes CV values and puts them into the array $\text{cv}_\text{vals}$. An option value is requested from the option object and it and the CVs are passed over to the accumulator object $\text{acc}_\text{1}$. $\text{acc}_\text{1}$ has been previously set up to know how to use the CVs it has been passed.

The $\text{CVmanager}$ needs to know which CVs from the bank actually need to be generated. A registration-type process takes place in the $\text{App\_MC\_CV::SetValues()}$ method. On line 21.f the option object is passed to the $\text{CVmanager}$. It gives the $\text{CVmanager}$ the names of the CVs it requires. The $\text{CVmanager}$ notes these as CVs it needs to generate and passes back an array of index values to identify them by. On line 21.1g the option is passed to its accumulator object, $\text{acc}_\text{1}$. It gives the indexes to $\text{acc}_\text{1}$ enabling it to select only the CVs that are needed.

The $\text{CVmanager}$

The $\text{CVmanager}$ object is presented in Figure 21.2. It maintains a set of CV objects in the array $\text{CVs}_\text{1}$, created in the $\text{CVmanager}$’s constructor. This set is the CV bank; in this implementation it is hard-wired in. It constitutes all the CV objects for which the $\text{CVmanager}$ might possibly receive requests.

The $\text{CVmanager}$ holds two other arrays. There is an array of Strings, $\text{cvnames}_\text{1}$, holding the names of the CVs in the bank. $\text{cvnames}_\text{1}$ is set up in the constructor. The final array is a Boolean array, $\text{CV\_mask}_\text{1}$. A True entry in $\text{CV\_mask}_\text{1}$ means that the corresponding CV is to be computed. $\text{CV\_mask}_\text{1}$ is set up in the $\text{CVmanager::SetCVlink()}$ method (called by $\text{App\_MC\_CV}$). $\text{SetCVlink()}$ is passed an option object which it asks for the names of the CVs it requires. It looks these up in $\text{cvnames}_\text{1}$. If the name is found, the corresponding entry in $\text{CV\_mask}_\text{1}$ is set to True. The indexes of the names in $\text{cvnames}_\text{1}$ are passed back to the option object for safe keeping. They act as handles for objects to use to refer to particular CVs.

The $\text{CV objects}$

CV objects conform to the $\text{ICV}$ interface (not displayed). They have a name to identify them, with a $\text{Property}$ to extract it, and a $\text{Function}$ that returns the value of the CV computed from a sample path passed as an argument.

Figure 21.3 displays a typical CV object, $\text{CVGeoAverageRate}$. This computes the value $\text{CV}_{GA}$ of the control variate derived in the GBM framework from a discretely reset geometric average rate option (equation (20.35), page 325). The interface $\text{Function}, \text{cv}()$, returns the value of the CV as the difference between the discounted payoff computed from the stock price path and the explicit solution, $\text{cv\_val}_\text{1}$, precomputed in the $\text{SetValues()}$ method.

The $\text{option objects}$

The option object needs to be extended slightly so that it knows what CVs it needs. Figure 21.4 displays the $\text{PayoffArithCall}$ option object. It maintains a list of names, $\text{cv\_names}_\text{1}$, as a String array. A second array, $\text{CV\_inds}_\text{1}$, contains the indexes of the corresponding CVs in the $\text{CVmanager}$ object. $\text{CV\_inds}_\text{1}$ is set by the $\text{CVmanager}$ in the $\text{CVmanager::SetCVlink()}$ method. $\text{CV\_inds}_\text{1}$ is kept in the option object to be used by the accumulator object to pull out from the complete set of generated CVs only those values needed by the option.

$\text{GetCVnames()}$ is the getter for the array of CV names, called by the $\text{CVmanager}$. A better design would enable the option object to pass on not just the name of its CV but also a set of parameters, perhaps
Private CVs_() As ICV 'The CV bank
Private cvnames_() As String 'Names of the CVs in the bank
Private CV_mask_() As Boolean 'Computes CV values specified by the mask
Private K_ As Long 'total number of CVs
Private DoCVs_ As Boolean 'True if CVs are requested

Private Sub Class_Initialize()
    K_ = 0
    Call AddCV(New CVTerminalStock) 'Construct the bank
    Call AddCV(New CVGeoAverageRate)
    Call AddCV(New CVEuroCallBS)
    Call AddCV(New CVquadExotic)
    ReDim cvnames_(1 To K_) As String
    ReDim CV_mask_(1 To K_) As Boolean
    Dim K As Long
    For K = 1 To K_
        cvnames_(K) = CVs_(K).CVname
        CV_mask_(K) = False
    Next K
End Sub

Private Sub Class_Terminate()
    If K_ = 0 Then Exit Sub
    Dim K As Long
    For K = 1 To K_
        Set CVs_(K) = Nothing
    Next K
End Sub

Friend Sub SetValues(ByRef data As InputManager)
    DoCVs_ = data.CVs: If Not DoCVs_ Then Exit Sub
    Dim K As Long
    For K = 1 To K_
        Call CVs_(K).SetValues(data)
    Next K
End Sub

Property Get K() As Long: K = K_: End Property

Friend Sub SetCVlink(pay As IPayoff)
    If pay.K = 0 Then Exit Sub
    Dim cv_names() As String: Call pay.GetCVnames(cv_names)
    Dim lb As Long: lb = LBound(cv_names)
    Dim ub As Long: ub = UBound(cv_names)
    Dim CVinds() As Long: ReDim CVinds(lb To ub) As Long
    Dim i As Long
    For i = lb To ub
        CVinds(i) = FindIndex(cv_names(i), cvnames_)
        CV_mask_(CVinds(i)) = True
    Next i
    Call pay.SetCVinds(CVinds)
End Sub

Friend Sub Get_cvs(path() As Double, ByRef CVs() As Double)
    If Not DoCVs_ Then Exit Sub
    If K_ = 0 Then Exit Sub
    ReDim CVs(1 To K_) As Double
    Dim i As Long
    For i = 1 To K_
        If CV_mask_(i) Then CVs(i) = CVs_(i).CV(path)
    Next i
End Sub

Private Sub AddCV(obj As ICV)
    K_ = K_ + 1: ReDim Preserve CVs_(1 To K_) As ICV: Set CVs_(K_) = obj
End Sub

'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX structural XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

Figure 21.2 The CVmanager object
Implementing Control Variates

```vbnet
Implements ICV
Private X_ As Double
Private discount_ As Double
Private cv_val_ As Double
Private name_ As String

Private Sub Class.Initialize()
    name_ = "GeoAverage"
End Sub

Friend Sub ICV_SetValues(ByRef data As InputManager)
    discount_ = Exp(-data.rr * data.T)
    X_ = data.X
    cv_val_ = DiscreteStnGeometricAverageCall(data.N, data.T / data.N, X_,
    data.S_0, data.sigma, data.rr)
End Sub

Friend Property Get Icv_CVname() As String: Icv_CVname = name_: End Property

Friend Function ICV_cv(path() As Double) As Double
    Dim av As Double: av = GeometricAverage0(path) 'Excludes path(lb)
    ICV_cv = discount_ * my_max(0, av - X_) - cv_val_
End Function
```

Figure 21.3 A control variate object, CVGeoAverageRate

Program 21.3 A control variate object

a strike and a maturity time, to configure it. The CVmanager would then instantiate the correctly configured
option object.

GetCVinds() and SetCVinds() are the getter and setter methods for the CV index array, CVinds_.
SetCVinds() is used by the CVmanager to let the option have the indexes; GetCVinds() is called by
the accumulator object to obtain them from the option object.

The CV accumulator object

The accumulator object is responsible for computing the CV adjusted option value. Figures 21.5 and 21.6
display the object AccumulatorCV. Whenever a new discounted payoff and CV values are available they
are passed to AccumulatorCV using its update() method. When the option value and its standard error
are required they are obtained from AccumulatorCV using its val() and se() Properties.

When its update() method (Figure 21.6) is called the accumulator simply puts the values it is passed
into a pair or arrays, vals_ and CVs_, to be processed later. Only when an option value is requested
is it calculated, along with the standard error and the standard deviation, by the Private method,
ComputeValues().

ComputeValues() performs the regression in equation (20.3), page 318, using its composite regression
object, bet_. It then computes the adjusted payoff values, and calculates the option value and
standard error.

Here it is a bit lazy, and not too efficient, to store all the payoff values and CV values until they are
passed on to bet_. It is unnecessarily wasteful on storage. A better alternative might be just to accumulate
the various sums of squares and cross-product values that go into the regression formula. However a
slice-based Monte Carlo would still need to accept values as slices.
Figure 21.4  The option object, PayoffArithCall

21.2 NUMERICAL ASSESSMENT

We assess the use of control variates for the arithmetic average rate option and the exotic option in the plain GBM model. In each case a delta control variate is implemented for each option (with mixed results) as well as standard CVs. Using a delta CV here can be regarded as a benchmarking exercise. Delta CVs might normally be used with an auxiliary model.

21.2.1 CVs and the arithmetic average rate option

First we consider the use of standard control variates for the arithmetic average rate option and then we assess the effectiveness of a delta control variate.

Standard CVs

Three CVs are investigated: the terminal stock CV, \( CV_S \equiv CV_{S,N} \) (equation (20.6), page 320); the Black–Scholes call CV, \( CV_c \); and the tailored CV, \( CV_{GA} \), using the explicit geometric average rate option (equation (20.35), page 325).

\( CV_c \) is computed as

\[
CV_c = e^{-rT}(S_N - X)^+ - c_t
\]
where $c_t$ is the Black–Scholes call value. Since the average rate option has a call type payoff, and a higher terminal stock price is likely to be associated with a higher average, we expect the use of $CV_c$ to be beneficial.

The CVs can be used by themselves or in combination. Results are presented in Table 21.1 for both the ATM ($X = 100$) and OTM ($X = 150$) versions of the option. Timings were produced in the spreadsheet MC_AverageRate_CV_timings.xls. (In this sheet options themselves compute the values of their CVs.)
Friend Sub IAccumulator_SetCVinds(pay As IPayoff)
    K_ = pay.K: If K_ = 0 Then Call RaiseError(3, "AccumulatorCV", "bad K")
    Call pay.GetCVinds(CVinds_)
    Call ReDimArrays(M, K_)
End Sub

Friend Sub IAccumulator_update(val As Double, CVs() As Double)
    mm_ = mm_ + 1: If mm_ > M_ Then Call RaiseError(3, "AccumulatorCV", "bad mm")
    vals_(mm_) = val
    Dim K As Long
    For K = 1 To K_
        CVs_(_K, mm_) = CVs(CVinds_<_K)
    Next K
End Sub

Private Sub ComputeValues()
    If values_done_ Then Exit Sub
    Dim beta() As Double: ReDim beta(0 To K_) As Double '0 is the constant
    Call bet_.GetBeta(beta, vals_, cvs_, K_, mm_)
    Dim acc_vals As Double: acc_vals = 0#
    Dim acc_squs As Double: acc_squs = 0#
    Dim i As Long, K As Long
    For i = 1 To mm_
        Dim offset As Double: offset = 0#
        For K = 1 To K_
            offset = offset + beta(K) * cvs_(_K, i)
        Next K
        Dim adj_val As Double: adj_val = vals_(i) - offset
        acc_vals = acc_vals + adj_val
        acc_squs = acc_squs + adj_val * adj_val
    Next i
    val_ = discount_ * acc_vals / mm_
    Dim rad As Double: rad = acc_squs - acc_vals * acc_vals / mm_
    If rad < 0 Then
        se_ = 0#
        sd_ = 0#
    Else
        se_ = Sqr(rad) / mm_
        sd_ = Sqr(rad) / Sqr(mm_) 
    End If
    values_done_ = True
End Sub

Private Sub Checkmm()
    If mm_ < 2 Then Call RaiseError(3, "AccumulatorCV", "mm too small")
End Sub

Private Sub ReDimArrays(M As Long, K As Long)
    ReDim vals_(1 To M_) As Double
    ReDim CVs_(1 To K, 1 To M_) As Double
End Sub
The achievable efficiency gains are very good. By itself the stock CV, $CV_S$, it not particularly impressive, although a gain of a factor of 3 for the ATM option is not to be dismissed. This is the same order as using the call CV, $CV_c$.

There is no additional benefit to using $CV_S$ and $CV_c$ together. This is not surprising; the values of $(S_N - X)^+$ and $S_N$ are comonotonic over most of their domain.

The real gains come from using the tailored CV, $CV_{GA}$. By itself it yields a gain of over 1000 for the ATM option; used with $CV_c$ the gain exceeds 1600. These are excellent. Since the overall computation time is increased only marginally, the gain comes chiefly from the decrease in standard error – more than a factor of 40 in the $CV_c + CV_{GA}$ case.

Using $CV_S$ in conjunction with $CV_{GA}$ produces only a small enhancement, or none at all.

Gains for the OTM option are nothing like as great as those for the ATM option. Here also the tailored CV, $CV_{GA}$, is the most effective.

---

**Table 21.1** Standard control variates with ATM and OTM average rate options, $N = 16$

<table>
<thead>
<tr>
<th>Method</th>
<th>$X = 100$</th>
<th>$X = 150$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plain</td>
<td>6.09 (0.027)</td>
<td>0.0045 (0.0007)</td>
</tr>
<tr>
<td></td>
<td>[3.02]</td>
<td>[3.12]</td>
</tr>
<tr>
<td>$CV_S$</td>
<td>6.06 (0.015)</td>
<td>0.0045 (0.0007)</td>
</tr>
<tr>
<td></td>
<td>[3.12]</td>
<td>[3.23]</td>
</tr>
<tr>
<td></td>
<td><strong>2.8</strong></td>
<td><strong>0.9</strong></td>
</tr>
<tr>
<td>$CV_c$</td>
<td>6.06 (0.014)</td>
<td>0.0053 (0.0007)</td>
</tr>
<tr>
<td></td>
<td>[3.28]</td>
<td>[3.26]</td>
</tr>
<tr>
<td></td>
<td><strong>3.4</strong></td>
<td><strong>1.1</strong></td>
</tr>
<tr>
<td>$CV_S + CV_c$</td>
<td>6.06 (0.014)</td>
<td>0.0051 (0.0007)</td>
</tr>
<tr>
<td></td>
<td>[3.39]</td>
<td>[3.33]</td>
</tr>
<tr>
<td></td>
<td><strong>3.3</strong></td>
<td><strong>1.0</strong></td>
</tr>
<tr>
<td>$CV_{GA}$</td>
<td>6.0577 (0.00074)</td>
<td>0.0051 (0.0002)</td>
</tr>
<tr>
<td></td>
<td>[3.51]</td>
<td>[3.51]</td>
</tr>
<tr>
<td></td>
<td><strong>1108</strong></td>
<td><strong>10.3</strong></td>
</tr>
<tr>
<td>$CV_{GA} + CV_S$</td>
<td>6.0584 (0.00061)</td>
<td>0.0052 (0.0002)</td>
</tr>
<tr>
<td></td>
<td>[3.59]</td>
<td>[3.61]</td>
</tr>
<tr>
<td></td>
<td><strong>1279</strong></td>
<td><strong>14.5</strong></td>
</tr>
<tr>
<td>$CV_{GA} + CV_c$</td>
<td>6.0582 (0.00061)</td>
<td>0.0053 (0.0002)</td>
</tr>
<tr>
<td></td>
<td>[3.59]</td>
<td>[3.61]</td>
</tr>
<tr>
<td></td>
<td><strong>1615</strong></td>
<td><strong>13.6</strong></td>
</tr>
<tr>
<td>$CV_{GA} + CV_S + CV_c$</td>
<td>6.0586 (0.00061)</td>
<td>0.0051 (0.0002)</td>
</tr>
<tr>
<td></td>
<td>[3.65]</td>
<td>[3.66]</td>
</tr>
<tr>
<td></td>
<td><strong>1545</strong></td>
<td><strong>15.5</strong></td>
</tr>
</tbody>
</table>
Benchmarking the delta CV

We implement the delta CV, $CV_{GA,\delta}$ (equation (20.36), page 325). We do not expect $CV_{GA,\delta}$ to work well in this case as it is expensive and would be used only where no alternative is available. Since it is a tailored CV it is not likely to be of use to other options in a portfolio, so its marginal cost cannot be ignored.

Table 21.2 gives results for strikes 100 and 150, and for 4, 8, 16, 32 and 64 resets. These are computed in the spreadsheet MC_delta_GBM.xls.

We are pleasantly surprised by the ATM case. As the number of reset dates increases, so the power of the CV continues to improve, not quite doubling with each doubling of the number of reset dates. An efficiency gain of 32 at $N = 64$ is very welcome. The effect of doubling the number of time steps is to double (roughly) the computation time but it also increases the correlation of the CV estimate. In this case the increased correlation more than compensates for the additional computation time.

Unfortunately these results are not echoed in the OTM case. There is a great deal of noise in the results but it is clear that although there are reductions in standard error these are outweighed by the additional computation time.

In the GBM model one would always use $CV_{GA}$ directly and not the delta CV derived from it. However, the results shown here are promising for later, at least in the ATM case, when we consider using the delta CV with an auxiliary model.

### 21.2.2 Control variates with the exotic option

We use both the payoff matching CV, $d_{a,b}$, (equation (20.30), page 324) and the delta CV, $h_{a,b}$, (equation (20.31)) derived from it. Good results are expected from the payoff matching CV but not from the delta CV. The delta CV is expensive, in that

1. a hedge portfolio requires many time steps before the hedging error becomes small;
2. computing each delta requires four computations of the normal distribution function;
3. in any case, the correlation with option payoff is never likely to be as high as that of the payoff matching CV from which it is derived.

The full extent of the expense is discussed below.

### Table 21.2 Delta CV with the arithmetic average rate option

<table>
<thead>
<tr>
<th>Option</th>
<th>Method</th>
<th>$N = 4$</th>
<th>$N = 8$</th>
<th>$N = 16$</th>
<th>$N = 32$</th>
<th>$N = 64$</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATM (X = 100)</td>
<td>Plain</td>
<td>6.92 (0.030)</td>
<td>6.37 (0.028)</td>
<td>6.04 (0.027)</td>
<td>5.89 (0.026)</td>
<td>5.83 (0.026)</td>
</tr>
<tr>
<td></td>
<td>Delta</td>
<td>6.950 (0.0079)</td>
<td>6.345 (0.0054)</td>
<td>6.056 (0.0038)</td>
<td>5.915 (0.0027)</td>
<td>5.833 (0.0019)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[1.02]</td>
<td>[1.67]</td>
<td>[3.06]</td>
<td>[5.69]</td>
<td>[10.8]</td>
</tr>
<tr>
<td>OTM (X = 150)</td>
<td>Plain</td>
<td>0.016 (0.0012)</td>
<td>0.0080 (0.00084)</td>
<td>0.0050 (0.00075)</td>
<td>0.0040 (0.00059)</td>
<td>0.0026 (0.00044)</td>
</tr>
<tr>
<td></td>
<td>Delta</td>
<td>0.019 (0.0013)</td>
<td>0.0080 (0.00068)</td>
<td>0.0054 (0.00044)</td>
<td>0.0045 (0.00031)</td>
<td>0.0040 (0.00023)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[1.02]</td>
<td>[1.69]</td>
<td>[3.02]</td>
<td>[5.61]</td>
<td>[10.9]</td>
</tr>
</tbody>
</table>

### Table 21.2 (Continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Method</th>
<th>$N = 4$</th>
<th>$N = 8$</th>
<th>$N = 16$</th>
<th>$N = 32$</th>
<th>$N = 64$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0.20</td>
<td>0.32</td>
<td>0.57</td>
<td>0.69</td>
<td>0.69</td>
</tr>
</tbody>
</table>
Table 21.3 Efficiency gains E for the exotic option

Panel (a): Efficiency gains for the payoff matching CV, various a and b

<table>
<thead>
<tr>
<th>a</th>
<th>100</th>
<th>101</th>
<th>102</th>
<th>103</th>
<th>104</th>
<th>105</th>
<th>106</th>
<th>107</th>
<th>108</th>
<th>109</th>
<th>110</th>
</tr>
</thead>
<tbody>
<tr>
<td>99</td>
<td>1.8</td>
<td>2.2</td>
<td>2.6</td>
<td>3.3</td>
<td>4.4</td>
<td>6.4</td>
<td>10.3</td>
<td>14.0</td>
<td>11.0</td>
<td>8.2</td>
<td>7.1</td>
</tr>
<tr>
<td>100</td>
<td>—</td>
<td>2.5</td>
<td>3.1</td>
<td>4.0</td>
<td>5.7</td>
<td>8.9</td>
<td>13.9</td>
<td>11.7</td>
<td>8.0</td>
<td>6.2</td>
<td>5.6</td>
</tr>
<tr>
<td>101</td>
<td>—</td>
<td>—</td>
<td>3.2</td>
<td>4.1</td>
<td>5.4</td>
<td>6.1</td>
<td>6.5</td>
<td>5.6</td>
<td>4.3</td>
<td>4.0</td>
<td>3.7</td>
</tr>
</tbody>
</table>

Panel (b): Efficiency gains with a delta CV

<table>
<thead>
<tr>
<th>Method</th>
<th>N = 1</th>
<th>N = 2</th>
<th>N = 4</th>
<th>N = 8</th>
<th>N = 16</th>
<th>N = 32</th>
<th>N = 64</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plain</td>
<td>21.5</td>
<td>21.8</td>
<td>21.6</td>
<td>21.5</td>
<td>21.7</td>
<td>21.5</td>
<td>21.5</td>
</tr>
<tr>
<td></td>
<td>(0.11)</td>
<td>(0.11)</td>
<td>(0.11)</td>
<td>(0.11)</td>
<td>(0.11)</td>
<td>(0.11)</td>
<td>(0.11)</td>
</tr>
<tr>
<td></td>
<td>[0.44]</td>
<td>[0.63]</td>
<td>[0.94]</td>
<td>[1.59]</td>
<td>[2.94]</td>
<td>[5.50]</td>
<td>[10.64]</td>
</tr>
<tr>
<td></td>
<td>(0.11)</td>
<td>(0.10)</td>
<td>(0.089)</td>
<td>(0.076)</td>
<td>(0.061)</td>
<td>(0.047)</td>
<td>(0.035)</td>
</tr>
<tr>
<td></td>
<td>[1.66]</td>
<td>[3.25]</td>
<td>[6.16]</td>
<td>[11.9]</td>
<td>[23.5]</td>
<td>[46.6]</td>
<td>[93.2]</td>
</tr>
<tr>
<td>(E_1)</td>
<td>0.27</td>
<td>0.16</td>
<td>0.11</td>
<td>0.075</td>
<td>0.060</td>
<td>0.050</td>
<td>0.046</td>
</tr>
<tr>
<td>(E_N)</td>
<td>0.27</td>
<td>0.23</td>
<td>0.23</td>
<td>0.27</td>
<td>0.40</td>
<td>0.63</td>
<td>1.10</td>
</tr>
</tbody>
</table>

**Payoff-matching CV**

For \(X_1 = 100\) and \(X_2 = 110\) Table 21.3, panel (a), shows the efficiency gains achieved with \(d_{a,b}\) for various combinations of \(a\) and \(b\). The table is computed with \(N = 4\) time steps in both the plain and CV cases. The plain case took 0.938 seconds to run. The time taken in the CV case does not vary significantly with \((a, b)\), lying between 1.01 and 1.09 seconds.

The best speed-up is only a factor of 14 but this is none the less satisfactory; it is achieved very cheaply.

**Benchmarking the delta CV**

The delta CV, \(h_{a,b}\), is available, but we do not expect it to be very effective. Results are given in Table 21.3, panel (b). Two efficiency gains are shown for each \(N\). The first, \(E_1\), compares the delta CV with \(N\) time steps with the plain Monte Carlo with a single time step (the plain \(N = 1\) case); the second, \(E_N\), compares the delta CV with \(N\) time steps with the plain Monte Carlo also with \(N\) time steps.

Results are very poor. Although using \(h_{a,b}\) does reduce the standard error it does so only at very great expense. It is 8 times slower (with \(N = 64\)) to compute \(h_{a,b}\) than it is to run the plain case with the same number of steps. \(E_1\) declines as \(N\) increases. Adding additional time steps continues to reduce the standard error but the reduction is not sufficient to compensate for the increase in computation time. With \(E_N\), perhaps embodying a fairer comparison, there is no improvement until \(N = 64\) and even this is hardly worthwhile.

---

1. MC_exotic_CV.xls, a chopped down and modified version of MC_AverageRate_CV.xls, was used to produce the table.
21.3 SUMMARY

A tailored CV, like $CV_{GA}$, can work very well indeed. Other CVs are less effective. The terminal stock CV and Black–Scholes call CV provide some improvement for the average rate option, but nothing spectacular; the correlations are just not high enough. Nevertheless the gains are real and should be exploited.

For payoff-matching CVs to work well the payoff needs to be matched very closely: a rough match produces only rough efficiency gains. Further examples are given in Chapter 23.

21.4 EXERCISES

1. How effective are the terminal asset and Black–Scholes call CVs when used with the exotic option?

2. For the exotic option example in section 21.2.2 add in further piece-wise linear segments to the payoff-approximating piece-wise linear function to establish what further improvements can be made to the efficiency gain. That is, for $a < b < c \leq X_2$ and $0 < h < X_1$, define an approximating function

$$H_{a,b,c}(S) = \begin{cases} 
0, & S \leq a, \\
\frac{S - a}{b - a}h, & a < S \leq b, \\
h + \frac{S - b}{c - b}(X_1 - h), & b < S \leq c, \\
X_1, & c < S \leq c', \\
h + \frac{b' - S}{b' - c}(X_1 - h), & c' < S \leq b', \\
a' - \frac{a' - S}{a' - b'}h, & b' < S \leq a', \\
0, & a' < S,
\end{cases}$$

(21.2)

for $c' = 2X_2 - c$.

(a) What is the explicit value of the approximating option in this case?

(b) By experimenting with different sets of values for $(a, b, c, h)$ what order of efficiency gain can be achieved?

3. The arithmetic and geometric averages, $a^j$ and $g^j$, were defined in equations (20.33) and (20.34). It was asserted that they were highly correlated.

(a) Using a numerical experiment compute empirically the correlation between $a^j$ and $g^j$ when the underlying asset follows a geometric Brownian motion with drift $r = 0.03$ volatility $\sigma = 0.25$, and the number $N$ of equally spaced reset dates takes values 4, 8, 16, 32 and 64 up to time $T = 1$.

(b) For the same set of values, what is the correlation between $(a^j - X)^+ \text{ and } CV_{GA}^j$?
Importance sampling (IS) is a method that can produce very good efficiency gains for options (and models) where other methods are not so effective. For instance, consider an option that is far out of the money. With an ordinary method the proportion of sample paths that generate a non-zero payoff will be very small, and the standard error of the Monte Carlo estimate will be very high. Stratified sampling, or using control variates, can help, but cannot, as we have seen, produce significant variance reduction in these cases.

Importance sampling can overcome this problem. It samples from a well-chosen (hopefully) distribution in which more paths are in the money, yielding a lower standard error than the original distribution. When transformed back to the pricing measure the standard error remains low.

There are two main ways of obtaining benefit from importance sampling. The first, emphasized in the treatment in this chapter, is simply to use it with OTM options to ensure that more paths are in the money. The second is to enhance the variance reduction for options of any moneyness by matching the payoff-weighted density as closely as possible.

The first benefit is direct, crude and effective for the OTM options we are chiefly concerned with; the second can result in very great efficiency gains but these are much harder to achieve. It relies upon a very good match to the payoff-weighted density, depending sensitively on the option and model parameters.

We first present some theory on importance sampling, giving some examples of IS densities and weights. An implementation is given in section 22.4 and finally numerical results are presented in section 22.5.

We find that quite good efficiency gains are possible, even without tailored densities, but that the technique is ‘delicate’.

### 22.1 IMPORTANCE SAMPLING

Suppose that a European option has payoff $h(S_T)$ at time $T$ and that (for simplicity) interest rates are constant. If $f_T(S)$ is the density of the underlying asset at time $T$ then the value $c_t$ of the option at time $t < T$ is

$$c_t = e^{-r(T-t)} \int_0^\infty h(S) f_T(S) \, dS. \quad (22.1)$$

Suppose that $h(S)$ is non-zero only in regions where $f_T(S)$ is small. This would be the case for a far out-of-the-money call option with strike $X \gg S_0$ when $\Pr[S_T \geq X] = \int_X^\infty f_T(S) \, dS < \varepsilon$ is small.

The method works by transforming the integrand and making a compensating change in the density under which it is being integrated.

Suppose that, for $U \subseteq \mathbb{R}^Q$, there is a measure $\mathbb{F}$ on $U$ with distribution function $F$ and density $f$ and we want to evaluate

$$H(U) = \mathbb{E}_U^\mathbb{F} [h(u)] = \int_U h(u) \, dF = \int_U h(u) f(u) \, du. \quad (22.2)$$
The plain Monte Carlo estimate $\hat{H}_F(U)$ of $H(U)$ is

$$ \hat{H}_F(U) = \frac{1}{M} \sum_{j=1}^{M} h(u_j) , $$

(22.3)

where $u_j$ is sampled according to $f$.

Now suppose that there is a second measure $G$ on $U$ with density $g(u)$ such that $g(u) = 0 \Rightarrow f(u) = 0$. Write $\overline{U} = \{ u \in U \mid g(u) > 0 \}$ then

$$ H(U) = \int_U h(u) f(u) \, du = \int_U h(u) \frac{f(u)}{g(u)} g(u) \, du = \mathbb{E}_G^U \left[ h(u) \frac{f(u)}{g(u)} \right] , $$

(22.4)

where the expectation is taken with respect to the measure $G$. Hence an estimate $\hat{H}_G(U)$ of $H(U)$ can be found by sampling $h(u) f(u)/g(u)$ under $G$,

$$ \hat{H}_G(U) = \frac{1}{M} \sum_{j=1}^{M} h(u_j) \frac{f(u_j)}{g(u_j)} , $$

(22.5)

where $u_j$ is sampled according to $g$.

This technique, of evaluating $\mathbb{E}_F^V [h(u)]$ under $F$ as $\mathbb{E}_G^U [h(u) f(u)/g(u)]$ under $G$, is called importance sampling. Whether or not $\hat{H}_G(U)$ is a better estimate of $H(U)$ than $\hat{H}_F(U)$ depends on the choice of $g$ and the cost of computing the ratio $f(u)/g(u)$. Suppose, for instance, that $h(u) f(u)$ is everywhere positive and that $g(u)$ is proportional to $h(u) f(u)$, so that $h(u) f(u) = k g(u)$ for some constant $k$. Then

$$ \hat{H}_G(U) = \frac{1}{M} \sum_{j=1}^{M} h(u_j) \frac{f(u_j)}{g(u_j)} \equiv k $$

(22.6)

is computable immediately with zero variance from a single sample. Of course, since to find $g(u)$ one has already found $k = \int_U h(u) f(u) \, du = c$, this is not too helpful.

However suppose that it is possible to find a density $g$ that is almost proportional to $h(u) f(u)$, so that $k(u) = h(u) f(u)/g(u)$ is almost constant, then

$$ \text{var}(\hat{H}_G) = \frac{1}{M} \text{var} \left( h(u) \frac{f(u)}{g(u)} \right) = \frac{1}{M} \text{var}(k(u)). $$

(22.7)

If $g(u)$ is close to being proportional to $h(u) f(u)$, so that $k(u)$ is close to being constant, the variance of the importance sampled estimate can be very small.

### 22.2 VALUING AN OTM DIGITAL OPTION

As a concrete example let $d_t$ be the value at time $t$ of a digital option with payoff $1_{[L \leq S_T \leq U]}$ at time $T > t$ and that $S_t$ follows a GBM. Let $V = [L, U]$ and $v = [l, u]$ for $l = \ln(L)$, $u = \ln(U)$. Then

$$ d_t = e^{-r(T-t)} \int_{0}^{\infty} 1_{S \in V} \hat{f}(S) \, dS $$

(22.8)
where \( \hat{f}(S) \equiv \hat{f}(S | S_t) \) is the density of \( S_T \) at time \( T \). Set \( R = \ln(S) \) so that \( R = \ln(S) \sim N(\mu_T, \sigma^2_T) \) is normal with mean \( \mu_T = \ln(S_t) + (r - \frac{1}{2} \sigma^2)(T - t) \) and variance \( \sigma^2_T = \sigma^2(T - t) \). Writing \( f \) for the density of \( R \) we have

\[
d_t = e^{-r(T-t)} \int_{-\infty}^{\infty} 1_{r \in V} f(r) \, dr. \tag{22.9}
\]

This integration can of course be computed explicitly, but consider evaluating it by Monte Carlo. Write \( F \) for the distribution function of \( R \) and let \( q_l \) and \( q_u \) be the quantiles of \( N(\mu_T, \sigma^2_T) \) corresponding to \( l \) and \( u \), so that \( q_l = F(\ln(L)) \), \( q_u = F(\ln(U)) \). A plain Monte Carlo method computes

\[
\hat{d}_t = e^{-r(T-t)} \frac{1}{M} \sum_{j=1}^{M} 1_{R_j \in V}, \tag{22.10}
\]

where \( R_j \) is drawn from \( N(\mu_T, \sigma^2_T) \).

This is all well and good but suppose that \( S_t \ll L \) is far from the money, so that \( q_l \) is close to 1 and \( p_v = q_u - q_l \) is small. We have \( \mathbb{E}[1_{R \in V}] = p_v \) and \( \text{var}(1_{R \in V}) = p_v(1 - p_v) \sim p_v \) so, setting \( \delta = e^{-r(T-t)} \), we find that \( \mathbb{E}[\hat{d}_t] = \delta p_v \) and

\[
\text{var}(\hat{d}_t) \sim \frac{\delta^2}{M} p_v = \frac{\delta}{M} \mathbb{E}[\hat{d}_t]. \tag{22.11}
\]

As a function of the option value this variance is huge. If \( p_v = 10^{-2} \) then the ratio of the standard error \( se \equiv \sqrt{\text{var}(\hat{d}_t)} \) to the option value is

\[
\frac{se}{\delta p_v} = \frac{1}{\sqrt{M}p_v}. \tag{22.12}
\]

This does not get below a hundredth of the option value until \( M \) is 10 000 times the reciprocal of \( p_v \). For \( p_v = 10^{-2} \) this requires \( 10^6 \) draws; an enormous number for a standard error that is still quite high.

Importance sampling comes to our rescue. Set \( \mu = \frac{1}{2}(l + u) \) and \( \sigma = \frac{1}{2}(u - l) \) and sample under the distribution \( \mathcal{N}(\mu, \sigma^2) \). Writing \( g \) for the corresponding density we have

\[
d_t = e^{-r(T-t)} \int_{-\infty}^{\infty} 1_{r \in V} f(r) \frac{g(r)}{g(r)} \, dr \tag{22.13}
\]

and the Monte Carlo estimate becomes

\[
\tilde{d}_t = e^{-r(T-t)} \frac{1}{M} \sum_{j=1}^{M} 1_{R_j \in V} \frac{f(R_j)}{g(R_j)}. \tag{22.14}
\]

\( R_j \) where it is sampled under \( g \).

With equation (22.14) there is a non-zero payoff with probability 0.68 and when \( R \in V \) the payoff is \( p(R) \),

\[
p(R) = \frac{\sigma}{\sigma_T} \exp \left( -\frac{1}{2} \left( \frac{R - \mu_T}{\sigma_T} \right)^2 + \frac{1}{2} \left( \frac{R - \mu}{\sigma} \right)^2 \right), \tag{22.15}
\]
and we find that $\text{var}(\tilde{d}_t) < \text{var}(\hat{d})$. Indeed

$$\text{var}
\left(\frac{f(R)}{g(R)}\right) = \mathbb{E}^\mathbb{G}
\left[\frac{f(R)}{g(R)}\right]^2
- \mathbb{E}^\mathbb{G}\left[\frac{f(R)}{g(R)}\right]^2 \tag{22.16}$$

$$= \mathbb{E}^\mathbb{P}\left[\frac{f(R)}{g(R)}\right] - p_v^2. \tag{22.17}$$

But

$$\mathbb{E}^\mathbb{P}\left[\frac{f(R)}{g(R)}\right] < p_v \max_{R \in v} \frac{f(R)}{g(R)} \tag{22.18}$$

and note that

$$\max_{R \in v} \frac{f(R)}{g(R)} < \frac{f(l)}{g(l)} = f(l)/(l-u)\sqrt{\frac{\pi e}{2}} < p_v \sqrt{2\pi e} \tag{22.19}$$

since from the geometry of $f$, $f(l)(u-l) < p_v f(l)/f(u)$ and $1 \sim f(l)/f(u) < 2$, say, when $u-l$ is small. We conclude that

$$\text{var}(\tilde{d}_t) \sim \frac{\delta^2}{M} \text{var}\left(\frac{f(R)}{g(R)}\right) < \frac{\delta^2}{M} p_v^2 \left(\sqrt{2\pi e} - 1\right) = p_v \left(\sqrt{2\pi e} - 1\right) \text{var}(\hat{d}_t). \tag{22.20}$$

Since $p_v$ is very close to 0 the reduction in variance is considerable. The standard error is now less than a fraction $\frac{1}{\sqrt{M}} \sqrt{2\pi e} - 1$ of the option price, a very great improvement over equation (22.12).

### 22.3 CHOICES FOR THE IS DENSITY

Suppose an asset has process $S = (S_t)_{t \geq 0}$. Let $T = \{t_1, \ldots, t_N\}$ be a set of reset dates. Write $S_t$ for $S_t$ and set $S = (S_1, \ldots, S_N) \in \mathbb{R}^N$. Suppose we want to price a European path-dependent option with payoff $h(S)$ at time $T = t_N$. Its value at time $t_0 < t_1$ is $c$,

$$c = e^{-r(T-t_0)} \int_U h(S) f(S \mid S_0) dS, \tag{22.21}$$

where $f(S \mid S_0)$ is the density of $S$ on $U = \mathbb{R}^N$ conditional on the value $S_0$ of $S$ at time $t_0$.

Suppose that $S$ follows a GBM,

$$dS_t = rS_t dt + \sigma S_t dz_t. \tag{22.22}$$

We have

$$f(S \mid S_0) = \prod_{i=1}^N f(S_i \mid S_{i-1}) \tag{22.23}$$

where $f(S_i \mid S_{i-1})$ is the density of $S_i$ conditional on $S_{i-1}$. 

$S_i \mid S_{i-1} \sim \text{LN}(\mu_i, \sigma^2_\Delta)$ is log-normal, with

$$
\mu_i = \ln(S_{i-1}) + \left( r - \frac{1}{2} \sigma^2 \right) \Delta t,
$$

$$
\sigma^2_\Delta = \sigma^2 \Delta t,
$$

where $\Delta t \equiv \Delta t_i = t_{i+1} - t_i$, so that

$$
f(S_i \mid S_{i-1}) = \frac{1}{S_i \sigma_\Delta \sqrt{2\pi}} \exp \left( -\frac{1}{2} \left( \frac{\ln(S_i) - \mu_i}{\sigma_\Delta} \right)^2 \right).
$$

and the density of $f(S \mid S_0)$ is

$$
f(S \mid S_0) = \prod_{i=1}^{N} f(S_i \mid S_{i-1})
$$

$$
= \frac{1}{\sigma_\Delta^N (2\pi)^{N/2}} \left( \prod_{i=1}^{N} S_i \right)^{-1} \exp \left( -\frac{1}{2 \sigma_\Delta^2} \sum_{i=1}^{N} (\ln(S_i) - \mu_i)^2 \right).
$$

Now suppose we want to apply importance sampling to this option, sampling $S$ from a density $g$,

$$
c = e^{-r(T-t_0)} \int_{U} h(S) \frac{f(S \mid S_0)}{g(S \mid S_0)} g(S \mid S_0) \, dS.
$$

Write $q(S) \in \mathbb{R}$ for $h(S)f(S)$ (where we drop the explicit reference to $S_0$). We do not expect to find large efficiency gains unless $g(S)$ is approximately proportional to $q(S)$. Whether it is or not on an $N$-dimensional state space depends on the structure of $h$.

There are many possibilities for $g$. We suppose here that the density $g$ arises from sample paths of a GBM with process

$$
dS_i = \bar{\sigma}_i S_i \, dt + \bar{\sigma}_i S_i \, dz_i
$$

but this is not a necessary restriction. We allow both the drift and the volatility to be time-dependent and indeed to depend upon $S_i$. In discrete time equation (22.30) is evolved as

$$
S_{i+1} = S_i \exp \left( \left( \bar{\tau}_i - \frac{1}{2} \bar{\sigma}_i^2 \right) \Delta t + \bar{\sigma}_i (w_{i+1} - w_i) \right), \quad i = 0, \ldots, N-1,
$$

where $\bar{\tau}_i = \bar{\tau}_{i+1}$, $\bar{\sigma}_i = \bar{\sigma}_k$, and $w_{i+1} = w_{i+1}$ is a discretely sampled Wiener process.

Allowing $\bar{\tau}_i$ and $\bar{\sigma}_i$ to depend upon $S_i$ in some way, related to the payoff function $h$, is a very powerful idea. Glasserman (2004) discusses a number of cases, including the selection of $g$ for barrier options and average rate options. We do not pursue this in detail, describing only two generic possibilities for $g$. We let $g$ arise from the process (22.30) with $\{ (\bar{\tau}_i, \bar{\sigma}_i) \}_{i=0,\ldots,N-1}$ specified in particularly simple ways:

1. A constant GBM density. $(\bar{\tau}_i, \bar{\sigma}_i) \equiv (\bar{\tau}, \bar{\sigma})$ constant.
2. A terminally modified density. $(\bar{\tau}_i, \bar{\sigma}_i) \equiv (r, \sigma)$ for $i = 0, \ldots, N-2$, $(\bar{\tau}_{N-1}, \bar{\sigma}_{N-1}) \neq (r, \sigma)$.

\footnote{The time step $\Delta t$ does not have to be constant. We drop the $i$ subscript on $\Delta t$ for simplicity.}
Each case is considered in turn. The constant GBM density turns out to work perhaps surprisingly well for average rate options, although the efficiency gains are not spectacular.

The terminally modified density depends on $SN$ and $SN-1$; since it does not depend on $Si$ at other times it is not suitable for fully path-dependent options, and since it depends on $SN-1$ as well as $SN$ its variance is too high for it to be useful for vanilla calls maturing at time $tN$.

22.3.1 The constant GBM density

When $g$ is a density function of $(S_1, \ldots, S_N)$ arising from the GBM (22.30) with constant coefficients, $(\bar{r}_i, \bar{\sigma}_i) \equiv (\bar{r}, \bar{\sigma}) \neq (r, \sigma)$, particular problems can arise. Once $N$ becomes moderately large we find that unless $\sigma \sim \sigma$ the importance sampling density $g$ is likely to be very distant from the reference density $f$, even before we try to factor in $h$.

Set

$$\bar{\mu}_i = \ln (S_{i-1}) + \left( \bar{r} - \frac{1}{2} \bar{\sigma}^2 \right) \Delta t.$$  \hspace{1cm} (22.32)

The importance sampled Monte Carlo estimate is

$$\tilde{c} = e^{-r(T-t_0)} \frac{1}{M} \sum_{j=1}^{M} h(S^j) \frac{f(S^j | S_0)}{g(S^j | S_0)}$$

$$= e^{-r(T-t_0)} \frac{1}{M} \sum_{j=1}^{M} h(S^j) \exp \left( N \ln \left( \frac{\bar{\sigma}}{\sigma} \right) - \frac{1}{2} \sum_{i=1}^{N} \left[ (e_i^j)^2 - (\bar{\tau}_i^j)^2 \right] \right).$$ \hspace{1cm} (22.33)

where

$$e_i^j = \frac{1}{\sigma \Delta} \left( \ln(S_i^j) - \mu_i \right),$$

$$\tau_i^j = \frac{1}{\sigma \Delta} \left( \ln(S_i^j) - \bar{\mu}_i \right),$$ \hspace{1cm} (22.35)

are normalized increments.

The IS weight is $w^j_f = f(S^j | S_0) / g(S^j | S_0)$,

$$w^j_f = \exp \left( N \ln \left( \frac{\bar{\sigma}}{\sigma} \right) - \frac{1}{2} \sum_{i=1}^{N} \left[ (e_i^j)^2 - (\tau_i^j)^2 \right] \right).$$ \hspace{1cm} (22.37)

We call $w_f$ the constant GBM weight.

If $\bar{\sigma} = \sigma$, the expression for $w_f$ simplifies and becomes

$$w^j_f = \exp \left( \frac{r - \bar{r} - \frac{1}{2} \sigma^2}{\sigma^2} \left( \ln \left( \frac{S_N^j}{S_0} \right) - \left( \hat{r} - \frac{1}{2} \sigma^2 \right) T \right) \right),$$ \hspace{1cm} (22.38)

where $\hat{r} = \frac{1}{T} (\bar{r} + r)$.

The importance sampling weight in equation (22.38) depends only upon the initial and terminal values of $S$. It is significantly cheaper to compute than equation (22.37). In practice when using $w_f$ one usually
needs to set \( \bar{\sigma} = \sigma \). When \( \bar{\sigma} \sim \sigma \) and \( N \) is more than single figures, say, the density of \( g \) becomes skewed relative to that of \( f \) – unfortunately in directions that do not bring it closer to \( q \) – and \( w_J \) is not an effective weight.

22.3.2 The terminally modified density

When \( h \) depends only on \( S_N \) then an obvious idea is to throw away all of the sample path apart from the point \( S_N \), as if \( S_N \) were generated from \( S_0 \) in a single step. One sets

\[
c = e^{-r(T-t_0)} \frac{1}{M} \sum_{j=1}^{M} h(S_N^j) \frac{f(S_N^j \mid S_0)}{g(S_N^j \mid S_0)}
\]

(22.39)

\[
= e^{-r(T-t_0)} \frac{1}{M} \sum_{j=1}^{M} h(S_N^j) w_{CT}^j
\]

(22.40)

where

\[
w_{CT}^j = \frac{f(S_N^j \mid S_0)}{g(S_N^j \mid S_0)}
\]

(22.41)

\[
= \exp \left( \ln \left( \frac{\bar{\sigma}}{\sigma} \right) - \frac{1}{2} \left( \left( e^j \right)^2 - \left( \bar{\sigma}^j \right)^2 \right) \right)
\]

(22.42)

for

\[
e^j = \frac{1}{\sigma \sqrt{T}} \ln(S_N^j) - \mu, \quad \mu = \ln(S_0) + (r - \frac{1}{2} \sigma^2) T,
\]

\[
\bar{\sigma}^j = \frac{1}{\bar{\sigma} \sqrt{T}} \ln(S_N^j) - \bar{\mu}, \quad \bar{\mu} = \ln(S_0) + (\bar{r} - \frac{1}{2} \bar{\sigma}^2) T.
\]

(22.43)

We call \( w_{CT} \) the co-terminal weight. Since \( w_{CT} \) does not depend on points that do not contribute to the value of \( c \) one expects it to work well, and this is supported in part by the numerical results in section 22.5.

If \( h \) depends only on a subset of observations along the generated sample path then a similar idea applies. Suppose \( h \) depends on \( \{S_{i_k}\}_{k=1}^{K} \), with \( K < N \) and \( i_{k-1} < i_k, k = 2, \ldots, K \). Then compute the integral as

\[
c = e^{-r(T-t_0)} \frac{1}{M} \sum_{j=1}^{M} h(S_{i_1}^j, \ldots, S_{i_k}^j) \frac{f(S_{i_1}^j, \ldots, S_{i_k}^j \mid S_0)}{g(S_{i_1}^j, \ldots, S_{i_k}^j \mid S_0)}.
\]

(22.44)

When \( h \) depends on the entire sample path this approach has no advantages, but a related weight can work if the value of \( h \) is determined chiefly by \( S_N \) and \( S_{N-1} \). When \( (\bar{r}, \bar{\sigma}) = (r, \sigma) \) for \( i = 0, \ldots, N-2 \) equation (22.37) simplifies. All terms in the sum apart from the \( N \)th cancel out, so that the weight, \( w_T \), becomes

\[
w_T^j = \exp \left( \ln \left( \frac{\bar{\sigma}}{\sigma} \right) - \frac{1}{2} \left( \left( \frac{\ln(S_N^j) - \mu_N}{\sigma \sqrt{\Delta t}} \right)^2 - \left( \frac{\ln(S_N^j) - \bar{\mu}_N}{\bar{\sigma} \sqrt{\Delta t}} \right)^2 \right) \right)
\]

(22.45)

We call \( w_T \) the terminally modified weight. If \( c_t \) is an OTM vanilla European call with strike \( X \) one may choose \( (\bar{r}, \bar{\sigma}) = (\bar{r}_N, \bar{\sigma}_N) \) as functions of \( S_{N-1} \) to place \( S_N \) above the level \( X \). For \( S_N \)
to have mean \( k_r X, k_r > 1 \), one sets

\[
\tau = \frac{1}{\Delta t} \ln \left( \frac{k_r X}{S_{N-1}} \right). 
\] (22.46)

\( \sigma \) can be chosen so that \( S_N \) has mass above \( X \), for instance, by setting

\[
\bar{\sigma} \sqrt{\Delta t} \sim k_\sigma \ln (k_r) 
\] (22.47)

for some \( k_\sigma \sim 1 \).

With adaptive values of \((\tau_N, \bar{\sigma}_N)\) the weight \( w_T \) and its extensions may have potential uses, perhaps for options paying off on an average computed from a averaging period towards the end of their life, but we explore this no further.

### 22.4 IMPLEMENTING IMPORTANCE SAMPLING

How should importance sampling be implemented? As with CVs, should a Monte Carlo method always evolve a bank of IS processes for options to pick from as they choose?

We find in section 22.5 that constant GBM weights work reasonably well for average rate options, with \( \tau = \sigma \) and across a range of \( \tau \). It would be possible to evolve a relatively small bank of standard constant GBM processes that a book of OTM average rate options, with a variety of strikes, could choose from. This would avoid duplication while largely retaining a worthwhile level of efficiency gain.

However, we see in section 22.5 that in general the efficiency gain attained by importance sampling is very sensitive to the parameters chosen for the IS process. It is likely to be the case that a standard set of IS processes would not be sufficiently close to a sufficient number of options in the book to justify their overhead. If this is so, then it might be better to:

1. hive off the extreme options and others that could benefit from IS into a separate book with tailored IS;
2. countenance potential duplication by enabling each extreme option to specify separately its own IS process.

The cost of this could be great. If a large book can be valued off a single set of sample paths, then the additional cost of a further set of sample paths could result in a significant increase in the total cost.

The approach taken in the implementation described here is to allow each option to register its own individual IS density in a bank. The principles of the implementation are:

1. There is a separate IS path generator object, conforming to the interface \( \text{IISGenerator} \), for each option requesting importance sampling.
2. There is a special object, \( \text{ISmanager} \), that manages both the reference \( \text{IGenerator} \) object and also all the individual \( \text{IISGenerator} \) objects.
3. Individual options are supposed to know their own IS specifications. That is, they are able to supply a specification of an \( \text{IISGenerator} \) object that can generate suitable IS sample paths and weights for them.

We refer collectively to the reference \( \text{IGenerator} \) object and the set of \( \text{IISGenerator} \) objects as the path generators in the application.

To enable options to specify their IS path generators a UDT, \( \text{ISspec} \), is defined. It has as one of its members an \( \text{Enum} \), \( \text{ISdensity} \). \( \text{ISspec} \) and \( \text{ISdensity} \) are shown in Figure 22.1. The \( \text{Enum} \) specifies
what sort of IS density is being used; the rest of the data members in ISspec are parameters for the density. Using an Enum is convenient, but it is a confession that the application is not fully polymorphic.

An importance sampling Monte Carlo method is given in the spreadsheet MC_AverageRate_IS.xls. We shall look at each new and modified object in turn starting with the application object.

The application object

The new application object, App_MC_IS, is displayed in Figure 22.2. Its innovative feature is the presence of a path manager object, ism_, of type ISmanager, that organizes the various path generator objects in the application: the reference IGenerator object (here always of type PathGeneratorGBM) that generates paths under the pricing measure, and a set of IISGenerator objects that generate paths and weights under IS specifications provided by option objects. Options set up the ISmanager to enable it to generate IS paths for them, through the ISmanager::SetISlink() method on line 22.2g in App_MC_IS::SetValues(). We return to this later.

When App_MC_IS::Run() is executing, and it is iterating through each sample path, it first asks the Wiener path generator object for a Wiener sample path, as usual, on line 22.2a. This is now passed to ism_ on line 22.2b where it is used to update the path generators that ism_ manages. A path is requested from ism_, using the ism_.GetPath() method, on line 22.2c. An option is passed as an argument to GetPath() to enable ism_ to determine which path to return.

After the option computes its payoff from the path that ism_ gives it, the value is passed back to ism_ to be transformed to a value under the pricing measure; again, the option object is passed alongside the payoff value so that ism_ can determine which transformation to apply.

The ISmanager

The ISmanager object is shown in Figures 22.3 and 22.4. Its job is to manage a set of path generators. The reference IGenerator is maintained as the Private data member gen_, and the IS path generators in an array of IISGenerators, Gens_.

Options register with the ISmanager object by calling the SetISlink() method. If importance sampling is requested the ISmanager asks the option for a specification of an IISGenerator object in the form of a UDT of type ISspec. This is partially set up on line 22.3c by the reference IGenerator object, gen_, which knows about numbers and lengths of time steps, and then filled with process data – the density type, the drift, volatility and initial value, and the strike that could be used by an adaptive weight – by the
Private out_ As OutputCounter
Private wie_ As WienerGeneratorPlain
Private acc_ As IAccumulator 'accumulator for IS MC
Private ism_ As ISmanager 'IS path manager
Private pay_ As IPayoff
Private M_ As Long 'number of paths
Private N_ As Long 'number of time steps

Private Sub Class_Initialize()
Set out_ = New OutputCounter
Set wie_ = New WienerGeneratorPlain
Set ism_ = New ISmanager
End Sub

Private Sub Class_Terminate()
Set wie_ = Nothing
Set acc_ = Nothing
Set pay_ = Nothing
Set ism_ = Nothing
Set out_ = Nothing
End Sub

Friend Sub SetValues(ByRef data As InputManager)
Call out_.SetValues(data)
Call wie_.SetValues(data)
Call ism_.SetValues(data)
Set acc_ = data.GetAccumulator
Set pay_ = data.GetOption
Call ism_.SetISlink(pay_) 'g. Create appropriate IS object
M_ = data.M
N_ = data.N
End Sub

Friend Property Get val() As Double: val = acc_.val: End Property
Friend Property Get se() As Double: se = acc_.se: End Property

Friend Sub run()
Dim path() As Double: ReDim path(0 To N_) As Double 'Stock values
Dim WieVec() As Double: ReDim WieVec(0 To N_) As Double 'Wiener path
Dim val As Double
Dim i As Long
For i = 1 To M_ 'For each sample path
Call out_.OutputCounter(i)
Call wie_.GetWienerVec(WieVec) 'a. Wiener path
Call ism_.SetPath(WieVec, i) 'b. Give the path to ISmanager
Call ism_.GetPath(i, path, pay_) 'c. Get a path under IS for pay_
val = pay_.payoff(path) 'd. Compute payoff
Call ism_.IStransform(i, val, pay_) 'e. Rotate it back
Call acc_.update(val) 'f. Accumulate
Next i
End Sub

Figure 22.2 Monte Carlo with importance sampling: the application object, App_MC_IS
Private gen_ As IGenerator 'Non-IS reference Pathgenerator
Private Gens_() As IISGenerator 'Array of IS path generators
Private genSpec_ As ISspec 'spec of gen_
Private K_ As Long 'length of Gens_()
Private PathID_ As Long 'ref to current path
Private path_() As Double 'the current non-IS path

Private Sub Class_Initialize()
K_ = 0
PathID_ = 0
End Sub

Private Sub Class_Terminate()
Set gen_ = Nothing
If K_ = 0 Then Exit Sub
Dim K As Long
For K = 1 To K_
  Set Gens_(K) = Nothing
Next K
End Sub

Friend Sub SetValues(ByRef data As InputManager)
Set gen_ = data.GetPathGenerator 'a
Call gen_.GetSpec(genSpec_) 'b
End Sub

Friend Sub SetISlink(pay As IPayoff)
If Not pay.DoIS Then Exit Sub 'c
Dim spec As ISspec: Call gen_.GetSpec(spec)
Call pay.GetISspec(spec): Call AddIS(spec) 'd
pay.ISref = K_
End Sub

Friend Sub SetPath(wpath() As Double, i As Long)
PathID_ = i
Call gen_.GetPath(wpath, path_) 'e
If K_ = 0 Then Exit Sub
Dim K As Long
For K = 1 To K_
  Call Gens_(K).SetPath(wpath)
Next K 'f
End Sub

Friend Sub GetPath(i As Long, path() As Double, opt As IPayoff)
Call CheckI(i)
If opt.DoIS Then
  Dim ISref As Long: ISref = opt.ISref
  Call Gens_(ISref).WithPath(path) 'g
Else
  path = path_
End If
End Sub

Friend Sub IStransform(i As Long, ByRef val As Double, opt As IPayoff)
Call CheckI(i)
If Not opt.DoIS Then Exit Sub 'val is unchanged
Dim ISref As Long: ISref = opt.ISref
val = val * Gens_(ISref).GetWeight 'j
End Sub

Figure 22.3 The ISmanager object: structural and interface methods
option on line 22.3d. The ISmanager creates a New IISGenerator object and appends it to the end of Gens_ (in the Private method AddIS() in Figure 22.4). A handle to the newly created IISGenerator, in the form of its index in Gens_, is passed back to the option.

ISmanager has its own factory method, GetIISGenerator() (Figure 22.4), that manufactures IISGenerator objects according to the specification passed to it as an argument. Its location here in ISmanager is problematical. It has a knowledge of every type of IISGenerator conforming object, coupling it not only to those objects but also to the options that specify them. This factory method would not be present in a fully polymorphic application.

Each IISGenerator object is set up on line 22.41 with both the specification coming from the option, spec, and the specification of the reference IGenerator, genSpec_. This endows it with all the information it needs to generate not only IS sample paths but also the associated weights.

When a Wiener sample path is passed to the ISmanager object, in its SetPath() method, the path is passed on to each composited path generator object; to the reference IGenerator on line 22.3e and to the IISGenerator objects on line 22.3f. On line 22.3e the ISmanager object obtains from the reference PathGeneratorGBM gen_ the current plain path. This is kept in the array path_ to be passed to any object that does not require an IS path.

When subsequently the ISmanager object receives a request for a path, by having its GetPath() method called, the option object is asked for its handle on line 22.3g. This identifies the IS path generator in Gens_ that the ISmanager object returns the path from on line 22.3h. Later, when an IS weight is needed upon a call of IStranform(), the option is again asked for the handle on line 22.3i so that the correct IS path generator is asked to supply the IS weight on line 22.3j.

The IISGenerator objects

IS sample paths and weights are returned by objects satisfying the IISGenerator interface (Figure 22.5). There are: two methods enabling an IISGenerator object to be set up, SetValues() and SetValuesSpec(); a getter and setter pair for the sample path, SetPath() and GetPath(); and a getter for the IS weight, GetWeight(). SetValuesSpec() is the main set-up method; SetValues()
is present essentially only for consistency. SetValuesSpec() is called by the ISmanager’s method, ISmanager::GetISGenerator(). SetPath() is used to pass a Wiener sample path to the object so that it can generate an IS sample path from it. GetPath() returns the IS sample path.

An example of an IISGenerator conforming object is ISgeneratorCoterminal, shown in Figure 22.6. This computes the co-terminal weight defined in equation (22.41). For convenience it uses an IGenerator object, gen_, to generate GBM sample paths, and an object of type IISweight, wgt_, to compute the weight itself.

The SetValues() method is a stub. The object is set up when ISmanager::GetISGenerator() calls SetValuesSpec(). The first ISspec argument, spec, is used to set up gen_, on line 22.6a; both arguments are needed to set up wgt_ on line 22.6b.
When a Wiener sample path is received, SetPath() asks the composite IGenerator object, gen_, on line 22.6c, to construct and return an IS sample path. The IISweight object, wgt_, is then requested to compute a weight from it.

An example of an IISweight object is ISweightCoTerminal, displayed in Figure 22.7. It computes equation (22.41) for a sample path supplied by its GetWeight() method. Another example is ISweightJoint (Figure 22.8) that implements equations (22.37) and (22.38). GetWeight() computes and returns the IS weight, either from equation (22.37) or from equation (22.38). When $\bar{\sigma} = \sigma$ the Private method SigConstCase() returns an evaluation of equation (22.38), otherwise SigDiffCase() evaluates and returns the result of equation (22.37).

**The modified PathGeneratorGBM object**

Figure 22.9 displays the PathGeneratorGBM object. There is a single change to it from previous versions. A PathGeneratorGBM object can now be set up in one of two ways: either as before by calling its SetValues() method, taking the InputManager as an argument, or now by calling a new method, ...

```plaintext
'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
'XX ISweightCoTerminal
'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 Implements IISweight
 Private PLdriftT_ As Double
 Private PLsgrtT_ As Double
 Private driftT_ As Double
 Private sigrtT_ As Double
 Private d_sig_ As Double
 Private N_ As Long
'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Friend Sub IISweight_SetValues(ByRef data As InputManager): End Sub 'stub
Friend Sub IISweight_SetValuesSpec(spec As ISspec, plspec As ISspec)
   N_ = plspec.N
   Dim T As Double: T = N_ * plspec.dt
   Dim PLsig As Double: PLsig = plspec.sig
   Dim PLr As Double: PLr = plspec.r
   Dim sig As Double: sig = spec.sig
   Dim r As Double: r = spec.r
   d_sig_ = Log(sig / PLsig)
   PLdriftT_ = (PLr - 0.5 * PLsig * PLsig) * T
   PLsgrtT_ = PLsig * Sqr(T)
   driftT_ = (r - 0.5 * sig * sig) * T
   sigrtT_ = sig * Sqr(T)
End Sub
'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Friend Function IISweight_GetWeight(path() As Double) As Double
   Dim Rt As Double: Rt = Log(path(N_) / path(0))
   Dim d As Double: d = (Rt - driftT_) / sigrtT_
   Dim d_PL As Double: d_PL = (Rt - PLdriftT_) / PLsgrtT_
   IISweight_GetWeight = Exp(d_sig_ + 0.5 * (d * d - d_PL * d_PL))
End Function
'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
'XX end of file
'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

*Figure 22.7 The ISweightCoTerminal object*
Implements IISweight
Private PLsig_ As Double
Private PLr_ As Double
Private sig_ As Double
Private r_ As Double
Private PLdrift_ As Double
Private PLsgrt_ As Double
Private drift_ As Double
Private sigrt_ As Double
Private N_d_sig_ As Double
Private N_ As Long
Private dt_ As Double

Friend Sub IISweight_SetValues(ByRef data As InputManager): End Sub 'stub
Friend Sub IISweight_SetValuesSpec(spec As ISspec, plspec As ISspec)
    N_ = plspec.N
    dt_ = plspec.dt
    PLsig_ = plspec.sig
    PLr_ = plspec.r
    sig_ = spec.sig
    r_ = spec.r
    N_d_sig_ = N_ * Log(sig_ / PLsig_)
    PLdrift_ = (PLr_ - 0.5 * PLsig_ * PLsig_) * dt_
    PLsgrt_ = PLsig_ * Sqr(dt_)
    drift_ = (r_ - 0.5 * sig_ * sig_) * dt_
    sigrt_ = sig_ * Sqr(dt_)
End Sub

Friend Function IISweight_GetWeight(path() As Double) As Double
    If PLsig_ = sig_ Then
        IISweight_GetWeight = SigConstCase(path)
    Else
        IISweight_GetWeight = SigDiffCase(path)
    End If
End Function

Private Function SigDiffCase(path() As Double) As Double
    Dim sum As Double: sum = 0#
    Dim i As Long
    For i = 1 To N_
        Dim Rt As Double: Rt = Log(path(i) / path(i - 1))
        Dim d As Double: d = (Rt - drift_) / sigrt_
        Dim d_PL As Double: d_PL = (Rt - PLdrift_) / PLsgrt_
        sum = sum + d * d - d_PL * d_PL
    Next i
    SigDiffCase = Exp(N_d_sig_ + 0.5 * sum)
End Function

Private Function SigConstCase(path() As Double) As Double
    Dim var As Double: var = sig_ * sig_
    Dim fact1 As Double: fact1 = (r_ - PLr_) / var
    Dim fact2 As Double: fact2 = Log(path(N_) / path(0))
    Dim fact3 As Double: fact3 = -0.5 * N_ * dt_ * (r_ + PLr_ - var)
    SigConstCase = Exp(-fact1 * (fact2 + fact3))
End Function

End Sub

End Function

Private Function SigConstCase(path() As Double) As Double
    Dim var As Double: var = sig_ * sig_
    Dim fact1 As Double: fact1 = (r_ - PLr_) / var
    Dim fact2 As Double: fact2 = Log(path(N_) / path(0))
    Dim fact3 As Double: fact3 = -0.5 * N_ * dt_ * (r_ + PLr_ - var)
    SigConstCase = Exp(-fact1 * (fact2 + fact3))
End Function

End Function

'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
'XX end of file
'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

Figure 22.8 The ISWeightJoint object
Figure 22.9 The PathGeneratorGBM object

SetValuesSpec(), taking a parameter array as an argument. The second method has been introduced to enable option objects to be able to specify how an IS PathGeneratorGBM object, composited within an IISGenerator object, should be set up.

The ISspec UDT supplied as an argument to SetValuesSpec() contains everything a GBM PathGeneratorGBM object needs to know about itself (and more besides). The specification includes process parameters, \( \{S_0, r, \sigma\} \), and numerical parameters, \{N, \Delta t\}. A better design would replace the UDT with an object created specifically to carry and manage the parameter data.

The new option object

The option object has to be modified to enable it to specify an IS path generator. Figure 22.10 shows the PayoffCall object. It holds a UDT, ISspec_, that contains parameter values for a path generator, of a
type specified by the ISdensity Enum, that the option would like to have to generate its IS sample paths. There is a Sub, GetISSpec(), that gets ISSpec_.

In this implementation the ISSpec is taken directly from the front end by the InputManager. This is done to enable the application to operate flexibly, using different weights with different options. In real life an option would itself specify its ISspec.

The option holds the handle supplied by the ISmanager in ISref_. This is a Long set by the ISmanager with a call to the ISref setter and later requested by the ISmanager using the ISref setter.

The final IS-related data member is a Boolean, IS_, with value True if the option requests importance sampling. This has an associated getter called by the ISmanager.

In this implementation only one type of reference path generator and one type of IS path generator, both GBMs, exist. However, there is no reason why further types of process should not be allowed. The UDT ISspec could be extended easily to include more general processes. This could be incorporated with little additional work and no change to the structure of the application.
22.5 NUMERICAL ASSESSMENT

We value three options at two different levels of strike. The options are:

1. A vanilla European call.
2. The exotic quadratic payoff option (described in Chapter 20).
3. The benchmark average rate option.

The strikes are set so that the options are either ATM or considerably OTM. For the average rate option three cases are investigated, with \( N = 4 \), \( N = 16 \) and \( N = 64 \). In all cases we take \( S_0 = 100 \), \( r = 0.05 \), \( \sigma = 0.2 \) and \( T = 1 \). The ATM and OTM strikes for the vanilla call and the average rate option are set at \( X = 100 \) and \( X = 150 \). The ATM and OTM specification for the exotic option are discussed below in the relevant sections.

This gives a reasonable set of test cases. The average rate option is path dependent; the exotic option has payoffs only in a narrow range; the vanilla call is important in the market.

Each option is valued by IS using either co-terminal (where applicable) weights with \( N = 1 \) or constant GBM weights with \( N > 1 \). We find that the co-terminal weight works quite well for the non-path-dependent options, and very well indeed when the option is OTM. It cannot be used with the average rate option, but constant GBM weights can be. Constant GBM weights do not work very well for ATM options, even if they are path dependent. However for OTM options, constant GBM weights can produce efficiency gains, although not as great for non-path-dependent options as those obtainable with co-terminal weights.

The tables in this section present efficiency gains compared with a plain Monte Carlo with the same number of time steps. Individual standard errors and times are not given. Times are constant (within noise) for each table and are summarized with each table. As no bias is displayed for any of the options, no option prices are given. The times in seconds taken for the plain and IS Monte Carlo are denoted by \( t_{PL} \) and \( t_{IS} \), respectively, and the standard error of the plain Monte Carlo is denoted by \( se_{PL} \).

Efficiency gains are given to at most 1 decimal place. A gain of 0.0 indicates a very bad result: an efficiency loss of at least a factor of 20. Gains are subject to computational noise; Glasserman (2004) explains how this is influenced by the skewness of the weights distribution. Timings are computed in the spreadsheet MC_AverageRate_IS_timings.xls.

22.5.1 ATM options

The weights we investigate do not perform particularly well for ATM options. Some efficiency gains are possible but these are not exceptional. For non-path-dependent options co-terminal weights give much better results than constant GBM weights. Even for the average rate options the gains found with constant GBM weights are ‘modest’.

The vanilla call

A plain call benefits a little from IS even when it is ATM. Panel (a) in Table 22.1 gives results for co-terminal weights, implemented with a single time step. Panel (c) displays more detail in the region of the optimal parameter values.

The payoff weighted density, \( q \), is best matched with \( (\overline{r}^*, \overline{\sigma}^*) \sim (0.27, 0.15) \) (panel (c)) with a gain of \( \sim 18 \). The results are not so sensitive to the choice of \( (\overline{r}, \overline{\sigma}) \) although getting too far away from \( (\overline{r}^*, \overline{\sigma}^*) \)
The exotic option

The payoff function is \( h(S) = (X_1 - (S - S_2)^2)^+ \) (see Chapter 20); this example uses \((X_1, X_2) = (100, 110)\). The option pays off when \( S_1 \in [100, 120] \) so that it is ATM.

Results are given in Table 22.2. Panel (a) gives efficiency gains on a course mesh, panel (b) in an area close to the optimal parameter values.

Results are very sensitive to the IS parameter choice. Optimal values are \((\bar{\sigma}, \bar{\sigma}^*) \sim (0.09, 0.045)\) with gains of \(\sim 24\). Gains fall off sharply away from the optimal parameter values. Efficiency gains are closely linked to the geometry of the payoff region. Only when the IS density is concentrated over the interval \([100, 120]\) of non-zero payoff, with \(\bar{\sigma} \sim 0.1\) and \(\bar{\sigma}^* \sim 0.05\), is there a reasonable chance of a decent efficiency gain.

Results for \(N = 64\) constant GBM weight are not worth reproducing in a table. They are at best around 0.9 for \(\bar{\sigma} = \sigma\) and \(\bar{\sigma}\) in a fairly broad band around 0.1. The constant GBM weight density cannot match \(q\) for this option.

---

<table>
<thead>
<tr>
<th>Table 22.1</th>
<th>Efficiency gains, ATM vanilla call option: Co-terminal and constant GBM weights</th>
</tr>
</thead>
<tbody>
<tr>
<td>(N = 1)</td>
<td>(0.1) (0.15) (0.2) (0.25)</td>
</tr>
<tr>
<td>(\bar{\sigma})</td>
<td>(0.20) (0.0) (3.0) (3.6) (2.3)</td>
</tr>
<tr>
<td>(T)</td>
<td>(0.30) (0.7) (14.1) (5.6) (2.9)</td>
</tr>
<tr>
<td>(N = 64)</td>
<td>(0.19) (0.20) (0.21)</td>
</tr>
<tr>
<td>(\bar{\sigma})</td>
<td>(0.20) (1.6) (5.0) (1.8)</td>
</tr>
<tr>
<td>(T)</td>
<td>(0.30) (2.0) (7.7) (2.2)</td>
</tr>
</tbody>
</table>

(a) Co-terminal, \(N = 1\).
\((t^{PL}, se_{PL}) = (0.7, 0.046), t_{IS} = 1.1\).

(b) Constant GBM, \(N = 64\).
\((t^{PL}, se_{PL}) = (9.8, 0.047),
\((t^{opt}_{IS}, t^{\bar{\sigma}}_{IS}) = (11.6, 13.7)\).

(c) Co-terminal, \(N = 1\).
\((t^{PL}, se_{PL}) = (0.7, 0.046),
\(t_{IS} = 1.1\).
22.5.2 OTM options

IS works much better for OTM options. It is easy to change a drift parameter to ensure that more sample paths end in the money. Exceptionally large gains may sometimes be possible although, in general, it is difficult to match an option \( q \) with the densities we use here.

The vanilla call

Table 22.4 gives results for the OTM vanilla call option. Panels (a) and (c) use co-terminal weights, panel (b) constant GBM weights.
Table 22.4  Efficiency gains, OTM vanilla call option: Co-terminal and constant GBM weights

<table>
<thead>
<tr>
<th>$N = 1$</th>
<th>$\sigma$</th>
<th>$N = 64$</th>
<th>$\sigma$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.08 0.10 0.12</td>
<td>0.19 0.20 0.21</td>
<td></td>
</tr>
<tr>
<td>$\bar{\tau}$</td>
<td>0.45 3 17 34</td>
<td>0.45 22 45 23</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.50 12 79 101</td>
<td>0.50 26 55 26</td>
<td></td>
</tr>
<tr>
<td>$\tau$</td>
<td>0.55 110 210 148</td>
<td>0.55 28 59 28</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.60 43 75 77</td>
<td>0.60 25 54 26</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.65 9 21 33</td>
<td>0.65 21 44 23</td>
<td></td>
</tr>
</tbody>
</table>

(a) Co-terminal, $N = 1$.  
$(t_{PL}, s_{ePL}) = (0.7, 0.0088)$,  
$t_{IS} = 1.1$.

(b) Constant GBM, $N = 64$.  
$(t_{PL}, s_{ePL}) = (9.8, 0.0088)$,  
$(t_{IS}, t_{IS}^{\sigma}) = (11.6, 13.7)$.

Table 22.5  Efficiency gains, quadratic payoff option, OTM, co-terminal and constant GBM weights

<table>
<thead>
<tr>
<th>$N = 1$</th>
<th>$\sigma$</th>
<th>$N = 64$</th>
<th>$\sigma$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.05 0.10 0.15</td>
<td>0.19 0.20 0.21</td>
<td></td>
</tr>
<tr>
<td>$\bar{\tau}$</td>
<td>0.30 1.3 5.2 4.5</td>
<td>0.30 2.5 4.5 2.6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.35 12.7 8.7 5.8</td>
<td>0.35 3.0 5.2 3.0</td>
<td></td>
</tr>
<tr>
<td>$\tau$</td>
<td>0.40 47.4 10.7 6.2</td>
<td>0.40 3.2 5.5 3.1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.45 8.6 9.1 5.8</td>
<td>0.45 2.9 5.4 3.1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.50 0.8 5.1 4.5</td>
<td>0.50 2.9 4.8 2.8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.55 0.0 2.2 3.3</td>
<td>0.55 2.4 4.1 2.5</td>
<td></td>
</tr>
</tbody>
</table>

(a) Co-terminal, $N = 1$.  
$(t_{PL}, s_{ePL}) = (0.8, 0.047)$,  
$t_{IS} = 1.2$.

(b) Constant GBM, $N = 64$.  
$(t_{PL}, s_{ePL}) = (9.8, 0.047)$,  
$(t_{IS}, t_{IS}^{\sigma}) = (11.7, 13.7)$.

Table 22.4  Efficiency gains, quadratic payoff option, OTM, co-terminal and constant GBM weights

<table>
<thead>
<tr>
<th>$N = 1$</th>
<th>$\sigma$</th>
<th>$N = 64$</th>
<th>$\sigma$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.027 0.028 0.029 0.030 0.031 0.032 0.033 0.034 0.035</td>
<td>0.19 0.20 0.21</td>
<td></td>
</tr>
<tr>
<td>$\bar{\tau}$</td>
<td>0.390 83 101 121 143 143 160 157 147 139</td>
<td>0.39 8.3 10.1 12.1 14.3 14.3 16.0 15.7 14.7 13.9</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.395 138 171 196 215 232 218 199 190 165</td>
<td>0.35 9.8 12.1 14.3 16.0 15.7 14.7 13.9</td>
<td></td>
</tr>
<tr>
<td>$\tau$</td>
<td>0.400 112 133 159 174 177 167 171 164 148</td>
<td>0.40 12.1 14.3 16.0 15.7 14.7 13.9</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.405 58 66 74 80 88 92 95 95 93</td>
<td>0.45 17.4 20.5 22.7 24.9 26.1 27.3 28.5 29.7 30.9</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.410 29 34 39 45 50 51 56 59 60</td>
<td>0.45 17.4 20.5 22.7 24.9 26.1 27.3 28.5 29.7 30.9</td>
<td></td>
</tr>
</tbody>
</table>

(c) Co-terminal, $N = 1$: gains on a refined mesh.  
$(t_{PL}, s_{ePL}) = (0.8, 0.047)$, $t_{IS} = 1.2$.  
{(a) Co-terminal, $N = 1$.  
$(t_{PL}, s_{ePL}) = (0.7, 0.0088)$,  
$t_{IS} = 1.1$.}
For OTM call options large gains are possible. Optimal values for co-terminal weights are \((\bar{\sigma}^*, \sigma^*) \sim (0.55, 0.10)\) with a gain of over 200. However, small variations from optimal parameter values can reduce the gain considerably. Using \((\bar{\sigma}, \sigma) \sim (0.50, 0.08)\), for example, gives a gain only about one-twentieth the size.

For constant GBM weights with \(N = 64\) smaller gains are found; although good, they do not approach those found using co-terminal weights. Setting \(\tilde{\sigma} = \sigma\), gains are stable in the region of 50 ~ 60 for the range of \(\bar{\sigma}\) given in the tables. As soon as \(\tilde{\sigma}\) moves away from \(\sigma\), the gains reduce very significantly.

The exotic option

For the OTM case we take \((X_1, X_2) = (100, 150)\). This has a positive payoff only when \(S_1 \in [140, 160]\).

Table 22.5 summarizes results for co-terminal weights (panel (a)) and constant GBM weights (panel (b)). Panel (c) gives more detailed results for the co-terminal case around the optimal parameter values.

The greatest gains in the co-terminal case are around 230 with \((\bar{\sigma}^*, \sigma^*) \sim (0.395, 0.031)\). These parameters place the IS distribution right over the payoff region. The gains are very sensitive to small variations in \((\bar{\sigma}, \sigma)\). With \((\bar{\sigma}, \sigma) = (0.41, 0.029)\) the gains are only about one-sixth of the best available in the table.

For constant GBM weights, with \(N = 64\), some gains are possible, despite the poor choice of IS distribution, but this is only because it brings the mean of the distribution in line with the payoff region, \([140, 160]\). Since it is not possible to change the value of \(\tilde{\sigma}\) without destroying the fit to the \(n\)-dimensional reference density, we have to endure the best fit that a skewed distribution can achieve. There is no scope to find better efficiency gains.

The average rate option

Table 22.6 displays results. Panels (a) and (d) are for \(N = 4\), panels (b) and (e) for \(N = 16\), and panel (c) for \(N = 64\).

Gains are high for all three options even if \(\tilde{\sigma} \neq \sigma\). The payoff depends on every \(S_i\) so it seems that constant GBM weights are better able to match \(q\) here than elsewhere. For \(N = 4\) the greatest gain is \(\sim 90\) with the fairly large values \((\bar{\sigma}^*, \sigma^*) \sim (0.60, 0.24)\); for \(N = 16\) it is \(\sim 115\) with \((\bar{\sigma}^*, \sigma^*) \sim (0.60, 0.24)\); and for \(N = 64\) it is \(\sim 80\) with \((\bar{\sigma}^*, \sigma^*) \sim (0.55, 0.20)\).

Constant GBM performs better for this option with \(N = 16\) than in other cases. Even when \(N = 64\) the results are not too bad. The payoff function does not enable \(q\) to match \(g\) very well, so we need to have \(\tilde{\sigma} \sim \sigma\) but a reasonable fit is achieved for a wider range of values of \(\tilde{\sigma}\) than was possible in previous examples. The optimal IS parameter values are affected by the reset frequency. \(\bar{\sigma}\) is smaller with more resets, and \(\tilde{\sigma}\) is forced to be closer to \(\sigma\).

The consistency of the gains should not be taken to imply that constant GBM weights work well in this case; it means rather that constant GBM weights underperform by about the same amount everywhere.

22.5.3 Comparison of the weights

Adding in importance sampling increases execution times by up to around 50% when \(N = 1\), proportionately less (perhaps 10 – 20%) for \(N = 64\). Table 22.7 summarizes, informally, how each weight performs for each option. “✓✓” is very good; “✓” is indifferent; “×” means that it does not work at all. \(w_{CT}\) is not applicable to average rate options.

For OTM options IS can work very well – even with these untailored weights. \(w_{CT}\) gives large efficiency gains for non-path-dependent options; \(w_J\) for the path-dependent average rate option. For ATM options
Table 22.6  Efficiency gains, OTM average rate options, constant GBM weights

<table>
<thead>
<tr>
<th></th>
<th>$N=4$</th>
<th>$N=16$</th>
<th>$N=64$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\bar{\sigma}$</td>
<td>$\bar{\sigma}$</td>
<td>$\bar{\sigma}$</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>0.25</td>
<td>0.30</td>
</tr>
<tr>
<td>$\bar{\sigma}$</td>
<td>0.30</td>
<td>0.30</td>
<td>0.30</td>
</tr>
<tr>
<td></td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
</tr>
<tr>
<td></td>
<td>0.30</td>
<td>0.30</td>
<td>0.30</td>
</tr>
<tr>
<td></td>
<td>0.30</td>
<td>0.30</td>
<td>0.30</td>
</tr>
<tr>
<td>$\bar{\sigma}$</td>
<td>0.20</td>
<td>0.20</td>
<td>0.20</td>
</tr>
<tr>
<td></td>
<td>0.21</td>
<td>0.21</td>
<td>0.21</td>
</tr>
<tr>
<td></td>
<td>0.22</td>
<td>0.22</td>
<td>0.22</td>
</tr>
<tr>
<td></td>
<td>0.23</td>
<td>0.23</td>
<td>0.23</td>
</tr>
</tbody>
</table>

(a) Constant GBM, $N=4$.

(b) Constant GBM, $N=16$.

(c) Constant GBM, $N=64$.

(d) Constant GBM, $N=4$: more detail.

(e) Constant GBM, $N=16$: more detail.

$\hat{t}_{IS}, t_{IS}$

$\hat{t}_{IS}, t_{IS}$

(sePL)

(sePL)

(sePL)

$(t_{PL, se_{PL}}) = (1.23, 0.0016), (\bar{t}_{IS}, \bar{t}_{IS}) = (1.7, 1.9)$. 

$(t_{PL, se_{PL}}) = (3.0, 0.00077), (\bar{t}_{IS}, \bar{t}_{IS}) = (3.8, 4.3)$. 

$(t_{PL, se_{PL}}) = (10.0, 0.00053), (\bar{t}_{IS}, \bar{t}_{IS}) = (12.0, 14.0)$. 

$(t_{PL, se_{PL}}) = (1.23, 0.0016), (\bar{t}_{IS}, \bar{t}_{IS}) = (1.7, 1.9)$.
Table 22.7  Comparison of the weights

<table>
<thead>
<tr>
<th>Moneyness</th>
<th>Option</th>
<th>Average rate</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Call</td>
<td>Exotic</td>
</tr>
<tr>
<td>ATM:</td>
<td>$w_{CT}$: ✓</td>
<td>$w_{CT}$: ✓</td>
</tr>
<tr>
<td></td>
<td>$w_J$: ~</td>
<td>$w_J$: ×</td>
</tr>
<tr>
<td>OTM:</td>
<td>$w_{CT}$: ✓✓</td>
<td>$w_{CT}$: ✓✓</td>
</tr>
<tr>
<td></td>
<td>$w_J$: ✓</td>
<td>$w_J$: ~</td>
</tr>
</tbody>
</table>

the efficiency gains are much smaller. $w_{CT}$ provides small gains for non-path-dependent options, but $w_J$ is not really very effective, even for average rate options.

22.6 SUMMARY

Importance sampling applies with greatest immediate gain to options with payoffs that are extreme in some sense. Either they are rare, or else they are almost certain; far in or out of the money options; barrier options either close to the barrier or far from the barrier. Only IS can really help to value the OTM benchmark options. None of the other techniques comes near to achieving its efficiency gains. Unfortunately, each extreme option needs its own set of tailored IS parameters.

The difficulty of using importance sampling in practice with a book of options is that each option requires its own importance sampling distribution; results can be quite sensitive to this. An $\overline{\mathcal{F}}$ of 0.6 might work well for a benchmark average rate option, but it would not work for an OTM put on the same underlying.

A possibility is to simulate a bank of IS distributions and allow each option to specify which, if any, of the bank IS distributions should be used. This strategy could work well for a large book where the marginal cost of simulating a set of processes, for various values of $\overline{\mathcal{F}}$, was small.

A word Glasserman uses in his description of IS specification is ‘delicate’. I concur, as will many who have implemented the method; get the specification even slightly wrong and you might not only lose the gain you expected, you may also unexpectedly gain a loss. The sensitivity of the results to the choice of IS parameter values makes the method hard to use effectively in practice. As time to maturity changes, so does the value of $\overline{S}_T$ and so also do the calibrated parameters of the reference distribution – the interest rate and the implied volatility will not stay constant. Unless there is a way to automate the selection of IS parameters, so that between them the option and the reference distribution can determine them without the need for a lengthy search, using IS will not be effective. It is vital for a serious practical implementation of IS that IS distribution parameters should be determined endogenously.

22.7 EXERCISES

1. Section 22.2 discusses pricing an OTM digital option using importance sampling.

   (a) Implement an importance sampling method for this option.

   (b) Comparing standard errors with and without importance sampling, do they behave in the way suggested in the text?
2. Consider the European options with the following payoffs:

(a) \( H^1(S) = \left( \prod_{i=1}^{N} (S_i - X)^+ \right)^{1/N} \) for \( X < S_0 \);

(b) \( H^2(S) = \prod_{i=1}^{N} 1_{L \leq S_i \leq U} \) for \( L < S_0 < U \);

(c) \( H^3(S) = (S_N - S_{N/2})^+ \) (for \( N \) even).

How well do the standard weights perform with these options? Can you devise importance sampling methods that enhance the available efficiency gains?

3. Consider a vanilla put option with strike \( X < S_0 \). As \( X \) decreases towards zero the put becomes progressively more OTM.

(a) Devise an importance sampling method that values OTM puts.

(b) How well does the method perform as \( X \to 0 \)?

(c) Are there any advantages in this case to using non-GBM-based importance sampling distributions?

4. Find an option which the terminal modified density described in section 22.3.2 might work well with.
This page intentionally left blank
Combining Variance Reduction Methods

In previous chapters we have examined separately a number of variance reduction methods. In this chapter we bring the methods together, implementing them simultaneously, to assess their performance as part of an ensemble of speed-up methods.

We find that, mostly, the methods complement one another. Unfortunately there are circumstances in which they do not, and adding in a variance reduction method can reduce or even remove gains previously obtained from other methods. This is seen chiefly with importance sampling and control variates; a sub-optimal importance sampling distribution can reduce the correlation of an option value with what would otherwise be a good control variate, leading to a failure to fully realize expected efficiency gains.

After a discussion on the simultaneous implementation of control variates and importance sampling in section 23.1, we move on in section 23.2 to briefly discuss a combined application. Numerical results are presented in section 23.3.

The timings in the tables in this chapter are given in seconds.

23.1 COMBINING CV AND IS

Antithetic sampling and stratified sampling can be applied without difficulty to the generation of Wiener sample paths. The chief issue that arises here is how to combine control variates with importance sampling, both theoretically and in practice.

A good control variate owes its effectiveness to the concordance between its values and that of the option, and this will be true for a range of models and parameter values; it is an option-dependent, rather than a model-dependent, virtue. A geometric average rate option supplies a good control variate for an arithmetic average rate option because the payoffs are very similar. As long as its value can be computed explicitly in a model its does not matter exactly what model is used or what its parameter values are, it is almost certain to remain a good control variate.

However one thing that can destroy the efficacy of a CV is if the option is valued under one measure and the CV under another.

Let \( S \) be a sample path for the underlying asset and suppose that a CV can be written in the form

\[
d(S) = h^d(S) - v
\]

where \( h^d \) computes the value of the CV along the path \( S \) and \( v = \mathbb{E}[h^d(S)] \) offsets by the expected value.

All the CVs in Chapter 20 can be put into this form; for a delta CV, \( v \) is zero.

Suppose that \( c \) is a (European) option with discounted payoff function \( h^c \) so that

\[
c = \int_\Omega h^c(S) \, d\mathbb{P}
\]

\( ^1 \) Auxiliary model CVs aside.
under the reference pricing measure $\mathbb{F}$. This is operationalized in a plain Monte Carlo method as

$$
\hat{c}^{PL} = \frac{1}{M} \sum_{j=1}^{M} h^c(S^j)
$$

(23.3)

where $S^j$ is a discrete sample path drawn under (an approximation to) $\mathbb{F}$. With an IS measure $\mathbb{G}$ the IS estimate of $c$ is

$$
\hat{c}^{IS} = \frac{1}{M} \sum_{j=1}^{M} h^c(\hat{S}^j)w(\hat{S}^j)
$$

(23.4)

where, as in Chapter 22, $\hat{S}$ is a sample path drawn under $\mathbb{G}$ and $w$ is the IS weight.

A very big mistake would be to use $d$ as a CV for $\hat{c}^{IS}$. They are computed under different measures. Instead one must use $d^{IS}$, defined as follows. Under the IS measure one sets

$$
d^{IS}(\hat{S}) = h^d(\hat{S})w(\hat{S}) - v.
$$

(23.5)

$d^{IS}$ is a control variate since $v = \mathbb{E}_G[h^d(\hat{S})w(\hat{S})]$, and since

$$
\text{corr}_G\left(\hat{c}^{IS}, d^{IS}\right) = \text{corr}_G\left(h^c(\hat{S})w(\hat{S}), h^d(\hat{S})w(\hat{S})\right)
$$

(23.6)

$$
\sim \text{corr}_G\left(h^c(\hat{S}), h^d(\hat{S})\right)
$$

(23.7)

$$
\sim \text{corr}_G\left(h^c(S), h^d(S)\right)
$$

(23.8)

$$
= \text{corr}_F\left(\hat{c}^{PL}, d\right)
$$

(23.9)

it may even preserve the efficacy of the CV $d$ from the plain case.

In the examples below we find that $d^{IS}$, the CV $d$ under IS, often remains a good CV, but its effectiveness may be reduced.

### 23.2 IMPLEMENTING VARIANCE REDUCTION METHODS IN COMBINATION

This section looks first at an application combining all four variance reduction methods, focusing mainly on the interaction between CVs and IS. It then briefly comments on extending the application to value a book of options.

#### 23.2.1 The combined application

The application constructed here uses the Wiener generator objects from Chapters 17, 18 and 19, the CV objects from Chapters 20 and 21, and the IS objects from Chapter 22. The combined application in MS_speedups_all.xls.

The changes to the application arise mainly as a result of merging methods and objects from the previously separated versions. On the whole the changes are orthogonal but some adaptation is required.

As there are now a large number of polymorphic objects from various object hierarchies floating around the application, a level 5 factory object is also introduced (but not discussed).
The application object

App_MC_all, displayed in Figure 23.1, is the application object. It combines the variance reduction methods we have seen so far into a single application. Antithetic variates, stratified sampling and LD sampling are implemented in the Wiener generator objects. CVs are controlled by a CV manager, cvm_, and importance sampling by an IS manager, ism_.

Objects register with cvm_ and ism_ in App_MC_all::SetValues(), exactly as in Chapters 20 and 22.

In App_MC_all::Run() the option obtains its CVs from the CV manager, and its IS sample path from the IS manager, as it did in previous chapters.

After obtaining an option payoff value, val, on line 23.1d it is passed to the IS manager on line 23.1e to be weighted by the IS weight. On lines 23.1f, 23.1g, and 23.1h the CV manager is asked for the CV path statistics, which are passed to the IS manager to be weighted, and then passed back to the CV manager again to be corrected for the mean (see below). Finally, as before, the option value and the CVs are given to the accumulator.

The option objects

A typical option object, PayoffArithCall, is shown in Figure 23.2. It adds in the CV and IS methods from the option object in Chapters 21 and 22.

In this application, to enable tables to be produced similar to those discussed in section 23.3 that freely mix combinations of variance reduction methods and options, the option has its IS and CV settings given to it from the front-end via the InputManager. In practice the object would specify its own, optimized, settings; for instance the PayoffArithCall object would specify the array cv_names_ to contain "BScall" and "GeoAverage"; the IS settings would scale with X, N and T, and with S when this value is known to the option.

The various option objects are very similar, differing only in the payoff function. The common code, the IS and CV methods and data members identical in all option objects, could appropriately be separated out into distinct helper objects.

The IS manager

Only minor changes are required to the IS manager. In Chapter 22 a single method, IStransform(), was used to transform the option payoff by multiplying by the IS weight. Now we have to apply IS weights to the CVs as well as to the option payoff.

The solution used here is to define a pair of methods, ISweightScalar() and ISweightVector(), the first to transform the payoff, the second to transform an array of CV values. The new methods are shown in Figure 23.3. There is nothing complicated about them; note that ISweightVector(), although its array argument vec could be anything, exits without altering the array if the option has not requested CVs.

The IS manager does not precompute the entire set of IS weighted CVs. This might be efficient if, in a book of options, most CVs were likely to be used with most weights. The assumption underpinning the approach adopted here, computing IS weighted CVs only on demand, is that options have their particular, perhaps tailored, CVs and IS densities, so that a mass precomputation would be wasteful.

The CV manager

The CV manager requires changes that are slightly more substantive than those needed in the IS manager. There are two new methods, Get_cv_path() and Get_cvs_mean(), shown in Figure 23.4.
Private ctr_ As OutputCounter
Private acc_ As IAccumulator  'accumulator
Private wie_ As IWienerGenerator  'Wiener generator
Private pay_ As IPayoff  'payoff object
Private cvm_ As CVManager  'manages CV objects
Private ism_ As ISManager  'IS path manager
Private M_ As Long  'number of paths
Private N_ As Long  'number of time steps
Private Ncv_ As Long  'number of control variates
Private IApp_MC_all As App_MC_all

Private Sub Class_Terminate()
'Lines omitted
'All references Set to Nothing
End Sub

Friend Sub IApp_SetValues(fact As Factory)
  Set acc_ = fact.Accumulator
  Set pay_ = fact.payoff
  Set wie_ = fact.WienerGenerator
  Set ctr_ = fact.counter: Call ctr_.ReSetValues(8, 4, 0)
  Call ctr_.ClearCounter
  Set cvm_ = fact.GetCVManager
  Set ism_ = fact.GetISManager
  Call ism_.SetISlink(pay_) 'Register with IS manager
  Call cvm_.SetCVlink(pay_) 'Register with CV manager
  Call acc_.SetCVinds(pay_) 'Set up accumulator
  Ncv_ = cvm_.Ncv
  Dim data As InputManager: Set data = fact.Inputter
  M_ = data.M
  N_ = data.N
  Set data = Nothing
End Sub

Public Sub IApp_ReSet()
  Call acc_.Reset
  Call wie_.Reset
  Call ism_.Reset
End Sub

Private Sub IApp_Run()
  Dim WieVec() As Double: ReDim WieVec(0 To N_) As Double  'Wiener path
  Dim ispath() As Double: ReDim ispath(0 To N_) As Double  'IS asset path
  Dim cv_vals() As Double
  Dim val As Double
  Dim j As Long
  For j = 1 To M_
    Call ctr_.OutputCounter(j)
    Call wie_.GetWienerPath(WieVec)  'a. Wiener path
    Call ism_.SetPath(WieVec, j)  'b. Gives path to the path manager
    Call ism_.GetPath(j, ispath, pay_)  'c. Get a path generated under IS
    val = pay_.payoff(ispath)  'd. Get the payoff under IS
    Call ism_.ISweightScalar(j, val, pay_)  'e. Rotate back value
    Call cvm_.Get_cv_path(ispath, cvs)  'f. Compute CV path statistics
    Call ism_.ISweightVector(j, cvs, pay_)  'g. Rotate back CVs
    Call cvm_.Get_cvs_mean(cvs)  'h. Use to compute CVs
    Call acc_.update(val, cvs)  'i. accumulate
  Next j
End Sub

Figure 23.1 Combined speed-ups: the application object, App_MC_all
Private X_ As Double
Private discount_ As Double
Private Ncv_ As Long  'number of CVs
Private CVinds_() As Long  'indexes of CVs; each option looks after its own
Private cv_names_() As String  'names of cv's used by this object
Private IS_ As Boolean  'true if IS requested
Private ISspec_ As ISspec  'IS process specification
Private ISref_ As Long  'reference supplied by ISmanager
Private Sub IPayoff_SetValues(fact As Factory)
    Dim data As InputManager: Set data = fact.Inputter
    X_ = data.X
    discount_ = Exp(-data.r * data.T)
    Call SetISstuff(data)
    Call SetCVstuff(data)
    Set data = Nothing
End Sub

Private Sub SetISstuff(ByRef data As InputManager)
    IS_ = data.ImpS
    ISspec_.S0 = data.S0
    ISspec_.r = data.rIS
    ISspec_.sig = data.sigIS
    ISspec_.ISweight = data.IStype
    ISspec_.X = data.X
End Sub

Private Sub SetCVstuff(ByRef data As InputManager)
    Ncv_ = data.Ncv: If Ncv_ = 0 Then Exit Sub
    ReDim cv_names_(1 To Ncv_) As String
    Dim K As Long
    For K = 1 To Ncv_
        cv_names_(K) = data.CV(K)
    Next K
End Sub

Friend Function Get IPayoff_DoIS() As Boolean: IPayoff_DoIS = IS_: End Property
Friend Function Get IPayoff_ISref() As Long: IPayoff_ISref = ISref_: End Property
Friend Function Get IPayoff_Ncv() As Long: IPayoff_Ncv = Ncv_: End Property
Friend Function Get IPayoff_payoff(path() As Double) As Double
    Dim av As Double: av = ArithmeticAverage0(path)
    IPayoff_payoff = discount_ * my_max(0, av - X_)
End Function

Figure 23.2  A typical option object: PayoffArithCall
Implementing Models of Financial Derivatives

Friend Sub ISweightScalar(i As Long, ByRef val As Double, opt As IPayoff)
    Call CheckI(i)
    If Not opt.DoIS Then Exit Sub 'val is unchanged
    Dim ISref As Long: ISref = opt.ISref
    Dim wgt As Double: wgt = Gens_(ISref).GetWeight
    val = val * wgt
End Sub

Friend Sub ISweightVector(i As Long, ByRef vec() As Double, opt As IPayoff)
    Call CheckI(i)
    If Not opt.DoIS Then Exit Sub 'vec is unchanged
    If opt.Ncv = 0 Then Exit Sub 'assumes that vec is an array of cvs
    Dim ISref As Long: ISref = opt.ISref
    Dim wgt As Double: wgt = Gens_(ISref).GetWeight
    Dim lb As Long: lb = LBound(vec)
    Dim ub As Long: ub = UBound(vec)
    Dim k As Long
    For k = lb To ub
        vec(k) = vec(k) * wgt
    Next k
End Sub

Figure 23.3 The IS manager: methods replacing ITransform()

Private cvmeans_() As Double 'new data member. mean values of every cv

Friend Sub Get_cv_path(path() As Double, ByRef cvs() As Double)
    If Not DoCVs_ Then Exit Sub
    If Ncv_ = 0 Then Exit Sub
    ReDim cvs(1 To Ncv_) As Double
    Dim i As Long
    For i = 1 To Ncv_
        If CV_mask_(i) Then cvs(i) = CVs_(i).CV(path) + cvmeans_(i)
    Next i
End Sub

Friend Sub Get_cvs_mean(ByRef cvs() As Double)
    If Not DoCVs_ Then Exit Sub
    If Ncv_ = 0 Then Exit Sub
    Dim i As Long
    For i = 1 To Ncv_
        If CV_mask_(i) Then cvs(i) = cvs(i) - cvmeans_(i)
    Next i
End Sub

Figure 23.4 The CV manager: additional methods

Get_cv_path() returns an array of CV path statistics. These are the values $h^d(S)$ in equation (23.5). It is called by the application object so that the CV path statistics can be given to the IS manager. Get_cvs_mean() is used to correct for the mean value of the weighted path statistics. It is passed, ByRef, the values $h^d(S)w(S)$ in the array cvs. It returns in situ the mean-corrected values for $dIS$.

Get_cv_path() and Get_cvs_mean() are implemented using a new Private data member, an array, cvmeans_. This is set up in CVmanager::SetValues() to hold the precomputed mean values of the CVs.
Get_cv_path() adds on the mean value to the CV value to recover the path statistic; Get_cvs_mean() subtracts it back off again to get back the CV.

Each ICV conforming object has an additional getter for the mean value. For a delta control variate, and similar CVs, the mean is identically zero.

23.2.2 Extending the application to value a book

It is clear that the application described here extends to a book of options, perhaps using a framework analogous to that presented in Chapter 10. Option specifications, which can be read in from file, could contain a specification of the CVs and IS densities to accompany each option, or each option could know endogenously its own speed-up specification without having to be set up with it exogenously.

During the set-up phase each option has to register with the CV and IS manager objects, and configure its accumulator object. While run() is executing each option in the books needs to communicate with the CV and IS managers before passing the output (the option and CV values) to its accumulator. This is straightforward to arrange.

23.3 NUMERICAL ASSESSMENT

We compare various combinations of variance reduction methods with our three standard options: the vanilla call, the quadratic payoff exotic, and the average rate option with $N \in \{4, 16, 64\}$. Comparisons are made at two levels of strike: an ATM level and an OTM level. Parameter values are standard. Results are obtained with the spreadsheet MC_speedups_all.xls, discussed in section 23.2.

Importance sampling uses weights and parameter values found in Chapter 22 to be optimal for each case individually. $w_{CT}$ is used for the vanilla call and exotic option, where $N = 1$, and $w_J$ for the average rate options, with $N > 1$.

CVs are the geometric and call option CVs for the average rate options, and the terminal asset CV for the vanilla call. The exotic option uses the tailored payoff-matching CV with, in the notation of Chapter 20, $(a, b) = (99, 107)$ in the ATM case and $(a, b) = (139, 157)$ in the OTM case.

Stratification is always done with random plus LD sampling at $N/2$ levels.

As usual the entries in the tables are the option value, the standard error in round brackets, the time in square brackets, and the efficiency gain (over the plain case) in bold. Option values are shown only to the number of decimal places justified by the standard error.

Tables are split into panels. For instance, for average rate options each table has four panels, corresponding to the four possibilities of toggling antithetic sampling and importance sampling. In each case panels (a) and (c) are results without antithetic sampling, panels (b) and (d) are with; panels (a) and (b) are without IS, panels (c) and (d) are with.

The results of antithetic sampling are displayed only if these are not wholly negative. When there are efficiency gains with antithetic sampling these are obtained mainly because this method produces a second set of sample paths comparatively cheaply, and hence brings down costs rather than standard error. The saving in computation time is greater when $N$ is larger; when $N$ is small the saving is not often large enough to produce an overall gain.

The assessment is done in two parts: first for the ATM options and then for the OTM options.

23.3.1 ATM options

Table 23.1 gives results for the ATM vanilla call and exotic options. Tables 23.2, 23.3 and 23.4 give results for the ATM average rate options with $N = 4$, $N = 16$ and $N = 64$, respectively.
For our examples we find that generic IS methods perform without distinction for ATM options and perhaps should not be used at the same time as both a control variate and stratified sampling.

The single most effective variance reduction method for ATM options, by our criteria, is stratified sampling. The gains are good, it is generic (as used here) and it combines very well with other methods.

Gains for the long-step, non-path-dependent options, the vanilla call and the exotic, are truly enormous. Gains for average rate options are moderate; for options with fewer reset dates gains are less than for those with more reset dates.

CVs are the next best method after stratified sampling and should be used with stratified sampling; for some average rate options, CVs in isolation work better than stratification in isolation.

**The vanilla call**

The vanilla call (Table 23.1, panels (a) and (b)) receives greatest benefit from using stratified sampling with a control variate. The efficiency gain of $1.11 \times 10^6$ is very large but not unexpected for this type of long-step option. Antithetic sampling makes things worse; IS with stratified sampling makes things very much worse. IS reduces very considerably the gains made by stratified sampling on its own, from $1.11 \times 10^6$ down to 1990, a factor of 500.
Table 23.2 ATM average rate option, \( N = 4 \)

<table>
<thead>
<tr>
<th>CVs</th>
<th>Stratified</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No</td>
<td>(0.030)</td>
<td>(0.0060)</td>
<td>1.4</td>
</tr>
<tr>
<td></td>
<td>6.9400</td>
<td>6.9394</td>
<td></td>
</tr>
<tr>
<td>Yes</td>
<td>(0.0067)</td>
<td>(0.00050)</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>1470</td>
<td>1730</td>
<td></td>
</tr>
</tbody>
</table>

Panel (a): Without IS, without antithetic sampling

<table>
<thead>
<tr>
<th>CVs</th>
<th>Stratified</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No</td>
<td>(0.015)</td>
<td>(0.0065)</td>
<td>1.9</td>
</tr>
<tr>
<td></td>
<td>6.9390</td>
<td>6.9393</td>
<td></td>
</tr>
<tr>
<td>Yes</td>
<td>(0.00067)</td>
<td>(0.00048)</td>
<td>2.7</td>
</tr>
<tr>
<td></td>
<td>1070</td>
<td>1530</td>
<td></td>
</tr>
</tbody>
</table>

Panel (c): With IS, without antithetic sampling

<table>
<thead>
<tr>
<th>CVs</th>
<th>Stratified</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No</td>
<td>(0.015)</td>
<td>(0.0042)</td>
<td>1.8</td>
</tr>
<tr>
<td></td>
<td>6.9420</td>
<td>6.9392</td>
<td></td>
</tr>
<tr>
<td>Yes</td>
<td>(0.00069)</td>
<td>(0.00041)</td>
<td>2.5</td>
</tr>
<tr>
<td></td>
<td>1080</td>
<td>2500</td>
<td></td>
</tr>
</tbody>
</table>

Panel (d): With IS, with antithetic sampling

The exotic option

Results are given in Table 23.1, panels (c) without IS and (d) with IS. As we saw in Chapter 22, in isolation IS works well with the exotic option, and here it combines better with stratified sampling than in the case of the vanilla call, but still the results are ambiguous. The greatest gains are made with stratified sampling and a control variate. Used with stratified sampling and CVs, IS reduces the gain by a factor of 5. Overall speed-ups are an astounding \( 2.5 \times 10^9 \) (for CV with stratified sampling), surely an exceptional level of variance reduction.

The ATM average rate options

Table 23.2 gives results for the average rate option with \( N = 4 \). The biggest single contribution to the gain comes from CVs, of order \( 10^2 \) or so. Stratified sampling boosts the gain by a small factor. The effect of antithetic variates is very marginal. IS is only helpful in this case when used with antithetic variates and stratified sampling. The effects of the main speed-ups are additive; each brings improvement in combination.

Results for \( N = 16 \) are given in Table 23.3. Control variates and stratified sampling work together well, augmenting in combination their individual gains; CVs give the greatest gain, increased by a factor of 10 by stratified sampling. Antithetic variates are now a little more effective, increasing gains by perhaps a quarter or so on average. IS can be a liability, reducing the gains from CV; adding it to the other three methods almost halves gains from 24300 to 13100.
Table 23.3 ATM average rate option, $N = 16$

<table>
<thead>
<tr>
<th>CVs</th>
<th>Stratified</th>
<th></th>
<th>CVs</th>
<th>Stratified</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No</td>
<td>Yes</td>
<td></td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>6.08</td>
<td>6.057</td>
<td></td>
<td>6.08</td>
</tr>
<tr>
<td></td>
<td>(0.027)</td>
<td>(0.0014)</td>
<td></td>
<td>(0.027)</td>
</tr>
<tr>
<td></td>
<td>[3.1]</td>
<td>[5.3]</td>
<td></td>
<td>[2.4]</td>
</tr>
<tr>
<td></td>
<td>218</td>
<td></td>
<td></td>
<td>1.3</td>
</tr>
<tr>
<td></td>
<td>6.0575</td>
<td>6.0581</td>
<td></td>
<td>6.0568</td>
</tr>
<tr>
<td></td>
<td>(0.00061)</td>
<td>(0.00015)</td>
<td></td>
<td>(0.00061)</td>
</tr>
<tr>
<td></td>
<td>[3.8]</td>
<td>[6.0]</td>
<td></td>
<td>[3.1]</td>
</tr>
<tr>
<td></td>
<td>1540</td>
<td>15500</td>
<td></td>
<td>1930</td>
</tr>
</tbody>
</table>

Panel (a): Without IS, without antithetic sampling

<table>
<thead>
<tr>
<th>CVs</th>
<th>Stratified</th>
<th></th>
<th>CVs</th>
<th>Stratified</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No</td>
<td>Yes</td>
<td></td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>6.07</td>
<td>6.058</td>
<td></td>
<td>6.06</td>
</tr>
<tr>
<td></td>
<td>(0.015)</td>
<td>(0.0014)</td>
<td></td>
<td>(0.014)</td>
</tr>
<tr>
<td></td>
<td>[4.0]</td>
<td>[6.1]</td>
<td></td>
<td>[3.1]</td>
</tr>
<tr>
<td></td>
<td>2.6</td>
<td>173</td>
<td></td>
<td>3.4</td>
</tr>
<tr>
<td></td>
<td>6.0581</td>
<td>6.0578</td>
<td></td>
<td>6.0581</td>
</tr>
<tr>
<td></td>
<td>(0.00053)</td>
<td>(0.00015)</td>
<td></td>
<td>(0.00055)</td>
</tr>
<tr>
<td></td>
<td>[4.8]</td>
<td>[6.9]</td>
<td></td>
<td>[4.1]</td>
</tr>
<tr>
<td></td>
<td>1610</td>
<td>14100</td>
<td></td>
<td>1760</td>
</tr>
</tbody>
</table>

Panel (c): With IS, without antithetic sampling

Table 23.4 displays the final average rate case, with $N = 64$. Using antithetic variates is a definite plus, perhaps almost doubling the gain in some cases. Both stratified sampling and CVs give, individually, gains of the order of 2000 or 3000 or so, and these augment one another in combination. IS works well with CVs but has a negative effect when stratified sampling is also included.

23.3.2 OTM options

As in the ATM case we start by looking at the vanilla call and the exotic in Table 23.5, and then move on to the average rate options in Tables 23.6, 23.7 and 23.8.

We now find that, in contrast to the ATM case, IS is generally much more effective than CV. Antithetic variates add very little but should be included with non-long-step Monte Carlo. Stratified sampling works very well, still adding as much in the OTM cases as it does in the ATM case. Here it is importance sampling that works best with stratified sampling, and the control variate works less well.

The vanilla call

The OTM vanilla call gets its best gains from stratified sampling used with importance sampling. Table 23.5 illustrates; panel (a) gives results without IS and panel (b) with IS.
The terminal asset control variate adds no value in isolation, and reduces gains in combination. IS still does not work well with CV, but here it is the CV that should not be used.

The exotic option

Results for the exotic option are given in Table 23.5, panels (c) and (d). For the exotic option both IS and CV give large, similar, gains. The gain achieved by stratified sampling, $4.6 \times 10^{-5}$ when combined with IS, is spectacular. Both IS and CV augment the gains made by stratification and, despite the small drop when all three methods are used, CV and IS could be used together.

The OTM average rate options

For the three OTM average rate options antithetic variates have at best only a very small positive effect. Nevertheless the effect is real.

The $N = 4$ case is given in Table 23.6. In isolation, CVs have about half the effect of IS. The effect of stratified sampling is slight, perhaps at best doubling the gain. The greatest gains come from using all four methods together, but the gain is not quite as great as the ATM case. On the whole the methods combine additively, each bringing an improvement, although perhaps sometimes small.
Table 23.5 OTM vanilla and exotic options, $N = 1$

OTM vanilla call option, $N = 1$. (Explicit value: 0.3596298)

<table>
<thead>
<tr>
<th>CVs</th>
<th>Stratified</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>0.375</td>
</tr>
<tr>
<td></td>
<td>(0.0095)</td>
</tr>
<tr>
<td></td>
<td>[0.8]</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Yes</td>
<td>0.358</td>
</tr>
<tr>
<td></td>
<td>(0.0082)</td>
</tr>
<tr>
<td></td>
<td>[1.1]</td>
</tr>
<tr>
<td></td>
<td>1.0</td>
</tr>
</tbody>
</table>

Panel (a): Without IS

Panel (b): With IS

OTM quadratic payoff exotic option, $N = 1$. (Explicit value: 3.0154796)

<table>
<thead>
<tr>
<th>CVs</th>
<th>Stratified</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>2.99</td>
</tr>
<tr>
<td></td>
<td>(0.046)</td>
</tr>
<tr>
<td></td>
<td>[0.9]</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Yes</td>
<td>3.014</td>
</tr>
<tr>
<td></td>
<td>(0.0030)</td>
</tr>
<tr>
<td></td>
<td>[1.1]</td>
</tr>
<tr>
<td></td>
<td>203</td>
</tr>
</tbody>
</table>

Panel (c): Without IS

Panel (d): With IS

Table 23.7 gives results for $N = 16$. Antithetics have a small positive effect, enhancing gains by an extra one-third or so. Stratified sampling is not here a major performance enhancer. IS gives significantly better results than CV. The four methods compound nicely to give together greater gains than in isolation or in lesser combination.

The $N = 64$ case is shown in Table 23.8. Here again the methods work well together although when IS is not present adding antithetics to stratified sampling reduces the gain. IS contributes more than CV and about as much as stratified sampling. The best gains are a considerable 72 500 from a combination of all four methods.

23.3.3 Overview of results

Each of the four speed-up methods has a contribution to make to the success of a combined method. Table 23.9 summarizes, for each of the options we have examined, the best gains and the methods used to achieve them. ‘A’ stands for antithetics, ‘St’ stands for stratified sampling.

In every example, stratified sampling contributes to the best method. Antithetics contribute when the method is not long-step. IS is always used for OTM options; CV is always used whenever the option is
Table 23.6 OTM average rate option, \( N = 4 \)

<table>
<thead>
<tr>
<th>CVs Stratified</th>
<th>No</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>0.019</td>
<td>0.0181</td>
</tr>
<tr>
<td>(0.0014)</td>
<td>(0.00075)</td>
<td></td>
</tr>
<tr>
<td>[1.4]</td>
<td>[1.8]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.9</td>
<td></td>
</tr>
<tr>
<td>Yes</td>
<td>0.0192</td>
<td>0.0192</td>
</tr>
<tr>
<td>(0.00021)</td>
<td>(0.00016)</td>
<td></td>
</tr>
<tr>
<td>[1.9]</td>
<td>[3.0]</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>39</td>
<td></td>
</tr>
</tbody>
</table>

Panel (a): Without IS, without antithetic sampling

<table>
<thead>
<tr>
<th>CVs Stratified</th>
<th>No</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>0.0190</td>
<td>0.01902</td>
</tr>
<tr>
<td>(0.00014)</td>
<td>(7.8 \times 10^{-5})</td>
<td></td>
</tr>
<tr>
<td>[1.9]</td>
<td>[2.3]</td>
<td></td>
</tr>
<tr>
<td>78</td>
<td>202</td>
<td></td>
</tr>
<tr>
<td>Yes</td>
<td>0.01897</td>
<td>0.01903</td>
</tr>
<tr>
<td>(2.8 \times 10^{-5})</td>
<td>(2.0 \times 10^{-5})</td>
<td></td>
</tr>
<tr>
<td>[2.7]</td>
<td>[3.7]</td>
<td></td>
</tr>
<tr>
<td>1390</td>
<td>1930</td>
<td></td>
</tr>
</tbody>
</table>

Panel (c): With IS, without antithetic sampling

ATM. The CV used with average rate options is particularly good; perhaps this is the reason why it is part of the best method even for OTM average rate options. The generic IS implementation used here is not too good with ATM options, which may explain why IS is part of only one optimal set of methods in the ATM case.

Speed-ups are usually quite a bit less for OTM options than for ATM options (except for the special case of the exotic). The \( N = 4 \) average rate option would benefit considerably from a better, tailored, IS distribution. Speed-ups for long-step (\( N = 1 \)) options can be very good indeed, largely because of the effectiveness of stratified sampling. When stratified sampling cannot be effective, for instance for the OTM vanilla call where it is being used with a poor IS distribution, then gains are much less.

For the average rate options gains are increasing as \( N \) increases, mainly because LD sampling is able to contribute more to the option when the number of reset dates is greater.

Since the optimal choice of variance reduction methods depends on the moneyness of an option, to be able to specify its variance reduction methods an option needs to be aware of process data such as the current value of the underlying asset. An industrial Monte Carlo method would include a pre-processing phase in which options were able to tune their variance reduction specifications to current market conditions (and the model they were being priced with). In our application this could take place in \( \text{App\_MC\_all::SetValues()} \) in the \( \text{ISmanager::SetISlink()} \) method.
Table 23.7 OTM average rate option, \( N = 16 \)

<table>
<thead>
<tr>
<th>CVs</th>
<th>Stratified</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>No</td>
<td>0.0044</td>
<td>0.00521</td>
<td>(9.6 \times 10^{-5})</td>
</tr>
<tr>
<td></td>
<td>(0.00069)</td>
<td>[3.2]</td>
<td>[5.3]</td>
</tr>
<tr>
<td></td>
<td>–</td>
<td>31</td>
<td></td>
</tr>
<tr>
<td>Yes</td>
<td>0.0054</td>
<td>0.00517</td>
<td>(5.2 \times 10^{-5})</td>
</tr>
<tr>
<td></td>
<td>(0.00015)</td>
<td>[3.9]</td>
<td>[6.1]</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>91</td>
<td></td>
</tr>
</tbody>
</table>

Panel (a): Without IS, without antithetic sampling

Panel (b): Without IS, with antithetic sampling

<table>
<thead>
<tr>
<th>CVs</th>
<th>Stratified</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>No</td>
<td>0.0048</td>
<td>0.00477</td>
<td>(9.7 \times 10^{-5})</td>
</tr>
<tr>
<td></td>
<td>(0.00069)</td>
<td>[2.4]</td>
<td>[3.5]</td>
</tr>
<tr>
<td></td>
<td>1.3</td>
<td>46</td>
<td></td>
</tr>
<tr>
<td>Yes</td>
<td>0.0054</td>
<td>0.00520</td>
<td>(5.8 \times 10^{-5})</td>
</tr>
<tr>
<td></td>
<td>(0.00016)</td>
<td>[3.1]</td>
<td>[4.1]</td>
</tr>
<tr>
<td></td>
<td>19</td>
<td>111</td>
<td></td>
</tr>
</tbody>
</table>

Panel (c): With IS, without antithetic sampling

Panel (d): With IS, with antithetic sampling

### 23.4 SUMMARY

Stratified sampling should always be used, but there is no strong case for using antithetic variates; they could be dropped. The difficulty lies in establishing when CVs and IS should be used, individually or in combination. Table 23.10 summarizes the situation for our options and our particular operationalizations of the methods.

Stratification always works well, and is often chiefly responsible for efficiency gains. To get really exceptional gains, stratification must be combined with another method: a CV in the ATM case, and IS with CV in the OTM case.

Antithetic sampling usually brings advantages simply because it enables a set of sample paths to be constructed more cheaply. If there is a decrease in standard error, that is a welcome but not necessarily expected bonus. With few long-step options in a book, antithetic sampling would normally be used; only if a book can be valued with very few time steps should antithetics definitely be dropped.

In practice the gains made with antithetic sampling are not large; you could not be criticized for not bothering to implement them. If stratification is fully LD, for some reason, then antithetics should certainly not be used; you are likely to be damaging the low-discrepancy properties of your sample.

A good CV works wonderfully well, especially for ATM options. However, even with a CV proved to work well for ATM options, like the geometric average CV for average rate options, gains are less secure when the options are OTM.
Table 23.8  OTM average rate option, $N = 64$

<table>
<thead>
<tr>
<th>CVs Stratified</th>
<th>No</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVs No</td>
<td>0.0049</td>
<td>0.00298</td>
</tr>
<tr>
<td></td>
<td>(0.00077)</td>
<td>(4.1 × 10^{-5})</td>
</tr>
<tr>
<td></td>
<td>[9.8]</td>
<td>[16.3]</td>
</tr>
<tr>
<td></td>
<td>--</td>
<td>217</td>
</tr>
<tr>
<td>CVs Yes</td>
<td>0.0034</td>
<td>0.00340</td>
</tr>
<tr>
<td></td>
<td>(0.00015)</td>
<td>(1.2 × 10^{-5})</td>
</tr>
<tr>
<td></td>
<td>[11.2]</td>
<td>[17.7]</td>
</tr>
<tr>
<td></td>
<td>23</td>
<td>2230</td>
</tr>
</tbody>
</table>

Panel (a): Without IS, without antithetic sampling

<table>
<thead>
<tr>
<th>CVs Stratified</th>
<th>No</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVs No</td>
<td>0.00344</td>
<td>0.003518</td>
</tr>
<tr>
<td></td>
<td>(5.2 × 10^{-5})</td>
<td>(3.0 × 10^{-6})</td>
</tr>
<tr>
<td></td>
<td>[11.7]</td>
<td>[18.2]</td>
</tr>
<tr>
<td></td>
<td>189</td>
<td>35500</td>
</tr>
<tr>
<td>CVs Yes</td>
<td>0.00348</td>
<td>0.003468</td>
</tr>
<tr>
<td></td>
<td>(1.9 × 10^{-5})</td>
<td>(2.71 × 10^{-6})</td>
</tr>
<tr>
<td></td>
<td>[13.2]</td>
<td>[19.7]</td>
</tr>
<tr>
<td></td>
<td>1210</td>
<td>68000</td>
</tr>
</tbody>
</table>

Panel (c): With IS, without antithetic sampling

A poor IS density may nevertheless bring a small level of speed-up, used only by itself, but used together with stratification or with CVs it can reduce some of the advantage that these methods deliver. IS methods, in our implementations, work very well with OTM options, both in isolation and in combination with both CV and stratified sampling. However, for ATM options our generic implementations are not particularly helpful. It is likely that tailored IS would perform much better than the generic implementations we have been using.

To summarize: A Monte Carlo method should always implement stratified sampling and maybe antithetic variates. Individual options must assess whether they should be ‘CV-options’, ‘IS-options’ or...
Table 23.10 Comparison of variance reduction methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Strat.</th>
<th>CV</th>
<th>IS</th>
<th>Anti.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATM</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Call</td>
<td>✓✓</td>
<td>✓</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Exotic</td>
<td>✓✓</td>
<td>✓</td>
<td>✓✓</td>
<td>✓</td>
</tr>
<tr>
<td>Av. rate</td>
<td>✓✓</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>OTM</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Call</td>
<td>✓✓</td>
<td>×</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>Exotic</td>
<td>✓✓</td>
<td>✓</td>
<td>✓✓</td>
<td>×</td>
</tr>
<tr>
<td>Av. rate</td>
<td>✓✓</td>
<td>✓</td>
<td>✓✓</td>
<td>~</td>
</tr>
</tbody>
</table>

‘CV-IS-options’ (depending on what is available and what works). An ATM option might be more likely to prefer to use CV but not IS; and OTM option might be better off with IS or both. This would need to be established case by case.

23.5 EXERCISES

1. Consider a far in the money vanilla call; with our standard parameter values this could be a call with strike $X = 50$, say.

   (a) How good a CV is the terminal asset value CV in this case?

   (b) How good is the $w_{CT}$ IS weight?

   (c) When used with other speed-up methods in combination, and with stratified sampling, is the option an IS-option, a CV-option, or an option of some other type?

2. Consider the path-dependent European options with the following payoffs:

   (i) $H^A(S) = \frac{1}{N} \sum_{i=1}^{N} (S_i - X)^+ \text{ for } X < S_0$;

   (ii) $H^G(S) = \left( \prod_{i=1}^{N} (S_i - X_1)^+ \right)^{1/N} - X_2$ \text{ for } X_1 < S_0;

   (iii) $H^2(S) = \frac{1}{N} \sum_{i=1}^{N} ((S_i - X)^+)^2 \text{ for } X < S_0$.

   For each option:

   (a) Is $CV_{GA}$, or some variant, a good CV?

   (b) Is $w_J$ a good IS density?

   (c) Do $CV_{GA}$ and $w_J$ complement one another?

   (d) What the greatest speed-up that you can devise for each of these options?

3. For the generic examples of IS densities illustrated in this chapter is it possible to devise a method to enable OTM options to determine endogenously their optimal, or near optimal, IS density parameters?
To reduce simulation error, in the form of standard error, Part VI discussed various speed-up methods that, when used appropriately, can produce significant efficiency gains. The examples in that part were centred around processes whose SDEs could be solved exactly, so that they could be simulated without distribution error. In particular, GBM was used, almost platonically, to illustrate their effectiveness.

Unfortunately GBM is ineffective in the real world as a model for derivatives; models that recover market prices better, that fit to more of the volatility surface, are needed and are used. These models, being perhaps only a little more complex, may no longer have tractable exact solutions. An exact simulation, even if possible, may be too expensive for practical use. In such cases an approximate simulation method should be used: a discretization scheme that, although not exact, converges in some sense to the exact solution. By putting in sufficient effort one may get arbitrarily close to the exact solution: convergence without bias.

This part discusses discretization schemes and their convergence. Chapter 24 reviews several issues related to convergence and bias. For certain options, such as barrier options and options on extreme values, unless the number of time steps is infeasibly large, a Monte Carlo method produces biased results. The chapter presents a brief review of sources of bias and methods for their reduction. We address some of these in Chapter 25 where a number of general discretization schemes are presented and implemented. Their efficacy for some benchmark processes is assessed. The main focus is on Itô–Taylor discretization schemes although predictor–corrector schemes are also mentioned.

The remaining chapters explore bias and convergence in particular applications. Chapter 26 investigates two examples relevant to interest rate models (but also applicable more widely): the CIR square root volatility process, and the computation of a stochastic discount factor in the Vasicek model. These are widely used in credit, volatility and interest rate models but can present practical implementation difficulties. The implementation (by standard methods) of the Heston 2-factor stochastic volatility model is examined in Chapter 27. Heston, although widely used, is hard to implement effectively.

I would like to thank Kai Zhang for his contribution in reviewing the chapters and validating the software in this part. All errors are my own.
The Monte Carlo method is a probabilistic numerical integration method. Every time it runs you get a different number out. Because of this the distribution function $F$ of the simulated value is of great interest.

A somewhat crucial requirement is that the mean of $F$ coincides with the true underlying option value. If not, the method is biased. More generally ‘bias’ can refer to more or less any effect over and above simulation error. For instance when $F$ is skewed so that option values are systematically shifted away from their mean values (and every so often heavily shifted in the opposite direction), or option values are obtained after repeated regression in the Longstaff and Schwartz least squares Monte Carlo method (Part VIII), these effects are also sometimes referred to as bias.

Bias can be produced if the method is discretizing an underlying stochastic process with a non-exact discretization method. Generally this form of bias goes to zero as the number of time steps $N$ grows large (otherwise the discretization method is discretizing the wrong process). Unfortunately $N$ may have to be quite large for the bias to disappear.

Alternatively bias may be introduced when a payoff in continuous time is not accurately reproduced by the payoff in the discrete time approximation. The classic example is a Monte Carlo valuation of a continuous barrier option. The Monte Carlo method samples the underlying stochastic process at discrete times, so it is effectively valuing a discrete barrier option. The value of discrete barrier options converge only slowly to the value of the corresponding continuous barrier option, so an unmodified Monte Carlo method retains a significant bias until $N$ is large.

This chapter is a loose discussion of a number of issues under the general heading of bias. Section 24.1 gives some examples of how bias can arise in practical problems. The next section discusses some general bias reduction methods, and section 24.3 discusses bias reduction for continuous barrier options.

### 24.1 REDUCING BIAS

This section looks at bond option valuation in the CIR short rate model, showing various ways in which error, bias and inefficiency can arise. It starts with a general overview of bias.

#### 24.1.1 Sources of bias

It is possible to identify at least four sources of bias in a Monte Carlo valuation method.

1. **Discretizing an SDE.** If one is unable to solve an SDE exactly then, to simulate it, a discrete approximation needs to be used. This introduces discretization error, and the possibility of bias.
2. **Computing values of market observables.** If the state variables are not market observables, then values of market observable values need to be computed to enable the model to be calibrated to the market. This can introduce bias, particularly when the market observed values have to be computed numerically.
3. Computing discount factors. In a model with stochastic interest rates the value along a sample path of the discount factor $B_t(T)$,

$$B_t(T) \sim \hat{B}_t(T) = \exp \left( -\sum_{i=0}^{N-1} r_i \Delta t \right), \quad (24.1)$$

must be approximated, potentially introducing bias.

4. Payoffs dependent on values in continuous time. An option on a continuous running minimum requires a minimum along an entire sample path, not just at a set of discrete times. An inappropriate approximation will cause bias.

In this list bias comes at two stages. The first is at the discretization stage and the second at the integration stage.

At the discretization stage bias may be introduced if the drift of the process being discretized depends on the value of the process itself. The extent of the bias depends on the nature of the discretization. If there is an exact solution to the SDE then, since the distribution of the state variable is sampled exactly, there is no bias. By contrast, a simple Euler scheme will usually be biased.

Bias may be introduced at the integration stage if the option payoff depends non-trivially upon the entire sample path, rather than values just at a discrete set of times. This is exemplified by continuous barrier options. Fortunately there are techniques available for barrier options to reduce the bias. In general this is not the case.

24.1.2 Example: The Cox, Ingersoll and Ross model

To motivate the discussion consider using a Monte Carlo method to value bond options in the CIR model (Cox, Ingersoll and Ross (1985)). This is a short-rate model in which the short-rate process is

$$dr_t = \alpha (\mu - r_t) \, dt + \sigma \sqrt{r_t} \, dz_t. \quad (24.2)$$

The presence of $\sqrt{r_t}$ in the volatility term prevents the process becoming negative, although for some sets of parameter values 0 is accessible.

Suppose that a European bond call option with strike $X$ maturing at time $T$ is written on a PDB with principal 1 that matures at time $\bar{T} > T$. Suppose the time step is $\Delta_t = T/N$ and that $\bar{T} = \bar{N}\Delta_t$ for $\bar{N} < N \in \mathbb{Z}$. Set $t_i = t_0 + i \Delta_t$ and write $r_t = r_{t_i}$. A naive valuation method would discretize (24.2) using an Euler discretization,

$$r_{t+\Delta_t} = r_t + \alpha (\mu - r_t) \Delta_t + \sigma \sqrt{r_t} \sqrt{\Delta_t} \, \varepsilon_t, \quad \varepsilon_t \sim \mathcal{N}(0, 1), \quad (24.3)$$

and then proceed to:

1. evolve $r_t$ up to time $\bar{T}$ yielding a sample path $(r_0, \ldots, r_N, r_{N+1}, \ldots, r_{\bar{N}})$;
2. compute the value $\hat{B}_T(\bar{T})$ of the PDB at time $T$ along the sample path as

$$\hat{B}_T(\bar{T}) = \exp \left( -\Delta_t \sum_{i=N}^{\bar{N}-1} r_i \right); \quad (24.4)$$
(3) compute the payoff along the path as

\[ V = \exp \left( -\Delta_t \sum_{i=0}^{N-1} r_i \right) \left( \hat{B}_T - X \right)^+, \]  

(24.5)

and then repeat \( M \) times and take the average.

This approach is about as bad as it can get. Much can go wrong with it.

**It blows up**

Using Euler there is nothing to prevent the simulated \( r_t \) from becoming negative. The square root term in (24.3) then blows up. This could be avoided by setting \( r_t \equiv r^+_t = \max(r_t, 0) \) at every step but this is

(i) inefficient and
(ii) itself a source of bias.

An improvement is to evolve

\[ \tilde{r}_{t+\Delta t} = \tilde{r}_t + \alpha (\mu - \tilde{r}_t) \Delta_t + \sigma \sqrt{\tilde{r}_t \Delta_t} \epsilon_t, \quad \epsilon_t \sim \mathcal{N}(0, 1). \]  

(24.6)

and then to set \( r_t = \tilde{r}^+_t \). \( \tilde{r}_t \) may become negative but not \( r_t \). \( r_t \) computed this way has better distributional properties than the previous method.

**Bias in the drift**

Because it assumes that \( r_t \) is constant over each time step, the term \( r_t + \alpha (\mu - r_t) \Delta_t \) is not the true mean of \( r_{t+\Delta t} \mid r_t \). This can be corrected in this case by substituting the known first moment,

\[ \mathbb{E}[r_{t+\Delta t} \mid r_t] = e^{-\alpha \Delta t} r_t + (1 - e^{-\alpha \Delta t}) \mu, \]  

(24.7)

for the Euler drift. (In fact for CIR the full distribution of \( r_{t+\Delta t} \mid r_t \) is known: it is non-central chi-squared. See chapter 26.) Even using the true first moment there is still bias if only because of the correction required near zero.

**Bias in the discount factors**

The term \( \Delta_t \sum_{i=0}^{N-1} r_i \) approximates the integral \( I_T = \int_0^T r_s \, ds \). A better approximation could be made using, for instance, a trapezium rule. Alternatively in the Vasicek case the distribution of \( I_T \mid r_0 \) is known, so is the bridge distribution \( I_T \mid r_0, r_T \), and so is

\[ I_{0,T}^V = \mathbb{E}[I_T \mid r_0, r_T]. \]  

(24.8)

The Vasicek correction can be used to approximate the CIR case,

\[ I_T = \int_0^T r_s \, ds \sim \sum_{i=0}^{N-1} I_{t_i, t_{i+1}}^V, \]  

(24.9)
for suitable choices of parameter values. As an alternative at each step the discount factor over the period \([t_i, t_{i+1}]\) can be approximated by the PDB \(B_{t_i}(t_{i+1} | r_{t_i})\), computed in either the CIR or the Vasicek model (as an approximation) so that

\[
\hat{B}_{t_0}(t_N) \sim \prod_{i=0}^{N-1} B_{t_i}(t_{i+1} | r_{t_i})
\]

(24.10)

along each sample path.\(^1\)

**Foresight bias**

When \(\hat{B}_T(T)\) is computed within the Monte Carlo method from equation (24.4), with the payoff then computed with equation (24.5), one is guilty of foresight: the exercise decision is based upon a knowledge of the future sample path. This leads to vastly overstated prices. An American Monte Carlo method, such as Longstaff and Schwartz (Part VIII) must be used instead. In fact since in the CIR model bond prices can be computed explicitly, the discount term does not need to be computed with Monte Carlo. The values \(B_{t_0}(t_N)\) and \(B_{t_N}(t_N)\) can be used directly.

**Inefficient computation**

If the payoff \((\hat{B}_T(T) - X)^+\) is zero then \(V\) is zero, and no discount back to time \(t_0\) is required. If discounting back to time zero is performed in every case, then the computation is inefficient. In fact this is not such an issue in practice since, if many options are being valued simultaneously, it is likely that the discount factor will have to be computed in any case.

**Violation of put–call parity**

In the CIR model there are explicit solutions for the term structure of interest rates (and for bond options for that matter, but we let that pass). The Monte Carlo method, even if unbiased, will not, on any particular run, recover theoretical bond prices. The extent to which prices are not matched is the extent to which put–call parity for bond options is violated.

Methods of evolving a CIR process and of computing discount factors are discussed in Chapter 26, and foresight bias in Part VIII.

**24.2 BIAS REDUCTION METHODS**

We shall find it useful to distinguish between three different security values. The first is the market value, the second the theoretical model value, and the third is the method value generated from the model by a numerical method. Some models are capable of calibrating exactly to market prices so that the model value equals the market value, at least over a well-defined set of instruments. Generally the method value will not equal the model value; even the computation of an explicit solution may involve a very slight numerical error.\(^2\) For a Monte Carlo valuation one expects the method value to be within (a small multiple of) the standard error of the model value, but not better.

---

1 In fact the CIR integral \(I_T\) has an (approximate) exact simulation method (due to Glasserman and Kim (2008)). However, given the results of chapter 26, where it is found that even for small \(N\) a simple trapezium rule performs as well as an exact scheme (in the Vasicek model), we are happy to consider alternative schemes for the CIR model.

2 A routine commonly used to compute the normal distribution function \(N()\) is accurate only to 6 decimal places.
We mention three techniques for reducing bias. These are martingale simulation, martingale correction and, more generally, moment matching. The martingale correction is closely related to moment matching. The former matches method prices to model prices; the latter matches to market prices.

**Martingale simulation**

This technique reduces simulation bias by removing the drift term in asset price process. The idea is to evolve not the original process, or set of processes, but a transformation of them, which is driftless. Values of the original processes are then recovered from the driftless processes. A Libor market model example is discussed in Glasserman and Zhao (2000).

**Martingale correction**

A martingale correction forces method prices to equal model prices, at least for a certain set of instrument prices. It can ensure that those things whose expected discounted values should equal today’s market value (such as the values of hedging instruments) do in fact equal today’s market value.

For example (see Duan and Simonato (1998); Glasserman (2004)), suppose that a set of sample paths have been generated from a GBM, quite possibly using the exact solution of the SDE. Under the accumulator account numeraire the process $S_t$ has the property that $S_t = e^{-r(T-t)}E[S_T]$.

Write $S^j_i$ for the value of $S_t$ at time $t_i$ along the $j$th sample path and compute

$$\hat{S}_{0,i} = e^{-r(t_i-t_0)} \frac{1}{M} \sum_{j=1}^{M} S^j_i. \quad (24.11)$$

Of course $\hat{S}_{0,i} \neq S_0$; the simulated expected discounted future value does not equal the current asset value.

This leads to disaster as arbitrage is now built into prices. Let $p_t$ and $c_t$ be the values at time $t$ of European puts and calls on the same underlying asset, with current value $S_t$, with the same strike and maturity. Then put–call parity is the relationship

$$p_t + S_t = c_t + \text{Pv}(X) \quad (24.12)$$

where here $\text{Pv}(X) = e^{-(T-t)}X$ is the present value to time $t$ of $X$ received at time $T$. Put–call parity is a model-independent property.

Suppose that interest rates are constant and write $\hat{p}_t$, $\hat{c}_t$ and $\hat{S}_t$ for the Monte Carlo estimates of $p_t$, $c_t$ and $S_t$ computed simultaneously from the same set of sample paths. They will satisfy method put–call parity,

$$\hat{p}_t + \hat{S}_t = \hat{c}_t + \text{Pv}(X). \quad (24.13)$$

If $\hat{S}_t \neq S_t$ then it is not possible for market put–call parity, $\hat{p}_t + S_t = \hat{c}_t + \text{Pv}(X)$, to be satisfied. Attempts to trade simultaneously at prices $\hat{p}_t$ and $\hat{c}_t$ leave the trader open to arbitrage.

Applying a martingale correction ensures that $\hat{S}_{0,i} = S_0$ for every $i$ and hence that $\hat{p}_t$ and $\hat{c}_t$ satisfy market put–call parity. The procedure is to rescale every $S^j_i$ by setting

$$\overline{S}^j_i = S^j_i \frac{S_0}{\hat{S}_{0,i}}. \quad (24.14)$$
then
\[ \hat{S}_{0,i} = e^{-r(t_i-t_0)} \sum_{j=1}^{M} \hat{S}_j^i = S_0 \] (24.15)

and put–call parity (for instance) is no longer violated. This is the martingale correction for GBM applied to the current asset value. The adjusted values, \( \hat{S}_j^i \), can be used to value other instruments with the assurance that method prices are put–call parity arbitrage-free.

Note that being arbitrage free, although a fine principle, may not be the full story. If the degree of method error lies within bid–offer spread then the effort devoted to a martingale correction may not be worthwhile. Also, although calibration to a model price, or set of model prices, is guaranteed there is no guarantee that prices of other instruments have not been pulled out of line; bias may be introduced into their method values. You gain on the swings but lose on the roundabouts.

Another potential issue is the storage requirement of the method. If a path-wise evolution is used, then the entire set \( \{ S_j^i \}_{j=1}^{M} \) has to be generated and stored before the \( \hat{S}_{0,i} \) can be computed and valuations made based on \( \hat{S}_j^i \). However if slice-wise evolution is used then there is no storage overhead. \( \hat{S}_{0,i} \) can be computed as you go along and the \( S_j^i \) are immediately available.

**Moment matching**

Moment matching extends the concept of the martingale correction by allowing a Monte Carlo method to reproduce exactly the market values of a choice of instruments. It is not quite the same thing as the martingale correction. The martingale correction allows the method to match to model prices; moment matching allows the method to match to market prices even if these deviate from model prices. Moment matching may perhaps mis-name the method; price matching may better reflect the application we consider here.\(^3\)

As an example (discussed in Glasserman (2004)) consider an interest rate model where the state variable is the short rate \( r_t \). Suppose that a Monte Carlo procedure has generated a set of values \( \{ r^j_i \}_{j=1}^{M} \) for \( N \) steps along \( M \) sample paths. Set \( \hat{B}^i \) as \( \exp \left( -\frac{1}{\Delta t} \sum_{k=0}^{i-1} r^j_k \right) \). Method bond prices are

\[ \hat{B}^i = \frac{1}{M} \sum_{j=1}^{M} \hat{B}^i,j. \] (24.16)

If market bond prices are \( B^i = B_{t_0}(t_i) \) then of course \( \hat{B}^i \neq B^i \) even if theoretical model bond prices match market values.

Let

\[ f^i_{i+1} = -\frac{1}{\Delta t} \ln \left( \frac{B^{i+1}}{B^i} \right) \]

be the market forward rate for the period \([t_i, t_{i+1}]\) and

\[ f^i_{i+1} = -\frac{1}{\Delta t} \ln \left( \frac{\hat{B}^{i+1}}{\hat{B}^i} \right) \]

the method forward rate. \( f^i_{i+1} \) and \( \hat{f}^i_{i+1} \) can be used to adjust simulated rates to force \( \hat{B}^i = B^i \). Set

\[ \hat{r}^j_i = r^j_i + f^i_{i+1} - \hat{f}^i_{i+1}. \] (24.17)

---

\(^3\) In general the method can match not simply market prices but functions (moments) of data observed in the market.
Then
\[
\hat{B}^{i+1} = \frac{1}{M} \sum_{j=1}^{M} \exp \left( -\Delta t \sum_{k=0}^{i} \hat{r}_{k} \right)
\]
(24.18)
\[
= \frac{1}{M} \sum_{j=1}^{M} \hat{B}_{i+1,j} \prod_{k=1}^{i} \frac{B^{k+1}}{B^{k}} \frac{\hat{B}^{k}}{B^{i+1}}
\]
(24.19)
\[
= \hat{B}^{i+1} \frac{B^{i+1}}{B^{0}} \frac{\hat{B}^{0}}{\hat{B}^{i+1}} = B^{i+1}
\]
(24.20)
since \(B^{0} = \hat{B}^{0} = 1\). The effect of offsetting the short rates by \(\hat{r}_{i}^{j+1} - \hat{r}_{i}^{i+1}\) is to force simulated bond prices to equal market bond prices. The offset \(\hat{r}_{i}^{j}\) can now be used to price other interest rate products.

Note four things. First, the method can be used to match to model prices rather than market prices. This is entirely reasonable and is equivalent to a martingale correction. It decouples the method from a knowledge of the market; it is up to the model whether it fits the market or not.

Second, forcing a fit in this way destroys the conditional distributions of the \(r_{i}^{j}\). If market prices do not agree with model prices then this is a very bad idea; you might be recovering bond prices but other prices (for instance, for path-dependent options) will be biased. It seems reasonable to use moment matching if market prices are a mere perturbation away from model prices. Anything more and you are likely to be in trouble.

Third, if path-wise generation is used there could be a storage problem. Every value of \(r_{i}^{j}\) would need to be stored before any derivatives could be priced. However, just like the martingale correction case, slice-wise generation can be used since it does not have these problems.

Fourth, there is scope to fit to prices of other instruments in addition to bond prices. Suppose we set \(\hat{r}_{i}^{j} = r_{i}^{j} + p_{i}^{j} - f_{i}^{j+1}\), where \(p_{i}^{j} = \epsilon_{i}^{j} + f_{i}^{j+1}\), such that

\[
\frac{1}{M} \sum_{j=1}^{M} \hat{B}_{i+1,j} \exp \left( -\Delta t \epsilon_{i}^{j} \right) = \frac{1}{M} \sum_{j=1}^{M} \hat{B}_{i+1,j}.
\]
(24.21)

Clearly, if the \(\epsilon_{i}^{j}\) are close to 0 then they can be chosen so that bond prices are still matched. Now, however, there is the possibility to choose \(\epsilon_{i}^{j}\) so that other prices are fitted. For instance, fix \(N\) and consider bond options maturing at times \(t_{i}, i = 1, \ldots, N - 1\). Fix the strike \(X\) and write \(c_{i}\) for the model value at time \(t_{0}\) of the bond option maturing at time \(t_{i}\) with strike \(X\). First set \(\epsilon_{i}^{j} = 0\) and perturb the \(r_{i}^{j}\) to fit to bond prices, and continue to write \(r_{i}^{j}\) for the perturbed rates. Next use backwards recursion. Suppose that at time \(t_{i}\) offsets \(\{\epsilon_{k}^{j} \}_{k=i,\ldots,N-1}^{j=1,\ldots,M}\) have been found so that method values match bond prices for every \(i\) and also to bond option prices \(\hat{c}_{k} = c_{k}\) for \(k = i + 1, \ldots, N - 1\). Since

\[
\hat{c}_{i} = \frac{1}{M} \sum_{j=1}^{M} \hat{B}_{i,j}^{i} \left( \frac{\hat{B}_{N,j}^{i}}{\hat{B}_{i,j}^{i}} - X \right)^{+}
\]
(24.22)
\[
= \frac{1}{M} \sum_{j=1}^{M} X \hat{B}_{i,j}^{i} \left( \frac{\hat{B}_{N,j}^{i}}{\hat{B}_{i,j}^{i+1}} \exp \left( -\Delta t \hat{r}_{i}^{j} - \ln(X) \right) - 1 \right)^{+}
\]
(24.23)
one possibility is set
\[ \varepsilon^j_i = \begin{cases} \varepsilon^+, & r^j_i \geq -\frac{1}{\Delta t} \ln(X), \\ \varepsilon^-, & r^j_i < -\frac{1}{\Delta t} \ln(X). \end{cases} \] (24.24)

where \( \varepsilon^+ \) and \( \varepsilon^- \) have been chosen so that (i) equation (24.21) is satisfied and (ii) \( \hat{c}_i = c_i \). The recursion proceeds back to time \( t_0 \) finding a complete set of \( \varepsilon^j_i \).

To reduce bias it is important that the \( \varepsilon^j_i \) are chosen systematically so that the \( \varepsilon^j_i \) are, as a set, as close to zero as possible.

This technique enables a calibration to co-terminal bond option prices. It can be extended to matching bond options on fixed length underlyings, so a calibration to model ATM caplet prices is possible, and hence a calibration to a term structure of volatility. In fact there is nothing to prevent calibration to more than one strike at each maturity so that calibration to a smile is feasible.

Of course there are problems with the technique. The chief problem is the unintended introduction of bias elsewhere; you may match a set of chosen prices but you may now be way off on other prices. Hence the need to make perturbations as small as possible, and as spread out and balanced as possible.

There is some freedom over which instruments are chosen to match to. This may not be too much of an issue. In practice you want to match to the prices of your primary hedging instruments. Usually this means the underlyings (the FX rate, the bond prices or forward Libors, \textit{et cetera}) and the volatility (for instance matching to vanilla options or caplet prices, \textit{et cetera}).

One minor issue is that, as usual, sample paths are no longer independent so that standard errors can no longer be estimated directly. As in other examples the actual standard deviation of the Monte Carlo estimate has to be found long-hand by running the whole procedure a sufficient number of times to enable it to be computed as its sample value.

### 24.3 BIAS AND BARRIER OPTIONS

Barrier options are a classic example of instrument bias. When a barrier option knocks in or knocks out there is a large discrete change in its payoff. It is important to accurately sample the hitting time distribution. Unfortunately making a discrete time approximation does not adequately capture the distribution.

An ordinary simulation method cannot price barrier options without bias unless the number of time steps is very large. Consider the case of an up-and-out barrier option with barrier level \( U \). Given a set of sample paths, \( S^j = \{ S^j_i \}_{i=1}^{N}, j = 1, \ldots, M \), the naively computed option value is

\[ \hat{c}_0 = e^{-rT} \frac{1}{M} \sum_{m=1}^{M} h_T(S^j) \] (24.25)

where for an up-and-out call with strike \( X \)

\[ h_T(S^j) = \begin{cases} (S^j_N - X)^+, & \max_i \{ S^j_i \} < U, \\ 0, & \text{otherwise}. \end{cases} \] (24.26)

\( \hat{c}_0 \) is a high-bias estimate of the barrier option value, since the option may have been knocked out, in continuous time, in between times at which it has been observed.

When \( S \) is a GBM, Beaglehole, Dybvig and Zhou (1997) and El Babsiri and Noel (1998) devised a method of eliminating bias. Let

\[ M^S_{t,T} = \max_{s \in [t,T]} \{ S_s \mid S_t, S_T \} \] (24.27)
be the maximum of the bridge distribution over the interval \([t, T]\). For every sample path over each time step \([t_i, t_{i+1}]\) an additional simulation is made for \(M^S_{t_i, t_{i+1}}\).

Suppose that a draw \(M^j_i\) has been made from the bridge distribution \(M^S_{t_i, t_{i+1}}\) over the interval \([t_i, t_{i+1}]\) with end values \(S^j_i\) and \(S^j_{i+1}\). Set

\[
\tilde{h}_T(S^j) = \begin{cases} (S^j_N - X)^+, & \max_i \{M^j_i\} < U, \\ 0, & \text{otherwise.} \end{cases}
\]  

(24.28)

Then

\[
\hat{c}_0 = e^{-rT} \frac{1}{M} \sum_{m=1}^M \tilde{h}_T(S^j)
\]  

(24.29)

is an unbiased estimate of \(c_0\).

It is easy to sample \(M^S_{t,T}\). When \(S\) is a GBM then \(R_T = \ln(S_T/S_t)\) is a Brownian motion. Let \(M^R_{t,T} = \ln(M^S_{t,T}/S_t)\) then, setting \(U = \ln(U/S_t)\), one can show that

\[
\Pr[M^S_{t,T} \leq U \mid S_t, S_T] = \Pr[M^R_{t,T} \leq U \mid R_t, R_T]
\]  

(24.30)

\[
= 1 - \exp \left(-2 \frac{(U - R_t)(U - R_T)}{\sigma^2(T - t)}\right)
\]  

(24.31)

(see, for example, Karatzas and Shreve (1991) and El Babsiri and Noel (1998)).

The distribution (24.31) can be sampled easily using inverse transform. Let \(U \sim U[0, 1]\) be uniform, then

\[
\hat{M} = \frac{1}{2} \left( R_t + R_T + \sqrt{(R_t - R_T)^2 - 2\sigma^2(T - t) \ln(1 - U)} \right)
\]  

(24.32)

is a draw from the distribution of \(M^R_{t,T}\) and \(S_t \exp(\hat{M})\) is a draw from \(M^S_{t,T}\).

This technique effectively removes this form of bias. It clearly generalizes to other related options and some other, tractable, processes.

It may not be necessary to actually draw samples \(M^j_i\). Suppose that the probability

\[
p^j_i = \Pr[M^j_i \leq U]
\]  

(24.33)

is known, as it is when \(S_t\) follows a GBM, and suppose that for some \(j\) we have \(\max_i \{S^j_i\} < U\) so that the option has not knocked out along the \(j\)th sample path.

Set

\[
h'_T(S^j) = (S^j_N - X)^+ N^{-1} \prod_{i=0}^{N-1} p^j_i.
\]  

(24.34)

\(h'_T\) is the expected continuous barrier payoff along \(S^j\), conditional on the observations \(S^j_i\) at times \(t_i\). Setting

\[
\tilde{c}_0 = e^{-rT} \frac{1}{M} \sum_{m=1}^M h'_T(S^j)
\]  

(24.35)

leads to a much reduced standard error compared to equation (24.29).
24.4 SUMMARY

This chapter has highlighted several issues related to bias. Using the CIR process we discussed several ways in which simulation methods can go wrong. We moved on to describe some techniques that can, in certain instances, help to reduce bias, including a bias correction technique for barrier options.

The next chapter approaches a particular type of bias – discretization bias, produced when simulating an SDE that lacks an exact solution – describing a number of approximations that can be used to make simulation possible.

24.5 EXERCISES

1. Barrier option valuation.
   (a) How might the GBM bias correction method embodied by equation (24.29) be incorporated into an OOP barrier option valuation application? Implement the mechanism and verify that it works as advertised.

   (b) Implement the bias correction mechanism using equation (24.35). Compare the effectiveness of this method to that given by equation (24.29).

   (c) When $S_t$ has a Heston process it is not possible to sample exactly from the conditional maximum. Can an approximation, similar to equation (24.29), be devised? Could such an approximation reduce, if not eliminate, bias?

2. Suppose an underlying PDB matures at time $T$ with value $B_t(T)$ at time $t$. Consider a barrier option written on $B_t(T)$ maturing at time $T_1 < T$ with payoff $(B_t(T) - X)^+$, for some strike $X$, conditional upon it not having knocked out. The option knocks out if at any time $t$, $0 \leq t \leq T_1$, the yield to maturity on the bond exceeds $r^U$, that is, if $B_t(T) \leq \exp(-r^U(T - t))$.

   (a) Construct a Monte Carlo application to value this option in the Vasicek model. By investigating how the Monte Carlo value varies with the number of time steps, assess the extent of the discretization bias inherent in the Monte Carlo price.

   (b) Can a bias correction mechanism be devised, analogous to that in the GBM case?

   (c) Can the option be valued accurately in the CIR interest rate model?

3. If a Monte Carlo method values Black–Scholes European puts and calls simultaneously, but independently, how large is the deviation from put–call parity? Implement the martingale correction method given by equation (24.14).

   (a) How expensive is the method? Is it worth the effort?

   (b) Using a slice-based evolution method, what is the effect of the martingale correction on the prices of barrier options?

4. The moment-matching method applied to PDBs, equation (24.17), enforces put–call parity for European bond options. Implement the method to value European bond options in the Vasicek model.

   (a) Does it work?

   (b) What is the effect on the valuation of barrier bond options like that in exercise 2?
This chapter focuses on generating sample paths for state variables from increments in a set of driving Wiener process. If an SDE has no explicit solution then to generate sample paths an approximate method has to be used. There are a number of standard techniques based upon the application of Itô–Taylor expansions to the coefficient functions of the state variables’ SDEs. It is far better, of course, to avoid an approximate scheme if possible.

This chapter focuses on Itô–Taylor related schemes. Chapter 26 looks at moment-based schemes applied to specific examples. The main source of material for Itô–Taylor schemes is Kloeden and Platen (1995). These schemes, and others, are also discussed in Glasserman (2004), Jackel (2002) and Gatheral (2006).

The first three sections describe various important discretizations methods. Section 25.1 discusses convergence issues and presents four processes that we use subsequently as illustrations. The second section presents Itô–Taylor schemes in $Q$-dimensions, and the next looks at the forms they take in 1-dimension. Section 25.4 introduces predictor–corrector schemes.

Section 25.5 applies the approximations to two of the illustrative processes. It benchmarks with the GBM and Vasicek processes.

### 25.1 DISCRETIZATION AND CONVERGENCE

We assume that we are given a set of dates $T = \{t_i\}_{i=0,...,N}$, with $t_i < t_{i+1}$ for all $i$, and with $t_0 = 0$ and $t_N = T$. Write $\Delta_i = t_{i+1} - t_i$ and let $\delta = \max_i \{\Delta_i\}$ be the mesh size. If $\Delta_i$ is constant for all $i$ we write $\Delta$ for the common value.

Let $X_t \in \mathbb{R}^Q$ be a process in $Q$-dimensions with SDE

$$dX_t = a(X_t) \, dt + b(X_t) \, dz_t$$

(25.1)

for an $M$-dimensional Wiener process $z_t$. Our objective is to construct a discrete time simulated sample path $X^T = \{X_t\}_{t=0,...,N}$. Given a continuous time sample path of $z = \{z_t\}_{t \in [0,T]}$ we require a mechanism to produce from it a path $X^T(z)$ such that the simulated value $X_{t_N}$ is a good approximation, in some sense, to the continuous time value $X_T$. There are many possible criteria; it is natural to look at asymptotic criteria.

#### 25.1.1 Strong and weak convergence

Suppose that for some discretization scheme and criterion function $c$ we have

$$c(X_{t_N}, X_T) \leq \alpha \delta^\beta$$

(25.2)

for all mesh sizes $\delta$, then the discretization scheme converges at the rate $\beta$ with respect to the criterion $c$.

Two very important convergence criterion are

**Strong:**

$$c_s(X_{t_N}, X_T) = \mathbb{E}[|X_{t_N} - X_T|],$$

**Weak:**

$$c_w(X_{t_N}, X_T) = |\mathbb{E}[f(X_{t_N})] - \mathbb{E}[f(X_T)]|, \text{ for all } f \in \mathcal{F}.$$
Strong convergence requires \( X_{tN} \) to be close to \( X_T \) (path-wise for \( z \)). Weak convergence requires only that the distribution of \( X_{tN} \) be close to that of \( X_T \).

Weak convergence is defined with respect to some particular set of functions \( \mathcal{F} \). Typically \( f \) would need to be sufficiently differentiable, polynomial bounded, \textit{et cetera}.

Under mild conditions a given order of strong convergence may imply at least the same order of weak convergence. For derivative pricing where one is computing values of the form \( c = \mathbb{E} \left[ f(X_T) \right] \) one might expect that weak convergence would be the more appropriate measure. Unfortunately option payoff functions are often not everywhere differential and thus fail to satisfy the formal requirements of weak convergence, so the order of strong convergence is sometimes taken as the chief convergence criterion. However in practice one seems to find that weak convergence nevertheless occurs and that the weak convergence measure is still applicable despite its technical disqualification.

Both weak and strong convergence are measures of convergence of dispersion, not of bias. To measure bias one instead computes \( \mathbb{E} \left[ X_{tN} - X_T \right] \) and \( \mathbb{E} \left[ f(X_{tN}) \right] - \mathbb{E} \left[ f(X_T) \right] \).

### 25.1.2 Four examples

Let \( X = (X_1, \ldots, X_Q)' \) be the \( Q \)-dimensional process,

\[
dX_q = a^q(X) \, dt + \sum_{j=1}^{M} b_{q,j}^q(X) \, dz_j, \quad q = 1, \ldots, Q, \tag{25.4}
\]

for an \( M \)-dimensional Wiener process \( z = (z_1, \ldots, z_M)' \).

We illustrate the discretization schemes with four examples: GBM, the Vasicek Ornstein–Uhlenbeck process, and CIR in 1-dimension (with \( M = 1 \)), and Heston in \( Q \)-dimensions (with \( Q = M = 2 \)). GBM and Vasicek are extremely tractable; CIR is not, chiefly because it has no convenient strong solution. The Heston process, as you might imagine, inherits the worse aspects of each of its two 1-dimensional component processes.

Discretization schemes are often expressed in terms of derivatives of the coefficient functions \( a^q(X) \) and \( b_{q,j}^q(X) \). For convenience we list these for our example processes.

For the 1-dimensional processes we set \( a \equiv a^1 \) and \( b \equiv b_{1,1}^1 \), and let primes denote derivatives. For a 1-dimensional GBM,

\[
dX_t = rX_t \, dt + \sigma X_t \, dz_t, \tag{25.5}
\]

we have

\[
a(X) = rX, \quad a'(X) = r, \quad a''(X) = 0, \\
b(X) = \sigma X, \quad b'(X) = \sigma, \quad b''(X) = 0. \tag{25.6}
\]

The Vasicek SDE is

\[
dX_t = \alpha(\mu - X_t) \, dt + \sigma \, dz_t, \tag{25.7}
\]

so we have

\[
a(X) = \alpha(\mu - X), \quad a'(X) = -\alpha, \quad a''(X) = 0, \\
b(X) = \sigma, \quad b'(X) = 0, \quad b''(X) = 0. \tag{25.8}
\]

Similarly for the CIR process,

\[
dX_t = \alpha(\mu - X_t) \, dt + \eta \sqrt{X_t} \, dz_t, \tag{25.9}
\]
we have
\[ a(X) = \alpha(\mu - X), \quad a'(X) = -\alpha, \quad a''(X) = 0, \quad b(X) = \eta \sqrt{X}, \quad b'(X) = \frac{1}{2} \eta X^{-1/2}, \quad b''(X) = -\frac{1}{4} \eta X^{-3/2}, \quad (25.10) \]

The Heston stochastic volatility model has two state variables, \( X = (X_1, X_2)' \) (later we write \( S \) for \( X_1 \) and \( v \) for \( X_2 \)). It can be written in the form
\[
\begin{align*}
    dX_1 &= a^1(X) \, dt + b^{1,1}(X) \, dz_1 + b^{1,2}(X) \, dz_2, \\
    dX_2 &= a^2(X) \, dt + b^{2,1}(X) \, dz_2,
\end{align*}
\]
where \( z_1 \) and \( z_2 \) are independent, with
\[
\begin{align*}
    a^1(X) &= rX_1, \quad b^{1,1}(X) = \bar{\rho} \sqrt{X_2} \, X_1, \quad b^{1,2}(X) = \rho \sqrt{X_2} \, X_1, \\
    a^2(X) &= \alpha(\mu - X_2), \quad b^{2,1}(X) = 0, \quad b^{2,2}(X) = \eta \sqrt{X_2},
\end{align*}
\]
where \( \bar{\rho} = \sqrt{1 - \rho^2} \), so that \( X_1, X_2 \) are correlated with correlation \( \rho \).

In the general case we set
\[
\begin{align*}
    b^{i,j}_k &= \frac{\partial b^{i,j}}{\partial x_k}, \quad b^{i,j}_k &= \frac{\partial^2 b^{i,j}}{\partial x_k \partial x_l}, \quad \text{and} \quad a^i_k &= \frac{\partial a^i}{\partial x_k}. \quad (25.14)
\end{align*}
\]

The Heston model has \( a^1_1 = r, a^1_2 = 0, a^2_1 = 0, a^2_2 = -\alpha \) and
\[
\begin{align*}
    b^{1,1}_1 &= \bar{\rho} \sqrt{X_2}, \quad b^{1,1}_2 = \frac{1}{2} \bar{\rho} X_1, \quad b^{1,1}_{1,1} = 0, \quad b^{1,2}_{1,1} = 1, \quad b^{1,2}_{1,2} = \frac{1}{2} \bar{\rho} \, X_1, \quad b^{1,2}_{2,2} = -\frac{1}{4} \bar{\rho} \, X_3^{3/2}, \\
    b^{1,2}_1 &= \rho \sqrt{X_2}, \quad b^{1,2}_2 = \frac{1}{2} \rho X_1, \quad b^{1,2}_{1,1} = 0, \quad b^{1,2}_{1,2} = 1, \quad b^{1,2}_{2,2} = -\frac{1}{4} \rho \, X_3^{3/2},
\end{align*}
\]
where \( b^{2,2} = 0 \), \( b^{2,2}_1 = 1, b^{2,2}_{1,2} = 0, b^{2,2}_{2,2} = 1, \quad b^{2,2}_{2,2} = \frac{1}{4} \eta \).

### 25.2 ITÔ–TAYLOR DISCRETIZATION SCHEMES

Given a variable \( X = (X_1, \ldots, X_M)' \), following the process (25.4), whose exact solution we assume is unknown, and a value \( X_t \) for time \( t \), our aim is to construct a value \( X_{t+\Delta t} \), for time \( t + \Delta t \). To do this we are given a sample path for an \( M \)-dimensional Wiener process so that, for instance, the values \( \Delta z_j = z_j(t + \Delta t) - z_j(t), j = 1, \ldots, M \), are known.

From (25.4), when a strong solution to the SDE exists, we have
\[
X_q(t + \Delta t) = X_q(t) + \int_t^{t+\Delta t} a^q(X(u)) \, du + \sum_{j=1}^{M} \int_t^{t+\Delta t} b^{q,j}(X(u)) \, dz_j(u), \quad (25.16)
\]
where \( q = 1, \ldots, Q \). This representation is possible only when (25.4) has a strong solution, and for this to exist the functions \( a^q \) and \( b^{q,j} \) must obey linear growth and Lipschitz conditions.
A function \( f(x) \) satisfies the linear growth condition if for all \( x \) and some constants \( A_0 \) and \( A_1 \),
\[
|f(x)| \leq A_0 + A_1 |x|.
\] (25.17)

\( f(x) \) satisfies the Lipschitz condition if, for \( x \) and \( y \), there is a constant \( B \) such that
\[
|f(x) - f(y)| \leq B |x - y|.
\] (25.18)

These conditions place very severe restrictions on the functions \( a^q \) and \( b^{q,j} \). Their first derivatives must be bounded and they may go to infinity at most linearly in \( X \). Conditions for the existence of strong solutions to (25.4) are discussed in Øksendal (1998), Karatzas and Shreve (1991), Kloeden and Platen (1995), Glasserman (2004) and in many other places.

Discretizations stemming from approximations to the integrals in equation (25.16) using Itô–Taylor expansions are called Itô–Taylor schemes. These are the subject of this section. We discuss the Euler and Milstein schemes and an order 1.5 strong scheme. Later we present an order 2.0 scheme. These schemes are discussed in detail in Kloeden and Platen.

Note that the volatility term \( b(X) = \sigma \sqrt{X} \) in the CIR process does not obey the Lipschitz condition at zero, so, strictly, none of the Itô–Taylor schemes described here work with it. In Chapter 26 we apply them to CIR regardless, but with no assurance that either the theoretical convergence rates will be attained or indeed that any scheme will converge at all.

### 25.2.1 The Euler discretization (0.5 strong Itô–Taylor scheme)

The simplest idea is to suppose that over the small interval \([t, t + \Delta_t]\) the values of the functions \( a^q \) and \( b^{q,j} \) change little and can, as an approximation, be assumed to be constant at their time \( t \) values. This gives us
\[
X_q(t + \Delta_t) = X_q(t) + a^q(X(t)) \int_t^{t+\Delta_t} du + \sum_{j=1}^{M} b^{q,j}(X(t)) \int_t^{t+\Delta_t} dz_j(u)
\] (25.19)
\[
= X_q(t) + a^q(X(t))\Delta_t + \sum_{j=1}^{M} b^{q,j}(X(t))\Delta z_j
\] (25.20)

where \( \Delta z_j = z_j(t + \Delta_t) - z_j(t) \). This is just the Euler discretization of \( X \). To simulate a value at time \( t + \Delta_t \) from one at time \( t \), set \( X_q(t + \Delta_t) = X_q(t) + \Delta_E X_q(t) \) where
\[
\Delta_E X_q(t) = a^q(X(t))\Delta_t + \sum_{j=1}^{M} b^{q,j}(X(t))\Delta z_j, \quad \Delta z_j \sim N(0, \Delta_t).
\] (25.21)

Kloeden and Platen show that the Euler scheme is strong of order \( \frac{1}{2} \) and weak of order 1 (elaborating on the conditions under which this is true).

### 25.2.2 The Milstein discretization (1.0 strong Itô–Taylor scheme)

The Euler discretization effectively sets \( da^q = db^{q,j} = 0 \) over the interval \([t, t + \Delta_t]\). A better approximation can be obtained by investigating how \( a^q \) and \( b^{q,j} \) change over the interval \([t, t + \Delta_t]\) and making an approximation based on this.
Consider the integral \( J = \int_t^{t+\Delta_t} b^{q,j}(X(u)) \, dz_j(u) \) in equation (25.16). From Itô’s lemma

\[
\begin{align*}
    db^{q,j}(X) &= \left( \sum_{l=1}^Q b^{q,j,l}_i(X) a^l(X) + \frac{1}{2} \sum_{j_1,j_2=1}^Q b^{q,j}_{j_1,j_2}(X) \sum_{k=1}^M b^{l,j,k}(X) b^{l',k}(X) \right) \, dt \\
    &\quad + \sum_{l=1}^Q b^{q,j}_l(X) \sum_{k=1}^M b^{l,k}(X) \, dz_k,
\end{align*}
\]

so the integral is

\[
\begin{align*}
    J &= \int_t^{t+\Delta_t} \left( b^{q,j}(X(t)) + \int_t^u db^{q,j}(X(s)) \right) \, dz_j(u) \\
    &\sim b^{q,j}(X(t)) \Delta z_j + \int_t^{t+\Delta_t} \int_t^u \left( \sum_{l=1}^Q b^{q,j,l}_i(X(s)) \sum_{k=1}^M b^{l,k}(X(s)) \right) \, dz_k(s) \, dz_j(u)
\end{align*}
\]

to first order. Now freezing \( b^{q,j}_l(X(s)) \) and \( b^{l,k}(X(s)) \) at their time \( t \) values and setting

\[
I_{k,j} = \int_t^{t+\Delta_t} \int_t^u \, dz_k(s) \, dz_j(u)
\]

for the value of the iterated Itô integral, we get

\[
\int_t^{t+\Delta_t} b^{q,j}(X(u)) \, dz_j(u) \sim b^{q,j}(X(t)) \Delta z_j + \sum_{l=1}^Q \sum_{k=1}^M b^{q,j}_l(X(t)) b^{l,k}(X(t)) I_{k,j}.
\]

Running through the same analysis with \( a^q(X) \) we find that, to first order, the integral in \( a^q(X) \) in equation (25.16) remains unchanged. Hence, to first order,

\[
\Delta_M X_q = a^q(X) \Delta_t + \sum_{j=1}^M b^{q,j}(X) \Delta z_j + \sum_{j=1}^Q \sum_{k=1}^M b^{q,j}_l(X(t)) b^{l,k}(X(t)) I_{k,j}.
\]

This is the Milstein discretization scheme. To proceed one must be able to simulate the stochastic integral \( I_{k,j} \). Unfortunately exact simulation schemes, for general \( I_{k,j} \), are not known. It can be shown that

\[
\begin{align*}
    I_{j,j} &= \frac{1}{2} (\Delta z_j^2 - \Delta t), \quad k = j, \\
    I_{k,j} + I_{j,k} &= \Delta z_k \Delta z_j, \quad k \neq j.
\end{align*}
\]

(where we have written \( \Delta z_k^2 \) for \( (\Delta z_k)^2 \) and the moments of \( I_{k,j} \) are known; for instance, for \( k \neq j \), we have \( E[I_{k,j}] = \frac{1}{2} \Delta z_k \Delta z_j \).

In fact one may sometimes be able to substitute \( I_{k,j} \) with its mean, \( \frac{1}{2} \Delta z_k \Delta z_j \). When this is legitimate (see Kloeden and Platen (1995)) it is certainly a relief.
A simulation method that matches the first few moments of $I_{k,j}$, to sufficient accuracy to enable weak convergence of order 2, is the following. Simulate $I_{k,j}$ as

$$I_{k,j} \sim \frac{1}{2} (\Delta z_k \Delta z_j - v_{k,j})$$

(25.29)

where

$$v_{k,j} = \begin{cases} \Delta t, & \text{probability } \frac{1}{2}, \\ -\Delta t, & \text{probability } \frac{1}{2}. \end{cases}$$

(25.30)

Kloeden and Platen discuss conditions under which this approximation is valid.

When sufficiently fierce conditions on $a^q$ and $b^{q,j}$ are met, it can be shown (see Kloeden and Platen) that Milstein is order 1 both weak and strong.

**Milstein for $Q = M = 2$**

In the special case of $Q = M = 2$, with

$$dX_q = a^q \, dt + b^{q-1} \, d\tilde{z}_1 + b^{q-2} \, d\tilde{z}_2$$

(25.31)

for $q = 1, 2$, we have, writing out equation (25.27) in full,

$$\Delta_M X_q = a^q \Delta t + b^{q-1} \Delta \tilde{z}_1 + b^{q-2} \Delta \tilde{z}_2$$

$$+ \left( b^{1,1} b^{q-1}_1 + b^{2,1} b^{q-1}_2 \right) I_{1,1} + \left( b^{1,1} b^{q-2}_1 + b^{2,1} b^{q-2}_2 \right) I_{1,2}$$

$$+ \left( b^{1,2} b^{q-1}_1 + b^{2,2} b^{q-1}_2 \right) I_{2,1} + \left( b^{1,2} b^{q-2}_1 + b^{2,2} b^{q-2}_2 \right) I_{2,2}.$$ \hspace{1cm} \hspace{1cm} (25.32)

for $q = 1, 2$.

If $X_1$ and $X_2$ are correlated, with correlation coefficient $\rho$, then we can assume that $B = \{b^{q,j}\}_{q,j}$ is upper triangular, with $b^{2,1} = 0$ and $b^{1,2} = \phi b^{1,1}$ for a constant $\phi = \rho / \tilde{\rho}$, $\tilde{\rho} = \sqrt{1 - \rho^2}$. The discretization can then be written as

$$\Delta_M X_1 = a^1 \Delta t + b^{1,1} (\Delta \tilde{z}_1 + \phi \Delta \tilde{z}_2)$$

$$+ b^{1,1} b^{1,1}_1 (I_{1,1} + \phi (I_{1,2} + I_{2,1}) + \phi^2 I_{2,2}) + b^{2,2} b^{1,1}_2 (I_{2,1} + \phi I_{2,2}),$$

(25.33)

$$\Delta_M X_2 = a^2 \Delta t + b^{2,2} \Delta \tilde{z}_2 + b^{1,1} b^{2,2}_1 (I_{1,2} + \phi I_{2,2}) + b^{2,2} b^{2,2}_2 I_{2,2}.$$ \hspace{1cm} \hspace{1cm} (25.34)

This is the form that we apply to the Heston model in Chapter 26.

**25.2.3 The 1.5 strong Itô–Taylor scheme**

It is possible to construct higher order schemes but, as we see below, these immediately become complicated when $Q > 1$. In practice, these schemes are liable to be effective, where they work. Unfortunately they do not apply in a number of the situations that are frequently encountered in finance.
A 1.5 strong scheme (see Kloeden and Platen) is

\[
\Delta_{1.5}X_q = a^q \Delta t + \sum_{j=1}^{M} b^{q,j} \Delta z_j + \sum_{k,l=1}^{Q} \sum_{j_1,j_2=1}^{M} b^{k,j_1,j_2} b^{q,j_3} I_{j_1,j_2} \\
+ \sum_{j=1}^{M} \left( \sum_{k=1}^{Q} a^k b^{q,j}_k + \frac{1}{2} \sum_{k,l=1}^{Q} \sum_{j_1=1}^{M} b^{k,j_1,l,j_1} b^{q,j}_k b^{q,j}_k \right) I_{0,j} \\
+ \sum_{j=1}^{M} \sum_{k=1}^{Q} b^{k,j} a^q I_{j,0} \\
+ \left( \sum_{k=1}^{Q} a^k a^q_k + \frac{1}{2} \sum_{k,l=1}^{Q} \sum_{j=1}^{M} b^{k-j,l-j} a^q_{k,l} \right) I_{0,0} \\
+ \sum_{j_1,j_2,j_3=1}^{M} \sum_{k,l=1}^{Q} b^{l,j_1} \left( b^{k,j_2,j_3}_l + b^{k,j_2,j_3}_l \right) I_{j_1,j_2,j_3} \tag{25.35}
\]

where subscripts denote partial differentiation and, conditional on \( \Delta z_j, j = 1, \ldots, M \),

\[
I_{j_1,j_2,j_3} = \int_t^{t+\Delta t} \int_t^{s_2} \int_t^{s_1} dz_{j_1}(u) \, dz_{j_2}(s_1) \, dz_{j_3}(s_2), \tag{25.36}
\]

\[
I_{0,j} = \int_t^{t+\Delta t} \int_t^{s} du \, dz_{j}(s), \tag{25.37}
\]

\[
I_{j,0} = \int_t^{t+\Delta t} \int_t^{s} dz_{j}(u) \, ds, \tag{25.38}
\]

and

\[
I_{0,0} = \int_t^{t+\Delta t} \int_t^{s} du \, ds = \frac{1}{2} \Delta t^2. \tag{25.39}
\]

One can show (Kloeden and Platen) that

\[
I_{j,j,j} = \frac{1}{2} \left( \frac{1}{3} \Delta z_{j}^2 - \Delta \right) \Delta z_j, \tag{25.40}
\]

\[
I_{0,j} = \Delta z_j \Delta t - I_{j,0}, \tag{25.41}
\]

and \( I_{j,0} \) is normal, with

\[
(\Delta z_j, I_{j,0}) \sim N \left( 0, \begin{pmatrix} \frac{1}{2} \Delta t^2 & \frac{1}{2} \Delta t^2 \\ \frac{1}{2} \Delta t^2 & \frac{1}{3} \Delta t^3 \end{pmatrix} \right), \tag{25.42}
\]
so that, conditional on $\Delta z_j$, $I_{j,0}$ and $I_{0,j}$ can be simulated as

$$I_{j,0} \sim \frac{1}{2} \Delta_t \left( \Delta z_j + \frac{1}{\sqrt{3}} \Delta y \right),$$  
(25.43)

$$I_{0,j} \sim \frac{1}{2} \Delta_t \left( \Delta z_j - \frac{1}{\sqrt{3}} \Delta y \right),$$  
(25.44)

for $\Delta y \sim N(0, \Delta_t)$.

To achieve strong convergence the terms $I_{k,j}$ must also be simulated. Scheme (25.29) does not work for this purpose. Kloeden and Platen propose an approximation based on the Stratonovich integral, but we do not describe this here.

## 25.3 SCHEMES IN 1-DIMENSION

We investigate how the Itô–Taylor schemes described in section 25.2 apply in the special case of $Q = M = 1$. Later, in Chapter 27, we see how they are applied to the 2-dimensional Heston model.

### 25.3.1 The Euler and Milstein schemes in 1-dimension

For a 1-dimensional process $dX_t = a(X_t) \, dt + b(X_t) \, dz_t$ the Euler and Milstein discretizations are

$$\Delta_E X_t = a(X_t) \Delta_t + b(X_t) \Delta z_t,$$

$$\Delta_M X_t = a(X_t) \Delta_t + b(X_t) \sqrt{\Delta_t} \epsilon_t + \frac{1}{2} b(X_t) b'(X_t) (\Delta z_t^2 - \Delta_t),$$  
(25.46)

where $b'(X) = \partial b / \partial X$ and $\Delta z_t \sim N(0, \Delta_t)$.

For the 1-dimensional GBM process, the Euler and Milstein discretizations are

$$\Delta_E X_t = \mu X_t \Delta_t + \sigma X_t \Delta z_t,$$

$$\Delta_M X_t = \mu X_t \Delta_t + \sigma X_t \Delta z_t + \frac{1}{2} \sigma^2 X_t (\Delta z_t^2 - \Delta_t)$$

$$= \left( \mu - \frac{1}{2} \sigma^2 \right) X_t \Delta_t + \sigma X_t \Delta z_t + \frac{1}{2} \sigma^2 X_t \Delta z_t^2.$$  
(25.49)

For Vasicek, since $b' \equiv 0$, the Euler and the Milstein schemes are identical:

$$\Delta_E X_t = \Delta_M X_t = \alpha (\mu - X_t) \Delta_t + \sigma \Delta z_t.$$  
(25.50)

The Euler and Milstein schemes for the CIR process are

$$\Delta_E X_t = \alpha (\mu - X_t) \Delta_t + \eta \sqrt{X_t} \Delta z_t,$$

$$\Delta_M X_t = \left( \alpha (\mu - X_t) - \frac{1}{4} \eta^2 \right) \Delta_t + \eta \sqrt{X_t} \Delta z_t + \frac{1}{4} \eta^2 \Delta z_t^2.$$  
(25.52)

Note that Euler is order $\Delta_t$ in the drift but only order $\sqrt{\Delta_t}$ in the volatility (conceptually since $E[\Delta z_t^2] = \Delta_t$), and Milstein is order $\Delta_t$ in both. One expects Milstein to converge as least as fast as Euler and this is often confirmed by numerical experiment.
In practice neither of these schemes can be used directly with either CIR or GBM; these are positive processes but (depending on parameter values) the discretization may cause $X_t$ to accidentally – inevitably – become negative. To avoid this one can impose boundary conditions, perhaps in the form of flooring or reflecting the discretized process at zero or, preferably, one can discretize a transformed variable that may legitimately become negative.

**Discretizing a transformed variable**

Let $f: \mathbb{R}^+ \to \mathbb{R}$ be smooth and invertible and set $Y_t = f(X_t)$. A scheme for $Y_t$ yields a scheme for $X_t$. Set

$$X_{t+\Delta_t} = f^{-1}(Y_t + \Delta Y_t)$$

where $Y_t = f(X_t)$. $Y_t$ can safely become negative, but the transformation ensures that $X_t$ remains positive without introducing any additional bias. For some processes a convenient choice for $f$ is $f(x) = \ln(x)$.

**Using a rectification function**

Let $p: \mathbb{R} \to \mathbb{R}^+$ be a rectification function, for instance $p(x) = (x)^+$ or $p(x) = |x|$, that eliminates negative values. Suppose that $\Delta X$ is a discretization that may produce spurious negative values for a positive process $X$. To ensure that $X$ remains positive one may evolve a process $\overline{X}$, with $\overline{X}_0 = X_0$, defining $\Delta^\circ X_t = X_{t+\Delta_t} - X_t$ implicitly by

$$\overline{X}_{t+\Delta_t} = \overline{X}_t + \Delta \left( p(\overline{X}_t) \right),$$

$$X_{t+\Delta_t} = p(\overline{X}_{t+\Delta_t}).$$

Alternatively one could simply set $X_{t+\Delta_t} = p(X_t + \Delta X_t)$.

A number of schemes are compared by Lord, Koekkoek and van Dijk (2008). Whatever the choice, some bias, perhaps considerable, will be introduced. Using a rectification function should be avoided wherever possible; it is a bodge, a confession that nothing better can be done.

**25.3.2 Transforming the SDE**

As well as being used to ensure positivity, a transformation may also produce a simpler process with better convergence properties. Suppose $Y_t = g(X_t)$, where $g$ is invertible. If $Y_t$ has a simpler volatility term than $X_t$ then an Euler scheme for $Y_t$, transformed back to give a scheme for $X_t$, may have better convergence properties than Euler applied directly to $X_t$.

A trivial example is when $X_t$ is a GBM. An Euler scheme for $R_t = \ln(X_t)$ is exact, so the implied scheme for $X_t$ is also exact. If $X_t$ is a CIR process then an Euler scheme for $Y_t = \sqrt{X_t}$ is as good as a Milstein scheme for $X_t$.

In more detail, suppose that $X_t$ is CIR and set $Y_t = \sqrt{X_t}$, then the process for $Y_t$ is

$$dY_t = \frac{1}{2} \frac{1}{Y_t} \left( \alpha (\mu - Y_t^2) - \frac{1}{4} \eta^2 \right) dt + \frac{1}{2} \eta dz_t.$$  

(25.56)

An Euler discretization for $Y_t$ is

$$\Delta E Y_t = \frac{1}{2} \frac{1}{Y_t} \left( \alpha (\mu - Y_t^2) - \frac{1}{4} \eta^2 \right) \Delta t + \frac{1}{2} \eta \Delta z_t.$$  

(25.57)
Since $X_t = Y_t^2$ the scheme for $X_t$ implied by the Euler scheme for $Y_t$ is

\[
\Delta X_t = (Y_t + \Delta E Y_t)^2 - Y_t^2 = 2Y_t \Delta E Y_t + (\Delta E Y_t)^2
\]

(25.58)

\[
= \left( \alpha (\mu - Y_t^2) - \frac{1}{4} \eta^2 \right) \Delta t + \eta Y_t \Delta z_t + \frac{1}{4} \eta^2 \Delta z_t^2 + O(\Delta t^{3/2})
\]

(25.59)

\[
= \Delta_M X_t + O(\Delta t^{3/2})
\]

(25.60)

This $\Delta X_t$ is equivalent to first order to the Milstein scheme applied directly to $X_t$. However the Euler scheme in (25.57) looks just as complicated as the Milstein scheme in (25.52). Note also that just as the volatility term of $X_t$ fails the Lipschitz condition, so the drift term in (25.57) fails the linear growth condition.

One advantage to using a Milstein scheme for CIR is that it helps to reduce, or may even eliminate, negative values (Gatheral (2006)). The Milstein scheme for a CIR process, equation (25.52), can be re-expressed as

\[
X_{t+\Delta t} = \left( \sqrt{X_t} + \frac{1}{2} \eta \Delta z_t \right)^2 + \alpha (\mu - X_t) \Delta t - \frac{1}{4} \eta^2 \Delta t. \tag{25.62}
\]

It follows that $X_{t+\Delta t}$ will be positive if $X_t < \mu - \eta^2/4\alpha$. Even when $X_t \sim 0$, if

\[
d = \frac{4\alpha \mu}{\eta^2} > 1 \tag{25.63}
\]

then $X_{t+\Delta t}$ is always positive. Note the similarity of this condition to the Feller condition: the process (25.9) is guaranteed to remain strictly positive (in continuous time) if and only if $2\alpha \mu / \eta^2 > 1$.

If $X_t \geq \mu - \eta^2/4\alpha > 0$ then as long as $\frac{1}{2} \eta^2 \Delta t \ll X_t$ it follows that $X_{t+\Delta t}$ remains positive with high probability.

If $X_t$ is a process for the short interest rate with parameters $\alpha = 0.2$, $\mu = 0.05$, $\eta = 0.1$, say, then $d = 4 > 1$ and condition (25.63) holds. However if $X_t$ is a volatility process, perhaps with $\alpha = 0.4$, $\mu = 0.04$, $\eta = 0.4$, then $d = 0.4 < 1$ so the condition does not hold and the discretization may in principle generate negative values of $X_{t+\Delta t}$, even if the time step is small.

### 25.3.3 1.5 strong Itô–Taylor with $Q = M = 1$

For the 1-dimensional process

\[
dX_t = a(X_t) \, dt + b(X_t) \, dz_t \tag{25.64}
\]

the 1.5 strong Itô–Taylor scheme becomes

\[
\Delta_{1.5} X_t = a \Delta t + b \Delta z_t + bb' I_{1,1} + ba' I_{1,0} + \left( ab' + \frac{1}{2} b^2 b'' \right) I_{0,1} + \left( aa' + \frac{1}{2} b^2 a'' \right) I_{0,0} + (b(b')^2 + b^2 b'') I_{1,1,1,1} \tag{25.65}
\]
where primes denote differentiation. Using equations (25.40), (25.41), (25.39) and (25.43), one can simulate $X_t$ as

$$\Delta_{1.5} X_t = a \Delta_t + b \Delta z_t + \frac{1}{2} b b' (\Delta z_t^2 - \Delta_t)$$

$$+ \frac{1}{2} b a' \Delta_t \left( \Delta z_t + \frac{1}{\sqrt{3}} \Delta y \right) + \frac{1}{2} \left( a b' + \frac{1}{2} b^2 b'' \right) \Delta_t \left( \Delta z_t - \frac{1}{\sqrt{3}} \Delta y \right)$$

$$+ \frac{1}{2} \left( a a' + \frac{1}{2} b^2 a'' \right) \Delta_t^2$$

$$+ \frac{1}{2} \left( b (b')^2 + b^2 b'' \right) \left( \frac{1}{3} \Delta z_t^2 - \Delta_t \right) \Delta z_t \quad (25.66)$$

where $\Delta y \sim N(0, \Delta_t)$. In 1-dimension this simulation is easy since it does not involve difficult cross-terms in $I_{k,l}$, $k, l \neq 0$.

### 1.5 strong Itô–Taylor for GBM

For GBM one has $a' b = ab'$ and $b'' = 0$ so, using (25.41), the terms in equation (25.65) in $I_{1,0}$ drop out. Since also $a'' = 0$ equation (25.65) reduces to

$$\Delta_{1.5} X_t = a \Delta_t + b \Delta z_t + bb' I_{1,1}$$

$$+ ab' \Delta z_t \Delta_t + aa' I_{0,0}$$

$$+ b(b')^2 I_{1,1,1}. \quad (25.67)$$

so, for a GBM we arrive at

$$\Delta_{1.5} X_t = \left( r - \frac{1}{2} \sigma^2 \right) X_t \Delta_t + \sigma X_t \Delta z_t + \frac{1}{2} \sigma^2 X_t \Delta z_t^2 + \frac{1}{6} \sigma^3 X_t \Delta z_t^3$$

$$+ \frac{1}{2} r^2 X_t \Delta_t^2 + \left( r \sigma - \frac{1}{2} \sigma^3 \right) X_t \Delta z_t \Delta_t. \quad (25.68)$$

### 1.5 strong Itô–Taylor for Vasicek

Equation (25.65) simplifies to

$$\Delta_{1.5} X_t = a \Delta_t + b \Delta z_t + ba' I_{1,0} + aa' I_{0,0} \quad (25.69)$$

$$= \left( 1 - \frac{1}{2} \alpha \Delta_t \right) (\alpha(\mu - X_t) \Delta_t + \sigma \Delta z_t) - \frac{1}{2 \sqrt{3}} \sigma \alpha \Delta_t \Delta y \quad (25.70)$$

for $\Delta y \sim N(0, \Delta_t)$. 

1.5 strong Itô–Taylor for CIR

Applying equation (25.66) to the CIR process (25.9), there are some cancellations. For CIR, \( b(b')^2 + b^2 b'' = 0 \) so the term in \( I_{1,1,1} \) drops out and one obtains

\[
\Delta_{1.5} X_t = a \Delta_t + b \Delta z_t + bb'I_{1,1,1} + aa'I_{0,0} \\
+ ba' \Delta z_t \Delta_t + \left( ab' - ba' + \frac{1}{2} b^2 b'' \right) I_{0,1}
\]

\[
= \Delta_t \left( \alpha \mu - \frac{1}{4} \eta^2 - \frac{1}{2} \sigma^2 \right) + \Delta_t \left( \frac{1}{2} \sigma^2 \Delta_t - \alpha \right) X_t + \eta \sqrt{X_t} \Delta z_t \\
- \alpha \eta \sqrt{X_t} \Delta z_t \Delta_t + \frac{1}{4} \eta^2 \Delta z_t^2 \\
+ \frac{1}{4} \eta \left( \left( \alpha \mu - \frac{1}{4} \eta^2 \right) \frac{1}{\sqrt{X_t}} + \alpha \sqrt{X_t} \right) \Delta_t \left( \Delta z_t - \frac{1}{\sqrt{3}} \Delta y \right)
\]

(25.71)

for \( \Delta y \sim \text{N}(0, \Delta_t) \).

In the CIR process, as we have observed, the volatility term is not Lipschitz. When \( X_t \) gets close to zero, as inevitably it will, the derivative \( b'(X) \) becomes unbounded. This is reflected in the scheme (25.72). The term in \( 1/\sqrt{X_t} \) prevents any hope of good behaviour in a numerical implementation, unless some adjustment (bodge) is made.

25.3.4 A 2.0 weak scheme in 1-dimension

The discretization scheme \( \Delta_{2.0}^w \),

\[
\Delta_{2.0}^w X_t = a \Delta_t + b \Delta z_t + bb'I_{1,1,1} + a'bI_{1,0} + \left( ab' + \frac{1}{2} b^2 b'' \right) (\Delta z_t \Delta_t - I_{1,0}) \\
+ \left( aa' + \frac{1}{2} a'' b^2 \right) I_{0,0,0},
\]

(25.73)

converges weakly to second order (Kloeden and Platen). This is just equation (25.65) with the term in the third-order integral \( I_{1,1,1} \) omitted.

For GBM, as before, the terms in equation (25.73) in \( I_{1,0} \) drop out, and setting \( a'' = 0 \) we have

\[
\Delta_{2.0}^w X_t = a \Delta_t + b \Delta z_t + \frac{1}{2} bb'(\Delta z_t^2 - \Delta_t), \\
+ \frac{1}{2} aa' \Delta_t^2 + ab' \Delta z_t \Delta_t.
\]

(25.74)

This results in

\[
\Delta_{2.0}^w X_t = \left( r + \frac{1}{2} r^2 \Delta_t - \frac{1}{2} \sigma^2 \right) X_t \Delta_t \\
+ \sigma (1 + r \Delta_t) X_t \Delta z_t + \frac{1}{2} \sigma^2 X_t \Delta z_t^2
\]

(25.75)

hardly more complicated that Milstein.
For the Vasicek process the scheme is just the 1.5 strong scheme (equation (25.70)) without the term in $\Delta y$ that simulates $I_{1,0}$,

$$\Delta_{2,0}^w X_t = \left(1 - \frac{1}{2} \alpha \Delta_t \right) \left(\alpha(\mu - X_t) \Delta_t + \sigma \Delta z_t\right).$$  \hspace{1cm} (25.76)

This is a simple bias correction to the standard Euler scheme.

In a weak scheme only the moment properties are important. In Kloeden and Platen’s simplified weak scheme $I_{1,0}$ is replaced with its expected value $\frac{1}{2} \Delta z_t \Delta_t$, and so one obtains

$$\Delta_{2,0}^w X_t = a \Delta_t + b \Delta z_t + \frac{1}{2} b b' (\Delta z_t^2 - \Delta_t),$$

$$+ \frac{1}{2} \left(ab' + a'b + \frac{1}{2} b''b^2\right) \Delta z_t \Delta_t$$

$$+ \frac{1}{2} \left(aa' + \frac{1}{2} a''b^2\right) \Delta_t^2.$$ \hspace{1cm} (25.77)

Applying this to the CIR process one finds

$$\Delta_{2,0}^w X_t = \Delta_t \left(\alpha \mu - \frac{1}{4} \eta^2 - \frac{1}{2} \alpha^2 \mu \Delta_t \right) + \left(\frac{1}{2} \alpha^2 \Delta_t - \alpha\right) \Delta_t X_t$$

$$+ \Delta z_t \left[\eta \left(1 - \frac{3}{4} \alpha \Delta_t \right) \sqrt{X_t} + \frac{1}{4} \eta \left(\alpha \mu - \frac{1}{4} \eta^2\right) \Delta_t \frac{1}{\sqrt{X_t}}\right]$$

$$+ \frac{1}{4} \eta^2 \Delta z_t^2.$$ \hspace{1cm} (25.78)

Just as in the 1.5 strong scheme (25.72), the term in $1/\sqrt{X_t}$ in (25.78) causes the scheme to blow up making it unworkable without modification.

### 25.4 PREDICTOR–CORRECTOR SIMULATION

Predictor–corrector schemes are alternative methods to correct the bias in the Euler scheme caused by the changing values of the drift and volatility over a time interval. They are weak of order 1 (Kloeden and Platen).

First consider the 1-dimensional case. Suppose that $dX_t = a(X_t) \, dt + b(X_t) \, dz_t$ and set

$$X_{t+\Delta t} = X_t + a(X_t) \Delta t + b(X_t) \Delta z_t.$$ \hspace{1cm} (25.79)

$X_{t+\Delta t}$ is the usual Euler value for $X$ at the next time step. Now it is used only as an approximation, a predictor, of the value of $X$ at the next step. This is used to compensate for the change in the value of $X$ over the interval $[t, t + \Delta t]$ by constructing better approximations to the integrals in the drifts and volatilities.
Choose weights $0 \leq w_a, w_b \leq 1$ and set
\[
X_{t+\Delta_t} = X_t + [w_a \underline{a}(X_{t+\Delta_t}) + (1-w_a)\bar{a}(X_t)]\Delta_t
+ [w_b \underline{b}(X_{t+\Delta_t}) + (1-w_b)b(X_t)]\Delta z_t
\]
\[
= X_{t+\Delta_t} - w_b \bar{b}(X_t) \frac{\partial \bar{b}}{\partial X}(X_t)\Delta_t
+ w_a[\underline{a}(X_{t+\Delta_t}) - \bar{a}(X_t)]\Delta t + w_b[b(X_{t+\Delta_t}) - b(X_t)]\Delta z_t
\] (25.80)

where $\underline{a} = a - w_b b(\partial b/\partial X)$. $X_{t+\Delta_t}$ is the simulated value corrected by altering the drift and volatility based upon a predicted value $\overline{X}_{t+\Delta_t}$ of $X$ for time $t + \Delta_t$. (The adjustment of $a$ to $\underline{a}$ is discussed in Kloeden and Platen.)

One is free to choose the weights $w_a$ and $w_b$ but often they are set to be $w_a = w_b = \frac{1}{2}$. The weights fix the balance between the explicit and implicit parts of the discretization.

In $Q$-dimensions things are a little more complicated, as usual, but not too bad. Let $X_t$ be a $Q$-dimensional process and $z_t$ an $M$-dimensional Wiener process with
\[
dX_t = a(X_t) \, dt + b(X_t) \, dz_t
\] (25.82)

where $a(X_t) = (a^1(X_t), \ldots, a^Q(X_t))^\prime$ and $b(X_t) = \{b^{i,j}(X_t)\}_{i,j=1,\ldots,M}^{q=1,\ldots,Q}$.

For the predictor we use Euler,
\[
\overline{X}_{t+\Delta_t} = X_t + a(X_t)\Delta t + b(X_t)\Delta z_t,
\] (25.83)
as before, where now $\Delta z_t = (\Delta z_t^1, \ldots, \Delta z_t^M)^\prime$ is an increment in $M$ dimensions. For the corrector step choose weights $w_a$ and $w_b$ (the same weight for every drift and for every volatility) and set
\[
X_{t+\Delta_t} = X_t + [w_a \underline{a}(X_{t+\Delta_t}) + (1-w_a)\bar{a}(X_t)]\Delta t
+ [w_b \underline{b}(X_{t+\Delta_t}) + (1-w_b)b(X_t)]\Delta z_t
\] (25.84)

where $\underline{a} = (\underline{a}^1, \ldots, \underline{a}^Q)^\prime$ with
\[
\underline{a}^q(x) = a^q(x) - w_b \sum_{i=1,\ldots,Q} b^{i,j}(x)b^{q,j}(x), \quad q = 1, \ldots, Q,
\] (25.85)
and $b^{q,j}(x) = (\partial/\partial x^i) b^{q,j}(x)$ as usual.

Predictor–corrector schemes for the three illustrative 1-dimensional processes are very straightforward.

**Predictor–corrector for GBM**

We have
\[
\underline{a}(X) = a(X) - w_b b(X) \frac{\partial b}{\partial X} = (r - w_b \sigma^2)X
\] (25.86)
so
\[
X_{t+\Delta_t} = \overline{X}_{t+\Delta_t} - w_b \sigma^2 X_t \Delta t + w_a(r - w_b \sigma^2)(\overline{X}_{t+\Delta_t} - X_t)\Delta t
+ w_b \sigma (\overline{X}_{t+\Delta_t} - X_t)\Delta z_t
\]
\[
= X_t + (1 + w_a(r - w_b \sigma^2)\Delta t + w_b \sigma \Delta z_t)\Delta E X_t - w_b \sigma^2 X_t \Delta t.
\] (25.87)
**Predictor–corrector for Vasicek**

We have \( \bar{a}(X) \equiv a(X) \) since \( \partial b / \partial X = 0 \). Hence, setting \( X_{t+\Delta t} = X_t + \Delta E X_t \) to be an Euler predictor,

\[
X_{t+\Delta t} = X_t + \Delta E X_t, \quad (25.89)
\]

\[
= X_t + (1 - w_a \alpha) \Delta E X_t, \quad (25.90)
\]

Note that when \( w_a = \frac{1}{2} \), this is identical to the 2.0 weak scheme.

**Predictor–corrector for CIR**

The modified drift is

\[
\bar{a}(X) = a(X) - w b(X) \frac{\partial b}{\partial X} = a(X) - \frac{1}{2} w b \eta^2, \quad (25.91)
\]

hence

\[
X_{t+\Delta t} = X_t + \frac{1}{2} w b \eta^2 \Delta t - w_a \alpha [X_{t+\Delta t} - X_t] \Delta t
\]

\[
+ w b \eta \left[ \sqrt{X_{t+\Delta t}} - \sqrt{X_t} \right] \Delta z_t, \quad (25.92)
\]

where the predictor \( X_{t+\Delta t} \) is Euler as usual.

### 25.5 NUMERICAL ASSESSMENT FOR BENCHMARK PROCESSES

GBM and Vasicek have exact solutions to their SDEs so they can be used to benchmark the various discretization schemes we have encountered. This can guide us when we look at Heston in Chapter 26. We find that, as a whole, the methods behave as expected.

\( M = 100,000 \) sample paths were generated. Wiener sample paths were stratified at the final time and then filled by Brownian bridge. Each path \( \{ z_i \}_{i=0,...,N} \) was evolved up to time \( T = 3 \) in \( N = 2^{10} \) steps. On the \( K \)th run, for \( K = 0, \ldots, 10 \), paths were sampled at intervals \( 2^K \) so that by comparing results as \( K \) varied the rate of convergence could be estimated.

Three functions were used to test weak convergence: the call payoff function, \( f^c(S) = (S - X)^+ \) (with \( X = 100 \)); the identity function \( f^M(S) = S \), returning the mean value; and the variance function, \( f^V(S) = (S - \mu)^2 \) where \( \mu \) is the exact expected value of \( S \). Only the second and third functions were used with the Vasicek process.

**GBM**

The spreadsheet MC_convergence_GBM.xls implements a number of discretization schemes for GBM. Parameter values were \( r = 0.05 \), \( \sigma = 0.2 \) and \( S_0 = 100 \).

Panel (a) of Table 25.1 summarizes convergence results for GBM. It shows convergence rates and also the absolute error at the finest resolution.

Figure 25.1 plots \( \log_2(\text{error}) \) against \( K = \log_2(N) \) where \( \text{error} \) is \( c_e \) in the top panel and \( c_w \) in the bottom (for the call function \( f^c \)). The rates of convergence are the slopes of the lines.

Convergence is generally regular and consistent with our expectations. Everything works as anticipated for this tractable process. Euler, Milstein and 1.5–strong all converge at around their theoretical rates. The
**Table 25.1** Comparison of discretization schemes: GBM and Vasicek

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Equation</th>
<th>Convergence rate</th>
<th>Convergence rate</th>
<th>Absolute error</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Strong</td>
<td>Weak</td>
<td>Strong</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call  Mean  Var.</td>
<td>Call  Mean  Var.</td>
<td>Call  Mean  Var.</td>
</tr>
<tr>
<td>Euler</td>
<td>(25.47)</td>
<td>0.48  1.06  0.98</td>
<td>0.94</td>
<td>0.24</td>
</tr>
<tr>
<td>Milstein</td>
<td>(25.49)</td>
<td>0.96  0.98  0.98</td>
<td>0.97</td>
<td>0.0051</td>
</tr>
<tr>
<td>1.5 strong</td>
<td>(25.67)</td>
<td>1.44  1.82  1.88</td>
<td>2.22</td>
<td>3.7 x 10^-5</td>
</tr>
<tr>
<td>2.0 weak</td>
<td>(25.75)</td>
<td>0.96  1.41  1.33</td>
<td>1.59</td>
<td>0.0015</td>
</tr>
<tr>
<td>P-C</td>
<td>(25.87)</td>
<td>0.91  0.94  0.99</td>
<td>1.01</td>
<td>0.0018</td>
</tr>
</tbody>
</table>

Panel (a): Convergence, GBM

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Equation</th>
<th>Convergence rate</th>
<th>Convergence rate</th>
<th>Absolute error</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Strong</td>
<td>Weak</td>
<td>Strong</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call  Mean  Var.</td>
<td>Call  Mean  Var.</td>
<td>Call  Mean  Var.</td>
</tr>
<tr>
<td>Euler/Milstein</td>
<td>(25.50)</td>
<td>1.03  1.01  1.01</td>
<td>1.01</td>
<td>5.0 x 10^-7</td>
</tr>
<tr>
<td>1.5 strong</td>
<td>(25.70)</td>
<td>1.05  1.49  1.50</td>
<td>1.50</td>
<td>3.0 x 10^-7</td>
</tr>
<tr>
<td>2.0 weak</td>
<td>(25.76)</td>
<td>2.39  2.08  2.02</td>
<td>2.02</td>
<td>2.1 x 10^-11</td>
</tr>
<tr>
<td>P-C (0.25)</td>
<td>(25.90)</td>
<td>1.07  1.01  1.00</td>
<td>1.00</td>
<td>2.5 x 10^-7</td>
</tr>
</tbody>
</table>

Panel (b): Convergence, Vasicek

predictor–corrector method converges in theory at order 1.0, and this is verified. The simplified 2.0 weak scheme appears to converge strongly at 1.0 and weakly at 1.5.

In this example the 1.5 strong scheme outperforms the other methods. Not only does it converge at a faster rate, it also has the smallest errors at the final time. The differences in errors are considerable. The error reduction over the Euler scheme is a factor of over 4000 for the call payoff function and almost 80000 for the variance function. The next best performer, the 2.0 weak scheme, is worse by a factor of 40 or so.

Compared to Euler, Milstein has significantly better strong convergence, as expected, but very similar weak performance. Predictor–corrector is a little better than Milstein, by a factor of maybe 2 or 3, but nothing spectacular.

**Vasicek**

The spreadsheet MC_convergence_OU.xls was used to compute convergence results for the Vasicek O-U process. Parameter values appropriate for an interest rate model were used: $\alpha = 0.2$, $\mu = 0.05$, $\sigma = 0.02$ and $r_0 = 0.04$. Four schemes were compared and the results are given in Table 25.1, panel (b). Since the predictor–corrector scheme with $w_a = \frac{1}{2}$ is equivalent to the 2.0 weak scheme, the table shows results for predictor–corrector scheme with $w_a = \frac{2}{4}$.

The Euler/Milstein and the predictor–corrector schemes converge order 1.0 both weak and strong, as expected. The 1.5 strong scheme converges order 1.5 weakly but only order 1.0 strong. The 2.0 weak scheme converges order 2.0 both weak and strong.

These results are reflected in the absolute errors. With $N = 2^{10}$ steps up to time $T = 3$ the error for the 2.0 weak scheme is a factor of $10^4$ less than the Euler/Milstein scheme. The 1.5 strong scheme has weak error several hundreds of times better than Euler/Milstein but the error on the strong test is only slightly less.
Panel (a): GBM, strong convergence

Panel (b): GBM, weak convergence

Figure 25.1 Convergence, GBM
25.6 SUMMARY

It is usually possible to find some kind of discretization scheme for any (reasonable) stochastic process, but this may not be an Itô–Taylor based scheme. The issue is not only about the rate of convergence but also about the standard error and the bias, and indeed whether a scheme can be applied successfully in the first place.

Euler usually works for processes met in finance but converges only slowly with possibly high error. It is usually possible to find a better scheme, even with the same rate of convergence, that better matches the distribution of the underlying state variable. Milstein is not always an improvement. In our benchmark cases all the Itô–Taylor based schemes work well and higher order convergence is achieved.

Higher order Itô–Taylor schemes are complex; they can work well when they work at all but with some common processes, such as CIR, that do not satisfy their conditions, they may even blow up. Correcting for this is awkward, and is unsatisfactory.

25.7 EXERCISES

Consider the following processes

(i) \( dX_t = \alpha(\mu - X_t^2) \, dt + \sigma \, dz_t, \quad 0 < \alpha, \)
(ii) \( dX_t = (a - X_t)(X_t - b) \, dt + \sigma \, dz_t, \quad 0 < a < b, \)
(iii) \( dX_t = \sin(\alpha X_t) \, dt + \sigma \, dz_t, \)
(iv) \( \frac{dX_t}{X_t} = \alpha(\mu - X_t) \, dt + \sigma \, dz_t, \quad 0 < \alpha, \)
(v) \( dX_t = \mu X_t \, dt + (a - X_t)(X_t - b) \, dz_t, \quad 0 < a < b, \)

and the following discretization schemes

(a) Euler, \( \text{equation (25.45)}, \)
(b) Milstein, \( \text{equation (25.46)}, \)
(c) 1.5 strong, \( \text{equation (25.66)}, \)
(d) 2.0 weak, \( \text{equation (25.73)}, \)
(e) Predictor–corrector, \( \text{equation (25.81)}. \)

1. How does each scheme apply to each process? In their implementations are any bodges required? Why are these needed? Can you relate these to failures in the linear growth or Lipschitz conditions?

2. How effective is each scheme for each process. Construct plots (similar to Figure 25.1) and estimate the rates of convergence for each scheme. Do the schemes attain their theoretical convergence rates? Are there circumstances in which any of the schemes blow up? (To test weak convergence use the three test functions \( f^c, f^M, \text{and} f^V \) as in section 25.5.)

3. Which scheme seems best for each process? In general how would you assess the performance of each scheme across the processes as a set?

4. The analysis of the GBM schemes in Table 25.1 is for an at-the-money option. Do the methods continue to work as effectively for in-the-money or out-of-the-money options? Repeat the exercise for strikes \( X \in \{80, 90, 110, 120\} \). What conclusions do you draw?
Chapter 25 examined Itô–Taylor based discretization schemes. This chapter looks at schemes based loosely on moment matching. The approach taken here is not systematic but more case-study oriented.

We investigate two examples. Section 26.1 looks at the CIR process. It develops moment-matching schemes and applies them, along with some Itô–Taylor schemes, presenting numerical results. Section 26.2 investigates the approximation of discount factors by simulation in the Vasicek model.

We find that simple schemes often outperform more complex schemes. We find that a log-normal moment-freezing scheme for CIR works well in our examples, and that to simulate discount factors a trapezium-based integration scheme is simple, fast and accurate.

### 26.1 THE CIR PROCESS

This process is a minor test-bed for simulation techniques. It has known distributions although they are expensive to simulate. We describe a number of schemes for this process and compare their performance. A useful survey article is Andersen, Jäckel and Kahl (2009) and various schemes are also compared by Lord, Koekkoek and van Dijk (2008).

A strong solution to the CIR SDE cannot be constructed. This means that it is not possible to generate an exact CIR process sample path, in a natural way, from a Wiener sample path. The speed-up methods in Part VI, based on Wiener sample paths, can be used only with approximate schemes. It is therefore extremely important to understand the behaviour of approximate schemes, not only to reduce bias, but also to enable fast Monte Carlo to be implemented with confidence.

We look at exact simulation (accurate, slow, not amenable to speed-ups); normal approximation (easy but less accurate); log-Euler approximation (unusable in this case); moment freezing (perhaps better than normal); log-normal approximation (first order and reasonable); and 2.0 order weak (first order, but bodged). Numerical results are given. We find that rates of convergence are sensitive to the parameters of the process; in particular it appears that the difficulty of numerical simulation is influenced by whether or not the parameters of the processes satisfy the Feller condition.

Anticipating its application in the Heston model as a stochastic volatility we write the CIR process as

\[
dv_t = \alpha(\mu - v_t) \, dt + \eta \sqrt{v_t} \, dz_t,
\]  

(26.1)

using \( v \) as the symbol for the state variable.

**Exact simulation for \( v_t \)**

When \( v_t \) follows a CIR process it is possible to simulate exactly from the distribution of \( v_{t+\Delta_t} \mid v_t, v_{t+\Delta_t} \) has a non-central \( \chi^2 \) distribution. Set

\[
d = \frac{4\alpha \mu}{\eta^2}, \quad \lambda = \frac{4\alpha}{\eta^2 (1 - e^{-\alpha \Delta_t})} v_t,
\]  

(26.2)

(26.3)
then

$$\Pr [v_{t+\Delta t} \leq v | v_t] = F_{d,\lambda}^{\chi^2}(\frac{4\alpha}{\eta^2(1-e^{-\alpha\Delta t})} v)$$  \hspace{1cm} (26.4)$$

where $F_{d,\lambda}^{\chi^2}$ is the non-central $\chi^2$ distribution function for a variable with $d$ degrees of freedom and non-centrality parameter $\lambda$.

This can be sampled directly. Set $c = \eta^2(1-e^{-\alpha\Delta t})/4\alpha$. It is convenient to separate out the case when $d \leq 1$.

For $d > 1$ let $Z \sim N(0, 1)$ and $X \sim \chi^2_{d-1}$ be a draw from a $\chi^2$ distribution with $d-1$ degrees of freedom, then set

$$v_{t+\Delta t} = c \left( X + \left( Z + \sqrt{\lambda} \right)^2 \right).$$  \hspace{1cm} (26.5)$$

When $d \leq 1$ let $N \sim \text{Poisson}(\lambda/2)$ be a Poisson variate with mean $\lambda/2$ and let $X \sim \chi^2_{d+2N}$, then set

$$v_{t+\Delta t} = cX.$$  \hspace{1cm} (26.6)$$

(One can simulate a $\chi^2_v$ as a gamma variate, $\chi^2_v \sim \gamma(v/2,2)$.)

Note that this $d$ also appears in equation (25.63), Chapter 25. When $d > 1$ the generated value $v_{t+\Delta t}$ is strictly positive.

This method is exact but it is slow – unfortunately too slow for routine use when short steps are needed, but plausible when long-step Monte Carlo is possible.

### Normal and log-normal approximations

The first and second moments of $v_t$ are known. They are

$$m = \mathbb{E}[v_{t+\Delta t} | v_t] = v_t e^{-\alpha\Delta t} + \mu \left( 1 - e^{-\alpha\Delta t} \right),$$  \hspace{1cm} (26.7)$$

$$s^2 = \text{var} [v_{t+\Delta t} | v_t] = v_t \frac{\eta^2}{\alpha} \left( e^{-\alpha\Delta t} - e^{-2\alpha\Delta t} \right) + \mu \frac{\eta^2}{2\alpha} \left( 1 - e^{-\alpha\Delta t} \right)^2.$$  \hspace{1cm} (26.8)$$

Using this distributional information one can simulate by the approximation $v_{t+\Delta t} | v_t \sim N(m, s^2)$, so that

$$v_{t+\Delta t} = m + s\varepsilon, \quad \varepsilon \sim N(0, 1).$$  \hspace{1cm} (26.9)$$

This is unbiased and cheap, and potentially more accurate than Euler which approximates $v_{t+\Delta t}$ as a normal variate but with the wrong moments. Unfortunately a symmetric distribution is not able to match a highly skewed distribution like $v_{t+\Delta t} | v_t$ very well at all.

A log-normal approximation supposes that $v_{t+\Delta t} | v_t \sim LN(\mu, \sigma^2)$ for some mean $\mu$ and variance $\sigma^2$, so that $v_{t+\Delta t}$ can be simulated as $v_{t+\Delta t} = \exp (\mu + \sigma \varepsilon)$ for some $\varepsilon \sim N(0, 1)$. The mean and variance of a log-normal distributed variate $X \sim LN(\mu, \sigma^2)$ are

$$\mathbb{E}[X] = \exp \left( \mu + \frac{1}{2} \sigma^2 \right),$$  \hspace{1cm} (26.10)$$

$$\text{var}[X] = \left( \exp (\sigma^2) - 1 \right) \exp (2\mu + \sigma^2).$$  \hspace{1cm} (26.11)$$
Given values $m$ and $s^2$ for $\mathbb{E}[X]$ and $\text{var}[X]$ one obtains

$$
\sigma^2 = \ln \left( 1 + \frac{s^2}{m^2} \right),
$$
(26.12)

$$
\mu = \ln (m) - \frac{1}{2} \ln \left( 1 + \frac{s^2}{m^2} \right),
$$
(26.13)

so $v_{t+\Delta t}$ can be simulated as

$$
v_{t+1} = m \exp \left( -\frac{1}{2} \sigma^2 + \sigma \varepsilon \right), \quad \varepsilon \sim \mathcal{N}(0, 1),
$$
(26.14)

where $\sigma = \sqrt{\ln (1 + s^2/m^2)}$.

This has the advantage of matching the heavily skewed distribution of $v_{t+\Delta t} \mid v_t$ a little better than the symmetric normal approximation.

**Log-Euler approximation**

This is a simple transformation scheme. Set $Y_t = -\ln(v_t)$. Then the process for $Y_t$ is

$$
dY_t = \left( \alpha - \left( \alpha \mu - \frac{1}{2} \eta^2 \right) e^{Y_t} \right) dt - \eta e^{\frac{1}{2} Y_t} d\zeta_t.
$$
(26.15)

An Euler discretization for $Y_t$,

$$
\Delta_E Y_t = \left( \alpha - \left( \alpha \mu - \frac{1}{2} \eta^2 \right) e^{Y_t} \right) \Delta_t - \eta e^{\frac{1}{2} Y_t} \Delta \zeta_t,
$$
(26.16)

yields a discretization for $v_t$ in the usual way. Note that if the Feller condition is not met then the coefficient of $e^{Y_t}$ in the drift is positive and the process is tending to increase

$$
\Delta v_t = e^{-Y_{t+\Delta t}} - e^{-Y_t} = e^{-Y_t} (e^{-\Delta_E Y_t} - 1).
$$
(26.17)

One should not expect this scheme to perform better than a normal approximation.

**Moment freezing**

Given $v_t$ one freezes the volatility over the interval $[t, t + \Delta t]$ at its time $t$ value and evolves using exact Gaussian moments based on it. For the process $d v_t = \alpha (\mu - v_t) \, dt + \sigma \, d\zeta_t$ we have

$$
m = \mathbb{E}[v_{t+\Delta t} \mid v_t] = v_t e^{-\alpha \Delta t} + \mu \left( 1 - e^{-\alpha \Delta t} \right),
$$
(26.18)

$$
s^2 = \text{var} [v_{t+\Delta t} \mid v_t] = \frac{\sigma^2}{2\alpha} \left( 1 - e^{-2\alpha \Delta t} \right),
$$
(26.19)

so this method evolves $v_t$ by setting

$$
v_{t+\Delta t} = m + s^2 \varepsilon, \quad \varepsilon \sim \mathcal{N}(0, 1)
$$
(26.20)
where $s'$ is obtained from $s$ by replacing $\sigma$ with $\eta \sqrt{v_t}$,

$$ (s')^2 = \frac{\eta^2}{2\alpha} \left( 1 - e^{-2\alpha \Delta t} \right) v_t. $$  

### Implementation issues

The spreadsheet MC_convergence_CIR_simul.xls implements a number of discretization schemes for CIR. Most implementations are straightforward and require no explanation. Unfortunately several are problematic. In practice the method they embody cannot be used with confidence with the CIR process.

The log-Euler scheme blows up when it attempts to compute $\exp(Y)$ with too large a value of $Y$. Figure 26.1 shows the procedure `GetFinalS()` in the class module `PCIRlnEuler` that evolves CIR by log-Euler. $dt_\text{inc}$ is the length of time corresponding to an increment of $\text{inc}_\text{time steps}$.

The evolution is bodged by explicitly incorporating a cap on the value of $Y$. Even so the method explodes with some (reasonable) combinations of parameters. It should not be used.

The 1.5 strong and 2.0 weak schemes also need adjustment. The implementation of the 2.0 weak scheme is given in Figure 26.2. It uses not one bodge but two combined; a rectification function `my_pos()` is used with a floor on the value of $v$ to prevent negative or zero values occurring.

The choice of the fudge parameter $s_bounce$ is delicate. If it is either too small or too large, relative to the parameters of the model, then the process either blows up or becomes heavily biased. However, the mere fact of its existence is sufficient to condemn the method. As implemented, these schemes should be avoided if the parameter values can change.

### Comparison of simulation techniques

Table 26.1 summarizes the results. Since there is no practical strong solution only weak convergence is shown. Two cases are given, $d < 1$ and $d > 1$, for the functions $f^M$ and $f^V$. Parameter values for the case $d > 1$ are $\alpha = 0.2$, $\mu = 0.05$, $\eta = 0.1$, and for $d < 1$ we use $\alpha = 0.4$, $\mu = 0.04$, $\eta = 0.4$, with $v_0 = 0.04$ and $T = 3$ in both cases. Note that with these values the Feller condition fails in the first case and holds in the second, so that in the second case the process (in continuous time) remains strictly positive.

```
'Coefficient values set in set-up procedures:
c1_ = alpha_ * dt
c2_ = (0.5 * eta_ * eta_ - alpha_ * mu_) * dt
c3_ = -eta_

Friend Function IPG_GetFinalS(WieVec() As Double) As Double
  Dim y As Double: y = -Log(v0_)
  Dim y_cap As Double: y_cap = 10# 'bodge to prevent overflow
  Dim i As Long
  For i = inc_ To N_ Step inc_
    Dim dz As Double: dz = WieVec(i) - WieVec(i - inc_)
    Dim ey As Double: ey = Exp(0.5 * my_min(y, y_cap))
    y = y + c1_ + c2_ * ey + c3_ * ey * dz
  Next i
  IPG_GetFinalS = Exp(-y)
End Function
```

**Figure 26.1** Implementing the log-Euler method
Applications to Models 421

'Coefficient values set in set-up procedures:
c0_ = (alpha_ * mu_ - 0.25 * eta_ * eta_ - 0.5 * alpha_ * alpha_ * mu_ * dt_) * dt_
c1_ = (1# - alpha_ * mu_ - 0.5 * alpha_ * alpha_ * mu_ * dt_) * dt_
c2_ = eta_ * (1# - 0.75 * alpha_ * dt_)
c3_ = 0.25 * eta_ * (alpha_ * mu_ - 0.25 * eta_ * eta_) * dt_
c4_ = 0.25 * eta_ * eta_

Friend Function IPG_GetFinalS(WieVec() As Double) As Double
Dim V As Double: V = v0_
Dim s_bounce As Double: s_bounce = 0.001 'too small or large and get bias,
Dim i As Long
For i = inc_ To N_ Step inc_
    Dim dz As Double: dz = WieVec(i) - WieVec(i - inc_)
    Dim sig As Double: sig = Sqr(my_pos(V))
    If sig = 0# Then sig = s_bounce 'should be parameter dependent
    V = c0_ + c1_ * V + (c2_ * sig + c3_ / sig) * dz + c4_ * dz * dz
Next i
IPG_GetFinalS = my_pos(V)

Figure 26.2  Implementing the 2.0 weak scheme

Table 26.1  CIR: comparison of discretization schemes, weak convergence

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Equation</th>
<th>$d &lt; 1$</th>
<th>$d &gt; 1$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$f^M$</td>
<td>$f^V$</td>
</tr>
<tr>
<td>Euler</td>
<td>(25.51), page 406</td>
<td>-1.29</td>
<td>-0.88</td>
</tr>
<tr>
<td>Milstein</td>
<td>(25.52), page 406</td>
<td>-0.50</td>
<td>-0.93</td>
</tr>
<tr>
<td>1.5 strong</td>
<td>(25.72), page 410</td>
<td>-1.15</td>
<td>-2.47</td>
</tr>
<tr>
<td>2.0 weak</td>
<td>(25.78), page 411</td>
<td>-1.13</td>
<td>-2.43</td>
</tr>
<tr>
<td>P-C</td>
<td>(25.92), page 413</td>
<td>0.03</td>
<td>-0.18</td>
</tr>
<tr>
<td>Normal</td>
<td>(26.9)</td>
<td>-0.16</td>
<td>-0.13</td>
</tr>
<tr>
<td>Log-normal</td>
<td>(26.14)</td>
<td>0.26</td>
<td>-0.96</td>
</tr>
<tr>
<td>Log-Euler</td>
<td>(26.17)</td>
<td>0.13</td>
<td>-1.49</td>
</tr>
<tr>
<td>Freezing</td>
<td>(26.20)</td>
<td>-0.21</td>
<td>0.01</td>
</tr>
</tbody>
</table>

$M = 100000$ Wiener sample paths are used, stratified at the final time and filled in with a Brownian bridge. The full sample paths have $2^{10}$ time steps. They are sampled at various intervals to get paths of length $N = 2^K$ steps, $K = 1, \ldots, 10$. The table shows rates of convergence as $\log_2(N)$ increases from 1 to 5. This range avoids issues with going from time 0 to time 3 in a single time step and also avoids the region where a residual simulation error dominates. The log-Euler scheme blows up in the $d > 1$ case.

The values of the mean and variance are known explicitly. To compute the weak criterion,

$$c_w(X_{tN}, X_T) = | \mathbb{E}[f(X_{tN})] - \mathbb{E}[f(X_T)] |,$$

these are used instead of a simulated value of $\mathbb{E}[f(X_T)]$. This is necessary since in the absence of a strong solution to the SDE it is not possible to generate $X_T$ naturally in continuous time from increments in a Wiener sample path.
The consequence of this is that the values given in the table are subject to a residual level of simulation error which does not reduce to zero as $N$ increases. When a strong solution is available $c_w(X_{tN}, X_T)$ can be estimated as

$$
\tilde{c}_w = \left| \frac{1}{M} \sum_{j=1}^{M} f\left(X_{tN}^j\right) - \frac{1}{M} \sum_{j=1}^{M} f\left(X_T^j\right) \right|
$$

(26.23)

$$
= \left| \frac{1}{M} \sum_{j=1}^{M} \left( f\left(X_{tN}^j\right) - f\left(X_T^j\right) \right) \right|
$$

(26.24)

where $j = 1, \ldots, M$ labels the sample paths. Whatever the rate of convergence (as long as there is convergence) this goes to zero as $N \to \infty$. However, if $c_w(X_{tN}, X_T)$ is estimated as

$$
\tilde{c}_w = \left| \frac{1}{M} \sum_{j=1}^{M} f\left(X_{tN}^j\right) - \mathcal{F} \right|
$$

(26.25)

where $\mathcal{F} = \mathbb{E}[f(X_T)]$ is the theoretical value and not a simulated value, then $\tilde{c}_w$ goes to zero only in $M$.

Results are sensitive to whether or not the Feller condition is satisfied. Convergence is usually around 1 for most schemes but there is considerable variation and the table does not tell the full story; it must be interpreted very cautiously. More informative are graphical convergence plots. In practice $\tilde{c}_w$ decreases as $N$ increases until the error is at a level similar to the standard error in $\tilde{c}_w$. The effect can be seen in Figure 26.3. The figure plots $\log_2(error)$ against $K = \log_2(N)$ for the nine discretization schemes. The figure shows two cases, one where the Feller condition is satisfied, and one where it is not.

Panel A shows convergence for the mean function at $T = 3$ in the $d > 1$ case (corresponding to the penultimate column in Table 26.1). The Euler, Milstein and predictor–corrector schemes converge as a group, at roughly order 1. These lie above, so are slower than, the next group, the normal, moment-freezing and 1.5 strong and 2.0 weak schemes. Finally the log-normal scheme is in a group by itself.

The schemes decrease in error until a level of around $2^{-17}$ to $2^{-18}$; then the error stays around this level. This represents the residual simulation error in this illustration. The log-normal scheme starts (and remains) at this level. This is not surprising as it is set up to match the mean of the CIR process.

The errors in the normal and moment-freezing schemes stay close together. This is seen often, and is not unexpected. The moment-freezing scheme is a refinement of the normal scheme, but does not represent a radical improvement over it.

The Euler and Milstein errors are usually close. In panel A Milstein is better than Euler, but in panel B Euler is better. This is not unusual; depending on parameter values and the choice of function it is quite possible for Euler to beat Milstein. However the results here are typical; both are slow, and only a little worse than the predictor–corrector.

The 1.5 strong and 2.0 weak schemes often, as here, have very similar errors. However their (joint) performance can be quite variable Sometimes they are worse than Euler; at other times, as in panel B, they can be as good as (if not better than) log-normal.

Panel B shows a contrasting case where $d < 1$ and $T = 0.1$, with the variance function. Zero is accessible; one anticipates that schemes will perform less well. Indeed the general picture is more confused although, in the example of panel B, convergence is quite regular, for the most part. Residual simulation error seems to be around $2^{-26}$ although this is not clear. Convergence is order 1 for log-Euler,
Figure 26.3  CIR: weak convergence for approximate schemes
predictor–corrector, Euler and Milstein; order 2 for normal and moment freezing; and around 0.25 for 1.5 strong, 2.0 weak and log-normal.

The chief differences with panel A are that the performance of the 1.5 strong scheme is considerably worse; it (and Milstein too) is worse than Euler. By contrast 2.0 weak is as good as log-normal. In other cases predictor–corrector is often better than this illustration suggests, and 2.0 weak is often worse.

The panel shows a plot for log-Euler. In this case log-Euler did not explode, however its performance is the worst of the set.

Dismissing the log-Euler and the 1.5 strong and 2.0 weak schemes, for reasons previously discussed, one is left favouring the log-normal scheme, at least for the cases considered here. Its performance in our tests is better than Euler, Milstein and predictor–corrector, and we prefer it to moment freezing because its asymmetry is likely to better match the CIR distribution.

### 26.2 SIMULATING DISCOUNT FACTORS

Under the accumulator account numeraire, the value $c_t$ of a European option at time $t$ written on an asset $S_t$ with payoff $H_T(S_T)$ at time $T$ is given by

$$c_t = \mathbb{E} \left[ \exp \left( - \int_t^T r_s \, ds \right) H_T(S_T) \right]$$

(26.26)

where $r_t$ is the short interest rate and expectations are taken with respect to the associated equivalent martingale measure.

In a Monte Carlo method for a model with stochastic interest rates one must evolve both $r_t$ and $S_t$, and compute not only simulated values of $H_T(S_T)$ but also simulated approximations to the continuous time stochastic discount factor $B_t(T)$,

$$B_t(T) = \exp \left( - \int_t^T r_s \, ds \right),$$

(26.27)

from a simulated path for the short rate $r = (r_{t_0}, \ldots, r_{t_N})$.

An easy approximation for $B_t(T)$ is to set

$$B_t(T) \sim \hat{B}_t^H(T) = \exp \left( - \sum_{i=0}^{N-1} r_{t_i} \Delta_t \right).$$

(26.28)

This approximates the integral

$$J = \int_t^{t+\Delta_t} r_s \, ds \mid r_t, r_{t+\Delta_t}$$

(26.29)

as

$$J \sim \hat{J}_t^H = r_t \Delta_t,$$

(26.30)

setting $\hat{B}_t^H(T) = \exp (- \sum_{i=0}^{N-1} \hat{J}_t^H)$, but this can be improved easily.

We consider three approaches to constructing approximations to $B_t(T)$. The first focuses on approximations $\hat{J}$ to $J$; the second tries to approximate $B_t(T)$ directly; and the third approximates $J$ but tries to correct for the bias introduced in taking an exponential.
Approximations to J

The approximation \( \hat{J}^H_t \) ignores the knowledge we have of the value of \( r_{t+\Delta_t} \). One possibility is to use instead a trapezium rule, setting

\[
J \sim \hat{J}^T_t = \frac{1}{2} (r_t + r_{t+\Delta_t}) \Delta_t,
\]

(26.31)

constructing \( \hat{B}^T_t (T) = \exp \left(- \sum_{i=0}^{N-1} \hat{J}^T_i \right) \).

Another approach uses an Itô–Taylor approximation. Suppose that \( dr_t = a(r_t) \, dt + b(r_t) \, dz_t \) and let \( J = \int_t^{t+\Delta_t} r_s \, ds \), then when the SDE has a strong solution we can write

\[
J = \int_t^{t+\Delta_t} \left( r_t + \int_t^s \, dr_u \right) ds
\]

(26.32)

\[
= \int_t^{t+\Delta_t} r_t \, ds + \int_t^s (a(r_u) \, du + b(r_u) \, dz_u) \, ds
\]

(26.33)

\[
= r_t \Delta_t + \int_t^{t+\Delta_t} \int_t^s a(r_u) \, du \, ds + \int_t^{t+\Delta_t} \int_t^s b(r_u) \, dz_u \, ds.
\]

(26.34)

With the approximation \( da = db = 0 \) on the interval \([t, t+\Delta_t]\) one has \( J \sim \hat{J}^E \) where

\[
\hat{J}^E = \hat{r}_t \Delta_t + a(r_t) \int_t^{t+\Delta_t} \int_t^s \, du \, ds + b(r_t) \int_t^{t+\Delta_t} \int_t^s \, dz_u \, ds
\]

(26.35)

\[
= r_t \Delta_t + \frac{1}{2} (a(r_t) \Delta_t + b(r_t) \Delta z_t) \Delta_t
\]

(26.36)

\[
= \frac{1}{2} (r_t + \hat{r}_{t+\Delta_t}) \Delta_t
\]

(26.37)

where \( \hat{r}_{t+\Delta_t} = r_t + a(r_t) \Delta_t + b(r_t) \Delta z_t \) is an Euler predictor for the value of \( r \) at time \( t + \Delta_t \). Then we set \( \hat{B}^E_t (T) = \exp \left(- \sum_{i=0}^{N-1} \hat{J}^E_i \right) \). Note that this resembles a predictor–corrector scheme.

By expanding out the integral in \( b \) this scheme can be refined a little to give a scheme of full second order. Let \( J_2 = \int_t^{t+\Delta_t} \int_t^s b(r_u) \, dz_u \, ds \), then

\[
J_2 = \int_t^{t+\Delta_t} \int_t^s \left( b(r_t) + \int_t^u \, db(r_s') \right) \, dz_u \, ds
\]

(26.38)

\[
= b(r_t) \int_t^{t+\Delta_t} \int_t^s \, dz_u \, ds + \int_t^{t+\Delta_t} \int_t^s \int_t^u \, b'(r_s') \, dr_s' \, dz_u \, ds
\]

(26.39)

\[
\sim b(r_t) I_{1,0} + b'(r_t) b(r_t) \int_t^{t+\Delta_t} \int_t^s \int_t^u \, dz_u' \, dz_u \, ds
\]

(26.40)

to second order. The iterated Itô integral \( I_{1,1,0} = \int_t^{t+\Delta_t} \int_t^s \int_t^u \, dz_u' \, dz_u \, ds \) can be approximated by \( \frac{1}{6} (\Delta \Delta z^2 - \Delta_z) \) (Kloeden and Platen) which has the same moments as \( I_{1,1,0} \) sufficiently accurate, to a sufficient number, to enable a 2.0 weak approximation.
This yields the scheme \( \hat{J}^2 \)

\[
\hat{J}^2 = \frac{1}{2}(r_t + \hat{z}_t^M) \Delta_t,
\]

where

\[
\hat{z}_t^M = r_t + a(r_t) \Delta_t + b(r_t) \Delta z_t + \frac{1}{2} b(r_t) b'(r_t) (\Delta z_t^2 - \Delta_t)
\]

is a first-order predictor of \( r \) at time \( t + \Delta_t \), and the approximation to \( B_t(T) \) is \( \hat{B}_t^2(T) = \exp (- \sum_{i=0}^{N-1} \hat{J}_i^2) \).

Note the difference in the correction term in \( \hat{z}_t^M \) compared to the usual Milstein correction term.

**Direct approximations to \( B_t(T) \)**

If bond prices are explicitly computable in a model, so that

\[
B_t(r_t, T) = \mathbb{E} \left[ \exp \left( - \int_t^T r_s \, ds \right) | r_t \right]
\]

is known, then one may use explicit values, setting the approximation to be

\[
\hat{B}_t^X(T) = \prod_{i=0}^{N-1} B_{t_i}(r_{t_i}, t_{i+1}).
\]

This can be regarded as an elaboration of the scheme \( \hat{B}_t^H(T) \).

\( \hat{B}_t^X(T) \) does not use information about the value of \( r_{t_i+\Delta t} \). In some tractable models, such as Gaussian affine models, the value of the conditional expectation

\[
D_{t,T}(r_t, r_T) = \mathbb{E} \left[ \exp \left( - \int_t^T r_s \, ds \right) | r_t, r_T \right]
\]

is known. For the Vasicek model this has been computed by Gandhi and Hunt (1997) (see below). One can then set

\[
\hat{B}_t^{GH}(T) = \prod_{i=0}^{N-1} D_{t_i, t_{i+1}}(r_{t_i}, r_{t_{i+1}}).
\]

This is a more sophisticated version of the trapezium rule estimator, \( \hat{B}_t^T(T) \).

**Bias correction**

When an approximation \( \hat{J} \) is normal (as is the case for the Vasicek process) we know that

\[
\mathbb{E} \left[ \exp \left( - \hat{J} \right) \right] = \exp \left( -\mathbb{E}[\hat{J}] + \frac{1}{2} \text{var}[\hat{J}] \right).
\]
One may then correct for bias in \( \hat{B}_t(T) = \exp \left( - \sum_{i=0}^{N-1} \hat{J}_i \right) \) by approximating \( B_t(T) \) as

\[
B_t(T) \sim \hat{B}_t^C(T) = \hat{B}_t(T) \exp \left( \frac{1}{2} \sum_{i=0}^{N-1} \text{var}[\hat{J}_i] \right). \tag{26.48}
\]

Even when \( \hat{J} \) is not normal a construction of this type might be beneficial.

For instance, consider applying this correction to the \( \hat{J}^E_t \) scheme in equation (26.37). We have

\[
\text{var}[\hat{J}_t^E | r_t] = \text{var} \left[ \frac{1}{2} b(r_t) \Delta z_t \Delta t \right] = \frac{1}{4} b^2(r_t) \Delta^3_t \tag{26.49}
\]

so the bias-corrected value is

\[
\hat{B}_t^C(T) = \hat{B}_t^E(T) \exp \left( \frac{1}{8} \sum_{i=0}^{N-1} b^2(r_t) \Delta^3_i \right). \tag{26.50}
\]

This third-order correction is not likely to have a huge effect unless \( \Delta_t \) is large.

### 26.2.1 Application to the Vasicek term structure model

This is a model of the term structure with the short rate \( r_t \) as its state variable following the SDE

\[
dr_t = \alpha (\mu - r_t) \, dt + \sigma \, dz_t. \tag{26.51}
\]

It is highly tractable but unusable in practice without modification (at a minimum by making \( \mu \) a deterministic function of time).

The Vasicek SDE can be solved so the process can be evolved exactly. In fact \( r_T | r_t \sim N(\mu_{t,T}, \sigma^2_{t,T}) \) with

\[
\mu_{t,T} = r_t e^{-\alpha \tau} + \mu \left( 1 - e^{-\alpha \tau} \right), \tag{26.52}
\]

\[
\sigma^2_{t,T} = \frac{\sigma^2}{2\alpha} \left( 1 - e^{-2\alpha \tau} \right), \tag{26.53}
\]

where \( \tau = T - t \).

There are explicit solutions for bond prices so we are able to benchmark the various alternative discrete integration methods. Set \( \tau = T - t \) to be the time to maturity and set \( r_{\infty} = \mu - \sigma^2 / 2\alpha^2 \). \( r_{\infty} \) is the long rate in the Vasicek model. The exact bond price,

\[
B_t(T | r_t) = \mathbb{E} \left[ \exp \left( - \int_t^T r_s \, ds \right) | r_t \right], \tag{26.54}
\]

is simply \( B_t(T | r_t) = \exp (-r_t(T) \tau) \) where

\[
r_t(T) = r_t \frac{1 - e^{-\alpha \tau}}{\alpha \tau} + r_{\infty} \left( 1 - \frac{1 - e^{-\alpha \tau}}{\alpha \tau} \right) + \frac{\sigma^2 \tau}{4\alpha} \left( \frac{1 - e^{-\alpha \tau}}{\alpha \tau} \right)^2. \tag{26.55}
\]
The conditional discount factor, $D_{t,T}(r_t, r_T)$ in equation (26.45), can be computed explicitly (Gandhi and Hunt (1997)). It is

$$D_{t,T}(r_t, r_T) = \mathbb{E} \left[ \exp \left( -\int_t^T r_s \, ds \right) \mid r_t, r_T \right] = \exp \left( \alpha_t,T - \beta_t,T r_t - \gamma_t,T r_T \right)$$  \hspace{1cm} (26.56)

where

$$\beta_t,T = \gamma_t,T = \frac{1}{\alpha} \left( \frac{1 - e^{-\alpha \tau}}{1 + e^{-\alpha \tau}} \right)$$  \hspace{1cm} (26.57)

and $\alpha_t,T = A_t,T + B_t,T$ where

$$A_t,T = r_\infty \frac{1 - e^{-\alpha \tau}}{\alpha} - r_\infty \tau - \frac{\sigma^2}{4 \alpha} \left( \frac{1 - e^{-\alpha \tau}}{\alpha} \right)^2,$$  \hspace{1cm} (26.58)

$$B_t,T = \gamma_t,T \mu \left( 1 - e^{-\alpha \tau} \right) - \frac{\sigma^2}{4 \alpha} \frac{1 - e^{-2\alpha \tau}}{4 \alpha}.$$

### 26.2.2 Numerical assessment

Table 26.2 summarizes the results. Six cases are compared (since for Vasicek $\hat{B}^2 \equiv \hat{B}^E$). The equation reference is either to $\hat{B}$ or to the corresponding $\hat{J}$. The table shows the rate of convergence (computed over the interval $[1, 5]$ in $K$) and the error at $K = 10$. Errors are computed with $10^6$ stratified sample paths up to time $T = 3$. Parameter values are $\alpha = 0.2$, $\mu = 0.05$, $\sigma = 0.02$, with $r_0 = 0.04$.

Convergence rates are reasonable and errors appear similar, but as usual the table does not tell the full story. Convergence is plotted in Figure 26.4. $\log_2(error)$ for the six schemes are plotted against $\log_2(N)$. The figure omits the exact simulation $\hat{B}^X$ with one step as it is, by construction, both very accurate and irrelevant.

The results are striking. The residual simulation error seems to be around $10^{-21}$. The Gandhi and Hunt scheme is at the level of residual error immediately. The simple trapezium scheme catches up in only $2^4$ steps, and from this point there is no significant advantage to the Gandhi and Hunt scheme. The naive $\hat{B}^H$ scheme converges only slowly (order 1) to the residual level, appearing not to reach it even after $2^{10}$ steps. The Euler, corrected Euler, and exact schemes are only a little more accurate. They converge together also at order 1.

From these results there is no reason for an ordinary Monte Carlo application not to use the trapezium scheme $\hat{B}^T$. It is simple, accurate, with smaller errors than every other scheme except Gandhi and Hunt.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Equation</th>
<th>Errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\hat{B}^H$</td>
<td>(26.30)</td>
<td>$1.03$</td>
</tr>
<tr>
<td>$\hat{B}^T$</td>
<td>(26.31)</td>
<td>$1.75$</td>
</tr>
<tr>
<td>$\hat{B}^E$</td>
<td>(26.37)</td>
<td>$1.03$</td>
</tr>
<tr>
<td>$\hat{B}^C$</td>
<td>(26.50)</td>
<td>$1.03$</td>
</tr>
<tr>
<td>$\hat{B}^X$</td>
<td>(26.44)</td>
<td>$0.67$</td>
</tr>
<tr>
<td>$\hat{B}^{GH}$</td>
<td>(26.46)</td>
<td>$1.34$</td>
</tr>
</tbody>
</table>
If long-step Monte Carlo is being used then the additional accuracy of Gandhi and Hunt may be advantageous. The basic $\hat{B}^H$ scheme need never be used and, as they are dominated by $\hat{B}^T$, there seems little purpose in using either $\hat{B}^E$ or $\hat{B}^X$.

26.3 SUMMARY

This chapter has looked in some detail at a pair of test cases: simulating the CIR process and simulating discount factors in the Vasicek model.

In the former example we found the Feller condition to be crucial. Life is considerably simpler if it is satisfied so that zero is inaccessible. In our illustrations the log-normal moment-matching scheme performed well, better than the more sophisticated, more delicate, higher order Itô–Taylor schemes.

For simulating discount factors we come down in favour of a simple trapezium integration scheme. This is simple but converges rapidly. In some cases, where long-step Monte Carlo is being used, a Gandhi and Hunt scheme might be preferred, but for everyday use keep it simple!

26.4 EXERCISES

1. The CIR process is highly skewed. Simulate values for this process for time $T = 1$ using the exact discretization and also for the methods described in section 26.1. How closely is the distribution matched?
2. In the CIR model the value $B_t(T)$ of a pure discount bond is given by

$$B_t(T | r_t) = \exp (a(T - t) - b(T - t)r_t)$$  \hspace{1cm} (26.60)

where

$$a(\tau) = \frac{2\alpha \mu}{\sigma^2} \ln \left( \frac{2\gamma \exp^{\frac{1}{2}(\gamma + \alpha)\tau}}{\gamma + \alpha(\exp^{\gamma \tau} - 1) + 2\gamma} \right),$$  \hspace{1cm} (26.61)

$$a(\tau) = \frac{2(\exp^{\gamma \tau} - 1)}{(\gamma + \alpha)(\exp^{\gamma \tau} - 1) + 2\gamma}$$  \hspace{1cm} (26.62)

and $\gamma = \sqrt{\alpha^2 + 2\sigma^2}$. Several of the schemes described in section 26.2 apply, or can be extended to apply, to the CIR model. Using the known PDB value as a benchmark, establish the relative performance of these discount factor schemes in this case.

3. Recall the processes listed in equation (25.93), page 416. Can analogues of any of the moment-matching schemes for CIR, described in section 26.1, be applied to any of these processes? Do any of these schemes outperform the Itô–Taylor schemes implemented for these processes in exercise 1?
Valuation in the Heston Model

Immediately one moves away from a standard 1-factor model one encounters difficulties. In this chapter we investigate the implementation of the 2-factor Heston model (Heston (1993)). It is an excellent test-bed for assessing the effectiveness of any numerical method. Here we are concerned only with simulation, and with standard simulation methods at that, but the model poses huge challenges for any numerical technique.

Our exploration of Heston is limited. Sections 27.1 and 27.2 look at the effectiveness of standard discretization techniques; section 27.3 looks at efficiency gains achievable using stratification and control variates. The results are both disappointing and challenging. We discover that ordinary discretization schemes perform poorly. Short-step Monte Carlo requires too many steps per year to enable a really fast simulation. Accuracy is low, particularly when the Feller condition on the volatility process is not satisfied.

We find a similar picture when we go on to investigate speed-ups. If the Feller condition is not met gains are small, and not astonishingly large even when it is. Fully stratified sampling (at the final time only) is expensive and does not bring significant benefit; the presence of stochastic volatility reduces the effectiveness of bucketing with short-step Monte Carlo. Auxiliary model CVs prove to be largely ineffective; the correlations are just too low. In our examples only the geometric average auxiliary CV used with arithmetic options produced other than a minor speed-up. The gains made with the call option auxiliary CV are poor.

CVs arising directly from the Heston model can be effective (for instance, the payoff matching Heston CV for the exotic quadratic option) but, because of their potential expense, must be used with a little care.

The timings in the tables in this chapter are given in seconds.

### 27.1 DISCRETIZING THE HESTON MODEL

The Heston model is a widely used stochastic volatility model. Its state variables are the asset value $S_t$ and its volatility $v_t$ with processes

$$\begin{align*}
    dS_t &= rS_t \, dt + \sqrt{v_t} \, S_t \, dz^S_t, \\
    dv_t &= \beta(\mu - v_t) \, dt + \sigma \sqrt{v_t} \, dz^v_t, \\
    dz^S_t \, dz^v_t &= \rho \, dt.
\end{align*}$$

We can rewrite this in terms of two uncorrelated Wiener processes, $z^1$ and $z^2$, as

$$\begin{align*}
    dS_t &= rS_t \, dt + \sqrt{v_t} \, S_t \left( \tilde{\rho} \, dz^1_t + \rho \, dz^2_t \right), \\
    dv_t &= \beta(\kappa - v_t) \, dt + \sigma \sqrt{v_t} \, dz^2_t,
\end{align*}$$

where $\tilde{\rho} = \sqrt{1 - \rho^2}$, so that $z^v = z^2$ and $z^S = \tilde{\rho} z^1_t + \rho z^2_t$.

In practice it is usually better to simulate not $S_t$ but $R_t = \ln(S_t)$, whose process is

$$\begin{align*}
    dR_t &= \left( r - \frac{1}{2} \, v_t \right) \, dt + \sqrt{v_t} \left( \tilde{\rho} \, dz^1_t + \rho \, dz^2_t \right),
\end{align*}$$

with $R_0 = \ln(S_0)$. 

There is an exact simulation scheme, due to Broadie and Kaya (2006), subsequently improved by Glasserman and Kim (2008), but it is slow and we do not assess it, or implement it, here. Instead two types of discretization schemes are investigated, \( \text{Itô–Taylor} \) based and moment matching. In this section \( v_t \) is usually discretized by the log-normal approximation discussed in section 26.1. Since there is no strong solution to hand we look only at weak convergence, with the projection function \( f^\nu(S, v) = e^{-rT}(S - X)^+ \).

Direct higher order strong schemes for Heston are likely to be problematic since it is not Lipschitz: the derivative of the volatility term, \( \sigma \sqrt{v} \), becomes unbounded as \( v \to 0 \). Moment matching schemes are likely to be easier to implement and, indeed, turn out to be workable in practice.

### 27.1.1 \( \text{Itô–Taylor} \) and predictor–corrector schemes

This section discusses the Euler, Milstein, 1.5 strong \( \text{Itô–Taylor} \), and predictor–corrector schemes for Heston. All (apart from 1.5 strong \( \text{Itô–Taylor} \)) are straightforward to implement although the usual bodges (see Chapters 25 and 26) are required.

**The Euler scheme**

The plain direct Euler scheme is

\[
\begin{align*}
\Delta^S_E S_t &= r S_t \Delta t + \sqrt{v_t} S_t \left( \bar{\rho} \Delta z^1_t + \rho \Delta z^2_t \right), \\
\Delta_E v_t &= \alpha (\mu - v_t) \Delta t + \eta \sqrt{v_t} \Delta z_2.
\end{align*}
\]

Apart from fiddles to ensure that the processes stay positive, this is simple. It is better to apply Euler not to \( S_t \) but to \( R_t \) yielding

\[
\begin{align*}
\Delta^R_E R_t &= \left( r - \frac{1}{2} v_t \right) \Delta t + \sqrt{v_t} \left( \bar{\rho} \Delta z^1_t + \rho \Delta z^2_t \right), \\
\Delta_E v_t &= \alpha (\mu - v_t) \Delta t + \eta \sqrt{v_t} \Delta z_2.
\end{align*}
\]

The advantages of doing this are that no bodge is required to keep \( S \) positive and also, were \( v_t \) frozen, the scheme would be exact.

**1.0 strong \( \text{Itô–Taylor} \) scheme (Milstein)**

The full 1.0 strong scheme in equation (25.27), page 403, specializes down when \( Q = M = 2 \) to equations (25.33) and (25.34), page 404. Substituting in for the Heston drifts and volatilities the discretization simplifies to give

\[
\begin{align*}
\Delta^S_M S_t &= r S_t \Delta t + \sqrt{v_t} S_t \left( \bar{\rho} \Delta z^1_t + \rho \Delta z^2_t \right) \\
&\quad + \bar{\rho}^2 S_t v_t (I_{1,1} + \phi I_{1,2}) + \left( \rho \bar{\rho} S_t v_t + \frac{1}{2} \eta \bar{\rho} S_t \right) (I_{2,1} + \phi I_{2,2}). \\
\Delta_M v_t &= \alpha (\mu - v_t) \Delta t + \eta \sqrt{v_t} \Delta z_2 + \frac{1}{2} \eta^2 I_{2,2}.
\end{align*}
\]
with $\phi = \rho / \tilde{\rho}$, and this becomes

$$
\Delta M_{S_t} = r S_t \Delta_t + \sqrt{v_t} S_t (\tilde{\rho} \Delta z^1_t + \rho \Delta z^2_t) \\
+ \rho^2 S_t v_t (\Delta z^2_t - \Delta_t) \\
+ \frac{1}{2} \left( \rho^2 S_t v_t + \frac{1}{2} \eta \rho S_t \right) (\Delta z^2_t - \Delta_t) \\
+ \rho \tilde{\rho} S_t v_t \Delta z_t \Delta z_2 + \frac{1}{2} \eta \tilde{\rho} S_t I_{2,1}, 
$$

(27.13)

$$
\Delta M_{V_t} = a(\mu - v_t) \Delta_t + \eta \sqrt{v_t} \Delta z_2 + \frac{1}{4} \eta^2 \left( \Delta z^2_2 - \Delta_t \right). 
$$

(27.14)

where $I_{2,1}$ remains to be simulated.

It is better to apply the scheme directly to $R_t$. Now the derivatives $b^{q,i}_{1,1}$ vanish, so equation (25.27) becomes

$$
\Delta M_R = a^1 \Delta_t + b^{1,1} (\Delta z_1 + \phi \Delta z_2) + b^{2,2} b^{1,1}_2 (I_{2,1} + \phi I_{2,2}). 
$$

(27.15)

$$
\Delta M_V = a^2 \Delta_t + b^{2,2} \Delta z_2 + b^{2,2} b^{2,2}_2 I_{2,2}. 
$$

(27.16)

Replacing $I_{2,1}$ by its conditional expected value, $\frac{1}{2} \Delta z^1_t \Delta z^2_t$, this gives the simplified version

$$
\Delta M^R_{R_t} = \left( r - \frac{1}{2} v_t \right) \Delta_t + \sqrt{v_t} (\tilde{\rho} \Delta z^1_t + \rho \Delta z^2_t) \\
+ \frac{1}{4} \eta \tilde{\rho} \left( \Delta z^1_t \Delta z^2_t + \phi \left( (\Delta z^2_t)^2 - \Delta_t \right) \right), 
$$

(27.17)

$$
\Delta M_{V_t} = a(\mu - v_t) \Delta_t + \eta \sqrt{v_t} \Delta z_2 + \frac{1}{4} \eta^2 \left( (\Delta z^2_t)^2 - \Delta_t \right). 
$$

(27.18)

### 1.5 strong Itô–Taylor

The general 1.5 strong scheme in equation (25.35), page 405, can be written out in full for the Heston process. It is difficult to simulate because of the need to approximate $I_{j_1,j_2,j_3}$ for all $j_i$ (see exercise 1). Since in any case the poor behaviour of the CIR volatility process must be worked around, one is reluctant to pursue the Itô–Taylor approach to this order.

**Predictor–corrector for Heston**

With $Q = M = 2$, predictor–corrector is a little more finicky to implement. Set $\Delta z_t = (\Delta z^1_t, \Delta z^2_t)'$ and

$$
\begin{pmatrix}
  b^R_r \\
  b^V_r
\end{pmatrix} = \begin{pmatrix}
  b^{R,1}_r & b^{R,2}_r \\
  b^{V,1}_r & b^{V,2}_r
\end{pmatrix} = \begin{pmatrix}
  \tilde{\rho} \sqrt{v_t} & \rho \sqrt{v_t} \\
  0 & \eta \sqrt{v_t}
\end{pmatrix}, 
$$

(27.19)

$$
\begin{pmatrix}
  a^R_r \\
  a^V_r
\end{pmatrix} = \begin{pmatrix}
  r - \frac{1}{2} v_t \\
  \alpha (\mu - v_t)
\end{pmatrix}, 
$$

(27.20)
\[ \begin{align*}
\theta^R &= a^R - w_b \left( b^R b^R_1 + b^R b^R_2 + b^v b^R_1 + b^v b^R_2 \right) & (27.21) \\
&= a^R - \frac{1}{2} w_b \rho \eta, & (27.22) \\
\theta^v &= a^v - w_b \left( b^R b^v_1 + b^R b^v_2 + b^v b^v_1 + b^v b^v_2 \right) & (27.23) \\
&= a^v - \frac{1}{2} w_b \eta^2. & (27.24)
\end{align*} \]

The predictor–corrector equation for \( v_t \) is given by equation (25.92), page 413, and the full Heston scheme is
\[ R_{t+\Delta t} = R_{t+\Delta t} - \frac{1}{2} w_b \rho \eta \Delta_t \\
+ w_a \left[ a^R (v_{t+\Delta t}) - a^R (v_t) \right] \Delta_t + w_b \left[ b^R (v_{t+\Delta t}) - b^R (v_t) \right] \Delta z_t & (27.25) \\
= R_{t+\Delta t} - \frac{1}{2} w_b \rho \eta \Delta_t \\
- \frac{1}{2} w_a \left[ v_{t+\Delta t} - v_t \right] \Delta_t + w_b \left[ \sqrt{v_{t+\Delta t}} - \sqrt{v_t} \right] \left( \rho \Delta z_t^1 + \rho \Delta z_t^2 \right), & (27.26) \\
v_{t+\Delta t} = v_{t+\Delta t} - \frac{1}{2} w_b \eta^2 \Delta_t - w_a \left[ v_{t+\Delta t} - v_t \right] \Delta_t + w_b \eta \left[ \sqrt{v_{t+\Delta t}} - \sqrt{v_t} \right] \Delta z_t^2, & (27.27)
\]
where \( v_{t+\Delta t} \) is the \( v \)-predictor.

### 27.1.2 Moment-matching schemes

We established in Chapter 26 that a log-normal approximation can work well for CIR. We apply this to the volatility process \( v_t \) and then, conditional upon a realization of a sample path of \( v \), construct sample paths for \( S_t \) by two different methods: moment freezing, and a partial predictor–corrector.

Suppose that we have generated a sample path \( v = (v_{t_0}, \ldots, v_{t_N}) \) for \( v_t \). We look at each possibility in turn.

**Moment freezing**

The exact solution to the process
\[ dS_t = r S_t \, dt + \sigma S_t \, dz_t^S, \]  
with constant volatility \( \sigma \), conditional on \( S_t \), is \( S_{t+\Delta t} = S_t \exp((r - \frac{1}{2} \sigma^2) \Delta_t + \sigma \Delta z_t^S) \). The moment freezing approximation replaces \( \sigma \) with \( \sqrt{v_t} \) to obtain
\[ S_{t+\Delta t} = S_t \exp \left( \left( r - \frac{1}{2} v_t \right) \Delta_t + \sqrt{v_t} \Delta z_t^S \right). \]
Partial predictor–corrector

\( v \) is simulated independently of \( R \); \( R \) is simulated with a predictor and a corrector. \( v_{t+\Delta t} \) is generated directly from \( v_t \), by some method, then, instead of using a predictor value \( v_{t+\Delta t} \) in equation (27.26), one uses the simulated value \( v_{t+\Delta t} \) to obtain a corrected \( R \),

\[
R_{t+\Delta t} = R_{t+\Delta t} - \frac{1}{2} w_b \rho \eta \Delta t - \frac{1}{2} w_d \left[ v_{t+\Delta t} - v_t \right] \Delta t + w_b \left[ \sqrt{v_{t+\Delta t}} - \sqrt{v_t} \right] \Delta z^R_t. \tag{27.30}
\]

This is extremely easy to implement.

### 27.2 CONVERGENCE IN THE HESTON MODEL

We compare only variations on the standard schemes described earlier. In practice the Heston model really requires tailored schemes but a treatment of these is beyond the scope of this book. Nine schemes are compared: three Itô–Taylor schemes, three schemes based on log-normal moment matching on \( v \), and three schemes with a predictor–corrector component. These are listed in Table 27.1. The three groups of methods are labelled I-T, LN and PC respectively.

For a test of weak convergence we use the function \( f^c(S, v) = e^{-rT} (S - X)^+ \), the discounted payoff of a European call option, so that \( \mathbb{E}[f^c] \) is the European option value. This value can be computed explicitly (see equation (20.64), page 328).

Two cases are considered. In the first the Feller condition is satisfied and the volatility process is locked away from zero (‘zero lock-out Heston’), in the second it is not. Parameter values are:

<table>
<thead>
<tr>
<th>Case</th>
<th>Feller</th>
<th>( \alpha )</th>
<th>( \mu )</th>
<th>( \eta )</th>
<th>( d = 2\alpha \mu / \eta^2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZLH: Satisfied</td>
<td>0.2 0.05 0.1 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>non-ZLH: Not satisfied</td>
<td>0.4 0.04 0.4 0.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(27.31)

In each case \( S_0 = 100, v_0 = 0.04, X = 100, T = 3, r = 0.05 \) and \( \rho = -0.5 \). The explicit value of the call option is 21.326 in the first case and 20.014 in the second. (The options are at the money, so numerical problems with away from the money options are avoided.)

In section 27.3 a further case is considered. This appears in Albrecher et al. (2006) and uses \( \alpha = 1.5768, \mu = 0.0398, \eta = 0.5751 \) (giving \( d = 0.3795 \)) together with \( S_0 = 100, v_0 = 0.0175, r = 0.025 \) and \( \rho = -0.5711 \).

<table>
<thead>
<tr>
<th>Type</th>
<th>( v )-discretization</th>
<th>S-discretization</th>
<th>Equations</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>I-T</td>
<td>Euler</td>
<td>Euler on ( S )</td>
<td>(27.8) and (27.7)</td>
<td>Euler</td>
</tr>
<tr>
<td></td>
<td>Euler</td>
<td>Euler on ( R )</td>
<td>(27.10) and (27.9)</td>
<td>Euler-R-Euler</td>
</tr>
<tr>
<td></td>
<td>Joint Milstein on ( v ) and ( R )</td>
<td></td>
<td>(27.18) and (27.17)</td>
<td>Milstein</td>
</tr>
<tr>
<td>LN</td>
<td>Log-normal</td>
<td>Euler on ( S )</td>
<td>(26.14) and (27.7)</td>
<td>LN-Euler</td>
</tr>
<tr>
<td></td>
<td>Log-normal</td>
<td>Moment freezing on ( S )</td>
<td>(26.14) and (27.29)</td>
<td>LN-freeze</td>
</tr>
<tr>
<td></td>
<td>Log-normal</td>
<td>Milstein on ( R )</td>
<td>(26.14) and (27.17)</td>
<td>LN-Milstein</td>
</tr>
<tr>
<td>PC</td>
<td>PC: Euler predictors for ( v ) and ( R )</td>
<td></td>
<td>(27.27) and (27.26)</td>
<td>Euler-PC</td>
</tr>
<tr>
<td></td>
<td>Log-normal</td>
<td>Euler PC for ( R )</td>
<td>(26.14) and (27.30)</td>
<td>LN-PC</td>
</tr>
<tr>
<td></td>
<td>PC: Log-normal predictor for ( v ), Euler for ( R )</td>
<td></td>
<td>(27.27) and (27.26)</td>
<td>joint-PC</td>
</tr>
</tbody>
</table>

Table 27.1 Heston: discretization schemes
Results, computed on the spreadsheet MC_convergence_Heston_simul_call.xls are displayed in Figure 27.1. The graphs show convergence in $N$ as $N$ ranges from 1 to 1024 steps over 3 years. $M = 10000$ pairs of Wiener sample paths were simulated, stratified at the final time. Stratification was applied to a pair of independent Wiener processes, $z_1 t$ and $z_2 t$, which were then combined, as in equations (27.4) and (27.5), to give correlated process for $S$ and $v$ (so stratification may not be optimal for the combined, correlated, processes).

The upper panel in the figure shows the example when the Feller condition is satisfied, the lower when it is not. The panels plot $\log_2$(error) against $\log_2(N)$, as usual, so that slopes indicate rates of convergence.

The behaviour of the discretization methods is very variable. Errors generally decline until the level of residual simulation error is reached. This level is greater by a factor of 8 or so when the Feller condition is not met. In the lower panel residual error is large, at around 1–2% of the option value; in the upper panel it is more like 0.2%. The residual error level is reached at around 300 steps per year in these examples (with this $M$) but there is variation between methods.

The predictor–corrector based methods perform very badly in the non-ZLH case where Feller is not satisfied. They appear to be converging but only very slowly. In the ZLH case their convergence is also slower than the other methods, but they seem to converge although the Euler-PC method is behaving very erratically when $N$ is larger.

The simplest methods, Euler and Euler-R-Euler, perform surprisingly well. When there are more than 50 or so steps a year ($\log_2(N) = 6$) there is little to distinguish in these two examples between the three log-normal moment-matching methods and the two Euler-based Itô–Taylor schemes, Euler and Euler-R-Euler. For non-ZLH the Milstein methods converge more slowly than Euler and Euler-R-Euler, but for ZLH the picture is mixed.

From Figure 27.1 the best discretization method would appear to be LN-freeze. This is simple and, in the illustrations, its error decreases rapidly to the level of residual simulation noise.

Note that increasing $M$ does not automatically reduce the simulation error; it only reduces the level of residual simulation noise. It may also be necessary to increase $N$ so that simulation bias reduces down to the new, lower, level of residual noise.

### 27.3 OPTION VALUATION IN THE HESTON MODEL

Given a usable discrete simulation scheme, whose properties we at least partially understand, we can move on to investigate option pricing and the effectiveness of the speed-ups described in Part VI. We price five options: a vanilla call option, the exotic quadratic option, and three arithmetic average rate options. The discretization we use is LN-freeze. We give a comparison of using Heston CVs and auxiliary model CVs, with and without stratification.

As before we consider two sets of parameters, one with $d < 1$ and the other with $d > 1$. For $d > 1$ we use the ZLH parameters from (27.31) and for $d < 1$ we use the Albrecher et al. parameter values. We use 320 time steps.

On the whole auxiliary model CVs perform quite poorly, except in special cases. Table 27.2 gives empirical correlations, as $|\rho|$, between each CV and underlying discounted payoffs for the five options considered in this section (not all CVs apply to every option). Recall from Chapter 20 that a correlation $\rho$ gives a speed-up or order $1/(1 - \rho^2)$ (when there is no additional work involved) so that $\rho \sim 0.9$ is a speed-up of order 5 and $\rho \sim 0.99$ is a speed-up of order 50. Only four CVs achieve correlations significantly above 0.9: the geometric average auxiliary CV, in both direct and delta forms, the Heston payoff-matching CV, and the delta form of the GBM payoff-matching CV. We confirm below that only in these cases are significant gains achieved.
Figure 27.1  Heston: Convergence of discretization schemes, call price function
We briefly examine the changes to the code required for this example, and then go on to look at efficiency gains for each option in turn.

### 27.3.1 Modifications to the code

The spreadsheet MC_Heston_auxiliary_CV.xls implements the speed-ups discussed in this section. Since there may now be a choice of model for a CV the code has to change to enable us to specify which version of the CV is required. A typical CV object is shown in Figure 27.2. This is CVEuroCall1GBM. It implements the European call CV in the GBM model. CV objects know their model. Clients are able to ask the CV object what their model is by calling the CVmodel() Property. This returns a String, "GBM" in this case, specifying the model. This is held as a Private data member set up in the object constructor.

The CVmodel() Property is called by the CVmanager object. The modified version is shown in Figure 27.3. In our example application, options continue to specify which CV they want to use, even if this also includes a specification of a model. A better implementation would enable the model to be specified separately.

---

### Table 27.2 Correlations: CVs with option payoffs

<table>
<thead>
<tr>
<th>CV type</th>
<th>Model</th>
<th>Exotic</th>
<th>Call</th>
<th>Av, 4</th>
<th>Av, 16</th>
<th>Av, 64</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stock</td>
<td>GBM</td>
<td>0.38</td>
<td>0.71</td>
<td>0.69</td>
<td>0.67</td>
<td>0.66</td>
</tr>
<tr>
<td></td>
<td>Heston</td>
<td>0.41</td>
<td>0.82</td>
<td>0.76</td>
<td>0.71</td>
<td>0.72</td>
</tr>
<tr>
<td>European call</td>
<td>GBM</td>
<td>0.70</td>
<td>0.67</td>
<td>0.64</td>
<td>0.63</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Heston</td>
<td>0.89</td>
<td>0.84</td>
<td>0.83</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Geometric average</td>
<td>GBM</td>
<td>0.71</td>
<td>0.73</td>
<td>0.72</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Geometric average delta</td>
<td>GBM</td>
<td>0.96</td>
<td>0.96</td>
<td>0.93</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exotic, match</td>
<td>GBM</td>
<td>0.43</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Heston</td>
<td>0.997</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exotic, match, delta</td>
<td>GBM</td>
<td>0.84</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exotic, exact</td>
<td>GBM</td>
<td>0.44</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Panel (a): Case $d < 1$

<table>
<thead>
<tr>
<th>CV type</th>
<th>Model</th>
<th>Exotic</th>
<th>Call</th>
<th>Av, 4</th>
<th>Av, 16</th>
<th>Av, 64</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stock</td>
<td>GBM</td>
<td>0.10</td>
<td>0.93</td>
<td>0.85</td>
<td>0.81</td>
<td>0.80</td>
</tr>
<tr>
<td></td>
<td>Heston</td>
<td>0.13</td>
<td>0.91</td>
<td>0.83</td>
<td>0.80</td>
<td>0.79</td>
</tr>
<tr>
<td>European call</td>
<td>GBM</td>
<td>0.98</td>
<td>0.88</td>
<td>0.84</td>
<td>0.83</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Heston</td>
<td>0.90</td>
<td>0.85</td>
<td>0.85</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Geometric average</td>
<td>GBM</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Geometric average delta</td>
<td>GBM</td>
<td>0.998</td>
<td>0.997</td>
<td>0.98</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exotic, match</td>
<td>GBM</td>
<td>0.93</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Heston</td>
<td>0.998</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exotic, match, delta</td>
<td>GBM</td>
<td>0.98</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exotic, exact</td>
<td>GBM</td>
<td>0.92</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Panel (b): Case $d > 1$
The Sub CVManagerSV::AddCV() appends a new CV object to the array CVs(). Some CVs now come in more than one flavour, depending on the model they work with. There is both a CVEuroCallGBM CV object and a CVEuroCallHeston CV object. The latter computes the European call value in the Heston model, the former under GBM.

The Sub Get_cvs() is modified. In addition to the Heston asset value path, passed to Get_cvs() as an argument, it generates a GBM sample path from the Wiener increments that produced the Heston path. The tame path generator object, gbm_, is equipped with a procedure to produce these. When a CV value is requested, the CV object is asked for its model type. It is then passed the corresponding sample path.

### 27.3.2 The European call option

Since in the Heston model the value of a European call may be computed explicitly, if only by numerical integration, valuing a call by Monte Carlo is a benchmarking exercise. We value a call with \( X = 100 \), \( T = 1 \). The explicit value of the call option, to 3 decimal places, is 7.274 in the \( d < 1 \) case and 10.542 for \( d > 1 \).

We make a quick check to see if the degree of stratification in each dimension makes a difference to the result and then look at the effectiveness of CVs for this call.
Private CVs_() As ICV
Private cvnames_() As String
Private K_ As Long 'total number of CVs
Private DoCVs_ As Boolean 'Computes CV values specified by the mask
Private gbm_ As IPG

Private Sub Class_Initialize()
    K_ = 0
    Call AddCV(New CVGeoAverageRateGBM)
    Call AddCV(New CVquadExoticGBMExact)
    Call AddCV(New CVTerminalStockGBM): Call AddCV(New CVEuroCallGBM)
    Call AddCV(New CVquadExoticHestonMatch): Call AddCV(New CVquadExoticHeston)
    ReDim cvnames_(1 To K_) As String
    ReDim CV_mask_(1 To K_) As Boolean
    Dim k As Long
    For k = 1 To K_
        cvnames_(k) = CVs_(k).CVname
        CV_mask_(k) = False
    Next k
    Set gbm_ = New PGBM
End Sub

Private Sub Class_Terminate()
    Dim k As Long
    For k = 1 To K_
        Set CVs_(k) = Nothing
    Next k
    Set gbm_ = Nothing
End Sub

Friend Sub SetValues(ByRef data As InputManager)
    'No change
End Sub

Property Get k() As Long: k = K_: End Property

Friend Sub SetCVlink(pay As IPayoff)
    'No change
End Sub

Friend Sub Get_cvs(Spath() As Double, vpath() As Double, Swie() As Double, _
    ByRef CVs() As Double)
    If Not DoCVs_ Then Exit Sub
    If K_ = 0 Then Exit Sub
    Dim lb As Long: lb = LBound(Swie)
    Dim ub As Long: ub = UBound(Swie)
    Dim Gpath() As Double: ReDim Gpath(lb To ub) As Double 'GBM path
    Call gbm_.GetPath1(Swie, Gpath) 'Gpath is asset path under GBM
    ReDim CVs(1 To K_) As Double
    Dim i As Long
    For i = 1 To K_
        If CV_mask_(i) Then
            Select Case CVs_(i).CVmodel
                Case "GBM": CVs(i) = CVs_(i).CV(Gpath)
                Case "Heston": CVs(i) = CVs_(i).CV(Spath)
                Case Else: Call RaiseError(123, "CVmanagerSV", "Bad model")
            End Select
        End If
    Next i
End Sub

Private Sub AddCV(cv_obj As ICV)
    K_ = K_ + 1: ReDim Preserve CVs_(1 To K_) As ICV: Set CVs_(K_) = cv_obj
End Sub

'Figure 27.3  The modified CVManagerSV object
The effect of stratification

Table 27.3 shows the effect of stratification for ZLH and non-ZLH for three specifications of stratification. No other speed-ups were used.

A stratification of type \((M_1, M_2)\) means that there are a total of \(M = M_1M_2\) draws split between \(M_1\) buckets for the first dimension and \(M_2\) in the second. The uniform stratified sample is

\[
U = \{(u_i, u_j)\}_{i=1,\ldots,M_1, j=1,\ldots,M_2},
\]

where

\[
\begin{align*}
  u_i &\in \left[\frac{i-1}{M_1}, \frac{i}{M_1}\right] \\
  u_j &\in \left[\frac{j-1}{M_2}, \frac{j}{M_2}\right]
\end{align*}
\]

are drawn uniformly.

A pair \((u_i, u_j)\) is used to construct by inverse transform a Wiener sample pair \((z_i, z_j)\) for the final time \(T\), and then a pair of independent sample paths up to time \(T\) constructed by Brownian bridge. These are then transformed to give a pair of paths with the correct correlation. In this section we set \(z^S = z_i\) and \(z^V = \tilde{\rho} z_i + \rho z_j\), where \(\tilde{\rho} = \sqrt{1 - \rho^2}\), so that a stratification for \(z_i\) is a direct stratification for \(z^S\).

The results are consistent with those of Table 18.3, page 303: it seems better to stratify roughly evenly in each dimension. There is certainly no gain in emphasizing the volatility dimension, as implemented here, and the gain from distributing the stratification equally is greater than concentrating it solely in the asset dimension.

In our future analysis we use only the \((M_1, M_2) = (100, 100)\) stratification.

Efficiency gains

We use three CVs with the call: the Heston stock CV, the auxiliary model stock CV, and the auxiliary model call CV. Each CV is used by itself and a fourth case uses all four CVs simultaneously. The results are shown in Table 27.4. It presents both the \(d < 1\) and the \(d > 1\) cases, with and without stratification. Each entry shows, as usual, the simulated option value, the standard error in round brackets, the time taken (in square brackets) and the efficiency gain over the plain case (in bold).

In the \(d < 1\) example the speed-ups are poor and stratification is ineffective, bringing practically no benefit.

With \(d > 1\) the situation is very different. The CVs combine well together to bring additional benefit. Stratification is an effective efficiency enhancer, particularly with the Heston stock CV.

The effect of stratification

Table 27.3 shows the effect of stratification for ZLH and non-ZLH for three specifications of stratification. No other speed-ups were used.

A stratification of type \((M_1, M_2)\) means that there are a total of \(M = M_1M_2\) draws split between \(M_1\) buckets for the first dimension and \(M_2\) in the second. The uniform stratified sample is

\[
U = \{(u_i, u_j)\}_{i=1,\ldots,M_1, j=1,\ldots,M_2},
\]

where

\[
\begin{align*}
  u_i &\in \left[\frac{i-1}{M_1}, \frac{i}{M_1}\right] \\
  u_j &\in \left[\frac{j-1}{M_2}, \frac{j}{M_2}\right]
\end{align*}
\]

are drawn uniformly.

A pair \((u_i, u_j)\) is used to construct by inverse transform a Wiener sample pair \((z_i, z_j)\) for the final time \(T\), and then a pair of independent sample paths up to time \(T\) constructed by Brownian bridge. These are then transformed to give a pair of paths with the correct correlation. In this section we set \(z^S = z_i\) and \(z^V = \tilde{\rho} z_i + \rho z_j\), where \(\tilde{\rho} = \sqrt{1 - \rho^2}\), so that a stratification for \(z_i\) is a direct stratification for \(z^S\).

The results are consistent with those of Table 18.3, page 303: it seems better to stratify roughly evenly in each dimension. There is certainly no gain in emphasizing the volatility dimension, as implemented here, and the gain from distributing the stratification equally is greater than concentrating it solely in the asset dimension.

In our future analysis we use only the \((M_1, M_2) = (100, 100)\) stratification.

Efficiency gains

We use three CVs with the call: the Heston stock CV, the auxiliary model stock CV, and the auxiliary model call CV. Each CV is used by itself and a fourth case uses all four CVs simultaneously. The results are shown in Table 27.4. It presents both the \(d < 1\) and the \(d > 1\) cases, with and without stratification. Each entry shows, as usual, the simulated option value, the standard error in round brackets, the time taken (in square brackets) and the efficiency gain over the plain case (in bold).

In the \(d < 1\) example the speed-ups are poor and stratification is ineffective, bringing practically no benefit.

With \(d > 1\) the situation is very different. The CVs combine well together to bring additional benefit. Stratification is an effective efficiency enhancer, particularly with the Heston stock CV.
stratification the auxiliary call CV is the best CV used individually – better than the Heston stock CV. In combination an agreeable speed-up of 86 is achieved. With stratification the Heston stock CV is better, individually, than the auxiliary call CV. It seems that in the auxiliary model the stratification buckets end up being less regular than for the original Heston sample paths.

The best speed-up, 151, is achieved with all CVs and with stratification. We see below that stratification often, as in the $d < 1$ case here, conveys little or no advantage.

### 27.3.3 The exotic quadratic option

Since explicit prices for vanilla options are available in the Heston model there are two payoff-matching CVs available for the exotic quadratic option: a Heston payoff-matching CV and an auxiliary model payoff-matching CV.

Table 27.5 presents results. It uses the auxiliary and Heston payoff-matching CVs, and, since in the auxiliary model an explicit solution for the value of this exotic is known, also the exact auxiliary CV. A total of five case are shown: the plain case with no CVs; results for each of the three CVs used individually; and the final case in which every CV is used simultaneously. (The delta CV is considered separately in section 27.3.6.)

The results are unambiguous. First, only the payoff-matching Heston CV has any significant effect. The auxiliary exact CV is only as good as the auxiliary payoff-matching CV, and that is not too good at all.

Second, the gains in the $d < 1$ case are generally much worse than the $d > 1$ case. Surprisingly, the payoff-matching Heston CV gives good results in both the $d < 1$ case and the $d > 1$ case.
Third, stratification brings no benefits for the exotic option. Its cost is as great as the benefit from a slight reduction in standard error.

### 27.3.4 Valuing arithmetic average rate options

Only in this example does an auxiliary model CV really come into its own. Results are given in Table 27.6 for the $d < 1$ example, and in Table 27.7 for the $d > 1$ case. (Results for the delta CV are presented separately in section 27.3.6.) The performance of five CVs is compared with three average rate options (with 4, 16 and 64 resets up to final maturity at $T = 1$). In each table the top panel is without stratification, the bottom panel is with.

In the $d < 1$ example there is hardly any significant gain from any CV, either alone or in combination, and stratification brings no gain – indeed, results if anything are worse.

When $d > 1$, introducing stratification again reduces efficiency gains. The auxiliary geometric average CV works well with gains of around 50 for each option (consistent with the correlation shown in Table 27.2). In combination with other CVs the gains increase by a factor of 2 or 3.

#### Table 27.6

<table>
<thead>
<tr>
<th>Resets</th>
<th>Plain</th>
<th>Stock CV</th>
<th>Call CV</th>
<th>Geometric</th>
<th>All</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Auxiliary</td>
<td>Heston</td>
<td>Auxiliary</td>
<td>Heston</td>
</tr>
<tr>
<td>4</td>
<td>4.63</td>
<td>4.71</td>
<td>4.67</td>
<td>4.79</td>
<td>4.66</td>
</tr>
<tr>
<td></td>
<td>[12.2]</td>
<td>[13.1]</td>
<td>[13.1]</td>
<td>[13.1]</td>
<td>[13.1]</td>
</tr>
<tr>
<td>16</td>
<td>3.99</td>
<td>4.11</td>
<td>4.09</td>
<td>4.08</td>
<td>4.09</td>
</tr>
<tr>
<td></td>
<td>[12.7]</td>
<td>[13.5]</td>
<td>[13.5]</td>
<td>[13.6]</td>
<td>[13.6]</td>
</tr>
<tr>
<td>64</td>
<td>3.90</td>
<td>3.90</td>
<td>3.96</td>
<td>3.93</td>
<td>3.90</td>
</tr>
<tr>
<td></td>
<td>[12.1]</td>
<td>[13.0]</td>
<td>[13.0]</td>
<td>[13.0]</td>
<td>[13.0]</td>
</tr>
</tbody>
</table>

Panel (a): Average rate option, $d < 1$, no stratification

#### Table 27.7

<table>
<thead>
<tr>
<th>Resets</th>
<th>Plain</th>
<th>Stock CV</th>
<th>Call CV</th>
<th>Geometric</th>
<th>All</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Auxiliary</td>
<td>Heston</td>
<td>Auxiliary</td>
<td>Heston</td>
</tr>
<tr>
<td>4</td>
<td>4.63</td>
<td>4.69</td>
<td>4.69</td>
<td>4.74</td>
<td>4.71</td>
</tr>
<tr>
<td></td>
<td>[12.2]</td>
<td>[19.3]</td>
<td>[19.4]</td>
<td>[19.3]</td>
<td>[19.4]</td>
</tr>
<tr>
<td>16</td>
<td>3.99</td>
<td>4.06</td>
<td>4.06</td>
<td>4.02</td>
<td>4.10</td>
</tr>
<tr>
<td></td>
<td>[12.7]</td>
<td>[19.6]</td>
<td>[19.6]</td>
<td>[19.8]</td>
<td>[19.7]</td>
</tr>
<tr>
<td>64</td>
<td>3.90</td>
<td>3.90</td>
<td>3.95</td>
<td>3.96</td>
<td>3.87</td>
</tr>
<tr>
<td></td>
<td>[12.1]</td>
<td>[19.4]</td>
<td>[19.5]</td>
<td>[19.5]</td>
<td>[19.5]</td>
</tr>
</tbody>
</table>

Panel (b): Average rate option, $d < 1$, with stratification
Table 27.7  Efficiency gains in the Heston model: Average rate options, \( d > 1 \)

<table>
<thead>
<tr>
<th>Resets</th>
<th>Plain Stock CV</th>
<th>Call CV</th>
<th>Geometric All</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Auxiliary Heston</td>
<td>Auxiliary Heston</td>
<td>auxiliary</td>
</tr>
<tr>
<td>4</td>
<td>7.07 (0.092) [12.1]</td>
<td>7.02 (0.051) [13.0]</td>
<td>6.97 (0.043) [13.0]</td>
</tr>
<tr>
<td>-</td>
<td>3.3</td>
<td>3.0</td>
<td>4.2</td>
</tr>
<tr>
<td>16</td>
<td>6.16 (0.081) [12.2]</td>
<td>6.02 (0.048) [13.0]</td>
<td>6.07 (0.043) [13.0]</td>
</tr>
<tr>
<td>-</td>
<td>2.9</td>
<td>2.7</td>
<td>3.4</td>
</tr>
<tr>
<td>64</td>
<td>5.80 (0.077) [12.3]</td>
<td>5.89 (0.046) [13.0]</td>
<td>5.88 (0.044) [13.0]</td>
</tr>
<tr>
<td>-</td>
<td>2.6</td>
<td>2.5</td>
<td>2.9</td>
</tr>
</tbody>
</table>

Panel (a): Average rate option, \( d > 1 \), no stratification

<table>
<thead>
<tr>
<th>Resets</th>
<th>Plain Stock CV</th>
<th>Call CV</th>
<th>Geometric All</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Auxiliary Heston</td>
<td>Auxiliary Heston</td>
<td>auxiliary</td>
</tr>
<tr>
<td>4</td>
<td>7.07 (0.092) [12.1]</td>
<td>6.94 (0.036) [19.5]</td>
<td>6.96 (0.035) [19.5]</td>
</tr>
<tr>
<td>-</td>
<td>2.7</td>
<td>4.0</td>
<td>4.3</td>
</tr>
<tr>
<td>16</td>
<td>6.16 (0.081) [12.2]</td>
<td>6.10 (0.035) [19.5]</td>
<td>6.11 (0.038) [19.5]</td>
</tr>
<tr>
<td>-</td>
<td>1.8</td>
<td>3.4</td>
<td>2.9</td>
</tr>
<tr>
<td>64</td>
<td>5.80 (0.077) [12.3]</td>
<td>5.84 (0.038) [19.4]</td>
<td>5.87 (0.041) [19.4]</td>
</tr>
<tr>
<td>-</td>
<td>2.5</td>
<td>2.5</td>
<td>2.2</td>
</tr>
</tbody>
</table>

Panel (b): Average rate option, \( d > 1 \), with stratification

Only the auxiliary geometric average CV brings any significant gains, but although very worthwhile they are not enormous. In the \( d < 1 \) case maximum gains do not exceed 10; for \( d > 1 \) gains of over 100 are seen (in combination with other CVs).

27.3.5  Comparisons of efficiency gains

How much worse are the gains available with the Heston model compared to those obtained with GBM? How much harder is it to implement a stochastic volatility version of GBM compared to plain GBM?

Table 27.8 contrasts gains from the Heston model (for both the \( d < 1 \) and \( d > 1 \) examples) with those from plain GBM. The GBM case gains are taken from tables in Chapter 23, achieved using CVs and stratified sampling only.

The results are striking. Results for Heston are worse by orders of magnitude, often many orders of magnitude, compared to plain GBM. Conceptually Heston may be only a stochastic volatility perturbation of GBM, but numerically nothing can be done with the Heston model in any way comparable to that possible with plain GBM.
Table 27.8  Greatest efficiency gains: Heston and GBM

<table>
<thead>
<tr>
<th>Model</th>
<th>European call</th>
<th>Exotic quadratic</th>
<th>Average rate</th>
<th>4 resets</th>
<th>16 resets</th>
<th>64 resets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heston, d &lt; 1</td>
<td>4.7</td>
<td>189</td>
<td>7.6</td>
<td>6.1</td>
<td>5.6</td>
<td></td>
</tr>
<tr>
<td>Heston, d &gt; 1</td>
<td>151</td>
<td>210</td>
<td>150</td>
<td>124</td>
<td>121</td>
<td></td>
</tr>
<tr>
<td>GBM</td>
<td>1.1 × 10^6</td>
<td>2.5 × 10^9</td>
<td>1730</td>
<td>15 500</td>
<td>176 000</td>
<td></td>
</tr>
</tbody>
</table>

It makes a huge difference whether the Feller condition is satisfied or not. Unfortunately in practice, in the equity world, it is not.

27.3.6 Delta CVs

Delta CVs for the exotic quadratic payoff option and the average rate options were implemented. The delta CV for the exotic option is the delta of the payoff-matching portfolio under GBM, rather than the delta of the exact solution under GBM (see exercise 7); that for the average rate option is the delta of the GBM exact solution to a geometric average option.

The results are given in Table 27.9, with and without stratification. Plain results are not shown; they are given in previous tables.

The first observation is the relatively high cost of these CVs, roughly three to four times the cost of the plain method. The second is the relatively low gains achieved for the exotic option; these give the worst performance of any individual CV. The third observation is the high gains achieved with the average rate options. Gains increase as the number of reset dates increases. The gains when d < 1 are low but are nevertheless as good as, or better than, gains achieved by other individual CVs. In the d > 1 case, gains achieved for 16 and 64 resets are the highest individual gains, and with stratification exceed the gains made using all the other CVs combined.

It would seem that there may be situations where the delta CV outperforms the direct CV from which it is derived. This may not be surprising; a frequently rehedged approximate hedging portfolio could indeed have a higher correlation with the asset value it is hedging than a CV representing what is effectively an unadjustable portfolio, and this is confirmed by the correlations presented in Table 27.2.

One drawback with delta CVs, and partly the reason why they are being discussed separately from the other CVs, is that they combine poorly with some other CVs. It seems that because of near collinearity in payoffs, the regression matrix is close being non-positive definite, and the choldc() procedure in the RegressionOLS object throws. Instead of using an OLS procedure, a singular value decomposition should be used. This approach is adopted in Part VIII.

Table 27.9  Efficiency gains with delta CVs

<table>
<thead>
<tr>
<th>Stratification</th>
<th>d &lt; 1</th>
<th></th>
<th></th>
<th></th>
<th>d &gt; 1</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Exotic</td>
<td>Av, 4</td>
<td>Av, 16</td>
<td>Av, 64</td>
<td>Exotic</td>
<td>Av, 4</td>
<td>Av, 16</td>
<td>Av, 64</td>
</tr>
<tr>
<td>Without</td>
<td>37.6</td>
<td>4.70</td>
<td>4.11</td>
<td>3.94</td>
<td>22.49</td>
<td>6.99</td>
<td>6.083</td>
<td>5.868</td>
</tr>
<tr>
<td></td>
<td>(0.22)</td>
<td>(0.020)</td>
<td>(0.014)</td>
<td>(0.013)</td>
<td>(0.069)</td>
<td>(0.018)</td>
<td>(0.0069)</td>
<td>(0.0052)</td>
</tr>
<tr>
<td></td>
<td>[57.1]</td>
<td>[36.6]</td>
<td>[36.8]</td>
<td>[36.8]</td>
<td>[54.8]</td>
<td>[36.9]</td>
<td>[36.9]</td>
<td>[36.9]</td>
</tr>
<tr>
<td></td>
<td>0.7</td>
<td>2.6</td>
<td>3.9</td>
<td>3.9</td>
<td>5.8</td>
<td>8.8</td>
<td>46</td>
<td>73</td>
</tr>
<tr>
<td>With</td>
<td>37.6</td>
<td>4.69</td>
<td>4.08</td>
<td>3.91</td>
<td>22.46</td>
<td>6.97</td>
<td>6.095</td>
<td>5.872</td>
</tr>
<tr>
<td></td>
<td>(0.20)</td>
<td>(0.017)</td>
<td>(0.019)</td>
<td>(0.011)</td>
<td>(0.065)</td>
<td>(0.017)</td>
<td>(0.0053)</td>
<td>42</td>
</tr>
<tr>
<td></td>
<td>[63.5]</td>
<td>[43.0]</td>
<td>[43.1]</td>
<td>[43.1]</td>
<td>[62.7]</td>
<td>[43.3]</td>
<td>[43.0]</td>
<td>[43.1]</td>
</tr>
<tr>
<td></td>
<td>0.8</td>
<td>3.2</td>
<td>4.0</td>
<td>5.0</td>
<td>5.8</td>
<td>8.7</td>
<td>67</td>
<td>93</td>
</tr>
</tbody>
</table>
27.4 SUMMARY

The Heston model is hard to implement effectively. The CIR process is difficult and expensive to use both individually and as part of a larger model. The Heston model has explicit solutions for some options but they are explicit only in the requires-numerical-integration sense. They are very time consuming to compute. Discretization, bias and speed are very big issues for the Heston model.

This chapter has examined some discretization and variance reduction methods used with Heston. Long-step Monte Carlo is possible with Broadie and Kaya but is expensive; on the other hand, we have seen that short-step Monte Carlo requires a very large number of time steps to achieve accuracy. We found LN-freeze (implementing equations (26.14) and (27.29)) to be a reasonably effective compromise between simplicity and accuracy, with low bias quite quickly reducing to residual levels.

Once a sample path is available further variance reduction techniques may be applied. We have looked only at control variate methods and discovered that they are nothing like as effective for the Heston model as their counterparts are in a plain GBM setting. The most effective speed-ups are based on solutions intrinsic to the Heston model: payoff matching with Heston calls and puts. In some cases we have seen that a GBM auxiliary CV can also be effective. Delta CVs can provide reasonable speed-ups.

The degree of difficulty posed by the Heston model depends on whether or not the Feller condition is satisfied. If zero is accessible, then neither bias nor speed can be improved easily. On the other hand, if zero is inaccessible then prospects are not quite so dim.

The Heston model has been around for a while but only relatively recently has it begun to be used extensively and with the requirement of greater accuracy. Research on the numerical implementation of the Heston model is progressing, and one expects further improvements to continue to be made, but at the time of writing the use of simulation methods with Heston remains a challenging problem.

27.5 EXERCISES

1. Write down the 1.5 strong Itô–Taylor discretization for the Heston model. As you read this, are there in the literature any effective ways to simulate the $I_{j_1,j_2,j_3}$ terms? If you find any, implement the scheme, making whatever bodges seem least inappropriate. Does the scheme converge? Does it converge faster than order 0.5? Can it blow up?

2. This chapter has considered stratifying Wiener sample paths only at the final time. Replace the Wiener generator object with an LD-based generator, to enable LD sampling at times other than the final time (see Chapter 19). What order of speed-up is now available? How do the $d < 1$ and $d > 1$ cases differ?

3. Can any form of importance sampling be used with the Heston model?

4. What CVs could be used with Heston to value a barrier option? Implement your candidates. Do they work?

5. Would the results in, for example, Table 27.5 be significantly different if instead of the LN-freeze discretization a Milstein discretization had been used?

6. What is the Broadie and Kaya (2006) simulation method for Heston? Implement it. How expensive is it compared to other schemes considered in this chapter? To achieve a given level of residual simulation error how much more expensive is a one-step Broadie and Kaya simulation method compared to the short-step methods described in this chapter?

7. For the exotic quadratic payoff option construct a delta CV object based on the delta of the GBM exact solution. How well does it perform?
In this part we investigate the Longstaff and Schwartz least squares (LSLS) Monte Carlo method (Longstaff and Schwartz (2001)) for valuing American and Bermudan options. The exposition focuses almost entirely on the American and Bermudan put.

The LSLS method has the advantage of simplicity but, even so, using it effectively is not straightforward. We find that it is possible to value Bermudan puts quite accurately, with low standard error and low bias, but only by using control variate techniques. American puts can be valued only approximately, as it is very expensive to value Bermudan puts with sufficient reset dates so that their values lie within simulation noise of the value of the American option: a very high number of reset dates is required.

The part starts in Chapter 28 with a brief review of some issues surrounding the valuation of American and Bermudan options. The concepts underlying the LSLS method are presented. Chapter 29 discusses the construction of plain estimates to the early exercise boundary (EEB). Various sets of basis functions are implemented and compared. The resulting EEBs are compared to a reference boundary obtained from a lattice method, and their quality is assessed. Chapter 30 implements the plain LSLS method and shows that, with or without stratification, results are quite dismal. Finally, Chapter 31 shows how the performance of the LSLS method can be improved beyond recognition. It turns out that by using a control variate method (due to Rasmussen (2005)) a very accurate estimate of the EEB can be obtained and this is the key to obtaining fast, unbiased, option values.

We conclude that it is possible, using the LSLS/Rasmussen implementation described in this part, to value Bermudan puts quickly and accurately with little bias. Given accurate methods to value Bermudan puts with different numbers of reset dates, leading to a common final maturity date, it is possible to value American puts using path-wise Richardson extrapolation. See Chirayukool and Webber (2010).

I would like to thank Pokpong Chirayukool for his contribution in reviewing chapters in this part. All errors are my own.
This page intentionally left blank
No book on Monte Carlo is complete without a discussion of the application of the method to the valuation of Bermudan and American style options. A major application is in the money market where there is a crucial need to get out prices and hedges for Bermudan swaptions. Since in a Libor market model the underlying processes are many and complex, there is no alternative but to resort to Monte Carlo. We do not look at this particular application here; instead we focus purely on issues with the canonical American put.

The ambition of this part is limited. We describe only the Longstaff and Schwartz least squares (LSLS) Monte Carlo method (Longstaff and Schwartz (2001)) in any detail, investigating its strengths and weaknesses. Our emphasis is on its properties and on its effective implementation in VBA.

American options almost always have to be valued numerically. Either implicitly or explicitly, this means identifying the early exercise boundary: the values of the state variables that border a region within which it is optimal to exercise the option.

With lattice and PDE methods, when they can be used, it is easy to identify a numerical approximation to the early exercise boundary. You just evolve backwards from the final maturity time, making an optimal decision at each step whether to exercise or not. This immediately determines the location of a boundary separating values where exercise is optimal from those where it is not.

If lattice and PDE methods cannot be used, perhaps because either the option is path-dependent or there are too many state variables, then for better or worse Monte Carlo may have to be used. This presents problems. Monte Carlo normally works forwards towards the final maturity date; but since you don't know what future option values are, even in expectation, you cannot make optimal decisions.

Techniques for applying Monte Carlo to value American and Bermudan options usually work by evolving forwards to the final time and then working backwards, solving for the early exercise boundary and the option value as you go. For a Monte Carlo method to work backwards through time there are just two possible ways to evolve the underlying state variables; either to have an evolution that computes and stores all the sample paths for every time step, or to construct a backwards Brownian bridge in a slice-wise evolution. The latter is much more convenient but not always possible.

Two-pass methods (see Chapter 31) have a second, separate, simulation phase that computes option values from an early exercise boundary found in the first pass. The second pass can be either forwards or backwards.

There are two competing sources of error particular to valuing Bermudan and American options by Monte Carlo. The first is foresight error. If you make decisions that are influenced by a knowledge of what happens next, then you vastly overestimate the option value. The second is in getting the early exercise boundary wrong. A sub-optimal exercise decision results in an undervalued option.

A third source of error, not particular to American options, is a variation of ordinary discretization error. In this instance a Monte Carlo method uses a finite number of time steps and thus values a Bermudan option with that number of exercise times – not an American option. As this effect is quite serious we shall discuss it in section 28.1.2.

Even using LSLS, valuing a book of American options is still not really feasible. Each option has to estimate separately its own early exercise boundary, and this is very time consuming. In a two-pass method each option has an expensive pre-pass (first pass) to find the early exercise boundary. Sample paths are shared but each option has to perform its own individual estimation from them. (The second-pass sample paths can be shared in the usual way at marginal cost.)
The location of the EEB depends not only on the option but also on the asset process. In particular, for a GBM, it depends on \( r \) and \( \sigma \). As estimates of \( r \) and \( \sigma \) change from day to day (or minute to minute) so the EEB changes and must be re-estimated.

We start with a discussion of how American style options can be formulated and then look at issues with using Monte Carlo methods for their valuation. We move on to describe the LSLS method, showing how it may be applied to value American puts and European bond options as special cases of the general structure.

Throughout we deal with a single standard American option. This is ATM expiring in one year at \( T = 1 \) with strike \( X = 100 \) on an underlying asset following a GBM with current value \( S_0 = 100 \) and volatility \( \sigma = 0.2 \) with short-rate \( r = 0.05 \). The true value of this option (established using a lattice method) is 6.090370 to 6 decimal places.

We also consider Bermudan versions of the standard option. The Bermudan option with \( R \) reset dates (exercise times) can be exercised at times \( T_i = (i/R)T \) for \( i = 0, 1, \ldots, R \). The (lattice) values of the standard Bermudans are given in Table 28.1, page 453.

### 28.1 AMERICAN OPTIONS

This section gives some general background on American options. First we describe a general American option, and then discuss how Bermudan option values converge to American option values as the number of reset dates increases.

#### 28.1.1 Introduction and general formulation

American and Bermudan options have early exercise features that enable them to be exchanged at the holder’s discretion for other options, or for cash, at a set of dates, continuous or discrete, up to a final maturity date. For American options the range of dates is continuous from the start date up to the final maturity date (although as a variation there may be an initial period in which no exchange is allowed). For Bermudan options the set of possible exchange dates is discrete.

The classical example is the American put where the holder may at any time exchange an underlying asset for cash. For an asset worth \( S_t \) at time \( t \) that can be exchanged for cash with value \( X \), the payoff upon an exchange at time \( t \) is \( (X - S_t)^+ \).

There are many other examples in the market: American straddles (with payoff \(|S_t - X|\)); Bermudan swaptions; convertible bonds; instalment options, *et cetera*.

These have in common the need to make an optimal decision: at each exercise date, conditional on the value of the asset, should the option be exercised or not?

#### A general exercise structure

The American put is a very simple, but nevertheless challenging, example of an American-style option. In general an option could be exchanged for another option from among a set of other options. Exchanges might be possible only if some condition is met,\(^1\) and a rebate may be paid upon an exchange. We briefly describe a general American style option which can be valued using an extension of the basic LSLS method.

Let \( S_t \) be an underlying state variable taking values in state space \( S \). An American exercise structure on \( S_t \) is a set \( \mathcal{C} = \{c^q\}_{q=0,\ldots,Q} \) of instruments written on \( S \) together with a specification of how these

\(^1\) Callable bonds may have features of this sort.
instruments may be exchanged for one another. We suppose that \(c^0\) is the zero option, that is, an option that expires immediately with no payoff. At time 0 we suppose that the owner of this structure holds option \(c^1\).

Each option \(c^d \in C\) has an exchange structure that determines how it can be exchanged for other options in \(C\). There are two parts to the exchange structure. The first is an exchange specification that determines when an exchange into each other option in \(C\) can take place. We suppose there is a range of allowable values of \(S\) at which exchange is possible. The second, the rebate structure, is a specification of the rebate paid to the option holder upon an exchange. The rebate may depend on the option being acquired and on the prevailing value of the asset.

For two options \(c^q, c^p \in C\) let \(T^{q,p} \subseteq [0, T]\) be the set of times at which a counterparty holding \(c^q\) can potentially exchange it, at their discretion, for \(c^p\). (Since \(T < \infty\) we are not considering perpetual options.) For \(t \in T^{q,p}\) write \(S_t^{q,p}\) for the set of values of the asset for which the exchange is permitted. Set \(S^{q,p} = \{S_t^{q,p}\}_{t \in T^{q,p}}\). The set \(\mathcal{S} = \{S_t^{q,p}\}_{0 \leq q,p \leq Q, q \neq p}\) is an exchange structure on \(C\). It represents our attempt to model a conditional exchange structure.

The rebate structure is a function \(H_t^{q,p}(S) \in \mathbb{R}\) that determines the cash paid to the holder of option \(c^q\) if it is exchanged into option \(c^p\) at time \(t \in T^{q,p}\). At time \(T\) if the option holder holds option \(c^q\) then our convention is that the payoff to the option holder is \(H_t^{q,0}(S_T)\).

If \(T^{q,p} = \emptyset\) then \(c^q\) can never be exchanged into \(c^p\). If \(T^{q,p_1} \cap T^{q,p_2} \neq \emptyset\) and if for some \(t \in T^{q,p_1} \cap T^{q,p_2}\) we have \(S_t^{q,p_1} \cap S_t^{q,p_2} \neq \emptyset\) then there exist values of \(S_t\) at which \(c^q\) is simultaneously exchangeable into either \(c^{p_1}\) or \(c^{p_2}\). Should this occur we presume that the choice of what to exchange into is at the discretion of the option holder. (We assume that only a single exchange can take place at any instant of time.)

This defines a general American style option. Clearly a general American option could be very complicated, but the American put in a relatively simple special case. It can be exchanged only for cash and so is defined as follows. Let \(C = \{c^0, c^1\}\) where the option holder initially holds option \(c^1\), the American put. The exchange specification has \(T^{1,0} = [0, T]\) with \(S_t^{1,0} = \mathbb{R}^+\) for all \(t \in T^{1,0}\). The rebate structure is \(H_t^{1,0}(S) = (X - S_t)^+,\) defined for all \(t \in [0, T]\), the familiar put payoff function.

**Exercise strategies**

Suppose that at time 0 the option \(c^1 \in C\) is held. If the underlying state variables follow Markov processes, and if neither the American nor exchange structures are path-dependent, then under (to us) relatively mild conditions an exercise strategy is determined by a set of subsets \(\mathcal{P} = \{P^q_t\}_{q=1, \ldots, Q, t \in [0, T]}\) of \(\mathcal{S}\). For a given \((t, q)\) we have \(P^q_t = \{E^q_t, 0, \ldots, E^q_t, \bar{Q}\}\) such that \(\mathcal{S} = \bigcup_{P=0, \ldots, Q} E^q_t, P\). If at time \(t\) the counterparty holds option \(c^q\) then \(P^q_t\) determines the exercise strategy:

1. If \(S_t \in E^q_t, q\) then no exchange should be made.
2. If \(S_t \in E^q_t, P, p \neq q\), then an immediate exchange into \(c^p\) should be made.

\(\mathcal{P}\) is determined under optimality conditions that maximize the value of \(c^1\) at time 0.

Suppose that \(\mathcal{P}\) is optimal. \(C^q_t = E^q_t, q\) is called is called the continuation region. For \(S_t \in C^q_t\) the exercise time is the first exit time of \(S_t\) from \(C^q_t\). If \(S_t \in \overline{C^q_t} = \mathcal{S} \setminus C^q_t\) the option should be exercised immediately. Since more than one exchange may be possible at any one time \(\overline{C^q_t} \subseteq \bigcup_{p \neq q} E^q_t, P\) is divided into regions where the exchange into each \(c^p\) is optimal. If no exchange into \(c^p\) is possible at that time then \(E^q_t, P = \emptyset\).

If for \(p_1 \neq p_2\) we have \(E^{q_1, p_1} \cap E^{q_2, p_2} \neq \emptyset\) then exchange into either \(c^{p_1}\) or \(c^{p_2}\) is equally valuable.

The boundary \(\partial E^{q_t, P} = C^q_t \cap \partial E^{q_t, P}\) is the early exercise frontier for exercise from \(c^q\) into \(c^p\) at time \(t\).
For an American put it is possible to show that \( P_t^1 \equiv P_t = (E_t^0, C_t) \) where

\[
C_t = (S_t^*, \infty) \subseteq \mathbb{R}^+,
\]

and \( E_t^0 = \mathbb{R}^+ \setminus C_t = [0, S_t^*] \), for some critical value \( S_t^* \). That is, at every time \( t \leq T \) there is a critical value \( S_t^* \) such that it is optimal to exercise if and only if \( S_t \leq S_t^* \).

For a Bermudan put there is similar result: if \( t_i \) is an exercise time then there is a critical value \( S_t^* \) such that one exercises at time \( t_i \) if and only if \( S_{t_i} \leq S_t^* \).

If the location of the early exercise frontier is known then it is easy to value American options, or at least a Bermudan approximation to the American option, by Monte Carlo. One exercises, with known payoff, when the boundary is hit.

If the early exercise frontier is not known then implicitly an approximation is made to it so that it is possible to determine when, according to the method, the option should be exercised. Since the frontier determined by the method is only an approximation to the theoretical boundary, and is therefore sub-optimal, the option holder will exercise suboptimally and the option will tend to be undervalued.\(^2\) The challenge for valuing American options by Monte Carlo is to establish the location of the frontier with accuracy sufficient for the purposes of the valuation.

### 28.1.2 Convergence of Bermudan values to American values

Part of the options’ folklore (for some folk) is that the value of a Bermudan option converges rapidly to that of an American option as the number of exercise times increases. This is not true. Figure 28.1 shows convergence of standard Bermudan put values, with \( R \) reset dates, to the value of the standard American put. The Bermudan values, \( p_R \), are computed in the spreadsheet Lattice_Bermudan_benchmark.xls on a trinomial lattice; estimated values to 5 decimal places are given in Table 28.1. Exercise times are spaced evenly between 0 and the final maturity time \( T \) (and are not, for instance, weighted towards the final time). The \( x \)-axis shows \( \log_2(R) \). As \( R \) increases the value of the Bermudan is converging to that of the American put.

Table 28.1 shows that differences in values remain significant even with a large number of exercise times. With 64 resets (so that \( \log_2(64) = 6 \)) the value of the Bermudan is \( \sim 6.08118 \), roughly 0.15% below the value 6.09037 of the American. This difference can be detected easily with a reasonably accurate Monte Carlo method.\(^3\)

Figure 28.2 plots \( \ln(\text{error}) \) against \( \ln(R) \). The values are computed in Lattice_Bermudan_benchmark.xls on a trinomial lattice with either 51200, 102400 or 204800 time steps. The plot has three lines, one for each lattice specification. The slopes of the lines give the rates of convergence to the American value. The figure shows that convergence is uniform at a rate \( \sim 1 \). Applying Richardson extrapolation one obtains the estimate given above of the American put value.

Using a lattice method it is easy to find (an approximation to) the early exercise boundary. Figure 28.3 shows the early exercise boundary for standard Bermudan puts with from 2 to 1024 reset dates. The markers are the critical values of the asset at each reset date below which the option should be exercised. The markers have been joined up to emphasize the shape of the boundary but, of course, the option can be exercised only on a set of discrete times. From the initial time each boundary rises gradually at first but increases steeply close to the final maturity time.

---

\(^2\) Although if the location of the boundary has been established by a method employing some degree of foresight then the option value will be biased high.

\(^3\) Later we present results whose standard errors are better than \( 1 \times 10^{-4} \). On the lattice only a Bermudan with more than about 5000 reset dates has a value this close to the American value. The implication is that to value an American option with bias obscured by this degree of simulation noise an LSLS Monte Carlo method would require at least 5000 time steps.
Figure 28.1  Convergence of Bermudan option values to an American value as the number of reset times, $R$, increases: price against $\log_2(R)$

<table>
<thead>
<tr>
<th>$R$</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>American</th>
</tr>
</thead>
</table>

The boundary has been computed on a lattice with 204,800 time steps. Even at this level of refinement the space step on the lattice is of the order of $10^{-2}$ so that the boundary is accurate only to 2 decimal places or so.

As the number of resets increases, so the level of the boundary goes down. Empirically this is at a rate roughly $O\left(-\frac{1}{2}\right)$ in the number of reset times. The figure shows a limiting curve estimated by Richardson extrapolation with this convergence rate at 64 reset dates. This lies a little over 1 below the boundary for the 64 reset Bermudan. This results in an American option value less than 0.01 greater than the Bermudan option value.

In a plain Monte Carlo method, simulation noise usually swamps the bias due to having too few exercise times. However when effective speed-ups are used it is easy to reduce the standard error to levels where the bias is obvious. This has enormous implications for cost. When accuracy is important not only must a large number of sample paths be used but, to reduce bias, a large number of time steps must also be used. Since the regression step is very expensive the effect upon the performance of the algorithm is alarming. Numerical examples are given in Chapter 30.
28.1.3 Valuing a mimicking barrier option

Consider a barrier option with barrier \( \{b_t\}_{t \in [0,T]} \) such that \( S_0 > b_0 \) and let \( \tau \) be the first hitting time to the barrier (or \( T \), whichever come first). Let \( X \geq \max_{t \in [0,T]} \{b_t\} \) be a strike and define the payoff \( H^b_\tau \) to the option at the hitting time to be

\[
H^b_\tau = \begin{cases} 
X - b_\tau, & \tau < T, \\
(X - S_T)^+, & \tau = T.
\end{cases}
\]  

(28.2)

The hitting time \( \tau \) determined by the barrier is a stopping time for an American put with strike \( X \). It is an optimal stopping time only if \( b_t \) is the early exercise boundary of the American put. For given strike \( X \), and fixed parameters for the process followed by \( S_t \), the value of this mimicking barrier option is maximized when \( b_t = b^*_t \) is the early exercise boundary of the American put.

The spreadsheet MC_American_CV_one_pass_set_barrier.xls uses Monte Carlo to compute values of Bermudan versions of this mimicking barrier option. The object OptionFlexiKO has a barrier level set after being read in from the front end. We use it to value a set of options \( v^k \), \( k \geq 0 \).

We quantify the effect on option prices of using a perturbation of the true EEB. Let \( \{b^*_t\}_{i=1,...,64} \) be the early exercise boundary for the standard Bermudan option with 64 exercise dates, \( t_i = (i/64)T \), computed by a lattice method (and plotted in Figure 28.3). Set \( b^k = \{b^k_t\}_{i=1,...,64} \) where \( b^k_t = e^{-k(T-t)}b^*_t \), so that
Early exercise frontiers: Bermudan options

Figure 28.3 Convergence of early exercise frontiers

\[ b^0 = b^*, \] and let \( v^k \) be the value of the mimicking Bermudan option with barrier \( b^k \). \( v^k \) is maximized when \( b^k \) is closest, in some sense, to the true early exercise boundary of the Bermudan. Let \( v^* = 6.08118 \) be the value of the Bermudan put computed on the lattice.

Figure 28.4 plots differences \( v^k - v^* \) computed by the spreadsheet for various values of \( k \) close to zero.\(^4\) The error bars represent ±1 standard error. When \( |k| \leq 0.00125 \) is small the differences are around \( 2 \times 10^{-5} \), within a fraction of the standard error (~6 \times 10^{-5}) of zero. However, when \( |k| = 0.01 \) the error is large, around \( 1 \times 10^{-3} \), easily visible as bias in the option value. The effect on the barrier level is also noticeable: we have \( b^*_1 \sim 82.054 \) but \( b^{0.01}_1 \sim 81.251 \), a difference of ~1%. By contrast \( b^{0.00125}_1 \sim 81.954 \), a percentage difference of ~0.1%.

The lesson is that small inaccuracies in the early exercise boundary may not affect the valuation too much, particularly if only a moderate number of sample paths is being used, and any error is hidden by simulation noise, but larger deviations rapidly become plain. The chief objective of a Monte Carlo method is to be able to use, implicitly, an accurate estimate of the early exercise boundary.

28.2 MONTE CARLO AND AMERICAN OPTIONS

This section very briefly reviews some ideas about American Monte Carlo methods but it is chiefly concerned with a description of the LSLS method in section 28.2.3.

\(^4\) The method uses 16 stratification times, \( M = 16 \times 10^6 \), and a CV based on a European put (see Chapter 31).
28.2.1 General formulation

As usual we assume that there are \( N \) time steps, \( 0 = t_0 < t_1 < \ldots < t_N = T \) and \( M \) sample paths. The times \( T = \{ t_i \}_{i=1,...,N} \) are a discrete reset structure in the sense that we assume that actions can be taken only at times \( t \in T \).

Let \( S_{i,j} \) denote the asset value at time \( t_i \) along the \( j \)th sample path. We use the following notation:

- \( V_{i,j}^q \) – The value of option \( c^q \) at a point \((S_{i,j}, t_i)\).
- \( H_{i,j}^{q,p} \) – The rebate paid to the holder of option \( c^q \) upon exchange into option \( c^p \) at the point \((S_{i,j}, t_i)\), if exchange is possible. \( H_{i,j}^{q,p} \) is interpreted as the coupon received by, or premium paid to, the option holder at the point \((S_{i,j}, t_i)\) if no exchange is made.
- \( Q_{i,j}^q \) – The continuation value of the option \( c^q \): the value of \( c^q \) at \((S_{i,j}, t_i)\) if it is not exchanged at \( t_i \) but optimally exchanged at times \( t_{i+1}, \ldots, t_N \).

Set

\[
V_{i,j}^{q,p} = H_{i,j}^{q,p} + Q_{i,j}^p, \tag{28.3}
\]

for all \( q, p \). \( V_{i,j}^{q,p} \) is the value of the option, initially held as \( c^q \) at \((S_{i,j}, t_i)\), if exchanged for option \( c^p \) (or, if \( q = p \), not exchanged). If an optimal exchange decision is made the value of the option
at \((S_{i,j}, t_i)\) when \(c^q\) is held is
\[
V_{i,j}^q = \max_{p=0, \ldots, Q} V_{i,j}^{q,p},
\]
with exchange being to any \(c^{p^*}\) such that \(V_{i,j}^q = V_{i,j}^{q,p^*}\).

If it is possible to find, or at least to approximate, the \(Q_{i,j}^q\) then the option values \(V_{i,j}^q\) can be found, implicitly defining an early exercise boundary.

In a lattice this is easy. One works backwards from the final time valuing every option simultaneously. Given a complete set of option values \(V_{i+1,j}^q\) for the \(j\)th level at time \(t_{i+1}\), continuation values \(Q_{i,j}^q\) for time \(t_i\) are
\[
Q_{i,j}^q = e^{-r_{i,j} \Delta t} \sum_{k=1}^K p_{i,j,k} V_{i+1,j+k}^q,
\]
where \(p_{i,j,k}\) is the branching probability from node \((i, j)\) at time \(t_i\) and level \(j\) to node \((i+1, j+k)\) at time \(t_{i+1}\), and \(r_{i,j}\) is the riskless rate at node \((i, j)\). Option values \(V_{i,j}^q\) for time \(t_i\) are just
\[
V_{i,j}^q = \max_{p=0, \ldots, Q} \{H_{i,j}^{q,p} + Q_{i,j}^p\},
\]
as above.

In a Monte Carlo method it is not possible to apply this approach directly. If iterating back and assuming that a complete set of \(V_{i+1,j}^q\) have been found for time \(t_{i+1}\), one is still at a loss as to what weights \(p_{i,j,k}\) should be used to iterate back to time \(t_i\).

### 28.2.2 Monte Carlo methods for the American put

There are a number of approaches to finding \(Q_{i,j}^q\) in a Monte Carlo method. Usually these are presented in the restricted context of the American put. A detailed review can be found in Glasserman (2004). In this case, where there is a single set of continuation values, the \(q\) index can be dropped.

Bundling algorithms (for instance, Barraquand and Martineau (1995)) approximate the \(Q_{i,j}\) at each \((i, j)\) by grouping together paths that have similar values of \(S_{i,j}\) at time \(t_i\).

Given \(\epsilon > 0\) let \(B_{i,j} = \{k \mid |S_{i,k} - S_{i,j}| < \epsilon\}\) and \(m_{i,j} = |B_{i,j}|\), then set
\[
\tilde{Q}_{i,j} = e^{-r_{i,j} \Delta t} \frac{1}{m_{i,j}} \sum_{k \in B_{i,j}} V_{i+1,k}.
\]
\(\tilde{Q}_{i,j}\) approximates \(Q_{i,j}\). It evaluates the expectation in (28.5) using equal weights on the future values from paths in that bundle.

Stochastic mesh methods (Broadie and Glasserman (1997), (2004)) use a double pass. The first pass determines an exercise boundary and also determines a high bias estimator. The second pass uses the boundary determined in the first pass to price the option, giving a low bias estimator. The true option value is probably somewhere in between.

Finally, the method discussed in this chapter, the LSLS functional fit method, approximates \(Q_{i,j}\) with a functional form. We review this next.
28.2.3 The LSLS Monte Carlo method

Given a value \( V^q_{i+1, j} \) for time \( t_{i+1} \) in a Monte Carlo method the continuation value at time \( t_i \) along the \( j \)th sample path is \( \hat{Q}^q_{i, j} = e^{-r \Delta t} V^q_{i+1, j} \). Unfortunately one cannot simply set

\[
V^q_{i, j} = \max_{p=0, \ldots, Q} \{ H^{q,p}_{i, j} + \hat{Q}^p_{i, j} \}
\]

as this value embodies perfect foresight over the next time step. The option is hugely overvalued. Instead one must make an exchange decision, to exchange \( q \) for \( p^* \) say, based on some other criteria, and then set

\[
V^q_{i, j} = H^{q,p^*}_{i, j} + \hat{Q}^{p^*}_{i, j}.
\]

If a functional form, \( Q^q(S) \), for the true continuation value functions were available then one would set \( \tilde{Q}^q_{i, j} = Q^q(S_{i,j}) \) and the exchange rule is

\[
\text{If } H^{q,p^*}_{i, j} + \tilde{Q}^{p^*}_{i, j} = \max_{p=0, \ldots, Q} \{ H^{q,p}_{i, j} + \tilde{Q}^p_{i, j} \} \text{ then exchange into } p^*.
\]

Longstaff and Schwartz (2001) (but see also Carriere (1996) and Tsitsiklis and Van Roy (2000)) devised a simple method for determining an approximation to \( Q_t \) that largely sidesteps the foresight issue. One chooses a functional form \( f \) to approximate \( Q_t \) so that at time \( t_i \) one has \( \hat{Q}^q_{i, j} = f(S_{i,j} | \theta^q_i) \). Parameters \( \theta^q_i \) are found by minimizing a RMSE criteria \( J^q_i \),

\[
\theta^q_i = \arg \min_{\theta} J^q_i(\theta),
\]

\[
J^q_i(\theta) = \sqrt{\frac{1}{M} \sum_{j=1}^{M} (\hat{Q}^q_{i, j} - f(S_{i,j} | \theta))^2}.
\]

This assumes that for fixed \( i \) the set of \( \hat{Q}^q_{i, j} \) are a noisy perturbation of the true continuation values \( Q^q_{i, j} \).

The Longstaff and Schwartz (LSLS) algorithm

The full algorithm is, in summary:

1. Simulate a set of paths \( S^j = \{S_0, \ldots, S_{N,j}\}, j = 1, \ldots, M, \) and set \( V^q_{N,j} = H^{q,0}_{N,j} = H_N^q(S_{N,j}) \).
2. At each \( t_i, 1 \leq i < N \), suppose that \( V^q_{i+1, j} \) are known for all \( q \) and \( S_{i+1, j} \). Set \( \hat{Q}^q_{i, j} = e^{-r \Delta t} V^q_{i+1, j} \). Find parameters \( \theta^q_i \) so that \( \hat{Q}^q_{i, j} \sim \tilde{Q}^q_{i, j} = f(S_{i,j} | \theta^q_i) \).
3. Find \( p^* \) such that \( H^{q,p^*}_{i, j} + \tilde{Q}^{p^*}_{i, j} = \max_{p=0, \ldots, Q} \{ H^{q,p}_{i, j} + \tilde{Q}^p_{i, j} \} \). This determines the exchange decisions at \( (S_{i,j}, t_i) \).
4. Set \( V^q_{i, j} = H^{q,p^*}_{i, j} + \tilde{Q}^{p^*}_{i, j} \).
5. At time \( t_0, \) set \( \hat{Q}^q_0 = e^{-r \Delta t}(1/M) \sum_{j=1}^{M} V^q_{1,j} \) and \( V_0 = \max_{p=0, \ldots, Q} \{ H^{q,p}_{0}(S_0) + \tilde{Q}^p_{0} \} \).

Note that one does not set \( V^q_{i, j} = H^{q,p^*}_{i, j} + \tilde{Q}^{p^*}_{i, j} \).
**LSLS and the American put**

The American put is a simple application of the general method. The American exercise structure is $C = \{c^0, c^1\}$ where $c^1$ is the American put. Dropping the $q$ superscript we write $V_{i,j}$ for $V_{i,j}^1$, $Q_{i,j}$ for $Q_{i,j}^1$, and $H_{i,j}$ for $H_{i,j}^0 = (X - S_{i,j})^+$, et cetera. Set $V_{N,j} = H_{N,j} = (X - S_{N,j})^+$. At a subsequent time step $t_i$, $1 \leq i < N$, set $Q_{i,j} = e^{-r \Delta t} V_{i+1,j}$ and, following the algorithm, find parameters $\theta_i$ so that $Q_{i,j} \sim \tilde{Q}_{i,j} = f(S_{i,j} \mid \theta_i)$.

We have

$$V_{i,j}^0 = H_{i,j} = (X - S_{i,j})^+, \quad (28.13)$$

$$V_{i,j}^1 = \tilde{Q}_{i,j} = f(S_{i,j} \mid \theta_i), \quad (28.14)$$

since $Q_{i,j}^0 \equiv H_{i,j}^1 \equiv 0$. The decision rule is to exercise if $V_{i,j}^0 > V_{i,j}^1$, that is, if $(X - S_{i,j})^+ > f(S_{i,j} \mid \theta_i)$. One then sets

$$V_{i,j} = \begin{cases} (X - S_{i,j})^+, & \text{if exercised,} \\ e^{-r \Delta t} V_{i+1,j}, & \text{if not exercised.} \end{cases} \quad (28.15)$$

**A simpler idea?**

Instead of adopting a seemingly complicated fitting procedure what is wrong with the following? At each time $t_i$ estimate a critical value $\hat{S}_i^*$ as

$$\hat{S}_i^* = \arg \max_S \frac{1}{M} \left( \sum_{S_{i,j} \leq S} (X - S_{i,j})^+ + \sum_{S_{i,j} > S} \tilde{Q}_{i,j} \right). \quad (28.16)$$

Note that the location of the critical value does not depend on the values $S_{i,j}$ of $S$ at time $t_i$ so in principle $\hat{S}_i^*$ is a legitimate estimate of critical value at time $t_i$.

$^5$ When $M$ is sufficiently large foresight bias will be small.

Unfortunately the early exercise boundary $\hat{S}^* = \{\hat{S}_i^*\}$ found by this procedure turns out to be very noisy. Since the procedure is costly – naively solving equation (28.16) iteratively is quadratic in $M$ – the expense rapidly becomes much too great. A better, faster, method, like LSLS, is required.

**Formulation in terms of cashflows**

Instead of iterating back, storing the values $\{V_{i,j}\}_{j=1,\ldots,M}$ from one step to the next, one can instead just keep track of when, along each sample path, the option is exercised and the payoff at that time. For each $j$ set

$$i_j = \min\{k \geq i \mid f(S_{k,j} \mid \theta_k) < H_{k,j}\}. \quad (28.17)$$

$i_j$ is the first time index on or after time $t_i$ when, on the $j$th sample path, the option is exercised.

We have

$$V_{i,j} = e^{-r(t_{i+1} - t_i)} H_{i,j}. \quad (28.18)$$

Along each sample path, instead of iterating the values $\{V_{i,j}\}_{j=1,\ldots,M}$ backwards at each step, iterate the vector of pairs $\{(H_{i,j}, t_i)\}_{j=1,\ldots,M}$ where initially $N_j = N$.

$^5$ A feature noted and exploited by Rasmussen (2005).
This alternative procedure involves the same amount of computation as the original but it enables the early exercise boundary to be found quite easily. For each $i$ set $S_i^* = \{S_{i,j} \mid 0_j = i\}$; this is the set of $S$ values at time $i$ at which the option is exercised. Set

$$S_i^* = \max_S \{S \in S_i^*\}.$$  \hspace{1cm} (28.19)

$S_i^*$ is the greatest simulated asset value at time $t_i$ at which the option is exercised.\(^6\) The early exercise boundary is the set $\{S_i^*\}_{i=1,\ldots,N}$. Equation (28.19) defines an early exercise boundary for an American (actually Bermudan) put. Other American style options, with different structures, may need more complex boundary specifications.

**Bond options**

There are a number of instances of instruments that at first sight do not appear to require an American Monte Carlo treatment but which in fact do. An example is a European bond option. At the maturity of the bond option the holder can choose whether or not to purchase an underlying bond. The difficulty arises if a model is being used in which bond values are not known explicitly. If a Monte Carlo method is being used, then the underlying bond value has to be estimated by fitting a functional form. Despite being a European option, an American Monte Carlo method must be employed to determine whether exercise is optimal.

Consider a European bond call option. In the language of the general framework let $C = \{c^0, c^1, c^2\}$ where $c^0$ is the zero option, $c^1$ the bond option, and $c^2$ the underlying pure discount bond.\(^7\) Let $T_1$ be the maturity date of the bond option and $T = T_2 > T_1$ the maturity date of the underlying bond. The exchange structure is defined by

$$T^{1,2} = \{T_1\}, \quad H_{T_1}^{1,2} = -X,$$

$$T^{1,0} = \{T_1\}, \quad H_{T_1}^{1,0} = 0,$$

$$T^{2,0} = \{T_2\}, \quad H_{T_2}^{2,0} = 1,$$

with all other specifications trivial. At time $T_1$ the bond option matures; either it expires worthless (being exchanged for the zero option), or the holder exchanges it for the bond, $c^2$, costing $X$ straightaway but receiving $1$ at time $T_2$.

Suppose for illustration that we are attempting to value this bond option in a short rate interest rate model, such as Vasicek (1977) or CIR (1985) – see James and Webber (2000) or Brigo and Mercurio (2001) – where the state variable is the short rate $r_t$.

Write $r_{i,j}$ for the value at time $t_i$ of $r_t$ along the $j$th sample path. Suppose that $T_1 = t_{i_1}$ and $T_2 = t_{i_2}$. Along the $j$th sample path the value at time $T_1$ of the underlying bond can be approximated as

$$\hat{B}_{i_1,j} = \hat{Q}_{i_1,j} = \exp \left(-\sum_{i=t_{i_1}}^{i_2-1} r_{i,j} \Delta t \right)$$

\hspace{1cm} (28.23)

\(^6\) $S_i^*$ is defined only when $S_i^*$ is non-empty. Set $S_i^* = 0$ if $S_i^* = \emptyset$.

\(^7\) Paying no coupons, and paying back $1$ at maturity.
(but see Chapter 26). A naive exercise decision rule is:

\[
\text{If } \hat{B}_{i,j} > X \text{ then exercise, else do not exercise.} \tag{28.24}
\]

Unfortunately this uses perfect foresight of the future path of interest rates along each sample path. The option value is vastly overestimated. Instead the LLSLS methodology can be applied to estimate the true bond value as a function of the value of the short rate at time \( T_1 \).

Exercise can occur only at time \( T_1 \). Set \( \tilde{B}_{i,j} = f(r_{i,j} | \theta_{i}) \) where \( \theta_{i} \) best fits \( f \) to \( \hat{B}_{i,j} \). The exercise rule is

\[
\text{If } \tilde{B}_{i,j} > X \text{ then exercise, else do not exercise.} \tag{28.25}
\]

The value \( V_{i,j} \) of the bond option at time \( T_1 \) is

\[
V_{i,j} = \begin{cases} 
(\hat{B}_{i,j} - X)^+, & \text{if exercised}, \\
0, & \text{if not exercised}. 
\end{cases} \tag{28.26}
\]

Note that the payoff, if exercised, may perfectly well be negative if this reflects the eventual payoff along the sample path.

### 28.3 SUMMARY

The American put is a simple example of an American style option. The LLSLS method can value American (actually Bermudan) puts, and can be extended to value other much more complex American style options, but it has limitations. The key idea is to find the continuation region by smoothing out the noise in the simulated continuation values. The underlying continuation value function \( Q_t(S) \) is approximated by fitting to the simulated sample of continuation values a suitably chosen functional form. The continuation region is found by comparing the estimate of \( Q_t(S) \) to the exercise values \( H_t(S) \). If \( Q_t(S) \) is too complicated, so that a reasonable fit is not possible, or lies outside the domain of applicability of the chosen functional form, then the method will fail.

Other options, perhaps not obviously requiring an American Monte Carlo method, can and should be valued using LLSLS.

### 28.4 EXERCISES

1. Consider a redeemable American bond option that can at any time be redeemed by the holder for an amount of cash, \( R \): at each moment in time the holder can take either a payoff of \( R \) or a payoff of \( (B_{i,j} - X)^+ \), or do nothing, where \( B_{i,j} = B_{t_i}(r_{i,j}) \) is the value at time \( t_i \) of a pure discount bond maturing at time \( T \) when the current short rate is \( r_{i,j} \). How could this option be placed into an exercise structure of the type described in this chapter?

2. Can the general framework be extended to include options exercisable at the discretion of either counterparty to the option? For concreteness consider the following redeemable, callable, convertible\(^8\) bond

---

\(^8\) A redeemable bond can be exchanged for cash at the discretion of the bond holder; a callable bond is exchangeable for cash at the discretion of the bond issuer; a convertible bond can be converted in stock at the discretion of the bond holder.
maturing at time $T$, issued by a company with share price $S_t$. Suppose that the size of the bond issue is negligible relative to the firm’s equity base so that the stock price and the bond price can be determined independently. Suppose that it can be

1. redeemed at times $t_1 < \ldots < t_N = T$, where $t_i = (i/N)T$, for an amount $R_i$ at time $t_i$, conditional upon $S_{t_i} \geq \frac{3}{2}S_0$;

2. callable at any time $t \in [t_1, t_N]$ for an amount $C$;

3. convertible at any time $t \in [t_1, t_N]$; if converted at time $t \in [t_i, t_{i+1})$ the bond holder receives $\gamma_i$ shares each with value $S_t$.

How would its value at time $t_0 < t_1$ be found?

3. Consider the following two Bermudan options, each exercisable at times $t_1 < \ldots < t_R = T$, where $t_i = (i/R)T$, $i = 1, \ldots, R$:

   (a) A Bermudan straddle with payoff $H_{t_i} = |S_{t_i} - X|$ if exercised at time $t_i$.

   (b) A Bermudan butterfly with payoff $H_{t_i} = \max(0, X_1 - |S_{t_i} - X_2|)$ if exercised at time $t_i$, where $0 < X_1 < X_2$.

Modify the spreadsheet Lattice_Bermudan_benchmark.xls to value these two options. Estimate the values of the corresponding American style options. How rapidly do the values of the Bermudan options converge to the corresponding American style options as $R$ increases? For concreteness suppose that $S_t$ follows a GBM with current value $S_0 = 100$, volatility $\sigma = 0.2$, and short rate $r = 0.05$, and suppose $X = X_2 = 105$ and $X_1 = 10$.

4. Consider the following two component option, maturing at time $T$. At time 0 the option holder owns an exchangeable call. If the holder holds the exchangeable call at maturity, she receives a payoff of $(S_T - X^c)^+$ for a strike $X^c$.

   The exchangeable call can be exchanged for an exchangeable put. If the exchangeable put is held at maturity the holder receives a payoff of $(X^p - S_T)^+$ for a strike $X^p > X^c$. The exchangeable put can be exchanged for the exchangeable call.

   An exchange from call to put or from put to call can be made at any time, and any number of exchanges can be made, except that once an exchange is made the holder has to wait for a period $\tau$ before a further exchange can be made.

   Suppose that $S_t$ follows a GBM with $S_0 = 100$, $\sigma = 0.2$ and $r = 0.05$, with $X^c = 90$, $X^p = 110$, $T = 1$ and $\tau = 0.1$.

   (a) Can this option be valued on a lattice? Describe the continuation and exchange regions.

   (b) How would this option be valued using an LSLS style method?

   Is valuation simplified if exchanges can be made only on a discrete set of dates, perhaps daily?
The LSLS method approximates the continuation value function \( Q_t(S) \) with a functional form,

\[
Q_t(S) \sim f(S \mid \theta_t),
\]

for some optimal set of parameters \( \theta_t \). The estimated value of the EEB (for the American put) at time \( t \) is the maximum value of \( S \) such that \( H_t(S) \geq f(S \mid \theta_t) \).

It is convenient to express the approximating function \( f \) as a sum of basis functions. However the choice of basis function is not clear cut. This chapter briefly discusses some of the issues involved and elaborates on their effect on valuation in the plain LSLS method.

We look first at various possible choices of basis functions. We mention the need to use singular value decomposition (SVD) instead of OLS to find coefficient values. Finally numerical examples are given of fits to continuation values, estimates of the early exercise boundary, and the effect on option pricing of errors in the EEB.

### 29.1 APPROXIMATING THE CONTINUATION VALUE FUNCTION

For equation (29.1) the parameters \( \theta \) are chosen to minimize a norm \( J(\theta) = \| \hat{Q}_{i,j} - f_{i,j}(\theta) \| \) where \( f_{i,j}(\theta) = f(S_{i,j} \mid \theta) \). Suppose that \( f \) is a linear combination of a small number of basis functions,

\[
f(S \mid \theta_i) = \sum_{k=1}^{K} \theta^k_i f_k(S),
\]

for some set of basis functions \( \{f_k\}_{k=1,\ldots,K} \) that spans (a suitable subspace of) \( C^\infty(\mathbb{R}) \) as \( K \to \infty \), and coefficients \( \theta_i = (\theta^1_i, \ldots, \theta^K_i)' \). When \( K \) is small solving equation (29.1) is a straightforward linear programming problem. This makes the optimization procedure explicit and fast (although ‘fast’ is a relative term).

For a fixed time \( t_i \) set \( Q = (\hat{Q}_{i,j})_{j=1,\ldots,M} \in \mathbb{R}^M \) and \( F = (f_k(S_{i,j}))_{j=1,\ldots,M} \in \mathbb{R}^{M \times K} \). We want to find a vector \( \theta = (\theta^k_i)_{k=1,\ldots,K} \in \mathbb{R}^K \) of coefficients such that \( \epsilon = Q - F\theta \) is minimized. A least squares linear regression solution finds an optimal \( \theta \) as

\[
\theta = (F'F)^{-1}F'Q.
\]

Unfortunately this direct approach fails and SVD must be used. See section 29.2.5.

### 29.2 CHOICES FOR BASIS FUNCTIONS

When \( f \) is expressed as a weighted sum of basis functions \( \{f_k\}_{k=1,\ldots,K} \), as in equation (29.2), the choice of basis functions may make a difference. Various choices are possible; the original Longstaff and Schwartz
paper considered several sets and others have been investigated subsequently. Here, for the American put, 11 sets of basis functions are compared. They fall into three groups:

1. **Families of orthogonal polynomials.** Three orthogonal families are used: Laguerre, Hermite and Legendre polynomials. Monomial powers are also included here as a fourth family even though they do not form an orthogonal set. Two versions of each family are used: plain and scaled. The plain versions are the classically defined versions of the polynomials. The scaled versions rescale the independent variable to obtain greater numerical stability.

2. **‘Natural’ basis functions.** A set of functions with a natural, financial, relationship to the underlying problem. These come in plain and scaled versions.

3. **B-splines.** General-purpose curve-fitting functions with no natural connection to any specific valuation problem.

We examine each group in a little more detail.

### 29.2.1 Orthogonal polynomials

Families $b = \{b_i\}$ of orthogonal polynomials satisfy an orthogonality relationship of the form

$$\int_D b_i(x)b_j(x)k(x)\,dx = \delta_{ij}n_i$$  \hspace{1cm} (29.4)

where $D$ is the domain over which the orthogonality relationship applies, $k(x)$ is the kernel function with respect to which the family is orthogonal, $\delta_{ij}$ is the Kronecker delta and $n_i$ is a normalization constant (a function of $i$). Properties$^1$ of several polynomial families are given in Table 29.1. Conventionally for

<table>
<thead>
<tr>
<th>Type</th>
<th>$b_1$</th>
<th>Recurrence relationship</th>
<th>$k(x)$</th>
<th>$D$</th>
<th>$n_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power:</td>
<td>$x$</td>
<td>$b_i(x) = x b_{i-1}(x)$</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>Legendre:</td>
<td>$x$</td>
<td>$b_i(x) = \frac{1}{i}[(2i - 1)x b_{i-1}(x) - (i - 1)b_{i-2}(x)]$</td>
<td>$1$</td>
<td>$[-1, 1]$</td>
<td>$\frac{2}{2i + 1}$</td>
</tr>
<tr>
<td>Laguerre:</td>
<td>$1-x$</td>
<td>$b_i(x) = \frac{1}{i}[(2i - 1 - x)b_{i-1}(x) - (i - 1)b_{i-2}(x)]$</td>
<td>$e^{-x}$</td>
<td>$[0, \infty)$</td>
<td>$1$</td>
</tr>
<tr>
<td>Hermite:</td>
<td>$x$</td>
<td>$b_i(x) = x b_{i-1}(x) - (i - 1)b_{i-2}(x)$</td>
<td>$e^{-\frac{1}{2}x^2}$</td>
<td>$(-\infty, \infty)$</td>
<td>$i!\sqrt{2\pi}$</td>
</tr>
<tr>
<td>Gegenbauer:</td>
<td>$2\alpha x$</td>
<td>$b_i(x) = \frac{1}{i}[(2x(i + \alpha) - 1)b_{i-1}(x) - (i + 2\alpha - 2)b_{i-2}(x)]$</td>
<td>$(1 - x^2)^{\alpha - \frac{1}{2}}$</td>
<td>$[-1, 1]$</td>
<td>$\frac{\pi 2^{1-2\alpha} \Gamma(i + 2\alpha)}{i^{3/2} i! (i + \alpha) \Gamma^2(\alpha)}$</td>
</tr>
<tr>
<td>Chebyshev (1st kind):</td>
<td>$x$</td>
<td>$b_i(x) = 2xb_{i-1}(x) - b_{i-2}(x)$</td>
<td>$(1 - x^2)^{-\frac{1}{2}}$</td>
<td>$[-1, 1]$</td>
<td>$\pi$, $i = 0$; $\frac{\pi}{2}$, $i \neq 0$.</td>
</tr>
<tr>
<td>Chebyshev (2nd kind):</td>
<td>$2x$</td>
<td>$b_i(x) = 2xb_{i-1}(x) - b_{i-2}(x)$</td>
<td>$(1 - x^2)^{\frac{1}{2}}$</td>
<td>$[-1, 1]$</td>
<td>$\frac{\pi}{2}$</td>
</tr>
</tbody>
</table>

$^1$ See Abramowitz and Stegun (1965).
polynomial families the first basis function has index 0, the second index 1, and so on. In each case the first basis function is constant, \( b_0 \equiv 1 \). The table shows the second basis function, \( b_1 \), the recurrence relationship between three successive basis functions satisfied by the family, and the values of \( D, k(x) \) and \( n_i \).

The first four families listed in Table 29.1 are implemented in the spreadsheets that accompany this part. In the VBA implementations the first few basis functions in each family are usually computed with explicit formulae. Subsequent basis functions, however, are computed using the recurrence relationship.

For the standard American and Bermudan puts considered in this part the scaled versions \( b_s^t(x) \) of power, Laguerre, Legendre and Hermite polynomials are obtained from the plain versions given in the table by setting \( b_s^t(x) = b_t(x') \) where \( b_t \) is the unscaled form and \( x' = (x - X)/S_0 \). This has the advantage that for reasonable values of strike \( X \) and initial value \( S_0 \) the scaled variable \( x' \) lies close to zero and, with very high probability, lies in \([-1, 1] \).

The plain versions of the orthogonal polynomials work very poorly and should not be used; the scaled versions should be used instead. The poor performance is not unexpected. The orthogonal families behave, asymptotically, like monomial powers. The \( k \)th monomial power of the asset value, when the asset value is chunky – close to 100 say – has value around \( 100^k \). As \( k \) increases, this rapidly becomes enormous. We see below that even for a simple American put perhaps at least six basis functions should be used (and maybe more) with the plain LSLS method. Any attempt to find coefficients to basis functions with values of the order \( 10^{12} \) to fit to a curve with values of order 10 is silly and should be expected to fail.

### 29.2.2 Natural basis functions

Natural basis functions are functions related to the problem structure. For an American put the value of a European put is a naturally related function, as is the underlying stock value. For other options, other functions would be natural choices. For instance, for a Bermudan swaption one could use (i) the underlying swap value, (ii) the underlying Libor value and (iii) the nearest forward Libor rate. If these values can be computed as functions of the state variables of the model, they could serve as basis functions.

Write \( BS(x) = BS(x \mid X, T, \sigma, r) \) for the value of the Black–Scholes European put with the specification corresponding to that of the American put and set \( y = BS(x) \). The natural basis we use is:

\[
\begin{align*}
b_N^0(x) &= 1, \\
b_N^1(x) &= y, \quad b_N^2(x) = x, \\
b_N^3(x) &= y^2, \quad b_N^4(x) = xy, \quad b_N^5(x) = x^2, \\
b_N^6(x) &= y^3, \quad b_N^7(x) = xy^2, \quad b_N^8(x) = x^2y, \quad b_N^9(x) = x^3, \\
&\vdots
\end{align*}
\]

The scaled version is obtained by replacing \( x \) by \( x' \) and \( y \) by \( y' = (y - X)/S_0 \).

### 29.2.3 B-splines

B-splines are a widely used family of general-purpose curve-fitting functions. They are used in derivatives applications to fit to the term structure of interest rates, or a term structure of ATM volatility, for instance. One very important feature is that they are local functions, zero outside a relatively narrow range. We shall be using cubic B-splines. At every value of \( x \) only four cubic B-splines are non-zero. This property makes
them very easy to work with; B-splines are one of the few sets of basis functions that OLS regression actually works with without (almost invariably) blowing up.

A spline of order \( K \) is a piece-wise polynomial function of order \( K \) that is differentiable \( K - 1 \) times everywhere. On an interval \([\xi_0, \xi_n]\), with knot points \( \xi_0 < \xi_1 < \ldots < \xi_{n-1} < \xi_n \), a \( K \)-order spline \( f^K(x) \) has the canonical representation

\[
f^K(x) = \sum_{j=0}^{K} a_j x^j + \sum_{i=1}^{n-1} b_i (x - \xi_i)^+^3
\]  

(29.6)

so that it is defined by a set of \( n + K \) parameters \( a_j, b_i \). The spline \( f^K \) is a polynomial of order \( K \) on every interval \((\xi_i, \xi_{i+1})\). It is differentiable only \( K - 1 \) times at the knot points \( \xi_i \) themselves.

As representation (29.6) is not often convenient, a B-spline representation is often used instead. We now fix \( K = 3 \) so that we consider only cubic splines. Define an additional set of six knot points, \( \xi_{-3} < \xi_{-2} < \xi_{-1} < \xi_0, \xi_0 < \xi_{n+1} < \xi_{n+2} < \xi_{n+3} \) and define

\[
b^S_i(x) = \sum_{k=i}^{i+4} \left( \prod_{j=4 \atop j \neq k}^{i+4} \frac{1}{\xi_j - \xi_k} \right) (x - \xi_k)^+^3
\]  

(29.7)

for \( i = -3, \ldots, n - 1 \). It is not hard to show that \( b^S_i(x) \) is non-zero only on the interval \([\xi_i, \xi_{i+4}]\). If \( x \in (\xi_k, \xi_{k+1}) \) then \( b^S_i(x) \) is non-zero only for \( i = k - 3, k - 2, k - 1, k \). For \( x \in [\xi_0, \xi_n] \) the set \( \{b^S_i\}_{i=-3}^{n-1} \) spans the set of (cubic) splines defined by the parameterization (29.6).

How are \( b^S_i \) used to fit to a set of points \( \{(S_j, c(S_j))\}_{j=1}^{M} \)? Fix \( n \) and set \( m_k = \lfloor (k/n)M \rfloor, k = 0, \ldots, n \), and suppose that every \( S_j \) lies within some bounded open interval \((\xi_0, \xi_n)\). For \( m \geq 1 \) write \( S_{(m)} \) for the \( m \)-th order statistic of the set of \( S_j \); the element \( S_{(m)} \) of \( S = \{S_j\}_{j=1}^{M} \) such that \([S \in S | S \leq S_{(m)}]\) = \( m \). For \( j = 1, \ldots, n - 1 \) choose \( \xi_j \in (S_{(m_j)}, S_{(m_j+1)}) \) so that, in the interval \([\xi_j, \xi_{j+1}]\), there lie \( m_j + 1 - m_j \sim M/n \) points.

Let \( l = \xi_1 - \xi_0 \) and set \( \xi_{-k} = \xi_0 + kl \) for \( k = -1, -2, -3 \). Similarly let \( u = \xi_n - \xi_{n-1} \) and set \( \xi_{n+k} = \xi_n + ku \) for \( k = 1, 2, 3 \).

A B-spline fit finds coefficients \( \lambda_i, i = -3, \ldots, n - 1 \), and for \( x \in [\xi_0, \xi_n] \) the functional approximation is

\[
f(x) = \sum_{i=-3}^{n-1} \lambda_i b^S_i(x).
\]  

(29.8)

Note that a B-spline fit requires a minimum of four B-spline basis functions.

### 29.2.4 Comparisons of basis functions

Six sets of basic functions are shown in Figure 29.1. Panels (a), (c), (e) and (f) show scaled power, Laguerre, Legendre and Hermite basis functions. Panel (a) shows B-spline basis functions and panel (d) the set of scaled natural basis functions. Each panel shows the first six basis functions from the family it illustrates.

B-spline functions are excellent for a wide range of fitting problem. Here though their flexibility may not be an advantage.
Figure 29.1 Sets of basis functions

(a) B-spline basis functions

(b) Scaled power basis functions

(c) Scaled Laguerre basis functions

(d) Scaled natural basis functions

(e) Scaled Legendre basis functions

(f) Scaled Hermite basis functions
Scaled Laguerre functions have shapes that look a little like the continuation value function. For this reason they may plausibly make a good set of basis functions; this turns out to be half true.

29.2.5 Practicalities: OLS and SVD

There are major difficulties in fitting basis functions using ordinary OLS techniques. There is a relatively large amount of data (at each time step there is one data point for each sample path) and a relatively small number of basis functions, so why should there be a problem? The issue is that the optimization surface is quite flat. Many different combinations of basis functions give numerically indistinguishable values of the criteria function. Operationalized, this results in an attempt being made to invert a matrix that, to within numerical indeterminacy, is singular. An OLS regression method cannot cope with this. When there are more than five or six basis functions an OLS method gives up.

Instead a singular value decomposition method must be used. This method is robust. It is more expensive, but it has the virtue of always working.

Our implementation of SVD is based on Press et al. (2007), transcribed from C++ to VBA. For a description of SVD see Gulub and Van Loan (1996) as well as Press et al.. SVD requires a tolerance level to be set. Too high a level results in sub-optimal coefficients; too small a level becomes expensive.

Even using SVD the unscaled version of some of the orthogonal polynomial families give very biased results once more than a few (8 or 9 or so) basis functions are used. One prefers a method to blow up entirely than to ‘work’ and give useless and meaningless results, but having identified the potentially problematic families we can avoid them.

29.2.6 Fitting to continuation values

Some examples of fits to the continuation value function produced by the basis functions are shown in Figure 29.2. These are taken from a run to value a 64 reset standard Bermudan put. For clarity this illustration uses only 2500 sample paths.

Panel (a) shows the regression line at time step 32 ($T = 0.5$), panel (b) at step 4 ($T = 1/16$). The panels show values $H_{i,j}$ of the payoff function, the scatter of continuation values, $\hat{Q}_{i,j} = e^{-r\Delta t}V_{i,j}$, and the regression lines produced by a B-spline fit with six basis functions. When the regression line lies below the exercise value, the option value is set to be the exercise value; when it lies above, the option value is set to be the continuation value.

The exact shape of the fitted curve is not of primary importance. It is used only to determine whether the option should or should not be exercised. Basis functions that are prone to produce wobbly fits are not necessarily worse than those that produce better looking fits, although one may express some doubts.

In panel (a) the regression line defines the critical asset value for time $T = 0.5$, the greatest value of $S$ at this time step at which a sample path is exercised, to be around $S^* = 84$. The regression line to the right of the critical value looks a little like an early exercise frontier. Unfortunately with so few sample paths it is a very poor approximation. Fortunately the algorithm uses only the estimate of the critical value, not the full regression curve.

Note that in panel (a) in the region $S \leq 74$ the regression line lies above the exercise value, so that the algorithm determines that the option should not be exercised at these points. This is an artifact; for the American put it is sub-optimal to exercise anywhere to the left of the critical value. The regression is

---

2 Two warnings: (1) Avoid using $l$ as a loop counter. In some fonts (Courier New, for instance) it looks almost identical to 1. (2) If to avoid the first problem you replace $l$ by $ll$ please make sure that you do not accidentally type $11$ instead.

3 The actual value is $\sim 85.12$. 


Figure 29.2  Continuation values and regression fit
attempting to fit through very few data points at this end; the best fit is achieved with sub-optimal exercise. With a more reasonable number of sample paths this feature becomes less significant although it does not disappear.

Panel (b) is just four steps away from the initial time. The algorithm determines that no paths are exercised. The regression line is quite wobbly but this does not influence the algorithm. The wobbles are an artifact of the fit (with this number of sample paths), not of the continuation value itself, which remains regular. Close to the initial time it becomes hard to fit the continuation value curve. Points are less dispersed, being clustered close to the initial value, and few points, if any, are far enough to the left to lie below the early exercise boundary.

29.3 THE EARLY EXERCISE BOUNDARY

Given a fit at each time $t_i$ the critical value of the asset, bordering the continuation regions, can be determined. For the American put it is found from equation (28.19), page 460. Examples of the early exercise boundary computed by the basic LSLS method (in the spreadsheet MC_American.CV.EExB.xls) are given in Figure 29.3. This shows how the approximation to the early exercise boundary improves as the number of sample paths increases. The figure shows approximations based on 2500, 50625, 250000 and $10^6$ sample paths with six scaled Laguerre basis functions (and plain Monte Carlo evolution without stratification). For small $M$ the estimates are very poor. They improve greatly as the number of sample paths increases. Errors persist at time steps close to the initial time since there are few observations of early exercise in this region. The feature was noted by Rasmussen (2005) who proposed a method to improve the simulation of the boundary in this region (which we do not implement here).

![Figure 29.3 Convergence to the early exercise boundary](image-url)
As an estimate of the error in the approximation to the early exercise boundary we compute the root mean squared error (RMSE) between the Monte Carlo and the lattice estimates. These are shown in Table 29.2. Three sets of RMSEs are given. The first is computed from all 64 reset dates, the second from reset dates 9 to 64 only, and the third from reset dates 17 to 64 only. The second and third estimates, by removing the noise from the initial segment of the curve, give a better indication of how the fit is improving over the bulk of the curve.

As the number of sample paths increases so the RMSE decreases – tending to a minimum level representing the bias in the estimate caused by having a finite number of basis functions. We investigate how this bias decreases and the number of basis functions we use increases. Errors in the early exercise boundary with \( M = 10^6 \) sample paths (generated with 16 low-discrepancy stratification times) with various numbers of basis functions are shown in Figure 29.4. Panel (a) shows results for B-spline basis functions; panel (b) for scaled Laguerre basis functions.

Table 29.3 shows average RMSEs from a number of estimation runs as the number of basis functions varies. The results are very noisy and the interpretation is not clear. For scaled Laguerre basis functions one may perhaps say that the fit is not getting worse. For B-splines it does not seem possible to say even that. This somewhat weakly quantifies the appearance of panel (a) in figure 29.4. The B-spline fits appear to remain noisy as the number of basis functions increases, so (scaled) Laguerre basis functions may be preferred.

As the number of basis functions increases the LSLS values, although biased low, converge to the true option value (Stentoft (2004)). We are concerned to establish a level of bias small enough to be obscured by simulation noise. This means we require sufficient

1. **Basis functions.** Too few and the fit is sub-optimal leading to a poor estimate of the early exercise boundary.
2. **Time steps.** Too few and the Bermudan we are valuing will not be a good approximation to the American option.
3. **Sample paths.** Too few and not only is (i) the standard error high, as in any Monte Carlo valuation, but (ii) the early exercise boundary will be poorly estimated.

### Table 29.2

Errors in the early exercise boundary as \( M \) increases (6 scaled Laguerre basis functions)

<table>
<thead>
<tr>
<th>RMSEs</th>
<th>2500</th>
<th>50 625</th>
<th>160 000</th>
<th>250 000</th>
<th>10^6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1–64</td>
<td>3.5</td>
<td>1.5</td>
<td>1.2</td>
<td>0.8</td>
<td>0.8</td>
</tr>
<tr>
<td>9–64</td>
<td>3.6</td>
<td>1.4</td>
<td>1.0</td>
<td>0.5</td>
<td>0.3</td>
</tr>
<tr>
<td>17–64</td>
<td>3.7</td>
<td>1.2</td>
<td>0.6</td>
<td>0.4</td>
<td>0.3</td>
</tr>
</tbody>
</table>

### Table 29.3

Errors in the early exercise boundary as \( M \) increases

<table>
<thead>
<tr>
<th>RMSEs</th>
<th>Scaled Laguerre</th>
<th>B-spline</th>
</tr>
</thead>
<tbody>
<tr>
<td># basis functions:</td>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td>1–64</td>
<td>0.8</td>
<td>1.2</td>
</tr>
<tr>
<td>9–64</td>
<td>0.3</td>
<td>0.4</td>
</tr>
<tr>
<td>17–64</td>
<td>0.3</td>
<td>0.2</td>
</tr>
</tbody>
</table>

---

4. Only observations for time steps where a Monte Carlo estimate exists were used.
5. In this case the sample paths were generated with 16 stratification times (with stratification specified as in Chapter 19).
Implementing Models of Financial Derivatives

Convergence to early exercise boundary,
Plain rollback, 16 stratifications, \( M = 1000000 \),
B-spline basis functions

Error in the early exercise boundary

Time step

-1
0
1
-1
-0.5
0.5
1.5

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Figure 29.4 Errors in approximations to the early exercise boundary

(a) 6, 12, and 24 B-spline basis functions

(b) 6, 12, and 24 scaled Laguerre basis functions
29.4 EFFECT ON VALUATION

It is easy to observe the effect on valuation of using a sub-optimal estimate early exercise boundary. We use the spreadsheet MC_American.CV_one_pass_set_barrier.xls to value a mimicking barrier option for the Bermudan put with 64 reset dates. Two examples are given. The first looks at error in the option value caused by error in the boundary as the number of sample paths used to estimate the boundary increases. The second investigates how error decreases as the number of basis functions increases.

Convergence in $M$

To gain an accurate estimate of the EEB in a plain LSLS method a large number of sample paths is needed. We estimate the EEB in runs using $M_1$ sample paths stratified at 16 times (and 6 scaled Laguerre basis functions). These EEBs are then passed in a separate run to an OptionFlexiKO object to value a mimicking barrier option. The second run uses $M = 10^6$ sample paths in every case (with stratified sampling and a put CV – see Chapter 31).

Results are given in Table 29.4. This shows option values against $M_1$, alongside the RMSEs for the EEBs relative to the lattice EEB. Allowing for stratification, standard errors in the option value are $\sim 0.00017$. Computation times for the mimicking barrier option valuation are shown separately to the time taken to estimate the EEB. It is clear that bias is decreasing as RMSE decreases and that increasing $M_1$ is absolutely necessary to decrease bias. However, even with $M_1 = 10^6$ sample paths, the maximum possible in a single run with the implementation used here, there is still bias of the order 0.001 in the option value. (We see in Chapter 31 how this can be decreased very considerably.)

Figure 29.5 displays $\ln(\text{error})$ against $\ln(M_1)$. The rate of convergence is roughly $O(-2/3)$. Were this rate to be maintained, then to reduce the error to 0.0001, around $M_1 = 3 \times 10^7$ sample paths would be required, and to reduce it to 0.00001, with this convergence rate, would need $\sim 10^9$ sample paths. It is not feasible to take so long to estimate the boundary.

Convergence in $K$

Estimates for the early exercise boundary were obtained from runs with either 6, 12 or 24 B-splines or scaled Laguerre basis functions. The boundary estimation passes used $10^6$ sample paths stratified at 16 times. For each combination of basis functions, three estimates of the early exercise boundary were obtained, resulting in boundaries similar to those in Figure 29.4.

The EEB estimates were used to value a mimicking barrier option valuation. In these valuation passes $16 \times 10^6$ sample paths were used (with stratified sampling at 16 times and a put CV). The resulting option values have a standard error of $\sim 7.5 \times 10^{-5}$.

Table 29.4 Bias in option values: effect of $M_1$. (Timings in seconds)

<table>
<thead>
<tr>
<th>$M_1$</th>
<th>2500</th>
<th>50625</th>
<th>$1.6 \times 10^5$</th>
<th>$2.5 \times 10^5$</th>
<th>$1 \times 10^6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>EEB time</td>
<td>1.7</td>
<td>35</td>
<td>110</td>
<td>172</td>
<td>743</td>
</tr>
<tr>
<td>Option value</td>
<td>6.0346 (0.00030) [117]</td>
<td>6.0760 (0.00030) [119]</td>
<td>6.0790 (0.00030) [119]</td>
<td>6.0785 (0.00030) [119]</td>
<td>6.0803 (0.00031) [119]</td>
</tr>
<tr>
<td>RMSE (17–64)</td>
<td>3.3</td>
<td>0.8</td>
<td>0.5</td>
<td>0.4</td>
<td>0.3</td>
</tr>
</tbody>
</table>
In(error)

Figure 29.5 Valuation error and $M_1$

Table 29.5 Bias in option values

<table>
<thead>
<tr>
<th>Basis type</th>
<th># Basis functions:</th>
<th>6</th>
<th>12</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scaled Laguerre:</td>
<td></td>
<td>6.0803</td>
<td>6.0804</td>
<td>6.0806</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{-21}</td>
<td>{-16}</td>
<td>{-14}</td>
</tr>
<tr>
<td>B-spline:</td>
<td></td>
<td>6.0797</td>
<td>6.0805</td>
<td>6.0800</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{-33}</td>
<td>{-15}</td>
<td>{-26}</td>
</tr>
</tbody>
</table>

Results are shown in Table 29.5. The number in curly brackets is the number of standard deviations the LSLS option value is away from the true value; this is a measure of bias. The differences in option values are significantly greater than the standard error. Every option value is biased low (recall the lattice value for this option is $v^* = 6.08118$) but for the scaled Laguerre basis functions the degree of bias is decreasing (slightly) as the number of basis functions increases.

For B-spline basis functions the bias increases going from 12 to 24 basis functions. The values closely reflect the RMSEs in the EEB estimates. It is possible that the flexibility of a B-spline fit is overfitting the continuation value function when $K$ is too large, resulting in nastier EEBs and worse option values.

We might conclude from Table 29.5 that B-splines may not be a suitable set of basis functions and that scaled Laguerre should be preferred, but the evidence here is slender. In fact when more sophisticated methods are employed we find that B-splines offer advantages over scaled Laguerre basis functions (see Chapter 31).

29.5 SUMMARY

For the standard Bermudan put we have seen that obtaining a reasonable fit to the early exercise boundary with a plain LSLS method is expensive, and we have noted difficulties in estimating the boundary at
times close to the initial time. There is significant noise in these estimates leading to biased option value estimates. When a good estimate of the early exercise boundary is available then a good approximation to the value of the corresponding option can be obtained.

Difficulties are encountered for American options. Direct estimates of their values are capped at the value of the corresponding Bermudan option with reset dates determined by the time steps in the Monte Carlo method. This establishes a minimum level of bias in the Monte Carlo method.

In the next chapter we move on to apply LSLS to valuation problems.

### 29.6 EXERCISES

1. Chebyshev polynomials of the first kind are described in Table 29.1. Implement these, in both plain and scaled forms, as basis functions in the LSLS method. Do they perform any differently to any other families of polynomials?

2. The payoff to an American straddle, if exercised at time $t \in [0, T]$, is $H_t = |S_t - X|$. What form does the early exercise boundary take for this option? Can the spreadsheet MC_American.xls be amended to enable it to compute the early exercise boundary for this instrument?

3. Let $0 < X_1 < X_2$. Consider the American option with payoff $H_t = \max(0, X_1 - |S_t - X_2|)$ if exercised at time $t \in [0, T]$. How can the early exercise boundary be computed using a LSLS method?

4. Consider the option described in exercise 4, page 462. Can this be valued using LSLS?

5. The SVD method is used to find coefficients for the basis functions. This makes fundamental use of a tolerance level $\text{tol}$ that, essentially, determines a cut off that prevents division by zero. In problems that arise from the LSLS method, how sensitive is the SVD method to the value of $\text{tol}$?
This page intentionally left blank
This chapter describes a one-pass sliced-based LSLS implementation. This turns out to be inaccurate, with high standard error. Bias is present, but is hidden within the simulation noise.

We note that execution times are large and speed-ups are required. Here we look at stratified sampling. The use of control variates is investigated in Chapter 31.

The method is option-by-option, in that each option has to compute its own early exercise boundary in an expensive set of operations. It is difficult to see how it might be possible to apply LSLS efficiently to value a book of American or Bermudan options.

The first section mentions some features of the VBA implementation of the plain LSLS method. The second introduces stratification and investigates the extent of the speed-ups available through this route. Finally we investigate using the 64 reset standard Bermudan option as a surrogate for the American put.

The timings in the tables in this chapter are given in seconds.

30.1 IMPLEMENTATION IN VBA

LSLS is naturally slice based. The application MC_American.xls is based on MC_example_v4b.xls in Chapter 7 equipped with a fully polymorphic factory from Chapter 12.

The American put object, OptionAmerican of type IOption, is not complicated; it is just a Bermudan option masquerading as an American option, where exercise is at every time step. No check is made to see whether exercise is optimal at time zero. Unlike other option objects, OptionAmerican::FetchSlice() returns a slice of continuation values, \( \hat{Q}_{0,j} \), not a slice of option values proper.\(^1\)

OptionAmerican is given in Figure 30.1. When a new slice is received (in the ReceiveNextSlice() method) first the payoff object pay_ is asked for the immediate payoff values. These are passed, along with the current stock values, to the backwards evolver object bck_ of type IRollback. This does all the hard work, returning option values for the current slice in Ovals_.

ReceiveNextSlice() is passed a discount factor, dis, to go back from the current time to time zero. This is kept in dis_. When the method has worked back to time index 1, the FetchSlice() method is called to obtain a slice of continuation values, discounted back from Ovals_, for time 0.

OptionAmerican expects strict backwards evolution. This is checked by the CheckT() procedure which throws if the current slice is not for the time immediately preceding the previous time.

Backwards evolution is handled by an evolver object of type IRollback. This is where the complexity of the LSLS method resides. It is a composite object inside the option object; it could – and should – be an object separate from the option object passed, as needed, to the option object. This would follow more closely the design outlined in chapter 10 (albeit slice-based rather than path-based). See exercise 1, page 511.

On the whole we use, for American and Bermudan puts, scaled Laguerre basis functions or B-splines. Other styles of American option might benefit from a different selection of basis function. Each option should be able to specify its preferred basis functions, and any parameters associated with them. One could define a mix-in interface class, IOptionBermudan, with a GetBasisSpec() method, amongst others. MC_American.xls does not have this interface but later implementations do (although not with a method GetBasisSpec()).

\(^1\) If \( S_0 \) is in the exercise region then every path would return the value \( X - S_0 \).
Implementing Models of Financial Derivatives

Private pay_ As IPayoff
Private bck_ As IRollback
Private T_ As Double
Private M_ As Long '# sample paths
Private N_ As Long '# time steps
Private OVals_() As Double 'Current option values
Private SVals_() As Double 'Previous stock values
Private last_i_ As Long 'Index of previous slice
Private dis_ As Double 'Current discount factor
Private dt_ As Double 'Time step
Private done_ As Boolean

Friend Sub IReusable_Reset()
    done_ = False
    last_i_ = 1 + N_
End Sub

Friend Sub IReusable_Initialise()
    ReDim OVals_(1 To M_) As Double
    CastReusable(bck_).Initialise
    Call CastReusable(Me).Reset
End Sub

Friend Property Let ICreatable_SetValues(fact As Factory)
    Set pay_ = fact.Create(New SynIPayoff)
    Set bck_ = fact.Create(New SynIRollback)
    M_ = fact.Value(New ParaNpaths)
    T_ = fact.Value(New ParaT)
    N_ = fact.Value(New ParaNsteps)
    dt_ = T_ / N_
    Call CastReusable(Me).Initialise
End Property

Private Sub CheckT(T As Double, dis As Double)
    If T / dt_ <> last_i_ - 1 Then Call RaiseError(123, "OptionAmerican", "bad T")
    last_i_ = last_i_ - 1
    dis_ = dis
End Sub

Figure 30.1 The OptionAmerican object
Three versions of the rollback object are supplied with MC_American.xls (and further IRollback are introduced later). The first, RollbackLS, exercises whenever \( H_{i,j} > \hat{Q}_{i,j} \). This corresponds to the general algorithm described in section 28.1. The second, RollbackLSmax, exploits the structure of the American put exercise boundary. We know that at each time \( t \) there is a critical value \( S^*_t \) such that the option is exercised if \( S_t \leq S^*_t \). The algorithm finds \( S^*_t \) from equation (28.19), page 460, and then exercises only if \( S_t < S^*_t \). The effect is to get rid of spurious exercise points that appear when \( S_t \) is low. For instance, we noted in Chapter 29 the presence of these points in Figure 29.2. This rollback object does away with them.

The third, RollbackSimple, is included for completeness. It implements equation (28.16), page 459, and is very slow and very inaccurate.

RollbackLSmax is displayed in Figures 30.2 and 30.3. It implements the cashflow formulation of equation (28.18), page 459.

Three arrays are stored whose values are updated at each iteration. TValues is an array of indexes of the times at which, along each sample path, the option is first exercised. These are the values \( i_j \) in equation (28.18). Cf_Vals is an array of cashflows containing the payoffs \( H_{i,j} \) received at time \( t_{i,j} \). Finally discount is an array of discount factors received for each time step.

On each step the NextValues() method is called by OptionAmerican::ReceiveNextSlice(). It has as arguments an array of asset values, \( Svals \), corresponding to \( S_{i,j} \), and an array of payoffs, \( Hvals \), corresponding to \( H_{i,j} \), that would be received were the option to be exercised. A Double, dis, is the discount factor for the current time.

NextValues() updates the three state arrays and returns (ByRef) an array of option values for time \( t_i \), Ovals, corresponding to \( V_{i,j} \). The discount factor is put into discount_. First continuation values are computed for the current time and put into Ovals; then the Private method ComputeRollover() is called to do the rollback proper.

ComputeRollover() first sets up some local arrays containing option values, et cetera, only for in-the-money paths (where \( H_{i,j} > 0 \)). This is to reduce the computation time involved in carrying out the LSLS regression; for the American put only ITM options might be exercised.

Next ComputeContValues() finds continuation values and ComputeSmax() finds the maximum \( S \)-value at which the option is exercised for this slice. Finally, for each sample path, if the decision is to exercise then the state arrays TValues_ and Cf_Vals_ are updated.

ComputeContValues() is the kernel of the rollback. The basis function object, 2 bas_, computes the basis function values \( F = (f_k(S_{i,j}))_{j=1,...,M} \) returning the result in the array B_values. The regression object, reg_, performs the regression, returning the result in the coefficient array coeffs, corresponding to \( \theta_i \). The coeffs array is used to construct the continuation value array, cont_values, corresponding to \( \hat{\theta} \), returned ByRef to ComputeRollover().

### 30.1.1 Valuation results

In Chapter 28 we investigated how the choice and number of basis functions influenced the estimation of the early exercise boundary. Here, more pertinently, we investigate the effect on option valuation. Figure 30.4 shows a jumble of option valuation plots as the number of basis functions increases from 3 to 24 (from 4 to 24 for B-spline basis functions). Results for 8 of our 11 sets of basis functions are presented (scaled natural and plain and scaled Legendre are omitted). The option is the standard 64 reset Bermudan put valued with 50,625 sample paths with plain Monte Carlo. The figure shows reference values for both the Bermudan and the standard American put.

---

2 The asset values have to be transferred to a new array, \( S\_values \), of the right length: IBasis::Funs() requires compatible array arguments. Basis functions are polymorphic, conforming to the IBasis interface. We do not discuss these objects here.
Implementing Models of Financial Derivatives

Private reg_ As IRegressor
Private bas_ As IBasis
Private Nbasis_ As Long
Private MAX_BASIS_ As Long
Private N_ As Long
Private M_ As Long
Private CurrentN_ As Long
Private Cf_Vals_() As Double 'Cashflow values
Private TValues_() As Long 'index of cashflow times
Private discount_() As Double 'Stored discount factors to time 0
Private X_ As Double

Implements IRollback
Implements ICreatable
Implements IReusable

Friend Sub IReusable_Reset()
CurrentN_ = 1 + N_ 'initialized to max value plus one...
Call SetToScalarL(TValues_, N_)
End Sub

Friend Sub IReusable_Initialise()
ReDim Cf_Vals_(1 To M_) As Double 'cash flows
ReDim TValues_(1 To M_) As Long 'Cashflow index values
ReDim discount_(0 To N_) As Double 'discount factors back to 0
Call CastReusable(Me).Reset
End Sub

Friend Property Let ICreatable_SetValues(fact As Factory)
Set reg_ = fact.Create(New SynIRegressor)
Set bas_ = fact.Create(New SynIBasis)
N_ = fact.Value(New ParaNsteps)
M_ = fact.Value(New ParaNpaths)
X_ = fact.Value(New ParaX)
Nbasis_ = fact.Value(New ParaNbasis)
If N > MAX_BASIS_ Then Call RaiseError(1119, name_, 'Too many basis elements')
Call CastReusable(Me). Initialise
End Property

Friend Sub IRollback_NextValues(ByRef Ovals() As Double, _
Svals() As Double, _
Hvals() As Double, _
Ovals() As Double, _
dis As Double)
CurrentN_ = CurrentN_ - 1
discount_(CurrentN_) = dis
If CurrentN_ = N_ Then 'Initial values
Cf_Vals_ = Hvals
Ovals = Hvals
Exit Sub
End If
Dim j As Long 'compute Ovals()
For j = 1 To M_
Ovals(j) = Cf_Vals_(j) * discount_(TValues_(j)) / dis 'continuation values
Next j
Call ComputeRollover(Hvals, Svals, Ovals, CurrentN_)
End Sub

Figure 30.2 The RollbackLSmax object: Interface implementations

The chief messages from the figure are

1. The plain versions of the Hermite, and power (and Legendre too – not shown) basis functions become very heavily biased once $K$, the number of basis functions, becomes moderately large. Option values dive to very low levels.
2. The behaviour of the remaining basis functions cannot, from this illustration, be told apart.
Private Sub ComputeRollover(H_values() As Double, S_values() As Double, _
ByRef Ovals() As Double, ind As Long)
Dim S_ind() As Long: ReDim S_ind(1 To M_) As Long 'Indexes of ITM SValues
Dim O_val() As Double: ReDim O_val(1 To M_) As Double 'Current ITM cont. values
Dim S_val() As Double: ReDim S_val(1 To M_) As Double 'ITM S values
Dim H_val() As Double: ReDim H_val(1 To M_) As Double 'ITM H values
Dim N_itm As Long: N_itm = 0 'Number of paths in the money at time i
Dim j As Long 'Set up ITM vectors
For j = 1 To M_
    If H_values(j) > 0 Then
        N_itm = N_itm + 1
        S_ind(N_itm) = j 'New ITM index found
        O_val(N_itm) = Ovals(j) 'Current in the money continuation values
        S_val(N_itm) = S_values(j) 'ITM S values
        H_val(N_itm) = H_values(j) 'ITM H values
    End If
Next j
If N_itm = 0 Then Call RaiseError(1115, name_, "No paths in the money")
Dim c_vals() As Double: Call ComputeContValues(N_itm, c_vals, S_val, O_val)
Dim Smax As Double: Smax = ComputeSmax(c_vals, H_val, S_val, N_itm)
Dim k As Long
For k = 1 To N_itm
    If S_val(k) < Smax Then
        TVals_(S_ind(k)) = ind
        CF_Vals_(S_ind(k)) = H_val(k)
        Ovals(S_ind(k)) = H_val(k)
    End If
Next k
End Sub
Private Sub ComputeContValues(N_itm As Long, ByRef cont_values() As Double, _
S_val() As Double, O_val() As Double)
ReDim cont_values(1 To N_itm) As Double
ReDim B_values(1 To N_itm, 1 To Nbasis_) As Double
ReDim coeffs(1 To Nbasis_) As Double
ReDim S_values(1 To N_itm) As Double
Dim k As Long
For k = 1 To N_itm
    S_values(k) = S_val(k) 'must be of compatible length
Next k
Call bas_.FunsM(B_values, S_values, Nbasis_) 'Construct B_values
Call reg_.run(B_values, N_itm, Nbasis_, O_val, coeffs) 'Get coeffs
For k = 1 To N_itm
    cont_values(k) = 0
    Dim j As Long
    For j = 1 To Nbasis_
        cont_values(k) = cont_values(k) + coeffs(j) * B_values(k, j)
    Next j
Next k
End Sub
Private Function ComputeSmax(c_vals() As Double, H_val() As Double, _
S_val() As Double, N As Long) As Double
ComputeSmax = 0
Dim k As Long
For k = 1 To N
    If H_val(k) > c_vals(k) Then ComputeSmax = my_max(S_val(k), ComputeSmax)
Next k
End Function

Figure 30.3 The RollbackLSmax object: private procedures
Implementing Models of Financial Derivatives

6.175
6.150
6.125
6.100
6.075
6.050
6.025
6.000
5.975
369 1 2

Number of basis functions

Figure 30.4 Option value as a function of the number of basis functions. \( N = 64, M = 50625 \)

3. Three basis functions are clearly too few. Option values are consistently biased low.
4. Once you have sufficient basis functions (perhaps five or six in this illustration) adding more basis functions makes no difference.
5. Simulation noise is far greater than the price difference between the Bermudan and the American option.
   For plain Monte Carlo with this number of sample paths, the simulated value of the standard Bermudan put has to be thought of as a good approximation to the American put.

   The last point needs immediate clarification. Simulation noise is considerable (ignoring the non-performing basis function families, standard errors are around 0.031–0.032 in every case), so great that it swamps the differences in value. When an improved method is used the difference in prices is obvious.

30.1.2 Stratified and LD sampling

Some speed-ups can be obtained with stratified sampling but it turns out that these are not great. Brownian bridge sample paths can be stratified, as discussed in Part VI. As in Part VI, sample paths are fully stratified at the final time \( T \) and at time \( T/2 \). At other stratification times low-discrepancy sampling is used. In the examples three sets of basis functions are used, chosen to reflect the three categories discussed in Chapter 29. These are B-spline, Laguerre (unscaled) and natural basis functions. In practice, scaled versions are preferred.

Standard errors are summarized in Table 30.1. Standard Bermudan puts with 2, 4, 8 and 16 reset dates are valued when stratified at 0, 2, 4, 8 and 16 times, with \( M = 1000000 \) sample paths using four B-spline
Table 30.1 Standard errors for standard Bermudans with increasing stratification

<table>
<thead>
<tr>
<th># Stratifications</th>
<th># Reset dates</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>0.0079</td>
<td>0.0075</td>
<td>0.0073</td>
<td>0.0073</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>0.0002</td>
<td>0.0036</td>
<td>0.0041</td>
<td>0.0040</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>–</td>
<td>0.0002</td>
<td>0.0038</td>
<td>0.0039</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>–</td>
<td>–</td>
<td>0.0002</td>
<td>0.0035</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>0.0002</td>
</tr>
</tbody>
</table>

basis functions. Standard errors are computed as the standard deviation of values from 100 individual Monte Carlo runs. As the number of stratification times increases so the standard error decreases. One observes that variance reductions achieved by stratification are not great. In the examples of Table 30.1, only a factor of 4 is achieved: stratification only approximately halves the standard error of the non-stratified values. Some of the results in this table are elaborated upon in section 30.1.3.

When a sample path is fully stratified, at every time step, the same effect is observed here as in Chapter 19. Results become apparently accurate – stratifying at every time yields a standard error of only 0.0002 – but in fact are heavily biased. The standard error is very low, but the option values one obtains are many multiplies of the standard error away from their true values. LD sampling should not be used in this way.

30.1.3 Benchmark valuation

To benchmark the plain and stratified Monte Carlo method we compute the values of standard Bermudan puts with a small number of exercise times.

Two resets: compound option benchmark

A Bermudan option exercisable only at the final time and at one other intermediate time is a compound option. When the underlying asset follows a GBM an explicit valuation formula is available (see Appendix B). This is a very convenient benchmark for the LSLS method (or, indeed, any American Monte Carlo method). There is a single time (the intermediate time) when the continuation values have to be estimated.

Write \( p_2(t \mid X, T_1, T_2) \) for the value at time \( t \) of the Bermudan put that can be exercised only at times \( T_1 \) and \( T_2 \), for \( t < T_1 < T_2 \). For the standard Bermudan put with two reset dates we set \( T_1 = \frac{1}{2} T \) and \( T_2 = T \). Figure 30.5 shows values for this option, \( p_{20}(S_0 \mid X, \frac{1}{2} T, T) \), computed by LSLS with three different sets of basis functions\(^3\) and the number of basis functions varying between 3 and 12. The true value of the option, computed by the formula, is 5.838710 to six decimal places. Panel (a) shows the case with no stratification, panel (b) the case with both times stratified. \( M = 1000000 \) sample paths were used in every run. When there is no stratification, standard errors are around 0.007–0.008; with full stratification, standard errors are around 0.0002. Since there is only a slight increase in computation time the efficiency gain is of the order 1000–1500. This gain is robust; it is relatively independent of the choice and number of basis function, and it is real as no LD stratification is involved. No bias is evident.

\(^3\)Throughout this chapter we switch between different sets of basis functions essentially at whim. Usually there is no apparent difference in performance. When there is, this is noted. Here we illustrate with Laguerre (not scaled), natural, and B-spline basis functions.
The detail is presented in Table 30.2. This shows option values, standard errors (in round brackets), and computation times in seconds (in square brackets), with and without stratification, and efficiency gains (in bold), for the three families of basis functions and for either 4 or 24 basis functions. Standard errors correspond to those given in Table 30.1.

Efficiency gains are very high but this is not unexpected. Every time step is stratified. There is no evidence of bias in these results. All values are within a reasonable multiple of the standard error of the true value.

**Eight resets**

The true value of the standard Bermudan put with eight reset dates, computed by a lattice method, is 6.02007 to 5 decimal places. Figure 30.6 shows the effect on the computed option value as the number
of stratification times increases, with various numbers of basis functions. As the number of stratification times increases so the standard error declines, but note that the effect is not even. Introducing two levels of (full) stratification reduces the standard error by roughly half (see Table 30.1). Moving to four stratification times (two full and two low discrepancy) does not significantly reduce the standard error. However, when all eight reset times are stratified the standard error falls to a low level.

Efficiency gains are given in Table 30.3 for option values computed with four B-splines basis functions (results are similar for other types and numbers) and # stratification times. Gains are modest until eight stratification times are used. Unfortunately, as we saw in Chapter 19, using low discrepancy to allow stratification at every reset date introduces a large degree on bias. It is clear from panel (d) in Figure 30.6 that option values are systematically biased high at a level of around 6.0205, approximately 2 standard errors away from the true value. This is in accord with the results of Chapter 19. It is recommended never

![Figure 30.6](image)

**Figure 30.6** Bermudan option valuation. $M = 1 000,000$, $N = 8$

<table>
<thead>
<tr>
<th>#</th>
<th>0</th>
<th>2</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Results:</td>
<td>6.0105</td>
<td>6.0178</td>
<td>6.0203</td>
<td>6.0192</td>
</tr>
<tr>
<td>Gain:</td>
<td>—</td>
<td>3.1</td>
<td>3.5</td>
<td>1250</td>
</tr>
</tbody>
</table>

Table 30.3  Efficiency gains: $R = 8$, four B-spline basis functions
to stratify completely using low discrepancy sampling. It seems safe to LD stratify up to only half the
number of time steps.
Note also that here it is only with 6 basis functions that option values settle down to their long-run level
(albeit biased). This is evidence that at least 6 basis functions should be used with this option.

30.2 VALUING THE AMERICAN PUT?

For the moment we take the standard Bermudan option with 64 reset dates as an approximation to the
American put option value. The value of the Bermudan option, to 5 decimal places, is 6.08118. We see
later that 64 reset dates is grossly insufficient for the approximation to be any good.

Convergence in the number of sample paths

One quickly verifies that the regressions employed by the method do not disturb the rate of convergence
of the standard error of the option value as a function of the number of sample paths. Table 30.4 gives
results for various values of \( M \) and number of basis functions, \( K \). (For B-spline basis functions; other
basis functions give similar results.) It can be confirmed that:

1. pricing error is \( O(-\frac{1}{2}) \) in \( M \);
2. computation time is \( O(-1) \) in \( M \) and \( \sim O(-7/4) \) in \( K \);
3. all prices are within one or two standard deviations of the true price, except when \( M = 2500 \); for
   sufficiently large \( M \) it is not possible to detect bias in the results.

Here and elsewhere we compute only up to a maximum of \( 10^6 \) sample paths on a single run. The method
is slice based; on the hardware platform used for these examples out-of-memory was encountered with
slightly greater than \( 10^6 \) sample paths.

<table>
<thead>
<tr>
<th>( M )</th>
<th>( K )</th>
</tr>
</thead>
<tbody>
<tr>
<td>2500</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>24</td>
</tr>
<tr>
<td>2500</td>
<td>6.4</td>
</tr>
<tr>
<td></td>
<td>6.2</td>
</tr>
<tr>
<td></td>
<td>6.4</td>
</tr>
<tr>
<td></td>
<td>6.6</td>
</tr>
<tr>
<td>( \times 10^3 )</td>
<td>( \times 10^3 )</td>
</tr>
<tr>
<td>50625</td>
<td>6.09</td>
</tr>
<tr>
<td></td>
<td>6.11</td>
</tr>
<tr>
<td></td>
<td>6.13</td>
</tr>
<tr>
<td></td>
<td>6.11</td>
</tr>
<tr>
<td>( \times 10^3 )</td>
<td>( \times 10^3 )</td>
</tr>
<tr>
<td>1000000</td>
<td>6.07</td>
</tr>
<tr>
<td></td>
<td>6.08</td>
</tr>
<tr>
<td></td>
<td>6.06</td>
</tr>
<tr>
<td></td>
<td>6.07</td>
</tr>
<tr>
<td>( \times 10^3 )</td>
<td>( \times 10^3 )</td>
</tr>
<tr>
<td>1000000</td>
<td>6.072</td>
</tr>
<tr>
<td></td>
<td>6.091</td>
</tr>
<tr>
<td></td>
<td>6.077</td>
</tr>
<tr>
<td></td>
<td>6.076</td>
</tr>
</tbody>
</table>
**Convergence with the number of basis functions**

It has been shown (Stentoft (2004)) that the LSLS method converges to a lower bound on the option value: it is a low-bias estimator. As the number of basis functions increases (towards a set of spanning basis functions) the bias decreases.

Figure 30.7 plots option values as the number of basis functions increases, and as the number of stratifications times increases from 0 to 16. Scaled Laguerre and B-spline basis functions are compared.

It is clear that using only 3 or 4 basis functions gives biased results. When stratified sampling is used there is bias if fewer than 5 or 6 basis functions are used. Later, when we examine errors in the early exercise boundary, we conclude that 12 or more basis functions are needed to get close to the true boundary.

**The effect of Stratification**

Efficiency gains are shown in Table 30.5 when 6 basis functions are used with scaled Laguerre or B-splines basis functions, for $M = 50625$. Standard errors were computed as the standard deviation of 100 replications of the Monte Carlo valuation.

![Figure 30.7 Bermudan option valuation. $M = 1000000, N = 64$](image)
Table 30.5  Efficiency gains: six basis functions. \( M = 50,625 \), \( R = 64 \)

<table>
<thead>
<tr>
<th>Basis type</th>
<th>Number of stratification times</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Plain</td>
</tr>
<tr>
<td>Scaled Laguerre:</td>
<td>6.06</td>
</tr>
<tr>
<td></td>
<td>(0.031)</td>
</tr>
<tr>
<td></td>
<td>[33.3]</td>
</tr>
<tr>
<td></td>
<td>−</td>
</tr>
<tr>
<td>B-spline:</td>
<td>6.09</td>
</tr>
<tr>
<td></td>
<td>(0.029)</td>
</tr>
<tr>
<td></td>
<td>[40.9]</td>
</tr>
<tr>
<td></td>
<td>−</td>
</tr>
</tbody>
</table>

Computation times are approximately constant at around 41 seconds for scaled Laguerre and 33 seconds for B-splines so efficiency gains are purely a result of a reduction in standard error (by a factor of about \( 1.7 \sim \sqrt{3} \)). The results in the table are typical; incorporating 16 levels of stratification with this option gives gains of around a factor of 3 or 4.

One may try to project from this table to establish how long it would take, in principle, to achieve results of any given degree of accuracy. For instance, to obtain a standard error of 0.005, so that one has a reasonable chance of being within 0.01 of the correct value (of 6.08118), would require roughly 10 times the number of sample paths, taking about 400 seconds in the B-spline case. To be within 0.001 of the correct value would take 100 times as long again, at over 10 hours, an unreasonable amount of time.

30.3 SUMMARY

Implementing a plain LSLS method is not too hard. Unfortunately, even with stratification, the results are quite poor. It is infeasibly expensive with plain LSLS to obtain results accurate to 3 decimal places, even for a 64 reset Bermudan put, let alone an American put.

It makes no difference, in the plain case, which set of basis function is used, or how many (as long as there are more than 5 or 6 and pathological sets are avoided).

To make progress something much better is required. In the next chapter we find that control variates come to our aid.

30.4 EXERCISES

1. Consider a flexible barrier option specified by a time-varying barrier level \( B_t, t \in [0, T] \), and time-varying payoff \( H_t \) if the barrier is hit at time \( t \). Construct an application to find values for options specified by:

   (a) \( B_t = a + b \exp(ct), a, b, c \geq 0; H_t = (X - B_t)^+, X > 0. \)

   (b) \( B_t = a + b \cos(ct), a, b, c \geq 0; H_t = |X - B_t|. \)

   (c) \( B_t = \begin{cases} X_1, & t \in [i/12, (i + 1)/12], i \in \mathbb{Z} \text{ even}, \\ X_2, & t \in [i/12, (i + 1)/12], i \in \mathbb{Z} \text{ odd} \end{cases}; H_t = [t/12]. \)

   In each case you may suppose that \( S_0 > B_0 \) and that the barrier is hit at the first time \( \tau \) such that \( S_\tau \leq B_\tau \). For concreteness suppose that \( S_\tau \) follows a GBM with \( r = 0.05 \) and \( \sigma = 0.2 \).
2. Consider an American put written on (a) a vanilla knock-in barrier option and (b) a vanilla knock-out barrier option. Writing \( c_t \) for the value of the underlying option at time \( t \), the payoff to the American put, if exercised at time \( t \), is \((X - c_t)^+\) where \( X \) is the strike. Construct an application to value these American puts. (Assume that the value \( c_t \) is available as an explicit formula.)

3. Richardson extrapolation to value an American put. Figure 28.3 (page 455) used Richardson extrapolation to project an American EEB from EEBs for Bermudan options. Can the same technique be used to obtain values for American options themselves?
   
   (a) Construct an application that can value simultaneously standard Bermudans with different numbers of reset dates.

   (b) Devise a means to use their values to infer a value for the American put.

   (c) Does it work?
Using CVs turns out to be highly effective both at reducing noise and, in a second mode of application, at reducing the bias that is then apparent. The best results are obtained in a two-pass method. A first pass computes the EEB, a second pass prices a mimicking barrier option using the barrier estimated in the first pass.

In turns out that, conceptually, the total simulation noise in an LSLS method has two components. The first is error in estimating the EEB; conditional upon a realization of the EEB, the second is the error in valuing the corresponding mimicking barrier option.

In a one-pass method, introducing an appropriate pricing CV is very effective at reducing the latter component but does nothing to the former. On any individual run the reported standard error is small, but since it does not include error in estimating the EEB it severely under-represents the true standard error. As usual this can be found only by replicating the valuation procedure a sufficient number of times and finding the standard deviation of the resulting set of option values.

To reduce the noise in the estimation of the EEB a second application of control variates is used. This reduces noise to such an extent that gains obtained in the first CV application become visible.

Here we apply the technique only to valuing standard Bermudan puts. Chirayukool and Webber (2010) take it one step further, showing how Richardson extrapolation enables American puts to be valued to much the same accuracy as Bermudan puts in only slightly worse times.

The first section discusses a set of CVs that can be used with the American put and presents pricing results. Section 31.2 describes the Rasmussen CV correction for estimating the EEB. Results from the two-pass method are given in section 31.3.

The timings in the tables in this chapter are given in seconds.

### 31.1 CONTROL VARIATES AND THE AMERICAN PUT

Since stratification produces only indifferent efficiency gains we investigate the use of control variates with the American put. We find that large gains are possible and can be increased even further using a two-pass method.

Write $p_t(S_t \mid X, T)$ for the value at time $t$ of a European put with strike $X$ and final maturity time $T > t$ when the asset value is $S_t$. Under GBM explicit formulae exist for both $p_t$ and for the standard Bermudan put with two exercise dates, $p_{2t}$. Either of these can be used as a control variate, either directly or indirectly when GBM is used as an auxiliary model. The use of $p_{2t}$ as a CV has been investigated by Chirayukool and Webber (2010).

**Stopping time CVs**

Sometimes a CV can be enhanced, in that its correlation with the target option value can be increased, if its value is sampled not at the final time but at a stopping time. For instance, the ordinary terminal put CV associated with the European put is

$$CV^p_T = e^{-rT}(X - S_T)^+ - p_0(S_0). \quad (31.1)$$
Now suppose that $\tau \leq T$ is a stopping time and define

$$CV^p_\tau = \begin{cases} e^{-r\tau} p_\tau(S_\tau) - p_0(S_0), & \tau < T, \\ e^{-r\tau}(X - S_\tau)^+ - p_0(S_0), & \tau = T. \end{cases} \quad (31.2)$$

The optional sampling theorem says that for a martingale $M_t$ and two stopping times $\sigma \leq \tau$ we have $M_\sigma = \mathbb{E}_\sigma[M_\tau]$. Since $e^{-r\tau} p_\tau(S_\tau)$ is a martingale it follows that $CV^p_\tau$ is a CV. We call $CV^p_\tau$ the put-tau CV.

For the American put a natural stopping time is the first hitting time to the early exercise boundary. Rasmussen (2005) shows both theoretically and numerically that $CV^p_\tau$ is a better CV than $CV^p_T$ and this is also confirmed by our results.

In addition to the two put CVs we also consider a pair of stock CVs and a set of CVs based on the standard Bermudan put with two reset dates.

**The stock CVs**

Let $CV^S_T$ and $CV^S_\tau$ be defined as

$$CV^S_T = e^{-rT} S_T - S_0, \quad (31.3)$$

$$CV^S_\tau = e^{-r\tau} S_\tau - S_0. \quad (31.4)$$

$CV^S_T$ is the usual stock CV; $CV^S_\tau$ is the stopping time enhanced version for the stopping time $\tau$.

**The Bermudan put CVs**

The Bermudan put $p2_t$ with exercise dates $T_1 = T/2$ and $T$ can be used as a CV. When $S$ follows a GBM an explicit solution for $p2_t$ exists (see Appendix B). $p2$ is exercised at time $T_1$ if $S_{T_1} \leq S^*_T$ for some critical stock value $S^*_T \leq X$. At that point the option ceases to exist.

Technically the Bermudan put CV is a trading strategy, $\hat{p}^2$, based on $p^2$. $\hat{p}^2$ differs from $p^2$ only in that if it is exercised at time $T_1$ then the cash received is invested at the riskless rate so that for $T_1 < t < T$

$$\hat{p}^2_t = \begin{cases} p_t, & S_{T_1} > S^*_T, \\ e^{r(t-T_1)}(X - S_{T_1/2})^+, & S_{T_1} \leq S^*_T. \end{cases} \quad (31.5)$$

This ensures that the CV is well defined on all sample paths (and consequently we now ignore the distinction we have just drawn between $p^2$ and $\hat{p}^2$).

$p^2_t$ can be used with a stopping time $\tau < T$. The CV is

$$CV^B_\tau = \begin{cases} e^{-r\tau} p^2_\tau(S_\tau) - p^2_0(S_0), & \tau < T_1, \\ e^{-rT_1}(X - S_{T_1}) - p^2_0(S_0), & T_1 \leq \tau \text{ and } S_{T_1} \leq S^*_T, \\ e^{-r\tau} p_\tau(S_\tau) - p_0(S_0), & T_1 \leq \tau < T \text{ and } S_{T_1} > S^*_T, \\ e^{-r\tau}(X - S_T)^+ - p_0(S_0), & \tau = T \text{ and } S_{T_1} > S^*_T. \end{cases} \quad (31.6)$$

\(^1\) More properly optional times.
We call $CV^B_\tau$ the Bermudan-\(\tau\) CV. \(p_{2\tau}\) can be used in four other ways, each giving a variant of $CV^B_\tau$.

1. The Bermudan-terminal CV, $CV^B_T$,

\begin{equation}
CV^B_T = \begin{cases} 
  e^{-rT_1}(X - S_{T_1}) - p_{20}(S_0), & S_{T_1} \leq S^*_{T_1}, \\
  e^{-rT}(X - S_T)^+ - p_{20}(S_0), & S_{T_1} > S^*_{T_1}.
\end{cases}
\end{equation}

This is a simple extension of the terminal put CV and, not surprisingly, does not give good results.

2. The Bermudan-\(T\) CV, $CV^B_{t;\tau}$,

\begin{equation}
CV^B_{t;\tau} = \begin{cases} 
  e^{-r\tau}p_{2\tau}(S_\tau) - p_{20}(S_0), & \tau < T_1, \\
  e^{-rT_1}(X - S_{T_1}) - p_{20}(S_0), & T_1 \leq \tau \text{ and } S_{T_1} \leq S^*_{T_1}, \\
  e^{-rT_1}p_{T_1}(S_{T_1}) - p_{20}(S_0), & T_1 \leq \tau \text{ and } S_{T_1} > S^*_{T_1}.
\end{cases}
\end{equation}

Here the stopping time \(\tau\) is capped by \(T_1\). At time \(T_1\) the CV is given its \(T_1\) value.

3. The plain Bermudan-\(T\) CV, $CV^{B,p}_{t;\tau}$,

\begin{equation}
CV^{B,p}_{t;\tau} = \begin{cases} 
  e^{-r\tau}p_{2\tau}(S_\tau) - p_{20}(S_0), & \tau < T_1, \\
  e^{-rT_1}(X - S_{T_1}) - p_{20}(S_0), & T_1 \leq \tau \text{ and } S_{T_1} \leq S^*_{T_1}, \\
  e^{-rT}(X - S_T)^+ - p_{20}(S_0), & T_1 \leq \tau \text{ and } S_{T_1} > S^*_{T_1}.
\end{cases}
\end{equation}

$CV^{B,p}_{t;\tau}$ is a simpler version of $CV^B_{t;\tau}$ and performs less well.

4. The Bermudan-early CV, $CV^p_\tau$.

Let \(P_t\) be the set of three instruments

\begin{equation}
P_t = \{p_{2\tau}, -p^*_t(S_t \mid S^*_T, T_1), -(X - S^*_T)\, d^*_t(S_t \mid S^*_T, T_1)\}
\end{equation}

where \(d^*_t(S_t \mid S^*_T, T_1)\) is the digital with payoff \(H_{T_1}\) at \(T_1\) given by

\begin{equation}
H_{T_1} = \begin{cases} 
  1, & S_{T_1} \leq S^*_{T_1}, \\
  0, & S_{T_1} > S^*_{T_1},
\end{cases}
\end{equation}

and \(p^*_t(S_t \mid S^*_T, T_1)\) is the European put expiring at time \(T_1\) with strike \(S^*_{T_1}\). Then define

\begin{equation}
CV^p_\tau = \begin{cases} 
  e^{-r\tau}P_t(S_\tau) - P_0(S_0), & \tau < T_1, \\
  -P_0(S_0), & T_1 \leq \tau \text{ and } S_{T_1} \leq S^*_{T_1}, \\
  e^{-rT_1}p_{T_1}(S_{T_1}) - P_0(S_0), & T_1 \leq \tau < T \text{ and } S_{T_1} > S^*_{T_1}, \\
  e^{-rT}(X - S_T)^+ - P_0(S_0), & \tau = T \text{ and } S_{T_1} > S^*_{T_1}.
\end{cases}
\end{equation}

Some explanation is required for $CV^p_\tau$. The portfolio \(P_t\) is essentially a knock-out put, expiring at time \(T\) with payoff \((X - S_T)^+\), that knocks out at time \(T_1\) if \(S_{T_1} \leq S^*_{T_1}\). The put and the digital cancel out the payoff to the Bermudan should it be optimal to exercise it at time \(T_1\). The name, Bermudan-early, refers to the payoff \(X - S_{T_1}\) at time \(T_1\) made if the Bermudan is exercised at that time. This has been removed and, effectively, taken instead at the earlier time \(\tau < T_1\).
Implementing Models of Financial Derivatives

We use, like Rasmussen, the first exercise time of the American put as the stopping time $\tau$. This is easily operationalized in an application. Recall from equation (28.17), page 459, the definition of $i_j$,

$$i_j = \min \{ k \geq i \mid H_{k,j} \geq \tilde{Q}_{k,j}\}$$

(31.13)

where $\tilde{Q}_{k,j} = f(S_{k,j} \mid \theta_k)$ is an optimal fit to the continuation values $\tilde{Q}_{k,j}$. $i_j$ is the index of the first time on or after time $t_i$ along the $j$th sample path that the algorithm determines it is optimal to exercise the option (conditional on not being exercised before time $t_i$). $0_j$ is the first exercise time along the $j$th path. We use $0_j$ as the discretely computed value of $\tau$. For instance, for the put-$\tau$ CV set $\tau_j = t_{0_j}$ and $p_j = p_{\tau_j}(S_{0_j,j})$. Then the value $CV^p_j$ of the put-$\tau$ CV on the $j$th sample path is

$$CV^p_j = e^{-r\tau_j} p_j - p_0(S_0).$$

(31.14)

### Implementing CVs

CV are implemented in the spreadsheet MC_American_CV_one_pass.xls. The application object, AppMC_CV, is shown in Figure 31.1. This is very similar to the CV application objects we have seen in previous chapters. There is an OptionWrapper object, wrp_, essentially a façade that wraps an option object and an associated accumulator object, and a CVmanager object. Links are set in the usual way in the SetValues() method (not shown).

An additional array is returned by the option object on each iteration, an array of Booleans, ex. This has an entry for each sample path, set to True if the algorithm determines that the option is exercised at that point on the path. ex is passed to the CVmanager object and on to each CV object where it is used to construct $t_{ij}$ at each step.

The CV objects are based on MC_speed-ups_all.xls from Chapter 23. The structure remains much the same as there. The CVmanager object is essentially unchanged. The individual CV objects need to be slice-based rather than path-based but they remain straightforward modifications of their corresponding option objects.

The mechanism by which options notify the CVmanager object of the CVs they require is unchanged from Chapter 21 and suffers from the advantages and disadvantages noted there.

A typical CV object, CVEuroPutBStau, is displayed in Figure 31.2. It implements a CV interface, ICV (not shown). In this application the CV objects are not factory-created and so are not endowed with the ICreatable interface (although in real life they are likely to be).

CVEuroPutBStau implements the put-$\tau$ CV. The key methods are ReceiveSlice() and GetCVs(). ReceiveSlice() is passed not only a slice of asset values, and index and discount data, but is also passed the array of Booleans, ex, that conveys whether or not the option has been exercised at that point.

CVEuroPutBStau maintains four state arrays that are updated when ReceiveSlice() is called. ex_times_ holds the earliest time along each sample path on which the option is currently exercised. This is the value $t_{ij}$. ex_vals_ contains the asset values, $S_{ij,j}$, at those times and ex_dis_ stores the discount factors for those times for each path. The fourth array, vals_T_, holds the exercise values at time $T$ for each path.

All that ReceiveSlice() does is to construct vals_T_ and to update the other three state arrays as new slices arrive. When slices for every time have been received, CVEuroPutBStau knows the exercise time along every sample path and the asset value at the time of exercise.

When GetCVs() is called it computes $p_j$ (in equation (31.14)) from ex_times_, ex_vals_, and ex_dis_ and returns a slice of CV values, ByRef, in CVvals.

Charts are produced in a special chart manager object, Chartmanager (not shown). An option scatter chart is displayed at every time step. Chartmanager pauses execution after it is displayed and prompts
the client whether they want to continue or to end. If the client wants to end an **End** statement is executed. This terminates program execution. Unfortunately this device is evil; **End** does not call any destructors so no cleaning up is done. I’ve used **End** as a shortcut; instead I should have raised an exception or used an orderly **Exit**.²

²As in writing, so in life. Shortcuts are taken to achieve deadlines and the code suffers. Please regard this defect as a pedagogical example. (See exercise 6.)
Implementing ICV
Implements IReusable

Private X_ As Double
Private T_ As Double
Private r_ As Double
Private sig_ As Double
Private M_ As Long

Private ex_times_() As Double 'current exercise times
Private ex_vals_() As Double 'stock value at exercise time
Private ex_dis_() As Double 'discount factor at exercise time
Private vals_T_() As Double 'CV value if exercised at final time
Private cv_mean_ As Double 'mean value of the CV
Private name_ As String

Private Sub Class_Initialize()
  name_ = "CVEuroPutBStau"
End Sub

Friend Sub ICV_SetValues(fact As Factory)
  r_ = fact.Value(New ParaR)
  sig_ = fact.Value(New ParaSigma)
  T_ = fact.Value(New ParaT)
  X_ = fact.Value(New ParaX)
  M_ = fact.Value(New ParaNpaths)
  Dim S0 As Double: S0 = fact.Value(New ParaS0)
  cv_mean_ = BSput(X_, T_, S0, r_, sig_, 0#)
  CastReusable(Me).Initialise
End Sub

Friend Property Get Icv_CVname() As String: Icv_CVname = name_: End Property
Friend Property Get Icv_CVmean() As Double: Icv_CVmean = cv_mean_: End Property

Friend Sub Icv_ReceiveSlice(dis As Double, slice() As Double, i As Long, _
  t As Double, ex() As Boolean)
  Dim j As Long
  If t = T_ Then
    For j = 1 To M_
      vals_T_(j) = dis * my_max(0, X_ - slice(j)) - cv_mean_
    Next j
  Else
    For j = 1 To M_
      If ex(j) And t < ex_times_(j) Then
        ex_times_(j) = t
        ex_vals_(j) = slice(j)
        ex_dis_(j) = dis
      End If
    Next j
  End If
End Sub

Friend Sub Icv_GetCVs(ByRef CVvals() As Double)
  Dim j As Long
  For j = 1 To M_
    If ex_times_(j) < T_ Then
      Dim t As Double: t = T_ - ex_times_(j)
      Dim bs As Double: bs = BSput(X_, t, ex_vals_(j), r_, sig_, 0#)
      CVvals(j) = ex_dis_(j) * bs - cv_mean_
    Else
      CVvals(j) = vals_T_(j)
    End If
  Next j
End Sub

Figure 31.2 The CVEuroPutBStau object
Efficiency gains

To assess the performance of the CVs we apply them to the mimicking barrier option with the lattice-derived 64 reset standard Bermudan barrier. The results are thus unaffected by uncertainties introduced by the construction of the EEB and represent a pure assessment of the CV performance. \( M = 10^6 \) sample paths were used, with no stratification. The plain case took 109 seconds to give a result of 6.088 with a standard error of 0.0071.

The CVs we have defined vary significantly in their efficacy. Table 31.1 gives results. Panel (a) gives correlations for single CVs alongside valuation results and efficiency gains. The plain Bermudan-\( T_1 \) CV, \( CV_{T_1}^{B,p} \), has a slightly higher correlation than the clever looking Bermudan-\( T_1 \) CV, \( CV_{T_1}^{B} \). Only the Bermudan-\( \tau \) CV, \( CV_{\tau}^{B} \), comes anywhere close to the put-tau CV, \( CV_{\tau}^{P} \), with its exceptional high correlation.

The Bermudan CVs take about 30\% longer to run than the plain Monte Carlo, but the other CVs do not add too much to the computation time. Even the put-tau CV takes only about 10\% longer than the plain case.

Efficiency gains reflect the correlations. The put-tau CV achieves a remarkable gain of over 500. The next best gain, a lowly 21, is obtained with the Bermudan-tau CV. The Bermudan-based CVs are expensive to compute. The Bermudan-early CV has an efficiency gain of less than 1 despite a correlation of 0.46.

Used in combination greater speed-ups are possible. Panel (b) of Table 31.1 shows efficiency gains achieved when the put-\( \tau \) CV is used with each of the other CVs. We see that the Bermudan-early CV works best in combination with the put-\( \tau \) CV; despite its cost, efficiency gains are significantly enhanced. The standard error is reduced to two-thirds while the computation time increases by only a quarter. By contrast most of the other CVs reduce the overall gain. (A small gain seems possible with the terminal stock CV but this is very marginal.)

Using the lattice EEB with a mimicking barrier option is absolutely necessary to make meaningful comparisons. Without doing this, too much noise is introduced in the EEB estimation, completely swamping the beneficial effect of the CVs. Using the mimicking barrier option, to get a standard error of 0.005 (with

<table>
<thead>
<tr>
<th>CV:</th>
<th>( CV_{T}^{S} )</th>
<th>( CV_{T}^{P} )</th>
<th>( CV_{T}^{P} )</th>
<th>( CV_{T}^{P} )</th>
<th>( CV_{T}^{P} )</th>
<th>( CV_{T}^{P} )</th>
<th>( CV_{T}^{P} )</th>
<th>( CV_{T}^{P} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correlation:</td>
<td>−0.72</td>
<td>−0.77</td>
<td>0.74</td>
<td>0.9991</td>
<td>0.84</td>
<td>0.89</td>
<td>0.82</td>
<td>0.46</td>
</tr>
<tr>
<td>Gain:</td>
<td>2.0</td>
<td>2.3</td>
<td>2.2</td>
<td>509</td>
<td>3.2</td>
<td>3.7</td>
<td>2.3</td>
<td>0.96</td>
</tr>
</tbody>
</table>

Panel (a): Single CVs

<table>
<thead>
<tr>
<th>CV:</th>
<th>( CV_{T}^{P} + CV_{T}^{S} )</th>
<th>( CV_{T}^{P} + CV_{T}^{S} )</th>
<th>( CV_{T}^{P} + CV_{T}^{P} )</th>
<th>( CV_{T}^{P} + CV_{T}^{P} )</th>
<th>( CV_{T}^{P} + CV_{T}^{P} )</th>
<th>( CV_{T}^{P} + CV_{T}^{P} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gain:</td>
<td>518</td>
<td>506</td>
<td>495</td>
<td>406</td>
<td>706</td>
<td>877</td>
</tr>
</tbody>
</table>

Panel (b): CVs in combination with the put-tau CV

\(^3\) The spreadsheet MC_American_CV_one_pass_set_barrier.xls is set up to do this.
16 stratification times and a put-\(\tau\) CV) only 2500 sample paths are needed. This gives a standard error of 0.0056 in \(\sim 0.45\) seconds. However using the full plain LSLS method, with the put-\(\tau\) CV and the same number of sample paths (and 16 stratifications), gives a value with a standard error of 0.03 (computed as the standard deviation of 100 replications) in \(\sim 6.5\) seconds – even though the reported standard error is only 0.007.

Of course the reported standard error is the standard error for the valuation of a barrier option whose barrier is the estimated EEB. From run to run the estimated barrier changes wildly, so that one gets a sequence of very accurate values to very inaccurate approximations to the true option. For a further illustration see Figure 31.7.

We recommend that Bermudan options should be valued routinely with the put-\(\tau\) and Bermudan-early CVs, but only if the EEB can be estimated accurately. This we tackle next.

### 31.2 CONTROL VARIATES AND THE EEB

The gains achieved from using a CV are completely pointless unless the EEB can be estimated accurately. It turns out that estimates of the EEB can be improved very considerably if a control variate is used to correct the raw continuation values before the LSLS regression is made. This important idea is due to Rasmussen (2005).

Let \(\hat{Q}_t = \{\hat{Q}_{i,j}\}_{j=1,...,M}\), \(\hat{Q}_{i,j} = e^{-r\Delta t}V_{i+1,j}\), be the set of continuation values at time \(t_i\). Write \(CV_{i,j} = CV_{t_i}(S_{i,j})\) for the value of a control variate at time \(t_i\) with asset value \(S_{i,j}\) and set

\[
\bar{Q}_{i,j} = \hat{Q}_{i,j} - \beta_i CV_{i,j}
\]

(31.15)

where, for each \(i\), \(\beta_i = \text{cov}(\hat{Q}_{i,j}, CV_{i,j})/\text{var}(CV_{i,j})\) is the regression coefficient. \(\bar{Q}_{i,j}\) will have the same expected value as \(\hat{Q}_{i,j}\) but, with a good choice of CV, will have much reduced variance.

Now one finds coefficients \(\theta_i\) such that \(\tilde{Q}_{i,j} = f(S_{i,j} | \theta_i)\) is a fit to \(\bar{Q}_{i,j}\) and these \(\tilde{Q}_{i,j}\) are used to determine the exercise decision. The option value is found accordingly,

\[
V_{i,j} = \begin{cases} 
H_{i,j}, & H_{i,j} \geq \tilde{Q}_{i,j}, \\
\tilde{Q}_{i,j}, & H_{i,j} < \tilde{Q}_{i,j}.
\end{cases}
\]

(31.16)

The method relies upon the availability of a good CV. Rasmussen uses a European put CV, and establishes its effectiveness. Set

\[
CV_t(S) = \begin{cases} 
e^{-r(\tau-t)}p_t(S) - p_t(S), & \tau < T, \\
e^{-r\tau}(X - S)^+ - p_t(S), & \tau = T.
\end{cases}
\]

(31.17)

where \(\tau\) is the first hitting time to the American put EEB conditional upon the asset having value \(S\) at time \(t\) and the put not having been exercised prior to time \(t\). To operationalize, set \(p_{i,j} = p_t(S_{i,j})\), with \(p_{N,j} = (X - S_{N,j})^+\), and define

\[
CV_{i,j} = e^{-r(t_{ij} - t_i)}p_{ij,j} - p_{i,j}
\]

(31.18)

where \(ij\) is given by equation (31.13). We proceed to implement \(CV_{i,j}\).
Control Variates and the LSLS Method

Implementing the CV rollback object

CV rollback is implemented in MC_American.CV_one_pass.xls. The CV rollback object, RollbackCVtau, is very similar to the plain rollback object, RollbackLSmax (Figures 30.2 and 30.3). Figures 31.3 and 31.4 display only the main differences with RollbackLSmax.

There is a new state array, CV_ex_. This contains the current values of $e^{-r_{ij}} p_{ij}$ at time $t_i$. It is initialized, in NextValues(), with values $e^{-rT}(X - S_{N,j})^+$ and then updated (in ComputeRollover()) whenever an earlier exercise decision is made along a sample path.

Adjusted continuation values are computed in NextValues() by AdjustOvals() (Figure 31.4). This computes $CV_{i,j}$ from CV_ex_, performs a regression to find $\beta_i$ (line 31.4a), and then constructs $Q_{i,j}$ (line 31.4b). Adjusted continuation values are returned in OvalsAdj. These are passed on to ComputeRollover() along with the unadjusted continuation values.

RollbackCVtau::ComputeRollover() differs only slightly from RollbackLSmax::ComputeRollover(). Continuation values are computed by ComputeContValues(), just as before; the only difference is that adjusted continuation values, in A_val, are passed to this procedure instead of unadjusted values. The Private Function ComputeCV() (Figure 31.4) returns the value $e^{-r_{ij}} p_{ij}$ of the put at an exercise point, discounted back to time zero.

ComputeRollover() updates the array CV_ex_ if it is found that an earlier exercise takes place.

Results: CV rollback

The results obtained by using this CV rollback are staggering. Figure 31.5 shows a typical set of continuation values, $\hat{Q}_i$, together with a set of corrected values, $\hat{Q}_i = \{\hat{Q}_{i,j}\}_{j=1,...,M}$. The corrected values are extremely tight compared with the plain values.

Fitting to the corrected values $\hat{Q}_i$ presents problems very different to those encountered in fitting to $\hat{Q}_i$. The narrowness of the corridor the fit is required to thread, and the fact that it is almost linear for most of its length, means that a large number of basis functions is likely to be required. A smaller number may be acceptable only if their shapes lie close to the target line.

Figure 31.6 shows errors between EEBs estimated with CV rollback and the lattice-derived EEB at each time step for B-splines and scaled natural basis functions. Each panel shows results for $M = 2500, 50625, 250000$ and $10^6$, with 16 stratification times. MC_American.CV.EExB.xls is used to compute these.

To the eye the fit seems to improve in each case as the numbers of basis functions and sample paths increases. This is confirmed by RMSEs, shown in Table 31.2. These are RMSEs computed from steps 17–64 for $K = 6, 12, 18$, and 24 basis functions, for scaled Laguerre, scaled natural and B-spline basis functions, for the case with $M = 10^6$ sample paths. As $K$ increases the RMSE is tending to decrease. With 12 or more basis functions the fit is very good, within a few hundredths of the true boundary level. This is approaching the degree of accuracy in the benchmark lattice EEB itself.

The fit is better for scaled natural and B-splines basis functions than for scaled Laguerre. Too few B-spline functions and the fit is awful: the wavy structure evident in Figure 31.6 panel (a) reflects the difficulty in making a fit with functions not possessing the right shape; once there are sufficient functions the fit is greatly improved.

4 Specify a “p” rollback type.
5 The plot, computed in MC_American.CV.EExB.xls, shows values computed at time $t_i = 59/64$, five steps into the valuation of a standard Bermudan put with 64 reset dates. For illustrative purposes it uses only $M = 2500$ sample paths.
Private bet_ As BetaComputer 'additional composite object
Private CV_ex_() As Double 'CV values at next exercise time, Redimed to 1 To M_
Private dt_ As Double 'time step
Private r_ As Double 'short rate
Private sig_ As Double 'sigma

Friend Sub IRollback_NextValues(ByRef Ovals() As Double, _
Svals() As Double, Hvals() As Double, _
dis As Double, ByRef ex() As Boolean)
    CurrentN_ = CurrentN_ - 1
    discount_(CurrentN_) = dis
    If CurrentN_ = N_ Then 'Initial values
        Cf_Vals_ = Hvals
        Ovals = Hvals
        CV_ex_ = Hvals
        Call MultiplyByScalar(CV_ex_, dis) 'discounted to time 0
        Call SetToBool(ex, True)
        Exit Sub
    End If
    Dim j As Long 'Set up Ovals()
    For j = 1 To M_
        Ovals(j) = Cf_Vals_(j) * discount_(TValues_(j)) / dis 'continuation values
    Next j
    Dim OvalsAdj() As Double: OvalsAdj = Ovals
    Call AdjustOvals(OvalsAdj, Svals, dis) 'corrected with cv
    Call ComputeRollover(Hvals, Svals, Ovals, OvalsAdj, CurrentN_, ex)
End Sub

Private Sub ComputeRollover(_
    H_values() As Double, S_values() As Double, ByRef Ovals() As Double, _
    Oadj() As Double, ind As Long, ByRef ex() As Boolean)
    'Lines omitted for brevity
    Dim N_itm As Long: N_itm = 0 'Number of paths in the money at time i
    Dim j As Long 'Set up ITM vectors
    For j = 1 To M_
        If H_values(j) > 0 Then
            A_val(N_itm) = Oadj(j) 'ITM ajusted cont. values
        End If
    Next j
    If k = 0 Then Call RaiseError(1115, name_, "No paths in the money")
    Dim c_vals() As Double: Call ComputeContValues(N_itm, c_vals, S_val, A_val)
    Dim Smax As Double: Smax = ComputeSmax(c_vals, H_val, S_val, N_itm)
    Dim k As Long
    For k = 1 To N_itm
        If S_val(k) < Smax Then 'Lines omitted for brevity
            CV_ex_(S_ind(k)) = ComputeCV(S_val(k), ind) 'update path CV value
        End If
    Next k
    For k = 1 To N_itm
        Ovals(S_ind(k)) = N_val(k)
    Next k
End Sub

Figure 31.3 The RollbackCVtau object: differences from RollbackLSmax
Private Sub AdjustOvals(ByRef OvalsAdj() As Double, Svals() As Double, dis As Double)
    Dim CVvals() As Double: ReDim CVvals(1 To 1, 1 To M_) As Double
    Dim tau As Double: tau = dt_ * (N_ - CurrentN_)
    Dim inflate As Double: inflate = 1 / dis
    Dim j As Long
    For j = 1 To M_ Step 1
        CVvals(1, j) = CV_ex_(j) * inflate - BSput(X_, tau, Svals(j), r_, sig_, 0#)
    Next j
    Dim beta() As Double: ReDim beta(0 To 1) As Double '0 is the constant value
    Call bet_.GetBeta(beta, OvalsAdj, CVvals, 1, M_) 'a.
    For j = 1 To M_ Step 1
        OvalsAdj(j) = OvalsAdj(j) - beta(1) * CVvals(1, j) 'b
    Next j
End Sub

Private Function ComputeCV(S As Double, t_ind As Long) As Double
    Dim Ttau As Double: Ttau = dt_ * (N_ - t_ind)
    ComputeCV = BSput(X_, Ttau, S, r_, sig_, 0#) * discount_(t_ind)
End Function

Figure 31.4 The RollbackCVtau object: additional Private procedures

![Adjusted and unadjusted continuation values](image1)

Figure 31.5 Unadjusted and CV adjusted continuation values
**Figure 31.6** CV rollback: errors in the estimation of the early exercise boundary. $M$ varies from 2500 to 1,000,000.
The effect of using the CV rollback is illustrated starkly in Figure 31.7. The panels, produced by MC_American_CV_one_pass.xls, show option values for the 64 reset standard Bermudan option computed in one pass with scaled Laguerre and B-spline basis functions, with up to 24 basis functions. Panel (a) is computed with plain rollback, panel (b) with CV rollback. Both cases use a put CV at the pricing stage. Computation times in panel (a) range from 15 to 127 seconds; times for panel (b) were around 25 seconds greater in every case.

The bias in panel (a) is acutely obvious, but this is entirely removed in panel (b). A consequence of having a good EEB is that the standard errors of the values in panel (b) are significantly less than those in panel (a). (This is visible to the eye and is discussed further below.)

Note that

1. Only with at least 7 or 8 basis functions is the bias removed.
2. It is obvious that it is the Bermudan put that is being valued and that its value cannot be used as an approximation to the value of the American put.
3. Standard errors in panel (b) reflect the standard errors of Table 31.1 (around 0.001, taking into account the reduced number of sample paths).

Figure 31.8 shows how RMSEs fall with $M$ as the number of basis functions varies.\textsuperscript{6} Once there are sufficient basis functions, 12 or so in these examples, adding additional basis functions decreases the

\textsuperscript{6} Found in MC_American_CV_EExB.xls.
Figure 3.18: Convergence of RMSE in $M$ and various numbers of basis functions. One-pass, put-put corrected EEB, no pricing CV.
RMSE by only a little; 6 basis functions are too few. Natural basis functions seem better able to fit to the EEB and reduce RMSE although B-splines also work well. Scaled Laguerre perform worst of the three.

The consequence for prices is shown in Figure 31.9. This shows ln-|pricing error| as a function of ln$(M)$, for $M = 2500, 50625, 250000$ and $10^6$, using a put-tau rollback, with and without a CV at the pricing stage. Panels (a), (b) and (c) (computed in MC_American_CV_EExB.xls) are obtained using the EEBs whose errors are plotted in Figure 31.6; for these no CV was used at the pricing stage. Panels (d), (e) and (f) (computed in MC_American_CV_one_pass.xls) show errors when a put-$\tau$ CV is used at the pricing stage. Results are noisy, particularly for small values of $M$.

A ln-error of $-4$ corresponds to an absolute error of $\sim 0.02$ and a ln-error of $-8$ to an absolute error of $\sim 0.0003$. Pricing error is decreasing as $M$ increases. With no pricing CV, the ln-error falls from about $-4$ to between $-7$ and $-8$; with a put-$\tau$ pricing CV, the ln-error falls from around $-6$ to between $-8$ and $-9$, say. The improvement for $M = 10^6$ is roughly a factor of 3.

Table 31.3 gives full results for B-spline basis functions (results for the other sets of basis functions are similar). Panel (a) presents results when there is no pricing CV, panel (b) when a put-$\tau$ pricing CV is used; panel (c) reports efficiency gains (compared to a plain LSLS B-spline estimate with the same $M$ and $K$) achieved when the put-$\tau$ pricing CV is used. The table reports raw standard errors (flagged with an asterisk); since 16 stratification times were used the true standard errors are a little less than the plain case. However the difference now is slight and we use the reported standard error as a surrogate for the true standard error. Panel (a) was produced in MC_American_CV_EExB.xls and panels (b) and (c) in MC_American_CV_one_pass.xls.

Computation times are quite similar between the two panels; real differences are submerged by noise. Using a put-$\tau$ CV adds very little to the expense if a put-$\tau$ rollback is already being used, and indeed the cost of a put-$\tau$ rollback is only slightly greater than the plain rollback. Standard errors reduce by roughly a quarter when a put-$\tau$ pricing CV is introduced and so efficiency gains (calculated using reported standard errors) reflect mainly the reduction in standard error. (Corresponding plain values are given in Table 30.4, page 486.)

We see that gains are around 400–500. A closer analysis reveals that the gains are split fairly evenly between (i) introducing a put-$\tau$ rollback and then (ii) adding a put-$\tau$ pricing CV.

To get a standard error of $\sim 0.005$ (with 12 B-spline basis functions and 16 stratification times) only around 2500 sample paths are needed. This gives a reported standard error of 0.006 in 8 seconds. A standard error of $\sim 0.0005$ can be had in only around 800 seconds. This is not small, but it is an amazing improvement compared to the plain case. In section 31.3 we do even better.

Adding a second CV at the pricing stage improves results still further. Table 31.4 gives results when both a put-tau and a Bermudan-early CV are used. $M = 50625$ sample paths were used with a put-$\tau$ rollback with no stratification. The efficiency gains reported in the table are relative to the plain case to those given in Table 30.4. Gains are roughly double those in Table 31.3 panel (c) and no doubt better CVs can yet be found.

One does not need to restrict oneself to a single CV in the rollback phase. Equation (31.15) can be extended to include several. However we did not note significant improvement in doing this.

### 31.3 A TWO-PASS LSLS

Using the same set of sample paths both to estimate the EEB and then to price from the estimated EEB, which is what the one-pass method has been doing, is unnecessarily restrictive. A two-pass method enables better results to be obtained. The first pass uses one set of sample paths to determine the early exercise boundary. The second pass uses an entirely new set of paths with this exercise boundary to compute the value of the option. The point is that since the second set of sample paths are not involved in an expensive set of
Implementing Models of Financial Derivatives

Figure 31.9 Convergence of Pricing Error: One-pass, Put-tau Corrected EEB, with and without a Pricing CV
Table 31.3  Put-tau rollback, priced with and without a put-tau CV, B-spline basis functions

<table>
<thead>
<tr>
<th></th>
<th>6</th>
<th>12</th>
<th>18</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>2500</td>
<td>6.118</td>
<td>6.112</td>
<td>6.090</td>
<td>6.074</td>
</tr>
<tr>
<td></td>
<td>(0.027*)</td>
<td>(0.026*)</td>
<td>(0.028*)</td>
<td>(0.028*)</td>
</tr>
<tr>
<td></td>
<td>[3.5]</td>
<td>[8.0]</td>
<td>[15]</td>
<td>[25]</td>
</tr>
<tr>
<td>50625</td>
<td>6.0712</td>
<td>6.0712</td>
<td>6.0757</td>
<td>6.0858</td>
</tr>
<tr>
<td></td>
<td>(0.0061*)</td>
<td>(0.0061*)</td>
<td>(0.0061*)</td>
<td>(0.0059*)</td>
</tr>
<tr>
<td></td>
<td>[68]</td>
<td>[150]</td>
<td>[290]</td>
<td>[490]</td>
</tr>
<tr>
<td>250000</td>
<td>6.0823</td>
<td>6.0839</td>
<td>6.0830</td>
<td>6.0857</td>
</tr>
<tr>
<td></td>
<td>(0.0028*)</td>
<td>(0.0027*)</td>
<td>(0.0027*)</td>
<td>(0.0027*)</td>
</tr>
<tr>
<td></td>
<td>[340]</td>
<td>[790]</td>
<td>[1470]</td>
<td>[2520]</td>
</tr>
<tr>
<td>1 000 000</td>
<td>6.0823</td>
<td>6.0812</td>
<td>6.0806</td>
<td>6.0809</td>
</tr>
<tr>
<td></td>
<td>(0.0014*)</td>
<td>(0.0014*)</td>
<td>(0.0013*)</td>
<td>(0.0013*)</td>
</tr>
<tr>
<td></td>
<td>[1410]</td>
<td>[3250]</td>
<td>[6010]</td>
<td>[9860]</td>
</tr>
</tbody>
</table>

Panel (a): B-splines, put-tau rollback, no CV

<table>
<thead>
<tr>
<th></th>
<th>6</th>
<th>12</th>
<th>18</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>2500</td>
<td>6.081</td>
<td>6.076</td>
<td>6.081</td>
<td>6.077</td>
</tr>
<tr>
<td></td>
<td>(0.0061*)</td>
<td>(0.0060*)</td>
<td>(0.0062*)</td>
<td>(0.0060*)</td>
</tr>
<tr>
<td></td>
<td>[3.4]</td>
<td>[8.1]</td>
<td>[15]</td>
<td>[24]</td>
</tr>
<tr>
<td>50625</td>
<td>6.079</td>
<td>6.081</td>
<td>6.082</td>
<td>6.081</td>
</tr>
<tr>
<td></td>
<td>(0.0014*)</td>
<td>(0.0013*)</td>
<td>(0.0013*)</td>
<td>(0.0013*)</td>
</tr>
<tr>
<td></td>
<td>[67]</td>
<td>[156]</td>
<td>[300]</td>
<td>[500]</td>
</tr>
<tr>
<td>250000</td>
<td>6.0793</td>
<td>6.0806</td>
<td>6.0814</td>
<td>6.0810</td>
</tr>
<tr>
<td></td>
<td>(0.0061*)</td>
<td>(0.0060*)</td>
<td>(0.0060*)</td>
<td>(0.0060*)</td>
</tr>
<tr>
<td></td>
<td>[330]</td>
<td>[770]</td>
<td>[1630]</td>
<td>[2460]</td>
</tr>
<tr>
<td>1 000 000</td>
<td>6.0799</td>
<td>6.0810</td>
<td>6.0816</td>
<td>6.0808</td>
</tr>
<tr>
<td></td>
<td>(0.0031*)</td>
<td>(0.0030*)</td>
<td>(0.0030*)</td>
<td>(0.0030*)</td>
</tr>
<tr>
<td></td>
<td>[1400]</td>
<td>[3100]</td>
<td>[6220]</td>
<td>[10500]</td>
</tr>
</tbody>
</table>

Panel (b): B-splines, put-tau rollback, put-tau CV

<table>
<thead>
<tr>
<th></th>
<th>6</th>
<th>12</th>
<th>18</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>2500</td>
<td>410</td>
<td>490</td>
<td>510</td>
<td>620</td>
</tr>
<tr>
<td>50625</td>
<td>330</td>
<td>450</td>
<td>510</td>
<td>530</td>
</tr>
<tr>
<td>250000</td>
<td>340</td>
<td>460</td>
<td>470</td>
<td>560</td>
</tr>
<tr>
<td>1 000 000</td>
<td>330</td>
<td>480</td>
<td>510</td>
<td>500</td>
</tr>
</tbody>
</table>

Panel (c): Efficiency gains, put-tau rollback, put-tau CV, relative to plain LSLS

regressions, the second pass is cheap. A much lower standard error is generated at marginal additional cost. As long as the expensive first pass estimates the EEB sufficiently accurately, the result will be unbiased.

Table 31.5, computed in MC_American_CV_one_pass_set_barrier.xls, illustrates. It show results when \( M_{\text{EEB}} \) sample paths are used to estimate the EEB and then an additional \( M_{\text{price}} \) sample paths are used to price the mimicking barrier option. \( M_{\text{EEB}} \) runs from 2500 to \( 10^6 \) and \( M_{\text{price}} \) from 2500 to \( 9 \times 10^6 \).
Twelve B-spline basis functions are used in the EEB estimation (results for other sets of basis functions are similar). Both passes use 16 stratification times.

As well as price, standard error and efficiency gain, the table also gives, in curly brackets, a relative pricing error, $\varepsilon_{\text{rel}} = (\hat{c} - c)/se$, where $\hat{c}$ is the Monte Carlo price, $c$ is the true price and $se$ is the standard error of $\hat{c}$. $\varepsilon_{\text{rel}}$ measures the distance of $\hat{c}$ away from $c$ in units of standard error. It is an indicator of bias; if $|\varepsilon_{\text{rel}}|$ is consistently greater than $\pm 2$, say, then bias may be present in the estimate.

The time taken for a combined run is the sum of times taken for the first pass and the second pass. These times are given (in square brackets) alongside the values of $M_{\text{EEB}}$ and $M_{\text{price}}$. Efficiency gains are relative to the plain LSLS method values with 12 B-splines and $M = M_{\text{EEB}}$.

The table is constructed with separate runs to compute the EEB and then to price the mimicking barrier option (see exercise 5).

One sees that the chief determinant of standard error is $M_{\text{price}}$; $M_{\text{EEB}}$ has no obvious effect on standard error. $M_{\text{EEB}}$ does, however, affect bias. For $M_{\text{EEB}} = 2500$ the EEB is estimated with significant noise. This is evident in $\varepsilon_{\text{rel}}$ for $M_{\text{price}} \geq 2.5 \times 10^5$. The Monte Carlo option price is many standard errors away...
from the true value. This bias is not detectable when $M_{\text{price}} \leq 50,625$; it is obscured by simulation noise. Bias is low when $M_{\text{EEB}} \geq 50,625$. Pricing error is, usually, a fraction of the standard error. There is no benefit, in this example, in having $M_{\text{EEB}} > 50,625$.

As $M_{\text{price}}$ increases so does the efficiency gain. $(M_{\text{EEB}}, M_{\text{price}}) = (50,625, 2.5 \times 10^5)$ takes only 180 seconds to achieve a reported standard error of only 0.0006, a gain of almost 2000 over plain LSLS. By contrast, a single-pass LSLS with $M = 2.5 \times 10^5$ takes 790 seconds to achieve the same standard error.

Notice that the results when $M_{\text{EEB}} = M_{\text{price}}$ are very similar to the one-pass results in Table 31.3, panel (b), second column. The two-pass method allows additional variance reduction at very low cost.

To see how the number of basis functions in the first pass affects the results, Figure 31.10 plots In-error against ln-$M_{\text{EEB}}$ with $M_{\text{price}}$ fixed at $9 \times 10^6$, as the number and type of basis function vary. Sample paths are stratified at 16 times, a put-tau pricing CV is used, and the EEB is computed using a put-tau CV.

As before, it is clear that 6 basis functions is too few to price without bias. B-splines give more consistent results, regardless of the number of basis functions used, as long as there are 12 or more. The In-error is around $-9$ to $-10$, better than scaled natural, and less erratic than scaled Laguerre. The error in all three cases is improving only erratically in $K$ and $M_{\text{EEB}}$ when more than a minimum number of each are used.

### 31.4 SUMMARY

A few years ago it was not possible to apply Monte Carlo for American options. Now the LSLS method enables quite accurate values of Bermudan puts to be found, and from these, using Richardson extrapolation (Chirayukool and Webber (2010)), equally accurate American put values can be obtained.

We have focused on the 64 reset standard Bermudan put, but what are the implications for valuing the standard American put directly? Using the put CV and stratified sampling it has been possible, with a tolerable number of sample paths, to value the Bermudan to a standard error of around $1 \times 10^{-4}$, with low bias.

At these levels of accuracy we can clearly distinguish between the values of the Bermudan and American options. Even if estimation error for the Bermudan EEB goes to zero one is still left with the pricing difference between the Bermudan and American options. The Bermudan EEB is nowhere near to the continuous time limit EEB needed for the American option: it has simply too few exercise dates. A Bermudan with 5120 reset dates has (lattice) value 6.09025. Only with around this number of time steps would an unbiased Monte Carlo estimate of the Bermudan value be likely to lie within one standard error (at 0.0001) of the American put value, but this number of time steps is a factor of 80 greater than that used for the benchmark 64 reset Bermudan considered in this chapter. Computation times would be 80 times greater, much too long for other than academic purposes. Fortunately extrapolation methods come to our rescue. Direct methods do not need to be used.

One is still left with bias caused by estimation error in the EEB of the Bermudan option itself. The problem of estimating the barrier at times close to the present has been ignored here, and would have to be addressed before progress can be made. (One could, for instance, follow suggestions made by Rasmussen (2005).)

One concludes that a Bermudan option can be valued reasonably accurately and (relatively) quickly. Pricing an American option by LSLS Monte Carlo is quite feasible using extrapolation.

---

7 The figure refers to $M_1 = M_{\text{EEB}}$. See MC_American_CV_one_pass_set_barrier.xls.
510 Implementing Models of Financial Derivatives

(a) Scaled Laguerre (b) Scaled natural (c) B-spline

Convergence of pricing error in $M_1$
First-pass: # = 16 put-tau corrected EEB,
Scaled Laguerre basis functions,
Second-pass: # = 16, put CV, $M = 9 \times 10^6$

Figure 31.10 Convergence of pricing error in $M_1$: two-pass, put-tau corrected EEB
31.5 EXERCISES

1. Chapter 10 presented an outline method shell that could incorporate various speed-ups. However we have not exploited the shell elsewhere in this book. How could the shell be used to accommodate the one-pass valuation of a Bermudan option with control variates?

2. Let \( \tau \) be the stopping time defined as the first hitting time to a (lower) barrier level \( U \). Let \( CV^U_\tau = e^{-r\tau} \rho_\tau(S_\tau) - p_0(S_0) \) be the put CV defined by this stopping time. What speed-ups are attainable using this CV? How does the efficiency gain depend upon the level \( U \)?

3. Let \( \tau \) be the first hitting time of the Bermudan option (with 64 reset dates) to its early exercise boundary. Define the CV \( CV^B_\tau = e^{-r\tau} b_U^U(S_\tau) - b_U^U(S_0) \) where \( b_U^U(S_\tau) \) is the value at time \( \tau \) of the down and in barrier option, with barrier level \( U \), knocking in to a European put with strike \( X \). How good a CV is this for the Bermudan put?

4. In a two-pass method the first-pass computes the early exercise boundary. Is there any way at all that two Bermudan puts, with different strikes and times of maturity, could compute their early exercise boundaries simultaneously with reduced duplicated computation?

5. The results in Table 31.5 were computed by combining two separate sets of results – they were not taken from runs of a single application. Construct a spreadsheet, taking \( M_{EEB} \) and \( M_{price} \) as input parameters, that conducts a two-pass LSLS Monte Carlo in one go.

6. The Chartmanager object in MC_American.CV_one_pass.xls uses, in certain circumstances, an evil \texttt{End} statement to terminate execution. How would you remove this defect?
This page intentionally left blank
This book has explored issues in software system design, and simulation valuation methods. It has used VBA with Excel, exploiting its language features to optimize the design and performance of numerical valuation systems. It has been a long journey, but the destination has been worthwhile, and the scenery on the way has at times been spectacular (outweighing, one hopes, the less attractive but necessary passages).

Where do we go from here?

Although the applications are in VBA the astute reader, versed in other languages, will have recognized how the concepts and designs described here transcribe almost directly to other contexts. Object-oriented design principles, ideas concerning the identification of and interaction between objects, abstract away from language specifics, and apply very broadly. They work because they relate first to the underlying problem, and only second to a particular programming language. The applications presented in this book can be built on and adapted to fit in with industrial, practical, valuation application systems.

The book valuation system, described in Chapter 10, is a shell, but can be integrated in with the necessary discretization and variance reduction techniques (described in Parts VI and VII) to apply to a wide range of models and methods. As new techniques and models are introduced, and new options require valuation and hedging, further objects can also be introduced, elaborating the basic structure. Although it is not possible to foretell in which directions new research, and the markets, will take us, the current structure is adaptable and flexible.

Only a few models have been investigated in this book: GBM, CIR, Heston, O–U. However it should be clear how other models – your favourite or your bank’s – can be incorporated. Stochastic interest rates, in an FX and equity context, have not been explored, but the structure here, with discount factors computed by an interest rate engine, can be included. Other topics not explored are various, including Lévy process and jump-diffusion models, models with copulas, Fourier transform methods, and elaborations on basic PDE and lattice methods. One hopes and expects that this book has nevertheless given a good grounding to the reader in the essentials of numerical implementation.

There is a limit on how far one may go with simulation methods; there is also a limit on how far one may go without them. The first limit is conveniently expressed as a function of the amount of time one needs to wait before getting an answer. The second in terms of the complexity of the market and the models needed to calibrate to it. Platforms get faster, but instruments become increasingly complex; the demands of the market soak up both computing power and methodological advances as soon as they become available.
One topic not touched upon, a serious omission, is multi-thread, multi-core computing. Techniques associated with these are going to become increasingly important. However a decent variance reduction technique, appropriately implemented, can mitigate the need to move in this direction. I believe, along with many others, in clever computing rather than clout-computing. I hope this book has helped a little to further the application of that belief in this particular area.
Appendices

The appendices contain supporting and background material. Appendix A reviews how to set up Excel and VBA for the purposes of this book. It also discusses some of the compiler problems that advanced users of VBA are likely to encounter.

Formulae for a number of options relevant to the material in this book are derived in Appendix B. These include discretely reset geometric average rate options. Appendix C discusses the library code modules that are used throughout the book, supporting the application code.

Appendix D gives an example and discusses some issues that arise in running DLLs from VBA. An example is included on the CD. DLLs are not used in this book; from its perspective this appendix is included for interest only. However, more advanced industrial users will need to be able to run code developed from other sources. DLLs provide a simple interchange device.

This book is intended to be used by readers who already have some familiarity with OOP techniques and by those who may have only a slight previous acquaintance. Appendix E provides an overview of object-oriented programming concepts to assist the reader less familiar with these ideas. It is designed to support the material presented in Chapters 5 and 6 where these ideas are met in earnest for the first time.

Appendices F and G describe level 0 and level 1 implementations of a lattice and a PDE method, similar in level to the Monte Carlo implementation described in Chapter 3. These provide a starting off point for a series of exercises in the main body of the book designed to enable the reader to apply the knowledge and techniques investigated in each chapter, there applied to Monte Carlo, to a pair of further practical valuation methods.

Another application stream is introduced in Appendix H. This describes several rather straightforward root-finding algorithms and a minimization algorithm. Having these available enables the reader to tackle a series of exercises based upon computing volatilities implied from market prices.
Appendix A  
VBA and Excel

In this appendix we discuss setting up VBA in Excel, and some of the VBA compilation issues that are sometimes encountered.

There are quite a number of online forums for VBA developers – too many, and perhaps too ephemeral, to list. VBA help can be obtained online from Microsoft sites. At the time of writing the proper site is


Although it contains a number of broken links and the navigation is not brilliant, it contains information, once you have found it.

A.1 SETTING UP EXCEL

You will have your own preferences for what you like Excel to look like. Here I describe my own.

A.1.1 Excel settings for Excel 2007

You should put the Developer menu into the ribbon. In Excel Options, tick “Show Develop tab” box in the Excel Options | Popular window. From the Excel Options | Customize window add the Visual Basic, Design Mode, Control Properties, and Code icons to the quick access toolbar (and any others you fancy) – see Figures A.1 and A.2.

To enable VBA macros, go to the Developer | Code | Macro Security window (Figure A.3) and select the desired security setting. In general, it is a bad idea always to trust macros, so I usually “Disable with notification”. (In previous versions of Excel set the security level to medium.) Make sure that “Trust access to the VBA project object model” is ticked.

A.1.2 VBA settings

There are some important options that must be set properly. It is absolutely necessary to set the “Require variable declaration” option. (Why this does not arrive already ticked by default I have no idea). Make the setting in the Editor tab of the Tools | Options window (Figure A.4). You may also want to select the “Break on All Errors” option from the General tab (Figure A.5) although this depends on what stage in its development an application has reached. The Project pane on the left-hand side of the VBA editor lists open VBA workbooks and add-ins. Add-ins may include, for example, AcrobatPDFMaker. Usually the add-ins can (and must) be ignored.

If you need access to the VBE object library, for instance because you are using spreadsheets implementing the polymorphic factory, then this has to be made available. Follow the steps given in section 11.1.

---

1 Code that relies on On Error Resume Next in routine execution, such as some versions of the polymorphic factory, or some file-handling idioms, will be inconvenienced. (A good reason to avoid these constructions.)
A.2 COMPILER PROBLEMS IN VBA

There are several well-known problems with the VBA compiler. A major issue concerns the behaviour of large VBA applications which from time to time can be subject to spurious and serious failure. The cause seems to be that maintenance and repeated compilations can erroneously leave behind old, superseded, chunks of compiled code. Ideally one would like to remove all compiled code from the project (a process in this context often referred to misleadingly as decompilation) then recompile cleanly. There are third-party applications, both commercial and freeware, that do this. If you do not have them available then the tedious solution is to periodically do the following by hand to each code module in the project:

1. Copy all the code into a text file.
2. Delete the module.
3. Insert another with the same name, and paste back in the code.
4. Compile the project (from the Debug menu).

With a large project this is very time consuming indeed but it will give you a clean compile and should make the problem go away.\(^2\)

\(^2\) Make sure you take a back-up copy of the workbook first.
The three main signals that indicate a problem are the “Error accessing file”, “Bad DLL calling convention” and “Excel has encountered a problem and needs to close” messages. Other messages may also be produced. One example is a “Type mismatch” error message where no type mismatch exists.

These messages can arise when there is a real underlying error, rather than a spurious problem with VBA. Here we assume it is the latter case.
520 Implementing Models of Financial Derivatives

Figure A.4 The Editor tab of the Tools | Options window

Figure A.5 The General tab of the Tools | Options window
Appendix A: VBA and Excel

The calamitous “Error accessing file” message

One of the most dangerous symptoms of corrupted compiled code is the “Error accessing file. Network connection may be lost” message. It is very serious; it can result in lost code. VBA appears to lose track of one or more code modules in a project. The affected modules are visible in the project explorer pane, but cannot be viewed, exported or edited.

There is no reliable cure. In the very worse case the application may even be completely lost. The only real solution is to create frequent back-ups, to clean-compile (as outlined above) fairly frequently, and to be prepared to lose work.

If the message appears then one possible way to salvage the disaster, after first backing-up the workbook, is to:

(1) Open the spreadsheet with “Enable macros” selected, if possible.
(2) Identify which modules are affected by systematically going through, trying to open each one.
(3) Close and re-open the spreadsheet in design mode, without enabling macros.
(4) Go into VBA and make minor changes to every affected module (in design mode you may now be able to open them).
(5) Compile.
(6) Save, close and re-open, with your fingers crossed.

Good luck.

The “Bad DLL calling convention” message

This is another VBA message that can strike apparently at random. VBA reports an error on a line of code, one that is likely to have nothing to do with any DLL, and this message is given.

The message is easier to cope with – try making a minor change to the affected module and recompiling – but it hints of darker days ahead. An alternative is also available: once the bad modules have been located, export them, then import them back in. As before, the proper solution is to undertake a clean compile.

The “Excel has encountered a problem and needs to close” message

You may see this when you open a workbook and enable macros. Untick “Recover my work” and go back into the application manually, this time not enabling macros. In no circumstances let Excel repair your workbook for you. You are likely to find that the repair has removed and probably lost your code modules.

Once in the workbook compile it, save, close and re-open. If this fails you will have to go through a clean recompile.

Spurious “Type mismatch” errors

A “Type mismatch” error where there is no mismatched type is an easier symptom to remove. Export the module, delete it, then import it back in. The fix, if it works, is likely only to be temporary; eventually – soon – a clean compile will be needed.

It is rumoured that Excel 2007 .xslm macro-enabled workbooks are not so susceptible to this problem. In the meantime there is a lot of code still in .xls workbooks.
This page intentionally left blank
This appendix reviews a number of valuation formulae used in the book. These are formulae for geometrically averaged average rate options, an option with a quadratic payoff, and a certain Bermudan option for which an explicit solution is available.

## B.1 GEOMETRICALLY AVERAGED AVERAGE RATE OPTIONS

This appendix reviews pricing formulae for discretely reset geometrically averaged average rate options on an asset following a geometric Brownian motion. These formulae are used to construct control variates for arithmetically averaged average rate options. Suppose that \( S = (S_t)_{t \geq 0} \) is a geometric Brownian motion with process

\[
dS_t = r S_t \, dt + \sigma S_t \, dz_t.
\]

We derive three formulae. The first case is for an option that is about to be issued, which has no part-average to account for; the second is a special case of the first in which reset dates are at regular intervals; the third is for options that have been alive for a while and which already have observations of the asset value contributing towards the final average.

### Discretely reset geometrically averaged average rate options

Let \( T = \{t_i\}_{i=1}^N \), with \( t_i < t_{i+1} \) for all \( i \), be a set of reset dates. Write \( S_i \) for \( S_{t_i} \) and suppose an average rate option pays off on the geometric average

\[
g_T = \left( \prod_{i=1}^N S_i \right)^{1/N} \tag{B.1}
\]

with payoff \( H_T = (g_T - X)^+ \) at time \( T \geq t_N \). This is a discretely reset geometrically averaged average rate call option.

When \( S_t \) is a geometric Brownian motion it is possible to show\(^1\) that, for times \( t < t_1 \), \( g_T \) is a log-normal variate. Specifically \( g_T \sim \text{LN} [\ln(S_t) + \mu T, \sigma^2 T] \) where

\[
\mu = r - \delta - \frac{1}{2} \sigma^2, \tag{B.2}
\]

\[
\tau = \frac{1}{N} \sum_{i=1}^N (t_i - t), \tag{B.3}
\]

\[
\sigma^2 \tau = \frac{\sigma^2}{N^2} \sum_{i=1}^N (2i - 1)(t_{N+1-i} - t), \tag{B.4}
\]

\(^1\) See Glasserman (2004).
with $\delta = \frac{1}{2}(\sigma^2 - \bar{\sigma}^2)$. Hence the value $v_t(X, T, S_t, r, \sigma)$ of a call option on $g^T$, maturing at time $T \geq t_N$, is given by a Black–Scholes type formula on an asset with initial value $S_t$, dividend yield $\delta$, volatility $\bar{\sigma}$, maturing at time $T$, corrected to reflect the time the payoff is actually made, $T$.

Writing $c_t(X, T, S_t, \delta, r, \sigma)$ for the value at time $t$ of a European call option maturing at time $T$ with strike $X$ on an asset $S$ with current value $S_t$, dividend yield $\delta$, volatility $\sigma$ and riskless rate $r$, we have

$$v_t(X, T, T, S_t, r, \sigma) = e^{r(\bar{T} - T + t)}c_t(X, \bar{T}, S_t, \delta, r, \bar{\sigma}).$$  \hfill (B.5)

Alternatively writing $F_t = e^{(r - \delta)T}S_t$ for the forward value of the average we have

$$v_t(X, T, S_t, \delta, r, \sigma) = e^{-r(T - t)}(F_tN(d_1) - XN(d_2)),$$  \hfill (B.6)

$$d_1 = \frac{1}{\sigma \sqrt{\bar{T}}} \ln \left( \frac{F_t}{X} \right) + \frac{1}{2} \bar{\sigma} \sqrt{\bar{T}},$$  \hfill (B.7)

$$d_2 = d_1 - \sigma \sqrt{\bar{T}}.$$  \hfill (B.8)

**A special case: regular reset dates**

As a special case of equation (B.6), suppose that we want to value an option that has $t_{i+1} - t_i = \Delta t$, constant for all $i$, at time $t = t_0 = t_1 - \Delta t$, maturing at $T = t_N$. Then

$$\bar{T} = \frac{1}{N} \sum_{i=1}^{N} i \Delta t = \frac{1}{2} (N + 1) \Delta t,$$  \hfill (B.9)

$$\bar{\sigma}^2 = \frac{\sigma^2}{N^2 \bar{T}} \sum_{i=1}^{N} (2i - 1)(N + 1 - i) \Delta t$$  \hfill (B.10)

$$= \frac{1}{3} \left( 2 + \frac{1}{N} \right) \sigma^2$$  \hfill (B.11)

and

$$\delta = \frac{1}{2} (\sigma^2 - \bar{\sigma}^2)$$  \hfill (B.12)

$$= \sigma^2 \left( \frac{1}{2} - \frac{1}{6} \left( 2 + \frac{1}{N} \right) \right)$$  \hfill (B.13)

$$= \frac{1}{6} \sigma^2 \left( 1 - \frac{1}{N} \right)$$  \hfill (B.14)

so

$$v_t(X, T, S_t, r, \sigma) = e^{-r(T - t)}(F_tN(d_1) - XN(d_2))$$  \hfill (B.15)

where

$$F_t = e^{\frac{1}{2}(r - \delta)(N + 1) \Delta t} S_t$$  \hfill (B.16)
and
\[
d_1 = \frac{1}{\bar{\sigma}\sqrt{T}} \ln \left( \frac{F_t}{X} \right) + \frac{1}{2} \bar{\sigma}\sqrt{T} \tag{B.17}
\]

with
\[
\bar{\sigma}\sqrt{T} = \sigma \sqrt{\frac{1}{6} \left( 2 + \frac{1}{N} \right) (N + 1)\Delta t}. \tag{B.18}
\]

**Part-way average rate options**

Average rate options need to be valued not only when they are issued but also part-way through their life. At a part-way time there will already be observations contributing to the ultimate value of \(g^T\). These must be taken into account.

Suppose that \(t_M < t < t_{M+1}\), for \(1 \leq M < N\), so that the option is observed part-way through its averaging period. Write \(Q = N - M\) for the number of remaining reset dates and write \(G_t = (\prod_{i=1}^{M} S_i)^{1/M}\) for the observed contribution to the average computed from dates \(t_1\) to \(t_M\). Set \(g = (\prod_{i=1}^{Q} S_{M+i})^{1/Q}\) for the unobserved average over \(t_{M+1}\) to \(t_N\). The option pays off against \(g^T = G_t^{M/N} g^{Q/N}\).

We know that \(g \sim \text{LN}[\ln(S_t) + \mu T, \bar{\sigma}^2 T]\) is log-normal where
\[
\mu = r - \delta - \frac{1}{2} \bar{\sigma}^2, \tag{B.19}
\]
\[
\delta = \frac{1}{2} (\bar{\sigma}^2 - \sigma^2), \tag{B.20}
\]
\[
\bar{\sigma}^2 = \frac{\sigma^2}{Q^2 T} \sum_{i=1}^{Q} (2i - 1)(t_{N+1-i} - t) \tag{B.21}
\]
\[
T = \frac{1}{Q} \sum_{i=1}^{Q} (t_{M+i} - t), \tag{B.22}
\]

and we can compute, writing \(t^+ = t_N - t - (Q - 1)\Delta t\) for the time to the next reset date,
\[
\bar{\sigma}^2 = \frac{\sigma^2}{Q^2 T^+} (Q^2 t^+ + \Delta t Q (Q - 1)(2Q - 1)/6), \tag{B.23}
\]
\[
T^+ = t^+ + \frac{1}{2} (Q - 1)\Delta t. \tag{B.24}
\]

If a random variate \(g \sim \text{LN}[\hat{\mu}, \hat{\sigma}^2]\) is log-normal then so is \(g^q \sim \text{LN}[q\hat{\mu}, q^2\hat{\sigma}^2]\); also for \(p \neq 0\) we have \((pg^q - X)^+ = p(g^q - X/p)^+\), so an option paying off on \(pg^q\) with strike \(X\), when \(g \sim \text{LN}[\ln(S_t) + \mu T, \bar{\sigma}^2 T]\), has value equivalent to \(p\) options with strike \(X/p\) on an asset with initial value \(S^p_t\), volatility \(q\hat{\sigma}\), riskless rate \(r\), and with a dividend yield \(\hat{\delta}\) satisfying
\[
q \left( r - \hat{\delta} - \frac{1}{2} \bar{\sigma}^2 \right) = r - \hat{\delta} - \frac{1}{2} q^2 \bar{\sigma}^2, \tag{B.25}
\]
so that
\[ \hat{\delta} = (1 - q)r + q\delta + \frac{1}{2}q\sigma^2(1 - q). \]  

(B.26)

Substituting in, setting \( q = Q/N \) and \( p = G_t^{M/N} \), and correcting for the actual payoff date, the value \( v_t(X, T, T, r, \sigma | G_t, M) \) of the part-way option is
\[ v_t(X, T, T, r, \sigma | G_t, M) = p e^{r(T - t)} c_t \left( \frac{X}{p}, T, \sigma^2, \hat{\delta}, r, q \right). \]  

(B.27)

An implementation of this formula is given in the spreadsheet MC_AverageRate_benchmark.xls.

**B.2 A QUADRATIC PAYOFF OPTION**

Consider a European option whose payoff \( H_T(S) \) at maturity time \( T \) is
\[ H_T(S) = \left( X_1 - (S_N - X_2)^2 \right)^+. \]  

(B.28)

This quadratic payoff option was used as an example in Chapters 20 and 21, and the derivation of an explicit formula for its value was set as an exercise on page 331. We obtain an expression for its value, \( c_T^e \).

It is a special case of an option with a general polynomial payoff \( H_T^p(S) \),
\[ H_T^p(S) = \begin{cases} 
    p(S), & S \in [l, u], \\
    0, & S \notin [l, u],
\end{cases} \]  

(B.29)

where
\[ p(S) = \sum_{i=0}^{N} a_i S^i \]  

(B.30)

is a polynomial function of \( S \). Notice that the payoff to this option is not constrained to be everywhere positive.

Suppose \( S_t \) follows a geometric Brownian motion
\[ dS_t = rS_t dt + \sigma S_t dz_t \]  

(B.31)

then the distribution of \( S_T \mid S_0 \sim LN[\mu, \sigma^2] \) is log-normal with
\[ \mu = \ln(S_0) + \left( r - \frac{1}{2} \sigma^2 \right) T, \]  

(B.32)

\[ \sigma^2 = \sigma^2 T, \]  

(B.33)

and the density \( f(S) \) of \( S \) is
\[ f(S) = \frac{1}{S\sigma\sqrt{2\pi}} \exp \left( -\frac{1}{2} \left( \frac{\ln(S) - \mu}{\sigma} \right)^2 \right). \]  

(B.34)
The value \( c_t \) of the general polynomial option is

\[
c_t = e^{-rT} \int_0^\infty H(S) f(S) \, dS
\]  
(B.35)

\[
e^{-rT} \sum_{i=0}^\infty a_i \int_\mu^\infty S^i f(x) \, dS
\]  
(B.36)

The integral \( \int_\mu^\infty S^i f(x) \, dS = \int_\mu^\infty S^i f(x) \, dS - \int_\mu^\infty S^i f(x) \, dS \) can be computed explicitly. We have

\[
\int_\mu^\infty S^k f(S) \, dS = \int_\mu^\infty \frac{S^k}{\sigma \sqrt{2\pi}} \exp\left( -\frac{1}{2} \left( \frac{\ln(S) - \mu}{\sigma} \right)^2 \right) \, dS
\]  
(B.37)

\[
= \int_\mu^\infty \frac{1}{\sigma \sqrt{2\pi}} \exp\left( (k-1) \ln(S) - \frac{1}{2} \left( \frac{\ln(S) - \mu}{\sigma} \right)^2 \right) \, dS
\]  
(B.38)

\[
= \int_\mu^\infty \frac{1}{\sigma \sqrt{2\pi}} \exp\left( -\frac{1}{2\sigma^2}(\ln^2(S) - 2(\mu + (k-1)\sigma^2) \ln(S) + \mu^2) \right) \, dS
\]  
(B.39)

Set \( \mu_k = \mu + (k-1)\sigma^2 \). Then

\[
\int_\mu^\infty S^k f(S) \, dS = \int_\mu^\infty \frac{1}{\sigma \sqrt{2\pi}} \exp\left( -\frac{1}{2\sigma^2}(\ln(S) - \mu_k)^2 + \mu^2 - \mu_k^2 \right) \, dS
\]  
(B.40)

\[
= \exp\left( \frac{\mu_k^2 - \mu^2}{2\sigma^2} \right) \int_\mu^\infty \frac{1}{\sigma \sqrt{2\pi}} \exp\left( -\frac{1}{2} \left( \frac{\ln(S) - \mu_k}{\sigma} \right)^2 \right) \, dS
\]  
(B.41)

The right-hand side of equation (B.41) can be computed quite easily. Change variable to \( y = (\ln(S) - \mu_k)/\sigma \) and set \( l_k = (\ln(l) - \mu_k)/\sigma \). Then

\[
\int_\mu^\infty \frac{1}{\sigma \sqrt{2\pi}} \exp\left( -\frac{1}{2} \left( \frac{\ln(S) - \mu_k}{\sigma} \right)^2 \right) \, dS
\]  

\[
= \int_{l_k}^\infty \frac{1}{\sigma \sqrt{2\pi}} \exp\left( -\frac{1}{2} y^2 \right) \sigma e^{\sigma y + \mu_k} \, dy
\]  
(B.42)

\[
= \int_{l_k}^\infty \frac{1}{\sqrt{2\pi}} \exp\left( -\frac{1}{2} \left( y - \sigma \right)^2 - 2\mu_k - \sigma^2 \right) \, dy
\]  
(B.43)

\[
= \exp\left( \mu_k + \frac{1}{2} \sigma^2 \right) \int_{l_k}^\infty \frac{1}{\sqrt{2\pi}} \exp\left( -\frac{1}{2} (y - \sigma)^2 \right) \, dy
\]  
(B.44)

\[
= \exp\left( \mu_k + \frac{1}{2} \sigma^2 \right) N\left( \frac{\mu_k + \sigma^2 - \ln(l)}{\sigma} \right)
\]  
(B.45)

Putting together equations (B.39) and (B.45), and noticing that

\[
\frac{\mu_k^2 - \mu^2}{2\sigma^2} + \mu_k + \frac{1}{2} \sigma^2 = k\mu + \frac{1}{2} k^2 \sigma^2,
\]  
(B.46)
Implementing Models of Financial Derivatives

we get

\[ \int_{l}^{\infty} S^k f(S) \, dS = \exp \left( k\mu + \frac{1}{2} k^2 \sigma^2 \right) N \left( \frac{\mu + k\sigma^2 - \ln(l)}{\sigma} \right) . \] (B.47)

Hence the value of the general polynomial option is

\[ c_t = e^{-rT} \sum_{k=0}^{N} a_k \exp \left( k\mu + \frac{1}{2} k^2 \sigma^2 \right) \times \left( N \left( \frac{\mu + k\sigma^2 - \ln(l)}{\sigma} \right) - N \left( \frac{\mu + k\sigma^2 - \ln(u)}{\sigma} \right) \right) . \] (B.48)

For the particular case of the quadratic option we have

\[ H_T^p(S) = \begin{cases} 
-S^2 + 2X_1 S + X_1 - X_2^2, & S \in [l, u], \\
0, & S \not\in [l, u].
\end{cases} \] (B.49)

where \( l = X_1 - \sqrt{X_2} \), \( u = X_1 + \sqrt{X_2} \). With these values of \( l \) and \( u \) the value \( c_t^e \) of the quadratic option is given by

\[ c_t^e = -e^{-rT} \exp(2\mu + 2\sigma^2) \left( N \left( \frac{\mu + 2\sigma^2 - \ln(l)}{\sigma} \right) - N \left( \frac{\mu + 2\sigma^2 - \ln(u)}{\sigma} \right) \right) \\
+ e^{-rT} 2X_1 \exp \left( \mu + \frac{1}{2} \sigma^2 \right) \left( N \left( \frac{\mu + \sigma^2 - \ln(l)}{\sigma} \right) - N \left( \frac{\mu + \sigma^2 - \ln(u)}{\sigma} \right) \right) \\
+ e^{-rT} (X_1 - X_2^2) \left( N \left( \frac{\mu - \ln(l)}{\sigma} \right) - N \left( \frac{\mu - \ln(u)}{\sigma} \right) \right) . \] (B.50)

MC_exotic_benchmark.xls contains an implementation of this formula.

**B.3 A BERMUDAN OPTION**

Suppose that \( t < T_1 < T_2 \) and let \( b_t \) be the value at time \( t \) of a Bermudan put exercisable at times \( T_1 \) and \( T_2 \) with strike \( X \). The value \( b_{T_1} \) of the Bermudan put just after time \( T_1 \) is

\[ b_{T_1}(S_{T_1}) = \max(X - S_{T_1}, \ p_{T_1}(S_{T_1} \mid X, T_2)) \] (B.51)

\[ = \max(0, \ p_{T_1}(S_{T_1} \mid X, T_2) + S_{T_1} - X) + X - S_{T_1} \] (B.52)

\[ = \max(0, \ c_{T_1}(S_{T_1} \mid X, T_2) + \text{Pv}_{T_1}(X, T_2) - X) + X - S_{T_1} \] (B.53)

by put–call parity, where \( \text{Pv}_{T_1}(X, T_2) = e^{-r(T_2-T_1)}X \) is the value at time \( T_1 \) of \( X \) received at time \( T_2 \). But \( \max(0, \ c_{T_1}(S_{T_1} \mid X, T_2) + \text{Pv}_{T_1}(X, T_2) - X) \) is the payoff at time \( T_1 \) of a compound call option with strike \( X_1 = X - \text{Pv}_{T_1}(X, T_2) \) exercising at time \( T_1 \) into a call maturing at time \( T_2 \) with strike \( X_2 = X \).
An explicit solution for the value $\hat{b}_t$ of this compound call option at times $t < T_1$ exists. In fact

$$\hat{b}_t(S_t) = S_t N_\rho(d_1, e_1) - X_2 e^{-rT_2} N_\rho(d_2, e_2) - X_1 e^{-rT_1} N(e_2)$$  \hspace{1cm} (B.54)

where $N_\rho(d, e)$ is the bivariate normal distribution function with correlation $\rho$,

$$d_1 = \frac{1}{\sigma \sqrt{T_2}} \ln \left( \frac{S_t}{X_2 e^{-rT_2}} \right) + \frac{1}{2} \sigma \sqrt{T_2},$$  \hspace{1cm} (B.55)

$$e_1 = \frac{1}{\sigma \sqrt{T_1}} \ln \left( \frac{S_t}{\hat{S} e^{-rT_1}} \right) + \frac{1}{2} \sigma \sqrt{T_1},$$  \hspace{1cm} (B.56)

with $d_2 = d_1 - \sigma \sqrt{T_2}$, $e_2 = e_1 - \sigma \sqrt{T_1}$, $\rho = \sqrt{\frac{T_1}{T_2}}$, and $\hat{S}$ solves

$$c_t(\hat{S}, X_2, T_2 - T_1) = X_1. \hspace{1cm} (B.57)$$

The value of the Bermudan option is then

$$b_t(S_t) = \hat{b}_t(S_t) + e^{-rT_1} X - S_t.$$

It is easy to solve for $\hat{S}$ in equation (B.57). Note that $\hat{S}$ lies in a bounded domain $X_1 < \hat{S} < X_1 + X_2$ and $c_t$ is monotonic in $\hat{S}$. Any of the root-finding algorithms discussed in Appendix H work well for this very tractable problem.

This formula is implemented in the spreadsheet CompoundOptionExplicit.xls.
There are several utility code modules used throughout this book. Usually new utility procedures are discussed when they are used for the first time but sometimes they are introduced silently. This section give a systematic list of the utility procedures in each utility code module.

The modules are LibUtility, LibMatrix, LibValidators, LibRandom, LibBrownianBridge, LibStats, LibTrig, LibVec, LibSobol, LibFactory, LibCopula and LibAverageRate, contained in the spreadsheet LibraryProcedures.xls. Note that sometimes cut-back versions of the full modules have been used in the application examples, or pick-and-mix extractions made from here and there, particularly in the early chapters of the book. Note that the modules are not independent; procedures in one module may call procedures in another.

Two further spreadsheets, complex.xls and quadrature.xls, contain a complex number object and a set of quadrature objects, respectively. It seems logical to place their descriptions in this appendix.

To call a library procedure it is best to qualify its name with the module name, for instance by writing

```
Call LibUtility.RaiseError(123, name_, mess_) '.
```

This is used in particular is used by a great deal of application code.

### C.1 THE UTILITY PROCEDURES

A number of the numerical procedures are transcriptions into VBA of *Numerical Recipes* code\(^1\) originally given in C. The code presented here is for pedagogical use only; I very strongly recommend the purchase of *Numerical Recipes* by anyone doing any serious numerical work. In particular, *Numerical Recipes* code works; although they have been tested, my transcriptions may have unwittingly introduced so far undetected errors.

The descriptions of each code module includes tables listing the signatures of the procedures to be found within them. All procedures are declared to be `Public`. To save space the `Public` qualifier is omitted from the tables.

Some of procedures return values in arguments passed `ByRef`. Other arguments are passed `ByVal`. To save space the qualifier `ByVal` is dropped from arguments in the tables.

---

**LibUtility**

This contains mathematical constants, arithmetic comparators, parity testers, error-raising utilities, and miscellaneous text-handling utilities, listed in Table C.1. The error-raising procedures are utilities that assist error handling. `RaiseError()` is a little spoonful of syntactic sugar, wrapping an `Err.Raise` statement; `ForwardError()` re-throws the `Err` object with some additional text added to it. `RaiseError()` in particular is used by a great deal of application code.

The text-handling utilities are used by some of the validators in LibValidators. `AddChar()` concatenates characters onto a `String`; `SpaceOutChars()` inserts spaces between adjacent characters in its `String`

---

\(^1\) Mostly from the second edition (Press *et al*., 2002), but with some from the third edition (Press *et al*., 2007).
### Table C.1  LibUtility: miscellaneous but useful procedures

#### Error Raising utilities

<table>
<thead>
<tr>
<th>Sub</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RaiseError</td>
<td>num As Long, source As String, mess As String</td>
</tr>
<tr>
<td>ForwardError</td>
<td>ByRef Nums() As Long, source As String</td>
</tr>
</tbody>
</table>

#### Miscellaneous text handling procedures

<table>
<thead>
<tr>
<th>Sub</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddChar</td>
<td>char As String, ByRef strg As String</td>
</tr>
<tr>
<td>SpaceOutChars</td>
<td>chars As String As String</td>
</tr>
<tr>
<td>PadOut</td>
<td>st As String, size As Long As String</td>
</tr>
<tr>
<td>PadSpaces</td>
<td>st As String As String</td>
</tr>
<tr>
<td>FindIndex</td>
<td>a As String, b() As String As Long</td>
</tr>
</tbody>
</table>

#### Mathematical constants

<table>
<thead>
<tr>
<th>Const</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PI</td>
<td>3.14159265358979</td>
</tr>
<tr>
<td>PI/2</td>
<td>1.5707963267949</td>
</tr>
<tr>
<td>2*PI</td>
<td>6.28318530717959</td>
</tr>
<tr>
<td>sqrt(2*PI)</td>
<td>2.506628274631</td>
</tr>
<tr>
<td>1/sqrt(Pi)</td>
<td>0.564189583547756</td>
</tr>
<tr>
<td>1/sqrt(2)</td>
<td>0.707106781186545</td>
</tr>
<tr>
<td>1/sqrt(2PI)</td>
<td>0.398942280401433</td>
</tr>
<tr>
<td>1/sqrt(3)</td>
<td>0.577350269189626</td>
</tr>
</tbody>
</table>

#### Cast to Object

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>anon</td>
<td>obj As Object As Object</td>
</tr>
</tbody>
</table>

#### Arithmetic comparators

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equals</td>
<td>(a As Double, b As Double) As Boolean</td>
</tr>
<tr>
<td>EqualsZero</td>
<td>a As Double) As Boolean</td>
</tr>
<tr>
<td>LessThanOrEquals</td>
<td>(a As Double, b As Double) As Boolean</td>
</tr>
<tr>
<td>my_max</td>
<td>(a As Double, b As Double) As Double</td>
</tr>
<tr>
<td>my_min</td>
<td>(a As Double, b As Double) As Double</td>
</tr>
<tr>
<td>my_max_long</td>
<td>(a As Long, b As Long) As Long</td>
</tr>
<tr>
<td>my_min_long</td>
<td>(a As Long, b As Long) As Long</td>
</tr>
</tbody>
</table>

#### Parity testers

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IsEven</td>
<td>N As Long) As Boolean</td>
</tr>
<tr>
<td>IsOdd</td>
<td>N As Long) As Boolean</td>
</tr>
</tbody>
</table>

The mathematical constants and arithmetic comparators are used almost universally.\n\n**Equals()** and\n\n**LessThanOrEquals()** compare two Doubles. If they are within a tolerance level (defined in LibUtility)\n\nof one another then they are deemed to be equal.\n\n**EqualsZero()** tests equality with zero.

The procedure **anon()** just casts its argument as an Object and returns it. This is not pointless: it enables functionality to be popped from objects created on the fly.

Because it contains so many widely used utilities it is intended that LibUtility is included routinely in most applications.
Table C.2 LibMatrix: procedures for linear regression

<table>
<thead>
<tr>
<th>Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sub ols_r(ByRef a() As Double, N As Long, M As Long, b() As Double, ByRef x() As Double)</td>
</tr>
<tr>
<td>Sub choldc(ByRef a() As Double, N As Long, ByRef p() As Double)</td>
</tr>
<tr>
<td>Sub cholsl(a() As Double, N As Long, p() As Double, b() As Double, ByRef x() As Double)</td>
</tr>
<tr>
<td>Sub Cholesky(ByRef a() As Double, ByRef b() As Double, N As Long)</td>
</tr>
<tr>
<td>Sub Invert_LT_matrix(ByRef a() As Double, ByRef b() As Double, N As Long)</td>
</tr>
</tbody>
</table>

LibMatrix

This contains some procedures to compute linear regression (Table C.2). The workhorse procedures, choldc() and cholsl(), are from Numerical Recipes.

LibValidators

LibValidators contains procedures that input from a spreadsheet front-end, validation functions used to validate input, and miscellaneous output functions. From Chapter 4 onwards they have been used in every application that inputs from a spreadsheet.

Table C.3 lists the procedures in four categories. The first panel gives the input utilities; the second panel lists validators used to confirm the type of Variants and return the value they contain; the third shows the validators used to test some basic numerical properties of their arguments; the fourth manipulates cells on the front-end.

The getters in the first panel come in two forms: the first form reads in from the active sheet; the second requires the client to specify the sheet from which the input is to be taken. The latter raise an error if the sheet does not exist. get_char() and get_char_wk() require input to be a single character from a list of acceptable characters in a String supplied as an argument.

There are two procedures that slightly extend this basic classification. get_long_in_range() checks that the input is a Long that lies in a specified range. get_char_force() is designed for interactive use. If an invalid character is entered on the sheet then instead of throwing, get_char_force() prompts the user for a valid character.

The validators in the middle panel are used by the input procedures. Items are read in from the front-end as Variants. The validators check if their argument can be converted to the type being validated for; if they can then the converted value is returned, otherwise an error is thrown.

LibRandom

This code module contains procedures that generate random variates. The generators, listed in Table C.4, produce variates from the exponential, beta, Poisson, normal, uniform, student, inverse gamma, gamma, and chi-squared distributions. A number of functions are transcribed from Numerical Recipes, as indicated in the table.

LibBrownianBridge

This library contains a number of procedures to be used with Brownian and Wiener bridges, including both distributions and samples. These are listed in Table C.5. Procedures may have a large number of arguments.

2 If the String is empty, any character is accepted.
### Table C.3  LibValidators: spreadsheet input utilities and validators

#### Inputters from a spreadsheet front-end

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Function get_double(x As Long, y As Long) As Double</code></td>
<td>'Double from (x, y)'</td>
</tr>
<tr>
<td><code>Function get_double_in_range(x As Long, y As Long, L As Double, U As Double) As Double</code></td>
<td>'Double from (x, y) in [L, U]'</td>
</tr>
<tr>
<td><code>Function get_bool(x As Long, y As Long) As Boolean</code></td>
<td>'Boolean from (x, y)'</td>
</tr>
<tr>
<td><code>Function get_long(x As Long, y As Long) As Long</code></td>
<td>'Long from (x, y)'</td>
</tr>
<tr>
<td><code>Function get_long_in_range(x As Long, y As Long, L As Long, U As Long) As Long</code></td>
<td>'Long from (x, y) in [L, U]'</td>
</tr>
<tr>
<td><code>Function get_char(x As Long, y As Long, vals As String) As String</code></td>
<td>'Character from (x, y)'</td>
</tr>
<tr>
<td><code>Function get_char_force(X As Long, y As Long, vals As String) As String</code></td>
<td>'Forces a valid char from (x, y)'</td>
</tr>
<tr>
<td><code>Function get_string(x As Long, y As Long) As String</code></td>
<td>'String from (x, y)'</td>
</tr>
<tr>
<td><code>Function get_double_wk(wk As String, x As Long, y As Long) As Double</code></td>
<td>'Double from wk.Cells(x, y)'</td>
</tr>
<tr>
<td><code>Function get_bool_wk(wk As String, x As Long, y As Long) As Boolean</code></td>
<td>'Boolean from wk.Cells(x, y)'</td>
</tr>
<tr>
<td><code>Function get_long_wk(wk As String, x As Long, y As Long) As Long</code></td>
<td>'Long from wk.Cells(x, y)'</td>
</tr>
<tr>
<td><code>Function get_char_wk(wk As String, x As Long, y As Long, vals As String) As String</code></td>
<td>'Character from wk.Cells(x, y)'</td>
</tr>
<tr>
<td><code>Function get_string_wk(wk As String, x As Long, y As Long) As String</code></td>
<td>'String from wk.Cells(x, y)'</td>
</tr>
<tr>
<td><code>Function get_path_wk(wk As String, x As Long, y As Long) As String</code></td>
<td>'Path from wk.Cells(x, y)'</td>
</tr>
</tbody>
</table>

#### Validate a Variant into a primitive type

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Function check_double(x As Variant) As Double</code></td>
<td>'Returns x as a Double'</td>
</tr>
<tr>
<td><code>Function check_bool(x As Variant) As Boolean</code></td>
<td>'Returns x as a Boolean'</td>
</tr>
<tr>
<td><code>Function check_long(x As Variant) As Long</code></td>
<td>'Returns x as a Long'</td>
</tr>
<tr>
<td><code>Function check_char(x As String, vals As String, msg As String) As String</code></td>
<td>'Returns x if it is in vals'</td>
</tr>
<tr>
<td><code>Function check_path(pth As String) As String</code></td>
<td>'Check if pth is a String'</td>
</tr>
</tbody>
</table>

#### Validate numerical properties

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Function Check_strictly_positive(x As Double, s As String) As Double</code></td>
<td>'x &gt; 0'</td>
</tr>
<tr>
<td><code>Function Check_positive(x As Double, s As String) As Double</code></td>
<td>'x &gt;= 0'</td>
</tr>
<tr>
<td><code>Function check_in_range(x As Double, L As Double, U As Double) As Boolean</code></td>
<td>'x ∈ [L, U]'</td>
</tr>
<tr>
<td><code>Function check_even(x As Long) As Boolean</code></td>
<td>'x even'</td>
</tr>
<tr>
<td><code>Sub ValidateIncreasing(a As Double, b As Double, s As String)</code></td>
<td>'a &lt;= b'</td>
</tr>
<tr>
<td><code>Sub ValidateStrictlyIncreasing(a As Double, b As Double, s As String)</code></td>
<td>'a &lt; b'</td>
</tr>
</tbody>
</table>

#### I/O to the Front-end

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Sub ClearColumn(sht As String, x As Long, y As Long)</code></td>
<td>'Clears consecutive non-empty cells'</td>
</tr>
<tr>
<td><code>Sub ReadColumn(ByRef vec() As Double, sht As String, x As Long, y As Long)</code></td>
<td>'Reads into vec from sht'</td>
</tr>
<tr>
<td><code>Sub WriteColumn(sht As String, x As Long, y As Long, dat() As Double)</code></td>
<td>'Writes dat to sht'</td>
</tr>
<tr>
<td><code>Sub WriteColumnL(sht As String, x As Long, y As Long, dat() As Long)</code></td>
<td>'Version for dat Long'</td>
</tr>
<tr>
<td><code>Sub WriteColumnIndexes(sht As String, x As Long, y As Long, lb As Long, ub As Long)</code></td>
<td>'Writes a sequence to sht'</td>
</tr>
</tbody>
</table>
Table C.4  LibRandom: random number generators

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rand_exp_VN() As Double</td>
<td>exponential</td>
</tr>
<tr>
<td>rand_beta_A_and_W(alpha As Double, beta As Double) As Double</td>
<td>beta</td>
</tr>
<tr>
<td>rand_beta_johnk(alpha As Double, beta As Double) As Double</td>
<td>beta</td>
</tr>
<tr>
<td>rand_beta_cheng(alpha As Double, beta As Double) As Double</td>
<td>beta</td>
</tr>
<tr>
<td>rand_beta(alpha As Double, beta As Double) As Double</td>
<td>beta</td>
</tr>
<tr>
<td>rand_Poisson(lambda As Double) As Double</td>
<td>Poisson</td>
</tr>
<tr>
<td>normal1() As Double</td>
<td>normal (polar rejection)</td>
</tr>
<tr>
<td>GetTwoNormals(ByRef n1 As Double, ByRef n2 As Double, rho As Double)</td>
<td>Two correlated normals</td>
</tr>
<tr>
<td>Brownian_bridge(ByRef bb, i As Long, N As Long, dt As Double, j As Long, z As Double)</td>
<td>Brownian bridge</td>
</tr>
<tr>
<td>stat_ran0() As Double</td>
<td>NR, uniform</td>
</tr>
<tr>
<td>ran2() As Double</td>
<td>NR, uniform</td>
</tr>
<tr>
<td>student0(V As Double) As Double</td>
<td>student</td>
</tr>
<tr>
<td>rand_ig(delta As Double, gamma As Double) As Double</td>
<td>inverse Gaussian</td>
</tr>
<tr>
<td>rand_gamma(alpha As Double, beta As Double) As Double</td>
<td>NR, gamma</td>
</tr>
<tr>
<td>rand_chi2(nu As Double) As Double</td>
<td>Chi^2</td>
</tr>
</tbody>
</table>

Table C.5  LibBrownianBridge: Brownian Bridge related procedures

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BB_hitting_time_dist(w0, wt, T, a, b, sigma)</td>
<td>'Hitting time dist. of the bridge to a linear barrier</td>
</tr>
<tr>
<td>BB_max_dist(w0, wt, T, b, sigma)</td>
<td>'Dist. of maximum of the Brownian bridge</td>
</tr>
<tr>
<td>BB_min_dist(w0, wt, T, b, sigma)</td>
<td>'Dist. of minimum of the Brownian bridge</td>
</tr>
<tr>
<td>BB_max_draw(w0, wt, T, sigma, u)</td>
<td>'Draw from the maximum of the Brownian bridge</td>
</tr>
<tr>
<td>BB_max_draw_vec(ByRef mn(), w0(), wt() T, sigma, u())</td>
<td>'Draw from the maximum of the bridge: array arguments</td>
</tr>
<tr>
<td>WB_max_draw(z0, zT, T, u)</td>
<td>'Draw from the maximum of a Wiener bridge</td>
</tr>
<tr>
<td>WB_max_draw_vec(ByRef mn(), z0(), zT() T, u())</td>
<td>'Draw from the maximum of a Wiener bridge: arrays</td>
</tr>
<tr>
<td>BB_min_draw(w0, wt, T, sigma, u)</td>
<td>'Draw from the minimum of the Brownian bridge</td>
</tr>
<tr>
<td>BB_min_draw_vec(ByRef mn(), w0(), wt() T, sigma, u())</td>
<td>'Draw from the minimum of the bridge: array arguments</td>
</tr>
<tr>
<td>WB_min_draw(z0, zT, T, u)</td>
<td>'Draw from the minimum of a Wiener bridge</td>
</tr>
<tr>
<td>WB_min_draw_vec(ByRef mn(), z0(), zT() T, u())</td>
<td>'Draw from the minimum of a Wiener bridge: arrays</td>
</tr>
</tbody>
</table>
To save space the table drops the type declaration of procedure arguments and the return type; these are always Double. In the table \( w_0 \) and \( w_T \) are end points of a Brownian bridge at times 0 and \( T \), and \( z_0 \) and \( z_T \) are end points of a bridge for a Wiener process; \( \sigma \) is the volatility of the Brownian motion and \( u \) is a uniform variate; array values are returned ByRef in the array \( mn \).

**LibStats**

The density and distribution functions (Table C.6), are for the normal, chi-squared, t, and beta distributions (and one of two other miscellaneous distributions). As usual a number of functions are transcribed from *Numerical Recipes*.

**LibTrig**

VBA has only a few trigonometric functions: \( \cos() \), \( \sin() \), \( \tan() \), and \( \arctan() \). This code module contains a set of trigonometric functions, listed in Table C.7, to supplement these. Most are modified from VB2TheMax Code Bank code, available at the time of writing at

http://www.devx.com/vb2themax/Door/18897?cat=752&type=1 \hspace{1cm} (C.2)

Although these procedures are not actually used in this book they, and the code bank itself, are an excellent resource. Please visit the VB2TheMax web site for author attributions. Derived mathematical functions are also given on the Microsoft VBA reference website.

**LibVec**

There are recurring vector operations for which it is useful to have stock procedures. This module contains some very basic, very simple, vector operations. They are given in Table C.8. Using these enables the client to avoid looping, encapsulating loop counters in a procedure, resulting in cleaner, friendlier, and sometimes even cheaper code.

Just a few procedures require comment. \( \text{NegateVec}() \) changes the sign of each element in the array supplied in the argument. \( \text{CompVec}() \) replaces each element \( a(i) \) of \( a \) by \( U - a(i) \). This is used, for instance to convert a vector containing a sample of uniform variates \( u_i \) to an antithetic sample, \( 1 - u_i \). Several procedures have versions (not listed in the table) that take and return Long arguments instead of Doubles. Finally, \( \text{RescaleToZeroMean}() \) subtracts a constant from each element of \( a \) so that \( a \) then has zero mean. It returns the value of the constant.

**LibSobol**

This has a single procedure,

\[
\text{Public Sub sobseq}(n\_dim As Long, ByRef X() As Double)
\]

that generates successive Sobol’ sequence numbers. It is a transcription from C into VBA of a *Numerical Recipes* procedure.
### Table C.6 LibStats: densities and distributions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>normal_cdf(X As Double) As Double</code></td>
<td>Normal distribution</td>
</tr>
<tr>
<td><code>n_probs(z As Double) As Double</code></td>
<td>Normal distribution (accurate)</td>
</tr>
<tr>
<td><code>Cody_erf(X As Double, jint As Integer) As Double</code></td>
<td>Error function due to Cody</td>
</tr>
<tr>
<td><code>cndev(u As Double) As Double</code></td>
<td>Inverse of normal dist. (Moro)</td>
</tr>
<tr>
<td><code>normal_inverse(u As Double) As Double</code></td>
<td>Inverse of normal dist. (Acklam)</td>
</tr>
<tr>
<td><code>Weibull_dist(x As Double, a As Double) As Double</code></td>
<td>Weibull distribution</td>
</tr>
<tr>
<td><code>Weibull_inv_dist(u As Double, a As Double) As Double</code></td>
<td>Inverse of Weibull dist.</td>
</tr>
<tr>
<td><code>GEV_dist(x As Double, a As Double) As Double</code></td>
<td>GEV distribution</td>
</tr>
<tr>
<td><code>GEV_inv_dist(u As Double, mu As Double, sig As Double, a As Double) As Double</code></td>
<td>Inverse of GEV dist.</td>
</tr>
<tr>
<td><code>Chi_squ_dist(x As Double, a As Double) As Double</code></td>
<td>Chi-squared distribution</td>
</tr>
<tr>
<td><code>gammp(a As Double, x As Double) As Double</code></td>
<td>NR, incomplete gamma function</td>
</tr>
<tr>
<td><code>gammq(a As Double, x As Double) As Double</code></td>
<td>NR, incomplete gamma function</td>
</tr>
<tr>
<td><code>gser(ByRef gamser As Double, a As Double, x As Double, ByRef gln As Double)</code></td>
<td>NR, incomplete gamma function</td>
</tr>
<tr>
<td><code>gcf(ByRef gammcf As Double, a As Double, x As Double, ByRef gln As Double)</code></td>
<td>NR, incomplete gamma function</td>
</tr>
<tr>
<td><code>gammln(X As Double) As Double</code></td>
<td>NR, ln of the gamma function</td>
</tr>
<tr>
<td><code>gamma_inverse(a As Double, p As Double, q As Double, ByRef ierr As Long) As Double</code></td>
<td>Inverse of gamma function</td>
</tr>
<tr>
<td><code>t_dist(x As Double, a As Double) As Double</code></td>
<td>t distribution</td>
</tr>
<tr>
<td><code>t_inv_dist(U As Double, a As Long) As Double</code></td>
<td>Inverse of t dist.</td>
</tr>
<tr>
<td><code>bi_variate_t_dist(dh As Double, dk As Double, r As Double, nu As Long) As Double</code></td>
<td>Bivariate t distribution</td>
</tr>
<tr>
<td><code>beta(xx As Double, ww As Double) As Double</code></td>
<td>NR, beta function</td>
</tr>
<tr>
<td><code>betafcf(x As Double, a As Double, b As Double) As Double</code></td>
<td>NR, incomplete beta function</td>
</tr>
<tr>
<td><code>beta inverse(yy0 As Double, aa As Double, bb As Double) As Double</code></td>
<td>Inverse of beta function</td>
</tr>
<tr>
<td><code>bi_variate_n_dist(x_lim As Double, y_lim As Double, rho As Double) As Double</code></td>
<td>Bivariate normal distribution</td>
</tr>
<tr>
<td><code>bi_variate_n_dist_upper(SK As Double, SK As Double, r As Double) As Double</code></td>
<td>Bivariate normal upper dist.</td>
</tr>
<tr>
<td><code>Cauchy_one_sided(x As Double, c As Double) As Double</code></td>
<td>Cauchy 1-sided density</td>
</tr>
</tbody>
</table>
Table C.7  LibTrig: trigonometric functions

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sec(X As Double) As Double</td>
<td>'Secant</td>
</tr>
<tr>
<td>Csc(X As Double) As Double</td>
<td>'Cosecant</td>
</tr>
<tr>
<td>Cot(X As Double) As Double</td>
<td>'Cotangent</td>
</tr>
<tr>
<td>SinH(X As Double) As Double</td>
<td>'hyperbolic sine</td>
</tr>
<tr>
<td>CosH(X As Double) As Double</td>
<td>'hyperbolic cosine</td>
</tr>
<tr>
<td>TanH(X As Double) As Double</td>
<td>'hyperbolic tangent</td>
</tr>
<tr>
<td>CotH(X As Double) As Double</td>
<td>'hyperbolic cotangent</td>
</tr>
<tr>
<td>SecH(X As Double) As Double</td>
<td>'hyperbolic secant</td>
</tr>
<tr>
<td>CscH(X As Double) As Double</td>
<td>'hyperbolic cosecant</td>
</tr>
<tr>
<td>ASin(X As Double) As Double</td>
<td>'arc sine</td>
</tr>
<tr>
<td>ACos(X As Double) As Double</td>
<td>'arc cosine</td>
</tr>
<tr>
<td>ACot(X As Double) As Double</td>
<td>'arc cotangent</td>
</tr>
<tr>
<td>ASec(X As Double) As Double</td>
<td>'arc secant</td>
</tr>
<tr>
<td>ACsc(X As Double) As Double</td>
<td>'arc cosecant</td>
</tr>
<tr>
<td>ASinH(X As Double) As Double</td>
<td>'hyperbolic arc sine</td>
</tr>
<tr>
<td>ACosH(X As Double) As Double</td>
<td>'hyperbolic arc cosine</td>
</tr>
<tr>
<td>ATanH(X As Double) As Double</td>
<td>'hyperbolic arc tangent</td>
</tr>
<tr>
<td>ACotH(X As Double) As Double</td>
<td>'hyperbolic arc cotangent</td>
</tr>
<tr>
<td>ASecH(X As Double) As Double</td>
<td>'hyperbolic arc secant</td>
</tr>
<tr>
<td>ACscH(X As Double) As Double</td>
<td>'hyperbolic arc cosecant</td>
</tr>
</tbody>
</table>

Other trigonometric functions

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>my_atan2(y As Double, X As Double) As Double</td>
<td>'Two argument atan()</td>
</tr>
</tbody>
</table>

LibFactory

This is a set of five casts, given in Table C.9, needed by applications that use the polymorphic factory. They assume that the interface types have been declared. This library should not be included in applications which do not use the factory since (i) it is not needed and (ii) the types it uses will not be recognized.

LibCopula

A number of copula distribution functions are contained in LibCopula. They are listed in Table C.10. Only the distribution functions for the Gaussian, student, Clayton, and a number of Archimedean copulas (Pareto, Ali-Mikhail-Haq, Gumbell–Barnett, Gumbel–Hougaard, and Frank) are included.

LibAverageRate

This library has in it implementations of a number of explicit solutions associated with average rate options. Table C.11 lists those implemented. They all assume that the underlying process is a geometric Brownian motion.

CtsGeoAvCall() and CtsGeoAvPut() compute continuously compounded geometric average call and put option values, with no average-to-date being carried. DisGeoAvCall() and DisStnGeoAvPut() value discretely reset geometric average part-way through their lives. The client can specify an average to date, and the time since the last reset date. The option specification is held in a UDT, AvRateUDT. DisStnGeoAvCall() and DisStnGeoAvPut() value the special case of a discretely reset geometric average that has just been issued and is carrying no average-to-date.
Table C.8  LibVec: simple vector functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Function GeometricAverage(a() As Double) As Double</code></td>
<td>Geometric average of a</td>
</tr>
<tr>
<td><code>Function ArithmeticAverage(a() As Double) As Double</code></td>
<td>Arithmetic average of a</td>
</tr>
<tr>
<td><code>Function VecSum(a() As Double) As Double</code></td>
<td>Sum of elements in a</td>
</tr>
<tr>
<td><code>Function VecMax(a() As Double) As Double</code></td>
<td>Maximum of elements in a</td>
</tr>
<tr>
<td><code>Function VecMin(a() As Double) As Double</code></td>
<td>Minimum of elements in a</td>
</tr>
<tr>
<td><code>Function VecMod(a() As Double) As Double</code></td>
<td>Modulus of a</td>
</tr>
<tr>
<td><code>Function VecSize(a() As Double) As Long</code></td>
<td>Returns # elements in a</td>
</tr>
<tr>
<td><code>Function InnerProduct(a() As Double, b() As Double) As Double</code></td>
<td>Inner product of a and b</td>
</tr>
<tr>
<td><code>Function RMSE(a() As Double, b() As Double) As Double</code></td>
<td>Root mean square error of a and b</td>
</tr>
<tr>
<td><code>Function StandardDeviation(a() As Double) As Double</code></td>
<td>Standard deviation of a</td>
</tr>
<tr>
<td><code>Function Correlation(a() As Double, b() As Double) As Double</code></td>
<td>Correlation between a and b</td>
</tr>
<tr>
<td><code>Sub NegateVec(ByRef a() As Double)</code></td>
<td>Toggles sign of elements in a</td>
</tr>
<tr>
<td><code>Sub CompVec(U As Double, ByRef a() As Double)</code></td>
<td>Complement of a() from U</td>
</tr>
<tr>
<td><code>Function RescaleToZeroMean(ByRef a() As Double)</code></td>
<td>Normalizes mean of a</td>
</tr>
<tr>
<td><code>Sub MultiplyByScalar(ByRef a() As Double, s As Double)</code></td>
<td>Multiplies a() by s</td>
</tr>
<tr>
<td><code>Sub SetToScalar(ByRef a() As Double, s As Double)</code></td>
<td>Sets a() to s</td>
</tr>
<tr>
<td><code>Sub SetToZero(ByRef a() As Double)</code></td>
<td>Sets a() to zero</td>
</tr>
<tr>
<td><code>Sub AddArrays(ByRef a() As Double, ByRef b() As Double)</code></td>
<td>Sets a(i) to a(i) + b(i)</td>
</tr>
<tr>
<td><code>Sub MinArrays(ByRef a() As Double, ByRef b() As Double)</code></td>
<td>Sets a(i) to min(a(i), b(i))</td>
</tr>
<tr>
<td><code>Sub MaxArrays(ByRef a() As Double, ByRef b() As Double)</code></td>
<td>Sets a(i) to max(a(i), b(i))</td>
</tr>
<tr>
<td><code>Sub SetToBool(ByRef a() As Boolean, s As Boolean)</code></td>
<td>Sets a() to s</td>
</tr>
<tr>
<td><code>Sub MaskSetToScalar(ByRef a() As Double, b() As Boolean, s As Double)</code></td>
<td>If b(i) then sets a(i) to s</td>
</tr>
<tr>
<td><code>Sub NotVec(ByRef a() As Boolean)</code></td>
<td>Sets a(i) to Not a(i)</td>
</tr>
<tr>
<td><code>Function VecAnd(a() As Boolean) As Boolean</code></td>
<td>Returns True if every a(i) is True</td>
</tr>
<tr>
<td><code>Sub MakeVecStep(x() As Double, lb As Long, ub As Long, x_min As Double, h As Double)</code></td>
<td>Makes x: starts at x_min, steps of h</td>
</tr>
<tr>
<td><code>Sub GetRow(A() As Double, b() As Double, j As Long)</code></td>
<td>Returns jth row of A in b</td>
</tr>
</tbody>
</table>
Table C.9  LibFactory: Casts required by the polymorphic factory

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CastSyn(obj As ISyn) As ISyn</td>
<td>'Casts to ISyn</td>
</tr>
<tr>
<td>CastPara(obj As IPara) As IPara</td>
<td>'Casts to IPara</td>
</tr>
<tr>
<td>CastReusable(obj As IReusable) As IReusable</td>
<td>'Casts to IReusable</td>
</tr>
<tr>
<td>CastCreatable(obj As ICreatable) As ICreatable</td>
<td>'Casts to ICreatable</td>
</tr>
</tbody>
</table>

Table C.10  LibCopula: copula distribution functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gaussian_copula(rho As Double, u1 As Double, u2 As Double) As Double</td>
<td></td>
</tr>
<tr>
<td>Student_copula(dof As Double, rho As Double, u1 As Double, u2 As Double) As Double</td>
<td></td>
</tr>
<tr>
<td>Independent_copula(u1 As Double, u2 As Double) As Double</td>
<td></td>
</tr>
<tr>
<td>Comonotonic_copula(u1 As Double, u2 As Double) As Double</td>
<td></td>
</tr>
<tr>
<td>Countermonotonic_copula(u1 As Double, u2 As Double) As Double</td>
<td></td>
</tr>
<tr>
<td>Clayton_copula(t_1 As Double, u1 As Double, u2 As Double) As Double</td>
<td></td>
</tr>
<tr>
<td>Arch_Pareto_copula(t_1 As Double, u1 As Double, u2 As Double) As Double</td>
<td></td>
</tr>
<tr>
<td>Arch_Pareto_generator(t_1 As Double, U As Double) As Double</td>
<td></td>
</tr>
<tr>
<td>Arch_AMH_copula(t_1 As Double, u1 As Double, u2 As Double) As Double</td>
<td></td>
</tr>
<tr>
<td>Arch_AMH_generator(t_1 As Double, U As Double) As Double</td>
<td></td>
</tr>
<tr>
<td>Arch_GH_copula(t_1 As Double, u1 As Double, u2 As Double) As Double</td>
<td></td>
</tr>
<tr>
<td>Arch_GH_generator(t_1 As Double, U As Double) As Double</td>
<td></td>
</tr>
<tr>
<td>Arch_GH_gen_cond_inv(t_1 As Double, U As Double) As Double</td>
<td></td>
</tr>
<tr>
<td>Arch_Frank_copula(t_1 As Double, u1 As Double, u2 As Double) As Double</td>
<td></td>
</tr>
<tr>
<td>Arch_Frank_generator(t_1 As Double, U As Double) As Double</td>
<td></td>
</tr>
<tr>
<td>Arch_Frank_gen_cond_inv(t_1 As Double, U As Double) As Double</td>
<td></td>
</tr>
<tr>
<td>Arch_GB_copula(t_1 As Double, u1 As Double, u2 As Double) As Double</td>
<td></td>
</tr>
<tr>
<td>Arch_GB_generator(t_1 As Double, U As Double) As Double</td>
<td></td>
</tr>
</tbody>
</table>

The module contains two helper Functions, BScall() and BSput(), that compute the standard Black–Scholes vanilla option values.

C.2 THE COMPLEX NUMBER OBJECT

The spreadsheet complex.xls contains two modules: a class module, complex, that defines a complex number object, and a standard module, ComplexHelpers, that contains a number of useful functions taking complex numbers as arguments. The methods and non-member library procedures are listed in Table C.12.

Since VBA cannot overload operators the syntax for using the complex object can become convoluted. An alternative to creating a complex object is just to use a UDT containing a pair of Doubles representing the real and imaginary parts, and to operate on the UDT entirely with library procedures. This is the approach taken by Rouah and Vainberg (2007), for instance. The advantage of using a UDT is that calculation is fast; the disadvantage is that expressions can get very unwieldy.

In the implementation of the complex object many methods return temporary instantiations. This risks efficiencies if the methods are not used in an application as intended. However fast an implementation, if used poorly in an application all that speed can be squandered: understand the implementation; precompute as much as possible; avoid creating temporaries.

It is hoped that the approach taken here is a reasonable compromise between speed and readability, but in speed-critical applications the UDT approach should be used.
Table C.11  LibAverageRate: Discrete average rate option valuation

Function CtsGeoAvCall(S0 As Double, r As Double, div As Double, sig As Double, T As Double, X As Double) As Double
Function CtsGeoAvPut(S0 As Double, r As Double, div As Double, sig As Double, T As Double, X As Double) As Double
Function DisGeoAvCall(data As AvRateUDT) As Double
Function DisGeoAvPut(data As AvRateUDT) As Double
Function DisStnGeoAvCall(N As Long, dt As Double, X As Double, S0 As Double, sig As Double, r As Double) As Double
Function DisStnGeoAvPut(N As Long, dt As Double, X As Double, S0 As Double, sig As Double, r As Double) As Double
Function BScall(X As Double, T As Double, S0 As Double, r As Double, sig As Double, div As Double) As Double
Function BSput(X As Double, T As Double, S0 As Double, r As Double, sig As Double, div As Double) As Double
Function BSdigital(X As Double, T As Double, S0 As Double, r As Double, sig As Double, div As Double) As Double
Function BScalldelta(X As Double, T As Double, S0 As Double, r As Double, sig As Double, div As Double) As Double
Function BSputdelta(X As Double, T As Double, S0 As Double, r As Double, sig As Double, div As Double) As Double
Table C.12 Methods and procedures for the Complex object

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Property Get Re() As Double</td>
<td>'Getter for real part</td>
</tr>
<tr>
<td>Property Get Im() As Double</td>
<td>'Getter for imaginary part</td>
</tr>
<tr>
<td>Property Let Re(a As Double)</td>
<td>'Setter for real part</td>
</tr>
<tr>
<td>Property Let Im(a As Double)</td>
<td>'Setter for imaginary part</td>
</tr>
<tr>
<td>Sub make(a As Double, b As Double)</td>
<td>'Sets the real and imaginary parts</td>
</tr>
<tr>
<td>Function conj() As complex</td>
<td>'Returns the complex conjugate</td>
</tr>
<tr>
<td>Function neg() As complex</td>
<td>'Returns the negative of Me</td>
</tr>
<tr>
<td>Function norm() As Double</td>
<td>'Returns the norm of Me</td>
</tr>
<tr>
<td>Function add(a As complex) As complex</td>
<td>'adds a to Me and returns</td>
</tr>
<tr>
<td>Function subtract(a As complex) As complex</td>
<td>'subtracts a from Me and returns</td>
</tr>
<tr>
<td>Function multiply(a As complex) As complex</td>
<td>'multiplies a by Me and returns</td>
</tr>
<tr>
<td>Function divide(a As complex) As complex</td>
<td>'divides Me by a and returns</td>
</tr>
<tr>
<td>Function Pow(N As Double) As complex</td>
<td>'computes and returns Me^N, N Double</td>
</tr>
<tr>
<td>Function Powc(a As complex) As complex</td>
<td>'computes and returns Me^a, a complex</td>
</tr>
<tr>
<td>Function cprint() As String</td>
<td>'returns a String representation of Me</td>
</tr>
</tbody>
</table>

The Complex object: non-member Library procedures

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function cmake(a As Double, b As Double) As complex</td>
<td>'Constructs a new complex</td>
</tr>
<tr>
<td>Function Sqrt(a As complex) As complex</td>
<td>'Returns the square root of a</td>
</tr>
<tr>
<td>Function Sq(a As complex) As complex</td>
<td>'Returns the square of a</td>
</tr>
<tr>
<td>Function cExp(a As complex) As complex</td>
<td>'Returns the exponential of a</td>
</tr>
<tr>
<td>Function CLn(a As complex) As complex</td>
<td>'Returns the log of a</td>
</tr>
</tbody>
</table>

C.3 QUADRATURE

The spreadsheet quadrature.xls contains a set of three integration objects, IntegratorTrapezium, IntegratorExtended and IntegratorSimpson, belonging to the IIntegrator hierarchy. They integrate, using the trapezium, extended trapezium, and Simpson methods, integrands conforming to the IIntegrand interface.

Given a function \( f(x) \) the aim is to compute the integral

\[
F_{a,b} = \int_a^b f(x) \, dx. \tag{C.4}
\]

Each method constructs a set of points \( a = x_0 < x_1 < \ldots < x_N < x_{N+1} = b \), computes \( f_i = f(x_i) \), \( i = 0, \ldots, N + 1 \), and then constructs an approximation \( \tilde{F}_{a,b} \) to \( F_{a,b} \) as

\[
\tilde{F}_{a,b} = \sum_{i=0}^{N+1} w_i f_i \tag{C.5}
\]

for some set of weights \( w = \{w_i\}_{i=0,\ldots,N+1} \).

For the three methods implemented here the grid points \( \{x_i\}_{i=0,\ldots,N+1} \) are equidistant with constant spacing \( x_{i+1} - x_i = h \) for all \( i \) and we write

\[
\tilde{F}_{a,b} = \frac{1}{h} \sum_{i=0}^{N+1} \tilde{w}_i f_i \tag{C.6}
\]
Private integrand_ As IIntegrand
Private integrator_ As IIntegrator
Private val_ As Double

Private Sub Class_Terminate()
Set integrand_ = Nothing
Set integrator_ = Nothing
End Sub

Friend Sub IApp_SetValues(ByRef data As InputManager)
Set integrand_ = data.GetIntegrand
Set integrator_ = data.GetIntegrator
End Sub

Friend Sub IApp_reset(): End Sub 'stub
Friend Property Get IApp_val() As Double: IApp_val = val_: End Property

Friend Sub IApp_run()
val_ = integrator_.integrate(integrand_)
End Sub

Figure C.1 Using the integrator objects

for normalized weights $\tilde{w} = \{\tilde{w}_i\}_{i=0,\ldots,N+1}$. The weights $\tilde{w}$ are

\[
\begin{align*}
\text{Trapezium:} & \\
\tilde{w}_0 &= \tilde{w}_{N+1} = \frac{1}{2}, & i &= 0, N + 1 \\
\tilde{w}_i &= 1, & i &= 1, \ldots, N.
\end{align*}
\]

\[
\begin{align*}
\text{Extended Trapezium:} & \\
\tilde{w}_0 &= \tilde{w}_{N+1} = \frac{5}{12}, & i &= 0, N + 1 \\
\tilde{w}_1 &= \tilde{w}_{N} = \frac{13}{12}, & i &= 1, N \\
\tilde{w}_i &= 1, & i &= 2, \ldots, N - 1.
\end{align*}
\]

\[
\begin{align*}
\text{Simpson (N odd):} & \\
\tilde{w}_0 &= \tilde{w}_{N+1} = \frac{1}{3}, & i &= 0, N + 1 \\
\tilde{w}_1 &= \frac{4}{3}, & i &= 1, 3, \ldots, N - 2, N \\
\tilde{w}_i &= \frac{2}{3}, & i &= 2, 4, \ldots, N - 3, N - 1.
\end{align*}
\]  \hspace{1cm} (C.7)

For an explanation of these weights see any numerical methods book (for instance, Burden and Faires (2004), Press et al. (2007), and many others).

Figure C.1 give an example of using the integrator objects. The App_quad_example object instantiates an IIntegrand and an IIntegrator object. It then asks the integrator to integrate the integrand. That is all there is to it.

To be able to integrate a function it has to be put into an object obeying the IIntegrand interface. An example is the $f1$ object shown in Figure C.2. There are two procedures in the interface.

\[
\text{IIntegrand}_f(x \text{ As Double}) \hspace{1cm} (C.8)
\]

returns a value of the integrand at the $x$-value $x$;

\[
\text{IIntegrand}_fvec(ByRef fvals() \text{ As Double}, u() \text{ As Double}) \hspace{1cm} (C.9)
\]
fills the array \( fvals \) with values of \( f() \) computed at \( x \)-values in the array \( u \).

An alternative syntax is possible. Instead of instantiating and destroying an integration object the way that \texttt{App_quad_example} does, clients can “pop” an integration with

\[
\text{val}_\_ = \text{CastIIntegrator}(\text{New IntegratorExtended}) \_ \\
\_ .\text{WithVals}(2, 12, 1001) .\text{integrate}(\text{integrand}_\_) \_ .
\]

(C.10)

In a single line this instantiates a \texttt{New IntegratorExtended} object, casts it to \texttt{IIntegrator}, sets its parameter values (the integration bounds and the number of steps) on the fly, and asks it to integrate \texttt{integrand}_\_, returning the value it computes. The \texttt{IntegratorExtended} object is temporary and is destroyed when it goes out of scope when the line finishes execution. This works because the \texttt{WithVals()} method returns a reference to \texttt{Me}; \texttt{CastIIntegrator()} is a standard casting procedure of a type discussed elsewhere.
Appendix D

Running DLLs from VBA

DLLs (dynamic-link libraries) are portable libraries that can be created by one application and used by another. In particular one can create a DLL in C++ then have its functions run from Excel by calling them from VBA.

Creating and using DLLs is not too difficult. For the purpose of this appendix we give only an example, with commentary. Much more detail can be found in Kimmel et al. (2004) who also discuss using the windows API.

Example

The spreadsheet dll_test.xls calls some functions in a DLL, testdll.dll. VBA declares and calls the DLL functions, passing them some arrays. The DLL functions do some manipulation with the arrays, and pass them back to VBA where they are then printed out.

Individual DLL functions have to be declared (as DLL functions) in VBA. The declaration statement gives the path to the DLL and the signature of the DLL function. The signature must be the VBA version of the signature of the original function (assumed here to have originated in C++).

Great care must be taken to specify correctly the argument types in both VBA and C++. If arguments (in VBA) are Double, Integer or String, either ByVal or ByRef, then in C++ they should be double or double *, int or int *, or char or char *.

Table D.1 gives the correspondence of these types between the C++ and VBA declarations. Generally an argument declared in VBA to be a ByVal primitive will correspond to the same primitive in C++. An argument declared ByRef corresponds in C++ to a pointer to that primitive.

To pass an array of Doubles to a DLL function the argument must be declared in VBA as

\[\text{ByRef x As Double} \quad \text{(D.1)}\]

and in C++ the DLL function would have a corresponding argument declared as double * x. When called in VBA the first element of the array is passed (ByRef) to the DLL function. When passing a String it must be declared in VBA to be fixed length and passed ByVal.

The signatures of the functions in the DLL testdll.dll, when they were created in C++, were

\[
\begin{align*}
\text{double Square1(double n, double * out); } & \quad \text{//a} \\
\text{void Square2(int n, double * out);} & \quad \text{//b} \\
\text{void Square3(int n, char * out);} & \quad \text{//c} \\
\text{void Square4(int * n);} & \quad \text{//d.}
\end{align*}
\]

(D.2)

These functions do weird things, but their signatures are the things of importance here. It so happens that Square1() assumes, without proof, that out is an array of (at least) 12 doubles; Square2() that out is an array of 7 doubles; and Square3() that out is an array of 10 chars. All four functions return results in their pointer arguments; Square1(), in addition, explicitly returns a double.
Table D.1  Correspondence of the data types

<table>
<thead>
<tr>
<th>To pass:</th>
<th>Declare in VBA</th>
<th>Declare in C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>A Double</td>
<td>ByVal x As Double</td>
<td>double x</td>
</tr>
<tr>
<td></td>
<td>ByRef x As Double</td>
<td>double * x</td>
</tr>
<tr>
<td>An Integer</td>
<td>ByVal x As Integer</td>
<td>int x</td>
</tr>
<tr>
<td></td>
<td>ByRef x As Integer</td>
<td>int * x</td>
</tr>
<tr>
<td>Array of Doubles</td>
<td>ByRef x As Double</td>
<td>double * x</td>
</tr>
<tr>
<td>A String</td>
<td>ByVal s As String</td>
<td>char * s</td>
</tr>
</tbody>
</table>

In VBA these functions are referred to by the names test1() to test4() with signatures

\[
\begin{align*}
\text{Function test1(ByVal n As Double, ByRef out As Double) As Double} & \quad \text{'}a' \\
\text{Sub test2(ByVal n As Integer, ByRef out As Double)} & \quad \text{'}b' \\
\text{Sub test3(ByVal n As Integer, ByVal out As String)} & \quad \text{'}c' \\
\text{Sub test4(ByRef n As Integer)} & \quad \text{'}d'.
\end{align*}
\]

(D.3)

The String argument in test3() is passed ByVal whereas the arrays of Doubles are all passed ByRef.

It is often convenient to define a wrapper object in VBA to mediate access to the DLL. This is not
strictly necessary, and certainly not required in the example, but it is good practice.

The wrapper object, DLLwrapperclass, is given in Figure D.1. It serves three purposes.

1. It tidies away the declarations of the DLL functions.
2. It hides the names of the DLL functions.
3. It can act as a façade, adapting the DLL functions’ interfaces to the needs of the application.

The syntax of the declaration statement for test1() in Figure D.1 is repeated in line (D.4):

\[
\begin{align*}
\text{Private Declare Function test1 Lib "D:\\testdll.dll" _} \\
\text{Alias "Square1" (ByVal n As Double, ByRef out As Double) As Double}. \\
\end{align*}
\]

(D.4)
test1() is the name the DLL function will go under in VBA. The full path to the DLL is given (and of
course must be given correctly).

The Alias clause is optional. Often the name declared in VBA is the same as the name used inside
the DLL; only use Alias if the name declared (test1) is different to the name the function goes by
(Square1) in the DLL. Being able to give an Alias makes it easy to avoid name clashes.

The signature is mandatory; empty brackets should be supplied if the function takes no arguments.

An example of using the wrapper class and the DLL it manages is given by the procedure CPP_DLL(),
illustrated in Figure D.2. Two fixed length arrays of Doubles and a fixed length String are declared. Their
sizes are greater than that assumed by the DLL functions, so operations performed on them will not stomp
on memory. The first element of the arrays of Doubles is passed by reference to the DLLwrapperclass
methods. This address is all that the DLL functions need.

There is telepathy between the DLL functions and the wrapper class. DLLwrapperclass has to know
exactly what the DLL functions are expecting to receive. If they do not get what they are expecting then
the code will blow up, taking Excel with it.
Appendix D: Running DLLs from VBA

Figure D.1 The DLLwrapperclass object

Figure D.2 The CPP_DLL() procedure

A VBA array knows its own size, but this information is lost when it – or rather when a reference to its initial element – is passed to the DLL. For safety the size of the array should also be passed to the DLL function.
This page intentionally left blank
Appendix E

Object-Oriented Programming

The purpose of this appendix is to provide some background material on objects and object-oriented programming. Much of it is general and not specific to VBA but all of it is relevant to understanding objects in VBA. It is intended that this appendix supports the object-oriented material presented particularly in Chapters 5 and 6.

Objects are user-defined types that

1. contain data defining the state of the object and
2. have procedures that act on and with the data.

Object-oriented concepts were introduced into programming languages as early as the 1960s (with Simula 67) but really became popular only in the 1990s with the widespread use of C++.

It is natural to have objects in a programming language. There are several independent motivations all pushing towards using objects.

- **Theoretical.** They are a natural extension of the notion of a type.
- **Developmental.** They are a natural development of the functionality embodied in a standard code module.
- **Practical.** They facilitate simpler design and cheaper maintenance in large projects.
- **Managerial.** They facilitate the management of teams of programmers on large projects.

Once you have objects, you find that it is relatively easy to extend their power enormously by letting different objects – sharing methods with the same names but doing different things – substitute for one another. You end up with code whose functionality can be toggled simply by passing it one object instead of another. This ability for the same code to do more than one thing, depending on the types of objects passed to it, is called polymorphism.

Objects have an associated set of concepts. Some of these are implementable in VBA, and some are not. The chief concepts are discussed here. First we provide a little motivation for having objects in our code, then in section E.2 we investigate what properties objects might possess, discussing some of the issues accompanying the introduction of user definable types with methods and at the same time presenting some basic object-oriented concepts. In section E.3 we say a little about how objects are implemented in VBA. Polymorphism is discussed in section E.3.2. Standard ways that objects can interact with one another – design patterns – are looked at in section E.4.

### E.1 MOTIVATION FOR OBJECTS

In Chapter 5 a StopWatch object was introduced. This replaced the yukky level 0 style timer used previously in Chapter 3. In this section we compare different ways of implementing the functionality of a stopwatch. The basic problem is given in Figure E.1. The procedure NakedMain() times the execution of two Subs, SubA() and SubC(). The timer is switched on for the execution of these two Subs and off for the execution of SubB() and SubD(). total_t accumulates the total time; partial_t is the time taken by individual pieces of code.

The disadvantages of the yukky-style timer are that it is intrusive: it introduces new variables into client code that have no right to be there. It is not re-usable: you have to add in the lines that accumulate the
time. It is very inconvenient to code in this style if more than one timer is needed at the same time; each timer requires its own set of variables.

A simple way of hiding the variables `total_t` and `partial_t` is to have them as `Static` variables inside a procedure. This might lead to the abominable code shown in Figure E.2. The `Function` `MonolithicTimerFn()` is passed an argument that toggles its functionality. An argument of 1 resets the timer, 2 starts it, 3 stops it, and 4 causes it to return the elapsed time. Classically, procedures of this sort would also return error codes, perhaps as arguments passed `ByRef`.

This code is really horrible. How is a client supposed to know what the various toggles do? Somewhere there has to be a reference manual, consulted every time some poor client has to look at `MonolithicMain()`. Also, only one timer can be active at a time. The procedure could no doubt be “extended” to enable it to start and stop more than one timer at a time, but it will be horrendous and error prone.

A significant and real improvement is the code shown in Figure E.3. The idea here is to declare the variables `total_t` and `partial_t` as `Private` global variables in a separate (standard) module, `ModuleTimer`. In the module are `Public` procedures that act with and on the data.

The procedure `ModuleMain()` is much clearer. The code numbers that desecrated `MonolithicMain()` are gone, replaced by clear function calls. This programming style would have been the preferred method in pre-OOP days. A C-style program may have done things this way. Note that here, unlike other places in the book,1 I have explicitly qualified procedure calls by the name of the module, writing `ModuleTimer.Start_Timer` instead of plain `Start_Timer`. I discuss this later.

This is almost a solution. The sole remaining difficulty is that, like `MonolithicTimerFn()`, only one timer can be active at one time. The actual solution is to replace the module by a full blown type. This enables one to write application code like that in Figure E.4. An object, `watch`, of type `StopWatch` is declared. Its methods are called to start and stop the timer, and to extract the elapsed time at the end.

Compare `ObjectMain()` with `NakedMain()`, `MonolithicMain()`, and `ModuleMain()`. `ObjectMain()` has all the advantages of `ModuleMain()` but in addition more than one stopwatch can

---

1 Perhaps I should have been following this practice somewhat more in those other places.
Private Function MonolithicMain(N As Long) As Double
    Call MonolithicTimerFn(1) 'reset the timer
    Dim i As Long
    For i = 1 To N
        Call MonolithicTimerFn(2) 'start the timer
        Call SubA
        Call MonolithicTimerFn(3) 'stop the timer
        Call SubB
        Call MonolithicTimerFn(2) 'start the timer
        Call SubC
        Call MonolithicTimerFn(3) 'stop the timer
        Call SubD
    Next i
    MonolithicMain = MonolithicTimerFn(4) 'fetch the time
End Function

Public Function MonolithicTimerFn(acode As Long) As Double
    Static NOT_FIRST_TIME As Boolean
    If Not NOT_FIRST_TIME Then
        Static total_t As Double: total_t = 0
        Static partial_t As Double: partial_t = 0
        Static Timer_running As Boolean: Timer_running = False
        Static Extra_time As Double: Extra_time = 0#
        NOT_FIRST_TIME = True
    End If
    MonolithicTimerFn = 0#
    Select Case acode
        Case 1: 'reset the timer
            total_t = 0: partial_t = 0: Timer_running = False
        Case 2: 'start the timer
            If Not Timer_running Then
                partial_t = timer
                Timer_running = True
            End If
        Case 3: 'stop the timer
            If Timer_running Then
                total_t = total_t + timer - partial_t
                Timer_running = False
            End If
        Case 4: 'fetch the time
            Extra_time = 0#
            If Timer_running Then Extra_time = timer - partial_t
            MonolithicTimerFn = total_t + Extra_time
    End Select
End Function

Figure E.2 MonolithicTimerFn(): uses a poor monolithic timer

be declared, be in existence, and be running at the same time. Each StopWatch maintains its own data, private and encapsulated.

The StopWatch object is highly re-usable. The client does not have to know anything about what goes on inside it; you just create a StopWatch and call the methods it exposes.

Objects can be regarded as extending the role of standard modules like ModuleTimer by letting the module define a type, and allowing the programmer to create instances of that type to execute the procedures
in the module. On the one hand, this can be regarded as a minor but convenient development of a standard module; on the other hand, it can be seen as a radical innovation, a qualitative ratchet-up in capability.

Meanwhile standard code modules can be regarded as proto-objects, that is, as quasi-objects not quite fully formed. (We see below that they behave a little like singleton objects.)

Figure E.3 ModuleMain(): uses a pre-OOP module implementation
Private Function ObjectMain(N As Long) As Double
  Dim watch As StopWatch: Set watch = New StopWatch
  Call watch.Reset_Timer 'reset the timer
  Dim i As Long
  For i = 1 To N
    Call watch.Start_Timer 'start the timer
    Call SubA
    Call watch.Stop_Timer 'stop the timer
    Call SubB
    Call watch.Start_Timer 'start the timer
    Call SubC
    Call watch.Stop_Timer 'stop the timer
    Call SubD
  Next i
  ObjectMain = Get_Elapsed_Time() 'fetch the time
  Set watch = Nothing
End Function

Figure E.4 ObjectMain(): uses an OOP-style timer

E.2 PROPERTIES OF OBJECTS

If you are going to define your own types, and to declare instances of them, what properties should they have? One can be guided by the properties of other types. We compare three categories of types: primitive types (Longs and Doubles, *et cetera*); extensions which, like UDTs, are already in the language; and proto-types like standard modules.

There are three categories of properties to consider: structural, conversion and functional. Structural properties concern object creation, assignment, copying and destruction; conversion is the ability to convert or cast an object into an object of another type; and functional properties concern how an object is used: the actions an object can take or which can be taken with an object.

Structural properties

Examples of creation, assignment and copying with primitives, here Doubles, are given in lines (E.1).

\[
\begin{align*}
\text{Dim } a \text{ As Double}, b \text{ As Double} & \quad \text{‘a. Default constructor,} \\
\text{a} & = 4.5 \quad \text{‘b. Assignment,} \\
b & = a \quad \text{‘c. Copy assignment,} \\
\text{Dim } d \text{ As Double} & = 4.5 \quad \text{‘d. Initialization, } \text{Illegal} \\
\text{Dim } d \text{ As Double} & = a \quad \text{‘e. Copy constructor. } \text{Illegal}
\end{align*}
\]

(E.1)

Primitives can be declared and assigned default value (line (E.1a)), assigned values (line (E.1b)) and copied with assignment to another variable of the same type (line (E.1c)).

Every instance of a type has a state.\(^2\) This is the set of values of its data members. For a primitive there is a single (scalar) value; for a UDT a set of values; for a module the state includes the values of globals

\(^2\) Even if it happens to be empty. Functor-like objects may be stateless.
and any Static variables held in Functions and Subs. Primitives, UDTs and data members in modules are all assigned default values when they are created. This is called default construction. The state can be changed later when items contributing to it are assigned new values.

The state of primitives and UDTs is Public. Anyone can access the value(s) that contribute to the state to inspect or change them. Modules can have Private data. Once data is allowed to be Private all data should be Private with access to it controlled by special-purpose getter and setter Functions. Data is hidden inside the module, encapsulated, safe.

In VBA, primitives cannot be initialized at the point of creation with values chosen by a client (unless they are constants, declared as Const). It is possible to initialize neither with a literal value (line (E.1d)) nor with a value copied from a primitive of the same (or different for that matter) type (line (E.1e)). Initializing an instance as a copy of a variable of the same type is called copy construction. Copy construction is allowed in C++, but is not generally permitted in VBA.

Nor can the programmer specify any code to be run at the point when a primitive is created or destroyed. You cannot destroy a primitive except by letting it pass out of scope.

Note that in contrast to line (E.1e) the code

```vba
Const d As Double = 4.5 'Legal
```

is perfectly legal; how else could you initialize a constant?

Restrictions very similar to those for primitives also apply to UDTs. UDTs require a definition of the type, and assignment of values to a UDT is one by one and not all at once, but otherwise the same rules apply.

Modules are not declarable but effectively have an automatic single instantiation. Although you can have many instances of a primitive type or UDT there is effectively exactly one instance, the same instance, of a module in existence at any time. At start-up any data members (global variables with module scope) are given default values. It is not possible to run any code associated with the module unless it is called from elsewhere; no code can be run automatically to initialize the module. For instance, hand-crafted code could be put into Workbook_Open() to call setters in the module, but this is external to the module.

**Conversion**

VBA can implicitly convert between primitive numeric types. Lines (E.3) illustrate implicit casting from a Double to a Long. The assignment on lines (E.3b) and (E.3c) are legal.

```vba
Dim a As Double, c As Long 'a, Default constructor,
a = 4.5 'b. Assignment,
c = a 'c. Conversion,
Dim d As Long = a 'd. Conversion constructor. Illegal
```

Some explicit conversion is possible using built-in conversion functions (like CDb1()) but, apart from these, conversion is possible only with user-defined procedures. It is illegal in VBA to construct an instantiation of a variable as a copy of a variable of a different type. Line (E.3d) does not compile. The process attempted by line (E.3d) is called conversion construction; it is permitted in C++ but not in VBA.

**Functional**

Functional properties include being able to apply operators and built-in functions to variables, and being able to pass them as arguments or return types for procedures.
Primitives can be passed as argument to Functions ByRef or ByVal, and only primitives can be passed as arguments to built-in VBA procedures (like Exp()). Only primitives work with operators; you can operate on them with operators (such as + and *). In VBA you can add two Doubles but you cannot extend the definition of + to work with UDTs or objects.

Modules have procedures. These can be Public or Private. Public procedures can be called by clients external to the module. They define the module’s interface. Private procedures are internal to the module; they cannot be seen outside the module. They define the module’s implementation.

A comparison of object properties

Table E.1 compares primitive types with UDTs and standard modules. It also shows what you can do with objects in VBA and C++.

Properties are summarized under four headings. Three properties – structural, conversion and functional – have been discussed above. The fourth, extended properties, are additional features possible with objects but not applicable to the other types.

Objects in VBA can do most things that either primitives, UDTs or modules can do. They do not allow conversions, nor can they work with operators, but everything else is possible. Like primitives you can instantiate and assign them; like modules you can have procedures that work on them.

It is possible to define objects that have no methods. These state-only objects are referred to as POD (plain old data). In VBA they are similar to UDTs but are more general. They can be used in situations where a UDT cannot.

The ability to have Private state is very important. It enables, or can enable, encapsulation. VBA objects naturally extend

(1) module-based encapsulation, by allowing multiple objects of the same type;
(2) encapsulation within functions with Staticss, since Class_Initialize() (see below) can circumvent the NOT_FIRST_TIME idiom and values are safely reset when the object is destroyed;
(3) aggregation of data within a UDT, as data in UDTs is Public, and objects can supplement their state with methods.

<table>
<thead>
<tr>
<th></th>
<th>Function Types in VBA</th>
<th>Objects VBA</th>
<th>Objects C++</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Primitives</td>
<td>UDTs</td>
<td>Module</td>
</tr>
<tr>
<td>Structural properties</td>
<td>Initialize (non-default)</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td></td>
<td>Destroy</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td></td>
<td>Copy</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td></td>
<td>Assign</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Private state</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Conversion properties</td>
<td>Implicit conversion</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td></td>
<td>Explicit conversion</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>Operational properties</td>
<td>Work with operators</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td></td>
<td>Can have methods</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td></td>
<td>Pass as arguments</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Have as return types</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Extended properties</td>
<td>Polymorphism</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td></td>
<td>Meta-class data</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>
Objects in C++ are more powerful than those in VBA. The programmer has full control over structural properties, being able to (re-)define copy assignment (by operator overloading), copy constructors, default constructors and copy converters. But beware: programming with objects in C++ is much trickier than in VBA.

E.3 IMPLEMENTING OBJECTS IN VBA

Full descriptions of VBA object capabilities and syntax can be found in sources such as Getz and Gilbert (2001), Green et al. (2007), Kimmel et al. (2004) and Lomax (1998). Chapters 5 and 6 describe in more detail some of the material outlined here.

Objects are defined in class modules rather than standard modules, but the rules are essentially identical to those for a proto-object in a standard module. The name of the object type is just the name of the class module. Procedures are called member functions or methods. When a client calls a member function it is said to be making a request of the object, or sending it a message.

First we look at the basic properties of objects, those that correspond to properties already expressed by primitives, UDTs and modules. Then we look at extended properties, those not found in the three basic type categories, including the most important extension – the ability to be polymorphic.

E.3.1 Implementing basic properties

Objects in VBA are accessible only by reference. A declaration introduces the name of a reference into the program. Initially a newly declared reference refers to Nothing. It must be set in a Set statement to refer either to an existing object or to a new instantiation. Lines (E.4) illustrate.

```vba
Dim ref1 As MyObject, ref2 As MyObject 'a,
Set ref1 = New MyObject 'b,
Set ref2 = ref1 'c.
```

Line (E.4a) declares references (ref1 and ref2) to objects of type MyObject. Line (E.4b) Sets the reference ref1 to point to a new instance of MyObject. The object is instantiated only on line (E.4b) in the Set-New statement. Finally line (E.4c) Sets ref2 to point at the same object as ref1.

Objects are destroyed only when all references to them cease to exist, either by going out of scope, or by being explicitly Set to Nothing.³

It is possible to assign one reference to another, but it is not possible in a single assignment to construct a fresh copy of an object and set the new reference to the new object. Assignment just assigns the reference, not what the reference is pointing to.

To execute a method belonging to the object just call it, qualified by the name of a reference to the object. For instance

```vba
Call ref1.CountToN(10) 'or whatever
```

executes the CountToN() method of (the object pointed to by) ref1. This is a straightforward extension of the syntax used both for modules and for UDTs.

VBA defines a special set of object procedures, called Properties, designed to act as setters and getters. These are useful syntactic sugar.

³ References should always be explicitly Set to Nothing. This is more future-proof.
In VBA structural features are not under programmer control. Objects have two special procedures, 
Class_Initialize() and Class_Terminate(). These are supplied by VBA as stubs that can 
be filled in by the programmer. Class_Initialize() executes when an object is instantiatted and 
Class_Terminate() when it is destroyed. They are not constructors and destructors in the C++
sense – executing them does not cause an object to be created or destroyed – they simply run immediately
after the object is created and immediate before an object is destroyed.

Neither procedure can take arguments. The fact that Class_Initialize() cannot take arguments
means that there is no mechanism available in VBA to permit objects to be created in a state determined
by a client. Copy construction, or any other form of initialization other than default initialization, is not
possible in VBA. There is no way to set the state of an object except through a setter of some sort, and
this breaks encapsulation.

Encapsulation can be preserved only if

1. the state can be set when the object is being constructed or
2. the object can restrict access to setter functions, enabling only certain other objects to call them.

In VBA neither of these two things is possible. Objects in VBA are inherently less safe than those
in C++.

It is not possible to extend the definitions of operators to enable them to have objects as arguments;
opponents work only with primitives. So-called operator overloading is possible in C++, but not in VBA.

E.3.2 Extended properties

There are two properties that none of the three basic VBA types possesses but which objects can have.
These are polymorphism and meta-class data. Polymorphism is possible in VBA but language mechanisms
to allow meta-class data do not exist.

Polymorphism

Polymorphism is the ability of one object to substitute transparently for another in client code, so that the
client does not know exactly which object type it is using. Polymorphism is implemented in VBA through
the use of interfaces. An interface specifies a type and a set of procedures. Another type conforming to
the interface, by Implementing it, has to implement all the specified procedures. It can then substitute
for the interface type in any place where it is used. When client code executes an interface procedure
it instead executes the procedure from the conforming object occupying that slot. Since all the specified
procedures are guaranteed to be present, everything works fine.

Interface objects are simply class modules with procedure stubs only, and perhaps declarations of data
members. As an example, consider writing an interface for a polymorphic functor class. Here a functor
is a function-like object that has only two procedures: run() and SetValues(). An interface class in
its entirety is shown in Figure E.5. This particular example has a run() procedure taking a Long as an
argument and returning a Double. The SetValues() procedure is provided to enable the functor to be
configured if required. Once set up the only method called for the object is run().

Conforming objects Implements the interface class. They must declare all procedures in the interface
class (and Properties for data declarations). Figure E.6 shows a realization as a functor of a function to
compute the value of π. The SetValues() procedure is not required by this particular object and is left
as a stub.

4 A similar example was used in Chapter 11 in an illustration of a polymorphic factory. There it was not implemented as a functor.
Implementing Models of Financial Derivatives

Figure E.5 An interface class for functor objects

```
Option Explicit
Public Sub SetValues(): End Sub
Public Function run(terms As Long) As Double: End Function
' end of file
```

Figure E.6 An object conforming to IFunctor

```
Option Explicit
' Pi_Euler. Conforms to IFunctor
Option Explicit
' Implements IFunctor
Implements IFunctor
Public Sub IFunctor_SetValues(): End Sub 'No implementation required.
Public Function IFunctor_run(terms As Long) As Double
    Dim sum As Double: sum = 0#
    Dim i As Long
    For i = 1 To terms
        sum = sum + (1# / i) / i
    Next i
    IFunctor_run = Sqr(6# * sum)
End Function
' end of file
```

Lines (E.6) illustrate polymorphism in action in client code.

```
Dim funct As IFunctor: Set funct = New Pi_Euler 'a,
Dim ret_value As Double: ret_value = funct.run(terms) 'b.
```

Line (E.6a) declares funct to be an IFunctor. It is Set to be a reference to an IFunctor conforming class, Pi_Euler, but any class that Implements IFunctor would do. Line (E.6b) is polymorphic. It executes funct's run() method without knowledge of which specific IFunctor object is being used.

Interfaces correspond to abstract base classes in C++. In C++ there is a related mechanism called inheritance that allows an object to inherit procedures from another. Objects can conform to more than one interface at a time. This is analogous to multiple inheritance in C++.

Strictly, any primitive type can substitute for another as a Variant. This could be viewed as polymorphism but it would be perverse. There are occasions when the sparing use of Variants is appropriate. Other than those, the use of Variants is a corruption of the type system and an invitation to write expensive and error-prone code.

**Meta-class data**

Meta-class data is data held in common by every instantiation of a type. It belongs to no instantiation in particular. For instance, a programmer might give a type a name. The name could be held as an item of
meta-class data and made available to every object of that type: for instance every option object could know that the name of its type was "Option". Meta-class data can also be used to keep track of how many instances of the type there are in existence.

In C++ this feature is implemented with static functions and data members. There is no comparable language feature in VBA. Instead meta-class data must be mimicked in VBA using meta-classes or something similar. Chapter 13 is an extended example of this mimicking device. However it is unsatisfactory and should be avoided.

**E.4 PATTERNS OF OBJECT USE**

Constructing an object-oriented application is about deciding, or designing, what objects there are to be in an application and then organizing them into sending messages to one another. ‘Organizing’ means

1. Creating them.
2. Setting them up so that the correct objects know about one another and can communicate.
3. Getting them to use one another, by sending messages to one another, following the channels designed by the programmer.

One characteristic of programming is that the same problems, or at least problems with the same structure, reappear in one guise or another. These standard problems have been solved many times in the past and now have standard solutions. These solutions are called design patterns, some of which are so important that they have been described and given names.

One of the earliest and most important collections of design patterns was given by Gamma, Helm, Johnson and Vlissides (1994), known as, for reasons I cannot guess, the Gang of Four.

Some patterns are so basic as to almost escape the designation. RAII (resource acquisition is initialization) is the name given to the idiom of obtaining the resources needed by the object in a constructor and releasing them in a destructor. We have used it throughout the book, for instance where references are Set in constructors and Set to Nothing in destructors.

Composition is the name given to the situation where one object keeps a reference to another as a (Private) data member. Composition often refers specifically to the case where the lifetime of the composited (contained) object is shorter than that of the containing object, and perhaps has no independent existence. Composition is used extensively in a very high proportion of applications.

Registration and call-back is a design pattern in which an object makes itself known to another, registering with it, and enabling the second object to execute some part of the first object’s interface (the call-back). Registration is to be thought of as a pro-active activity. The registering object instigates the registration, or at least is able to do so in response to a trigger.

Registration and call-back is an essential component of extended polymorphic applications where the scale of the application makes it impossible for the programmer to hard-wire in every association between objects. There are likely to be far too many paths to set to be incorporated every time a new object comes along. It is much better to set things up so that the objects themselves do the hard work of letting other objects know they exist.

In VBA, registration is possible through the VBIDE object. The polymorphic factory in Chapter 12 provides an example of this. The mechanism can be used far more extensively to enable general registrations and call-backs, but this is not so easy or natural to do.

We now describe several other patterns that have appeared in this book. These are the façade pattern, the adapter pattern, the decorator pattern, the factory pattern, the strategy and template patterns and the singleton pattern.
The façade pattern

You have a set of objects that work together. You now want to tidy things up a bit, so you construct an object whose sole purpose is just to wrap the other objects, giving them a single interface to the outside world, hiding away the objects themselves.

Figure E.7 represents a set of objects that are able to send messages to one another. Some of them communicate with the client. Client code sees a jumble of confusing interactions. It may be convenient to hide the objects away inside another object, as in Figure E.8. The advantages of doing this is that the outside world can no longer see any of the complications. These are encapsulated away.

We have used the façade pattern at several points in this book. For instance in Chapter 7 a façade object, \texttt{EvolverW} (Figure 7.8, page 114) is used to wrap two other objects, an \texttt{IWiener} and an \texttt{IGenerator}. Its sole purpose is to tidy away the contained objects.

The adapter pattern

You may have a large application with a well-defined and appropriate set of interfaces. You want to add some further functionality by adding in an object conforming to one of the interfaces. You have available a ready-made object that does the necessary work but which unfortunately does not satisfy your interface. Rather than rewrite the ready-made object to satisfy the required interface you can instead just wrap it in an adapter object. The adapter object conforms to the required interface and contains a reference to the ready-made object. When it receives a request it re-interprets it for its composite object. Output from the composite object is repackaged by the adapter object into the form required by the interface. The pattern is illustrated in Figure E.9. A boiler-plate object with a non-conforming interface is repackaged into an interface of the right sort by wrapping it in an adapter object.

Due to the nature of this book, where we have tried to integrate our applications with tailored code, there has not been too much call for the adapter pattern. However in other contexts it is used fairly frequently. For instance, there is plenty of boiler-plate code around to perform various numerical tasks. However, it does not necessarily fit with the objects one already has. In these cases writing an adapter wrapper class may be a lot easier than trying to modify the boiler-plate code.

![Figure E.7](image)

\textbf{Figure E.7}  A disparate set of intercommunicating objects
Figure E.8  The façade pattern: objects tidied away inside a wrapper class

Figure E.9  The adapter pattern
The decorator pattern

Suppose that you have an object responsible for some task. You now want essentially the same functionality but extended in some way; you want a little bit more functionality interposed either before object procedures run or after they finish, but you are able to keep the same interface.

Here you can add a further conforming object, satisfying the same interface, that contains the original object as a data member. The new object can wrap calls to the composited object with the additional functionality you require. Figure E.10 illustrates an inner object whose functionality is intermediated by the functionality of an outer object.

The design pattern in which an object conforming to an interface has a composited object conforming to the same interface is called the decorator pattern; the outer object decorates the functionality of the inner object. A classical example of this is in implementations of antithetic variate methods in Monte Carlo simulation (see Part VI). An object reifying a general compound option – an option exercisable into some other option – may contain as a composited option object the option it is exercised into. The functionality of the decorating object is greatly extended, and very different, to that of the decorated object.

Another example was given in Chapter 8. The IApp interface described in that chapter has two conforming objects, AppRepeater (Figure 8.7, page 125) and AppMC (Figure 8.8, page 126). AppRepeater contains a reference to another IApp object, AppMC, decorating with it additional functionality. In this example the AppRepeater::run() method causes the run() method of the composite IApp object to execute repeatedly.

The factory pattern

This is the name given to a way that objects can be created polymorphically in a language. You need to be able to create objects without the programmer having to hard-wire in object names (in places other than inside the object itself). In practice this means that objects have to be able to register themselves with the factory, possibly leaving a call-back that enables the factory to create the object. We saw in Chapter 12 how this can be accomplished in VBA using the VBIDE object. The mechanism is not particularly natural but it works.

The strategy and template patterns

These patterns refer to the ways in which algorithms are implemented. The strategy pattern refers to the algorithm as a whole; the template pattern to parts of the algorithm.
If an algorithm may change then it is better to put it inside a polymorphic object. When a better, or just different, algorithm comes along it is much easier to switch from one to the other. Some algorithms perform the same task but vary in speed or accuracy, depending on the values of certain parameters. Here again you want the flexibility to switch to the algorithm most suited to the conditions you have for it. Switching appropriately between algorithms is the strategy pattern.

An example is generating a beta variate. There are a number of different algorithms that can be used for this; the choice depends upon the parameters of the beta variate. The strategy pattern lets you select polymorphically the best algorithm for the job in hand.

The template pattern is related. This refers to delegating parts of an algorithm to other objects. The functionality of these parts can vary while the structure of the algorithm stays the same.

In a PDE context one may evolve backwards using, for instance, explicit or implicit solvers. A PDE method can be set up so that solvers may be used interchangeably, resulting in different methods but within the framework of a top-level common algorithm.

A lattice method can be trinomial or heptanomial; it may be pruned or unpruned. The template pattern allows you to modify these algorithm details while leaving the algorithm as a whole in place.

The singleton pattern

Here the objective is to ensure that one and only one instance of an object type can ever be created. This is not quite the same as requiring that only one instance of an object can exist at any one time. A singleton is a once and future object; it cannot be destroyed and be replaced by a substitute.

The need for a singleton arises in two main ways. First, it may correspond to an application entity of which only one exists; second, for structural reasons, only one instantiation of the object can exist in the code.

An example of the former is an object that manages a file. Having two objects each responsible for opening and closing the file would be disastrous.

For the latter consider the FactoryCreator object from Chapter 12. All factory registrations should register with the same instance of the FactoryCreator object. Later, all clients of FactoryCreator need to access the instantiation that contains the registrations. In Chapter 12 this is ensured by creating a unique instance of FactoryCreator and passing references to it around to clients, but instead the singleton pattern should be implemented. This would be easier to use and less error prone.

The spreadsheet 9c_singleton.xls shows one way in which the singleton idiom can be realized. It has an object type SingleClass that is implemented as a singleton. A standard code special module, SingletonMetaModule, instantiates the unique instance of SingleClass and tracks whether any attempts are made to create further instances of it.

Figure E.11 displays the class SingleClass. All it does is to store a String which has getter and setter Properties. The interesting material is in the constructor. When an instantiation occurs SingleClass checks this with SingletonMetaModule, calling its CheckSingleInstant() procedure, passing it a reference to itself.

The important work takes place in SingletonMetaModule, shown in Figure E.12. As a standard module only single instances of its data members and procedures are ever constructed. As Private data members, it holds a set of references to instantiations of individual singleton objects. In Figure E.12 there is a single singleton object data member, theSingleClassInstance, but as many as you like can be included.

Each singleton type has an associated instance Function and a checking Sub. The instance Function for SingleClass is SingleClassInstance(). All it does is to return a reference to an object referred to by theSingleClassInstance; if theSingleClassInstance has not yet been Set it Sets it as a reference to a New SingleClass object.
Implementing Models of Financial Derivatives

Figure E.11 The class module SingleClass

Private astring_ As String 'represents the application data
Private Sub Class_Initailize()
    Call SingletonMetaModule.CheckSingleInstant(Me) 'Checks the instant
    Call MsgBox("SingleClass instantiated") 'Just to illustrate
End Sub
Private Sub Class_Terminate()
    Call MsgBox("SingleClass terminated") 'Just to illustrate
End Sub

Property Get text() As String: text = astring_: End Property
Property Let text(txt As String): astring_ = txt: End Property

Figure E.12 The standard module SingletonMetaModule

The checking Sub for SingleClass is CheckSingleClass(). When CheckSingleClass() is called it checks to see if SingletonMetaModule already has a reference to a SingleClass object. If it does, then CheckSingleInstant() throws. Otherwise it sets the theSingleClassInstance reference to the reference it has just been passed. This way only one instantiation can ever be made (without throwing) and a reference to it always exists in SingletonMetaModule.

An initial instantiation of a SingleClass can be made either directly using a Set-New statement or using a call to the SingleClassInstance() method. Once an instantiation has been made, a reference to it can be accessed with a call to SingleClassInstance(), but any attempt to instantiate a second instance will throw.

A standard module is practically a singleton object in its own right. Conceptually VBA automatically instantiates a single object with the same name as the module. The standard module ModuleTimer,
in section E.1, is essentially a singleton. The link between standard modules and singletons would be strengthened if procedures and data in standard modules were by default Private and not Public.

By qualifying procedure names with the module name the intention is to reinforce the object-like nature of the module. This is very reminiscent of the syntax used in languages like C++ to call an object’s static methods: you qualify the name of the method not by the name of any particular instantiation of the object but by the name of the object type itself. I recommend that if a standard module is being used as a surrogate singleton object, then its procedure names should always be qualified in this way.

The mechanism implemented in 9c Singleton.xls is not ideal. SingletonMetaModule has to have separate instance and checking procedures for every singleton object in the application. It would be much better to handle this by a registration process. This can be done, but we do not do so here.

## E.5 SUMMARY

Objects are a vital part of any application that is more than just concept code.

We have represented a motivation for objects, and discussed their potential behaviour. We have seen what aspects of this are possible in VBA and how it is commissioned. Finally, we have briefly mentioned some of the ways that objects can relate to one another in applications.

Objects should be adopted by every reader as standard. Rather than write procedures, instead write methods. Use functors instead of functions. Make your code object-oriented.
This appendix presents a lattice method for derivative security valuation. The first section presents some general theory and then describes a particular trinomial lattice for asset processes following a geometric Brownian motion. The second section follows up by describing a poor, monolithic, implementation.

The purpose of this appendix is both to introduce a lattice method and also to provide a starting point for exercises that take the basic implementation and improve it, in stages, into a high-level application.

Only backwards induction methods are described. Forwards induction methods (Jamshidian (1991)) are important and powerful, but are not treated here.

### F.1 LATTICE METHODS

Consider a European style option, on an asset with value $S_t$ at time $t$, with payoff $H(S_T)$ at its maturity time $T$. Let $v_t$ be its value at time $t < T$. $v_t$ is determined by the fundamental option valuation formula (see equation (1.1) in Chapter 1)

$$v_t = \mathbb{E}_t \left[ H(S_T) \frac{P_t}{P_T} \right].$$

In equation (F.1), $P_t$ is a numeraire and $\mathbb{E}_t$ is the expectation with respect to the equivalent martingale measure associated with $P_t$. When $P_t$ is the accumulator account and the short rate is constant, so that $P_t = e^{rt}$, we have

$$v_t = e^{-r(T-t)} \mathbb{E}_t [H(S_T)].$$

In this case the expectation is computed with an asset price process $S = (S_t)_{t \geq 0}$ (for GBM)

$$dS_t = rS_t \, dt + \sigma S_t \, dz_t,$$

for a Wiener process $z_t$.

A lattice method computes $v_t$ by replacing the continuous time state variable $S$ by a discrete time state variable, $\hat{S}$, taking only a small number of possible values at each time step.

Fix a number of time steps $N$ and set $0 = t_0 < t_1 < \ldots < t_N = T$ where $t_{i+1} - t_i = \Delta t = T/N$ is a fixed time increment. $\hat{S}$ is defined only at times $t_i$.

Write $\hat{S}_i$ for $\hat{S}_{t_i}$ and suppose that $\hat{S} = (\hat{S}_i)_{i=1,\ldots,N}$ evolves as follows. Let $B \in \mathbb{N}$ be the order of branching, $\mathcal{U} = \{u_b\}_{b=1,\ldots,B} \subseteq \mathbb{R} \setminus \{0\}$ be a set of scaling factors, and $\{p_b\}_{b=1,\ldots,B} \subseteq (0, 1)$, $\sum_{b=1,\ldots,B} p_b = 1$, be a set of branching probabilities. On each step $\hat{S}$ can branch to one of $B$ different values at the next time. If $\hat{S}$ has value $s$ at time $t_i$ then at time $t_{i+1}$ $\hat{S}$ takes value $u_b s$ with probability $p_b$, $b = 1, \ldots, B$.

In a trinomial lattice one has $B = 3$; a binomial lattice takes $B = 2$. Lattice methods with higher order branching are also found in the literature.\(^1\)

---

\(^1\) Contributions of my own include Alford and Webber (2001) and McCarthy and Webber (2001).
Equation (F.1) can now, in principle, be solved iteratively. The steps are:

1. Evolve forwards constructing a full set of possible values for \( \hat{S}_i \), \( i = 1, \ldots, N \).
2. Evolve backwards evaluating equation (F.1) locally over the interval \([t_i, t_{i+1}]\) at every value of \( \hat{S}_i \).

**Evolving forwards**

This is easy. Write \( S_i \) for the set of values of \( \hat{S}_i \) at time \( t_i \) so that \( S_0 = \{S_0\} \) is the set made up of the asset value at time \( t_0 \). Given \( S_i \) one can iterate forwards to construct

\[
S_{i+1} = \{ s \in \mathbb{R} | s = uz \text{ for some } u \in \mathcal{U}, z \in S_i \}. \tag{F.4}
\]

\( S_i \) is called the slice at time \( t_i \). Elements of \( S_i \) are called nodes. Usually, as in the example we implement in section F.1.1, there is structure to the set \( \mathcal{U} \). In such a case it may be possible to construct \( S_i \) directly from \( S_0 = \{S_0\} \) without iteration.

**Evolving backwards**

Also easy. At each time \( t_i \) one constructs a set \( V_i \) of option values, one for each \( s \in S_i \). Given \( s \in S_i \) write \( v(s) \in V_i \) for the corresponding option value.

At time \( t_N = T \) one computes option values directly from the payoff function: for \( s \in S_N \) set \( v(s) = H(s) \). Now suppose that we have found values \( V_{i+1} \) for time \( t_{i+1} \). We compute \( V_i \) from \( V_{i+1} \) as follows.

Let \( s \in S_i \) and for \( b = 1, \ldots, B \) write \( s^b = u_b s \in S_{i+1} \). Set

\[
v(s) = e^{-r\Delta t} \sum_{b=1}^{B} p_b v(s^b). \tag{F.5}
\]

Equation (F.5) is exactly equation (F.2) applied to \( s \) at \( t_i \) where the option takes values \( \{v(s^b)\}_{b=1}^{B} \) at time \( t_{i+1} \).

One iterates back all the way from time \( t_N \) to time \( t_0 \). At time \( t_0 \) the value of the option is \( v(s) \), where \( s = S_0 \) is the sole member of \( S_0 \).

**American options**

It is simple to value American style options in a lattice. One iterates backwards checking the exercise value at each node. For \( s \in S_i \) now let \( v(s) \) denote the value of an American option and suppose that these have been found back to time \( t_{i+1} \). Equation (F.5) computes the value at time \( t_i \) at asset value \( s \), of the option if it is not exercised. This is called the continuation value. For an American option the holder may exercise and is assumed to do so if the exercise value is greater than the continuation value. Suppose that when the asset value is \( s \) at time \( t \) the option holder receives \( H(t, s) \) upon exercise. Set \( H_i(s) = H(t_i, s) \). Then for an American option, and for \( s \in S_i \), one sets

\[
v(s) = \max\{c(s), H_i(s)\}. \tag{F.6}
\]

where

\[
c(s) = e^{-r\Delta t} \sum_{b=1}^{B} p_b v(s^b) \tag{F.7}
\]

is the continuation value.
A trinomial lattice

This is a special case of the general lattice framework just described. Take $B = 3$ and, for convenience, relabel indexes to run from $-1$ to $1$. For a branching parameter $k > 1$ set

$$u_1 = \exp\left(\left(r - \frac{1}{2} \sigma^2\right) \Delta t + \sigma \sqrt{k} \Delta t\right), \quad p_1 = 1/(2k),$$
$$u_0 = \exp\left(\left(r - \frac{1}{2} \sigma^2\right) \Delta t\right), \quad p_0 = (k - 1)/k,$$
$$u_{-1} = \exp\left(\left(r - \frac{1}{2} \sigma^2\right) \Delta t - \sigma \sqrt{k} \Delta t\right), \quad p_{-1} = 1/(2k).$$

Given $S_0$ one can immediately write down the sets $S_i$. For instance

$$S_N = \left\{ \exp\left(\left(r - \frac{1}{2} \sigma^2\right) T + \sigma j \sqrt{k} \Delta t\right) \right\}_{j=-N,\ldots,N}. \quad (F.9)$$

In general the set $S_i$ is indexed by $j = -i, \ldots, i$ with

$$s_j = \exp\left(\left(r - \frac{1}{2} \sigma^2\right) i \Delta t + \sigma j \sqrt{k} \Delta t\right). \quad (F.10)$$

Suppose that $s \in S_i$. Note that

$$\mathbb{E}_t \left[ \hat{S}_{i+1} \mid s \right] = s \sum_{b \in \{-1,0,1\}} p_b u_b \sim s e^{r \Delta t} \quad (F.11)$$

and

$$\text{var}_t \left[ \hat{S}_{i+1} \mid s \right] = s^2 \sum_{b \in \{-1,0,1\}} p_b \left( u_b - e^{r \Delta t} \right)^2 \sim s^2 \left( e^{\sigma^2 \Delta t} - 1 \right). \quad (F.12)$$

These are the correct first and second moments for a geometric Brownian motion over time step $\Delta t$, so the first two moments of the discrete process agree with those of GBM.

The result holds only in the limit as $\Delta t \to 0$ and it is possible to prove that as $N \to \infty$ option values produced on the lattice converge to their continuous time equivalents.

In a lattice method one first rolls forwards computing values in the sets $S_i$, starting with $S_0$ and moving forwards to end with $S_N$. However, it may be more efficient, where possible, to first compute $S_N$ directly, and then roll backwards. This is true in the case of this lattice. Equation (F.9) shows us how to compute $S_N$. Then for any $s \in S_i$, since we have

$$s = e^{-\left(r - \frac{1}{2} \sigma^2\right) \Delta t} u_0 s, \quad (F.13)$$

a value $u_0 s \in S_{i+1}$ can be rolled back to a value $s \in S_i$.

This lattice recombines: the number of nodes at time $t_i$ is a linear function of $i$. This is an important computational property. In a totally non-recombining lattice the number of nodes $N_i = |S_i|$ at each step is $N_i = B^i$. This blows up rapidly for small $N$. The trinomial example has $N_i = 2i + 1$ which by contrast is very well controlled.

We refer to this lattice, with no speed-ups, bells, or whistles, as the plain (trinomial) lattice.

---

2 A classical result. More can be found in Heston and Zhou (2000).
F.1.2 Assessment of the lattice method

During the valuation phase a lattice method evolves backwards,\(^3\) like a PDE method. This means that it is easy to value American or Bermudan style options. This is a very big advantage of a lattice method. Disadvantages are

1. You cannot easily value path-dependent options.
2. Given a particular model it may be hard to devise an efficient recombining lattice.
3. Models with more than two or three state variables can also be implemented as lattices, but may have too great a memory requirement, and in any case may be hard to devise.
4. Convergence may be slow, particularly for American style options.

Computationally there are also some small problems with the basic method as described.

When evolving forwards the moments of the discrete process should match the true moments of the underlying continuous time process not, like equations (F.11) and (F.12), an approximation to them accurate only to first order. Matching correctly the true moments is essential for fast convergence.

If the interest rate is not constant but stochastic, then the lattice method needs to be extended. This is feasible but takes us too far away from our topic. A particular problem is that even if the value of the interest rate is known at a node \(s \in S_i\), with value \(r\) say, one cannot evolve back efficiently using equation (F.5). Discounting back using the value of \(r\) at the destination node leads to slower than necessary convergence.

A general feature is that if \(N\) is large the values of the asset generated at the boundaries of the slice \(S_N\) become very distant from the expected value of \(\hat{S}_N\). These distant values are reached with very low probability and can be ignored: one prunes the lattice of these nodes so that no calculation needs to be made with them. This speeds-up the lattice greatly, as we see below.

F.2 IMPLEMENTING A LEVEL 0 LATTICE METHOD

In this section we build a basic trinomial lattice application, following closely the steps we took in Chapter 3 for the Monte Carlo application. The general steps are

1. Create a front-end.
2. Create a main stub with I/O, and a button to run it.
3. Build a program outline.
4. Fill in the details.

The implementation we create values American or European puts or calls. It is very poor: it is monolithic and nasty. Please do not follow this type of design. I am doing it this way so that, in a series of exercises, the reader can think of how the implementation can be improved through the various levels, implementing them along the way.

We go through the steps in turn (combining steps 3 and 4).

Creating a front-end

There are three groups of input data: process parameters (the initial stock price \(S_0\), the short rate \(r\), and the stock return volatility \(\sigma\)); option parameters (the maturity time \(T\) and the exercise price \(X\), the exercise

---

\(^3\) Here we are considering only backwards induction methods, not forwards induction methods.
Appendix F: A Yukky Level 0 Monolithic Lattice Implementation

Figure F.1  The level 0 lattice front-end

Figure F.2  Lattice application: the main() stub

type (American or European) and the payoff type (call or put)); and lattice parameters (the number of time steps $N$ and branching parameter $k$). Output is the option value, the execution time, and a counter.

Figure F.1 shows the front-end.\(^4\) As usual the input and output is grouped in boxes to provide what is intended to be a user friendly structure. The button when clicked executes main(). Initially this is just a stub (Figure F.2). From the beginning, as always, banner comments are an integral part of the code.

Adding in the button

We provide a more detailed description of how an ActiveX button is added on to the front-end. The procedure is:

1. Go to the ActiveX controls drop-down Insert menu in the Developer tab and select the button control (Figure F.3).

\(^4\) This is the spreadsheet Lattice_application_v0.xls.
Figure F.3 Selecting an ActiveX control button

2. Click and drag the cursor on the front-end to create the button.
3. Associate code to the button by double-clicking on it. A stub CommandButton1_Click() Sub is created for the button and you are taken to it to edit. Insert the line Call main.

You are put automatically into design mode. Switch out of design mode when you end.

Add in the basic I/O

One can immediately add in code to input from the front-end, to calculate the elapsed time, and to output the time and the option value (Figure F.4). This is easy to do since there are concrete locations to read in from and to write to. Sensible names have been used. X_type and P_type (the exercise type and payoff type) are meant to have code-letter values, “a” or “e” for X_type, “c” or “p” for P_type, and are read in as Strings.

Further code will be added at positions (a), (b) and (c) in the figure.

Add in outline code for generating the lattice

The procedure is:
1. Generate stock values for the final time.
2. Compute payoffs.
3. Discount back through the lattice.
Public Sub main()
    Dim S_0 As Double: S_0 = Cells(9, 5).Value 'initial asset value
    Dim rr As Double: rr = Cells(10, 5).Value 'short rate
    Dim sigma As Double: sigma = Cells(11, 5).Value 'asset volatility
    Dim X As Double: X = Cells(14, 5).Value 'strike
    Dim T As Double: T = Cells(15, 5).Value 'maturity time
    Dim N As Long: N = Cells(9, 8).Value '# time steps
    Dim k As Long: k = Cells(10, 8).Value 'branching parameter
    Dim X_type As String: X_type = Cells(18, 5).Value 'a or e
    Dim P_type As String: P_type = Cells(19, 5).Value 'c or p
    Dim e_time As Double: e_time = Timer
    'Position (a) Initialising slices and probabilities
    'Position (b) Evolving the lattice
    'Position (c) Code for setting o_value goes here
    e_time = Timer - e_time
    Cells(14, 8).Value = o_value
    Cells(15, 8).Value = e_time
End Sub

Figure F.4 Lattice application: adding in I/O

We hold the values in arrays. Four arrays are needed: two to hold the asset values and option values for the current time (this_s and this_o), and two to hold the asset values and option values for the next time as they are being calculated (next_s and next_o). When the new asset values and option values have been found the time step rolls over: the new values become the current values from which the next set of values can be generated.

The four arrays need to be of size $2N + 1$. It is convenient to index them from $-N$ to $N$.

We also need to calculate branching probabilities. These are put into the array probs indexed from $-1$ to $1$.

To accomplish these tasks add in the code in Figure F.5 at position (a) in Figure F.4. There are four sections in this code. Lines F.5a to F.5d precompute useful variables. $dt$ is the time step $\Delta t$; disc_dt is

```vba
Public Sub main()
    Dim dt As Double: dt = T / N 'a
    Dim disc_dt As Double: disc_dt = Exp(-rr * dt) 'b
    Dim drift As Double: drift = (rr - 0.5 * sigma * sigma) * T 'c
    Dim vol As Double: vol = sigma * Sqr(k * dt) 'd
    Dim this_o() As Double: ReDim this_o(-N To N) As Double 'e
    Dim next_o() As Double: ReDim next_o(-N To N) As Double 'f
    Dim this_s() As Double: ReDim this_s(-N To N) As Double 'g
    Dim next_s() As Double: ReDim next_s(-N To N) As Double 'h
    Dim j As Long 'i
    For j = -N To N 'j
        this_s(j) = S_0 * Exp(drift + j * vol) 'k
        this_o(j) = Compute_PO(P_type, this_s(j), X) 'l
    Next j
    Dim probs(-1 To 1) As Double 'm
    probs(-1) = 0.5 / k 'n
    probs(0) = (k - 1) / k 'o
    probs(1) = probs(-1) 'p
End Sub
```

Figure F.5 Lattice application: slices and probabilities
the discount factor over the time step $\Delta t$, $e^{-r\Delta t}$; drift and vol are the terms $(r - \frac{1}{2}\sigma^2)T$ and $\sigma \sqrt{\Delta t}$ respectively appearing in equation (F.9).

The next group of four lines, lines F.5e to F.5h, declare the four slices. this_s and this_o are initialized in the third group, on lines F.5i to F.5l. Finally the last group, lines F.5m to F.5p, declares and initializes probs.

Line F.5k computes the set $S_N$, indexed by $j$, using equation (F.9). Line F.5l then computes $V_N$, the set of payoffs corresponding to $S_N$. Payoffs are returned by the procedure Compute_PO(). This can be added in straight away (Figure F.6). It uses a Select statement to toggle the payoff depending on the value of the payoff code $P_type$. my_max() is the usual utility function.

Constructing the array this_s for time $t_N$ is effectively rolling forwards through the lattice from time $t_0$ in a single step.

**Evolving back through the lattice**

We have constructed asset values and option values for time $t_N$. Now we evolve back through the lattice one time step at a time.

Suppose we have computed option values for time $t_i$ (held in the array this_o) and we want to find values for time $t_{i-1}$, putting them into the array next_o. First we write a loop to evolve backwards. Insert the code in Figure F.7 at position (b) in Figure F.4. This is the main loop, with index counting, stepping back from time $t_N$ to time $t_0$. Now add in the code indicated in Figure F.8 at position (d) in Figure F.7. This does the actual backwards evolution of this_o.

The outer loop, in $i$, loops back, constructing slices for time $t_{N-1}$ back to time $t_0$. It first computes new values, assigning them into next_o. Then the line

$$\text{this}_o = \text{next}_o$$

rolls over the slice to the next time step.

---

5 It is somewhat faster to use an If-Else statement instead of a Select statement, but the latter is more elegant.
Appendix F: A Yukky Level 0 Monolithic Lattice Implementation

```vba
' Evolving back.
Dim interval As Long: interval = 10
Cells(16, 8).Value = N

Dim i As Long
For i = N - 1 To 0 Step -1
    If i / interval = i \ interval Then Cells(16, 8).Value = i
    'Add in the following code
    For j = -i To i 'for each element of the slice
        Dim v As Double: v = 0
        Dim q As Long 'compute expected value
        For q = -1 To 1
            v = v + probs(q) * this_o(j + q)
        Next q
        next_o(j) = disc_dt * v 'discount
    Next j
    'Position (e) American comparison goes here
    this_o = next_o 'rollover
Next i
```

Figure F.8  Lattice application: Evolving back

The middle loop, in \( j \), loops through each element of \( \text{next}_o \) in turn, computing it from values in \( \text{this}_o \). It first computes the expected future value, with the \( q \) loop, and then discounts it back to the time of the next slice.

Once the time step loop has finished, the option value is just \( \text{this}_o(0) \). Insert the code

\[
o_value = \text{this}_o(0)
\]

at position (c) in Figure F.4 and the core functionality of the application is complete. The lattice will now compute values of European calls or puts.

**American options**

A further step is needed to enable the lattice to value American options. So far we are computing the continuation values from equation (F.7). For a European option the continuation value is itself the option value, but for an American option we need to introduce extra code to evaluate equation (F.6).

Add the code in Figure F.9 at position (e) in Figure F.8. \( \text{scale}_s \) is the value \( e^{-\left(r-\frac{1}{2}\sigma^2\right)\Delta t} \). Line F.9d implements equation (F.13) to roll back asset values. Lines F.9d and F.9e implement equation (F.6).

**F.2.1 Assessing the design**

We look first at the convergence properties of the lattice method and then at the design. It works pretty well as an algorithm to produce option values (albeit it can easily be made to run faster and to be generalized, in theory, to other contexts). The difficulty is in the quality of the implementation.

**Convergence results**

As \( N \) increases, option values produced on the lattice converge to their model values in continuous time. Table F.1 compares option values produced on the trinomial lattice with the option’s exact value. Two
options are compared: a European call and an American put, each with strike \( X = 100 \) and time to maturity \( T = 1 \) year. The initial asset value is \( S_0 = 100 \), the riskless rate is \( r = 0.05 \) and the volatility \( \sigma = 0.2 \). For the American put the “exact value” was computed by a Crank–Nicolson PDE method (see Appendix G). Results are shown for the plain lattice constructed in section F.2, and for a faster version described in the next section. Computation times (in seconds) are shown in brackets under the corresponding option values.

It is clear that convergence is haphazard and, in the plain lattice, very slow. The time taken is proportional to the number of nodes in the lattice which, for the plain lattice, grows quadratically in \( N \). For the plain lattice, with \( N = 51200 \) time steps, this is a cost of over 10 minutes for the European call, and almost an hour for the American put, but results are still not guaranteed to be accurate even to 4 decimal places.

Computation times can be improved very significantly using very simple techniques, but even so these results are not confidence inspiring. What is needed is a proper sophisticated lattice method.

### Speeding up the application

The basic application can be speeded up considerably. Five techniques are used here leading to the times shown in the “faster” columns in Table F.1. Note that there is no difference in the option values. The speed-ups are not at the expense of accuracy.

The four techniques are relatively trivial. First, instead of outputting the counter every 10 steps it is output every 100 steps. This yields a saving of 2\%. Second, the Select statement in `Compute_PO()` is replaced by an If-Else statement. In the American case this simple change gives a remarkable saving of around 10\%. Third, each For statement now explicitly has a Step clause. This decreases overall execution times by about 1\%. Fourth, the line

\[
next_s(j) = scale_s * this_s(j)
\]  

(F.16)

in the American comparison has been replaced by the line

\[
this_s(j) = scale_s * this_s(j)
\]  

(F.17)

and the explicit rollover statement removed. This gives a saving of about 2\% for the American case.

Total savings from these measures amount to around 15\%. Not trivial, but more is needed.

---

6 Strictly the counter could be removed altogether from the application without affecting the option valuation but, as we saw in Chapter 15, the added cost of a counter is slight and its utility to the user is great.
The final technique is much more important. This is pruning. Nodes at the extremes of the lattice, that are reached only with very low probability, do not contribute significantly to the option value and do not need to be computed. For $\ln(S)$ at time $T = t_N$ the lattice stretches out to a distance of $N\sigma \sqrt{k\Delta t}$ away from the mean. The standard deviation of $\ln(S)$ is only $\sigma \sqrt{t_N} = \frac{\sigma \sqrt{N}}{\Delta t}$, so the furthest node is

$$\frac{N\sigma \sqrt{k\Delta t}}{\sigma \sqrt{N\Delta t}} = \sqrt{Nk} \quad (F.18)$$

standard deviations away from the mean. With $k = 3$ and for $N = 300$ this is an enormous 30 standard deviations. Contributions from these nodes are utterly ignorable (unless the payoff function is pathological).

In practice only contributions from up to 8 or 10 standard deviations from the mean need to be considered. To reach out $\lambda$ standard deviations, one need have only $N_p$ space steps where

$$N_p \sigma \sqrt{k\Delta t} = \left\lceil \lambda \sigma \sqrt{N\Delta t} \right\rceil \quad (F.19)$$
Implementing Models of Financial Derivatives

Dim Nprune As Long:
If lambda = 0 Then 'a. lambda = 0 means no pruning
  Nprune = N
Else
  Nprune = CLng(lambda * Sqr(N / k)) 'b. Max. extent of the lattice
End If
Dim i As Long
For i = N - 1 To 0 Step -1 'for each time step
  If i / interval = i \\ interval Then Cells(16, 8).Value = i
  Dim Mi As Long: Mi = my_minl(i, Nprune) 'c. Set the pruning level
  For j = -Mi To Mi 'd. Capped range
    'et cetera

Figure F.10  Lattice application: implementing pruning

so \( N_p = \lceil \lambda \sqrt{N/k} \rceil \). Taking \( \lambda = 8 \) and \( k = 3 \) and with \( N = 2700 \), say, one computes \( N_p = 240 \). Only \( \pm 240 \) steps either side of the mean must be computed instead of \( \pm 2700 \), less than a tenth of the number.

On each time step calculations are made only in the range \([-M_i, M_i]\) where \( M_i = \min(i, N_p) \). It is easy to see that if the number of nodes in each slice is capped by \( N_p \) then the total number of nodes in the lattice increases in proportion to \( N^{3/2} \), a rate considerably less than the quadratic rate for the plain case.

The code to effect this is shown in Figure F.10. A parameter \( \lambda \), read in from the front-end, is the number of standard deviations away from the mean value of \( S \) at the final time at which the lattice is to be pruned. Line F.10b sets the value of \( N_p \), the program variable representing \( N_p \), to the Long adjacent to \( \lambda \sqrt{N/k} \). A value of \( \lambda \) of 0 means no pruning and \( N_p \) is set to \( N \). Line F.10c sets the value of \( M_i \). On line F.10d, \( j \) runs only from \(-M_i \) to \( M_i \) instead of from \(-i \) to \( i \), computing a much smaller number of nodes, a change repeated in the subsequent code (not illustrated; please refer to the spreadsheet).

The final two columns of Table F.1 present results for the faster lattice. We can now take many more time steps without burdening the computation times. The option values are identical to those computed in the plain case but the times are very significantly faster. The cost savings are enormous. The run time for the American option with \( N = 51200 \) time steps has gone down from almost an hour on the plain lattice to under 2 minutes on the faster lattice. Doubling the number of time steps increases the run time by a factor of only roughly \( 3 \sim 2^{3/2} \). Convergence is not uniform but it seems that results are accurate to within 5 decimal places for either option within 100 to 200 seconds computation time.

The computation time is still too great, considering the accuracy achieved, but the speed-up we have achieved is nonetheless extremely welcome.

The design

main() in full (for the plain lattice) is shown in Figure F.11. This is really horrible (and the faster lattice is even worse). It is a good example of a bad program. It is monolithic, hard to grasp in its entirety, and hard to maintain. As I was writing it, following the steps above, I was fighting with myself not at least to introduce stub procedures. Still, I persevered, dear reader, so that you would have the satisfaction of doing it yourself.

F.3 SUMMARY

It is very easy to throw together a simple lattice method. The risk is that you end up with something like the implementation described here. Please regard it as a symbol of what to avoid.

7 Spreadsheet Lattice_application_v0_faster.xls.
Sub main()
Dim S_0 As Double: S_0 = Cells(9, 5).Value 'initial asset value
Dim rr As Double: rr = Cells(10, 5).Value 'short rate
Dim sigma As Double: sigma = Cells(11, 5).Value 'asset volatility
Dim X As Double: X = Cells(14, 5).Value 'strike
Dim T As Double: T = Cells(15, 5).Value 'maturity time
Dim N As Long: N = Cells(9, 8).Value '# time steps
Dim k As Long: k = Cells(10, 8).Value 'branching parameter
Dim X_type As String: X_type = Cells(18, 5).Value 'a or e
Dim P_type As String: P_type = Cells(19, 5).Value 'c or p
Dim dt As Double: dt = T / N 'a
Dim disc_dt As Double: disc_dt = Exp(-rr * dt) 'b
Dim drift As Double: drift = (rr - 0.5 * sigma * sigma) * T 'c
Dim vol As Double: vol = sigma * Sqr(k * dt) 'd
Dim scale_s As Double: scale_s = Exp(-drift / N)
Dim this_o() As Double: ReDim this_o(-N To N) As Double 'e
Dim next_o() As Double: ReDim next_o(-N To N) As Double 'f
Dim this_s() As Double: ReDim this_s(-N To N) As Double 'g
Dim next_s() As Double: ReDim next_s(-N To N) As Double 'h
Dim j As Long 'i
For j = -N To N 'j
    this_s(j) = S_0 * Exp(drift + j * vol) 'k
    this_o(j) = Compute_PO(P_type, this_s(j), X) 'l
Next j
Dim probs(-1 To 1) As Double 'm
probs(-1) = 0.5 / k 'n
probs(0) = (k - 1) / k 'o
probs(1) = probs(-1) 'p
Dim e_time As Double: e_time = Timer
Dim interval As Long: interval = 10
Cells(16, 8).Value = N
Dim i As Long
For i = N - 1 To 0 Step -1 'for each time step
    If i / interval = i \ interval Then Cells(16, 8).Value = i
    For j = -i To i 'for each element of the slice
        Dim v As Double: v = 0
        Dim q As Long 'compute expected value
        For q = -1 To 1 'a
            v = v + probs(q) * this_o(j + q)
        Next q
        next_o(i) = disc_dt * v 'discount
    Next j
    if X_type = "a" Then 'For American options
        For j = -i To i 'b
            next_s(j) = scale_s * this_s(j) 'obtain asset values
        Next j
        Dim payoff As Double
        For j = -i To i 'c
            payoff = Compute_PO(P_type, this_s(j), X) 'Compute payoffs
            this_o(j) = my_max(this_o(j), payoff) 'American value
        Next j
    End If
Next j
Dim o_value As Double: o_value = this_o(0)
e_time = Timer - e_time
Cells(14, 8).Value = o_value
Cells(15, 8).Value = e_time
End Sub

Figure F.11 Lattice application, plain case: main()
This page intentionally left blank
Appendix G

A Level 1 Crank–Nicolson PDE Implementation

This appendix is a companion to Appendix F which presented a yukky lattice implementation. We present a level 1 implementation of a Crank–Nicolson PDE solver for 1-dimensional PDEs.

We start by indicating how PDEs arise as equations satisfied by option values. Section G.2 then presents the Crank–Nicolson solution method and section G.3 describes an implementation. An assessment of the method is made in section G.4. In section G.5 we briefly review the SOR (successive over-relaxation) backwards evolution method that potentially allows second-order convergence even for American options.

G.1 PDE METHODS FOR DERIVATIVE VALUATION

As usual we consider a European style option maturing at time $T$ with payoff function $H(S)$ paying off on the value of an underlying asset whose price process $S = (S_t)_{t \geq 0}$ follows a geometric Brownian motion,

$$dS_t = rS_t \, dt + \sigma S_t \, dz_t,$$

for a Wiener process $z_t$ and riskless rate $r$, under the risk-neutral measure. The value $v_t$ of the option at time $t$ is given by the fundamental option valuation formula

$$v_t = \mathbb{E}_t \left[ H(S_T) \frac{P_t}{P_T} \right]$$

where $P_t = e^{rt}$ is the accumulator account numeraire associated with the risk-neutral measure under which $S$ obeys equation (G.1).

Under these conditions, and more generally, the quantity $v_t$ satisfies a certain PDE. Suppose that $S$ follows the process (G.1), then for $P_t = e^{rt}$, and under certain continuity and boundedness conditions, $v_t$ satisfies equation (G.1) if and only if it satisfies the PDE

$$\frac{\partial v_t}{\partial t} + rS \frac{\partial v_t}{\partial S} + \frac{1}{2} \sigma^2 S^2 \frac{\partial^2 v_t}{\partial S^2} = rv_t$$

with boundary conditions $v_T(S) = H(S)$. This is the Feynman–Kac representation.\(^1\) For an American option we also require $v_t(S) \geq H(S)$ for all $t \leq T$.

Having an alternative characterization of $v_t$ is useful. Not only may it open up methods leading to explicit solutions but it also enables a further set of numerical methods – PDE finite difference methods – to be applied. We describe only the Crank–Nicolson finite difference method although many other methods exist. Some of these are discussed from a finance perspective in Wilmott (1998), Tavella and Randall (2000) and Duffy (2004), among others.

The advantage of finite difference methods is that where they work they work very well, and they are extremely well understood. They are reliable: you know, on the whole, where and why they go wrong.

\(^1\) More general versions of Feynman–Kac are presented in Øksendal (1998), and similar sources, including a discussion of conditions under which they hold.
They compute an entire array of option values for time \( t_0 \) for a set of values of the underlying asset; deltas and gammas can be read off along with the option value. They work very well for problems with a single state variable, and quite well for those with two state variables, but then it all starts to go wrong. For three or more state variables, accuracy is lost if you want answers within a reasonable amount of time.

An implicit disadvantage is that finite difference methods apply only to problems expressible as PDEs. If you cannot find a PDE for your path-dependent option, for instance, then you cannot use a PDE method.

### G.2 THE CRANK–NICOLSON FINITE DIFFERENCE METHOD

The Crank–Nicolson method can in principle be applied to any 1-dimensional diffusion PDE and generalizations to \( N \)-dimensional PDEs exist. Only the 1-dimensional case is discussed here. First we look at an application to the PDE (G.3) and then in section G.2.2 to a transformed and simplified version of (G.3).

The key feature of the Crank–Nicolson method, perhaps the main reason for its popularity (and it is popular), is that it is second-order accurate in \( N \). This rate of convergence is achieved only if the number of space steps increases with \( N \). The method convergences order 1 in computation time. An illustration is given in section G.4.

Numerical properties of the Crank–Nicolson method are discussed in detail in the finite difference literature and, to some extent, in the references previously given.

#### G.2.1 The general method

Finite difference methods solve PDEs like that in equation (G.3) in discrete time by permitting \( S \) to take only discrete values in a bounded domain \( S = [S_{\text{min}}, S_{\text{max}}] \subseteq \mathbb{R} \).

Choose the number of space steps, \( M \), and time steps \( N \). Set \( S_{\text{min}} = \mathcal{S}_0 < \mathcal{S}_1 < \ldots < \mathcal{S}_M = S_{\text{max}}, \) with \( \mathcal{S}_j - \mathcal{S}_{j-1} = \Delta S \) constant, and \( 0 = t_0 < \ldots < t_N = T, \) with \( t_i - t_{i-1} = \Delta t \) constant. The index values 0 and \( M + 1 \) are used to impose boundary conditions. Write \( v_{i,j} \) for \( v_t(\mathcal{S}_j) \). Only the values \( \{v_{i,j}\}_{i=0,\ldots,N}, \ j=0,\ldots,M+1 \) are computed.

Finite difference methods replace terms in equation (G.3) by approximations in discrete time and space. For the Crank–Nicolson method set \( \theta = \frac{1}{2} \) and use the following approximations, \( (i = 1, \ldots, N, \ j = 1, \ldots, M) \):

\[
\frac{\partial v_t}{\partial t} (S_j) \sim \frac{v_{i,j} - v_{i-1,j}}{\Delta t}, \quad (G.4)
\]

\[
\frac{\partial v_t}{\partial S} (S_j) \sim \theta \frac{v_{i+1,j} - v_{i,j}}{2\Delta S} + (1 - \theta) \frac{v_{i-1,j} - v_{i-1,j-1}}{2\Delta S}, \quad (G.5)
\]

\[
\frac{\partial^2 v_t}{\partial S^2} (S_j) \sim \theta \frac{v_{i,j+1} - 2v_{i,j} + v_{i,j-1}}{\Delta S^2} + (1 - \theta) \frac{v_{i-1,j+1} - 2v_{i-1,j} + v_{i-1,j-1}}{\Delta S^2}, \quad (G.6)
\]

\[
v_t (S_j) \sim \theta v_{i,j} + (1 - \theta) v_{i-1,j}. \quad (G.7)
\]

Substitute these into equation (G.3) and rearrange. Setting

\[
\mu_{i,j} = rS_{i,j}, \quad (G.8)
\]

\[
\eta_{i,j} = \frac{1}{2} \sigma^2 S_{i,j}^2, \quad (G.9)
\]

\[
\alpha = \frac{\Delta t}{\Delta S^2}, \quad (G.10)
\]
Appendix G: A Level 1 Crank–Nicolson PDE Implementation

\[ u_{i,j} = \alpha \left( \eta_{i,j} + \frac{1}{2} \mu_{i,j} \Delta S \right), \quad (G.11) \]
\[ m_{i,j} = -r \Delta t - 2\alpha \eta_{i,j}, \quad (G.12) \]
\[ l_{i,j} = \alpha \left( \eta_{i,j} - \frac{1}{2} \mu_{i,j} \Delta S \right), \quad (G.13) \]

we obtain, for \( j = 1, \ldots, M, \)

\[ \theta u_{i,j} v_{i,j+1} + (1 + \theta m_{i,j}) v_{i,j} + \theta l_{i,j} v_{i,j-1} \]
\[ = -(1 - \theta) u_{i,j} v_{i-1,j+1} + (1 - (1 - \theta)m_{i,j}) v_{i-1,j} - (1 - \theta) l_{i,j} v_{i-1,j-1}. \quad (G.14) \]

For \( j = 0 \) and \( j = M + 1 \) boundary conditions have to be applied. For instance, suppose that the deltas of the option at \( S_{\text{min}} \) and \( S_{\text{max}} \) are specified. For call options we might impose

\[ \frac{\partial v_t}{\partial S}(S_{\text{min}}) = 0, \quad \frac{\partial v_t}{\partial S}(S_{\text{max}}) = 1 \quad (G.15) \]

and for a put

\[ \frac{\partial v_t}{\partial S}(S_{\text{min}}) = -1, \quad \frac{\partial v_t}{\partial S}(S_{\text{max}}) = 0. \quad (G.16) \]

We assume that the domain \( S \) is sufficiently large so that these approximations can be made without introducing significant error.

In general, suppose that we have been supplied with constants \( \delta_{\text{min}} \) and \( \delta_{\text{max}} \) to take as values of the deltas at \( S_{\text{min}} \) and \( S_{\text{max}}. \) For every \( i \) we impose the following approximations on the grid

\[ \frac{v_{i,1} - v_{i,0}}{\Delta S} = \delta_{\text{min}}, \quad \frac{v_{i,M+1} - v_{i,M}}{\Delta S} = \delta_{\text{max}}. \quad (G.17) \]

Equations (G.14) and (G.17) can be combined and expressed in matrix notation. Write \( v_i = (v_{i,0}, \ldots, v_{i,M+1})', R = \text{diag}(r \Delta t), I = \text{diag}(1), \) and define

\[ \lambda_{\text{min}} = \delta_{\text{min}} \Delta S, \quad \lambda_{\text{max}} = \delta_{\text{max}} \Delta S. \quad (G.18) \]

Set

\[
\begin{pmatrix}
0 & 0 & 0 & 0 & \cdots & 0 \\
\theta u_{i,1} & 1 + \theta m_{i,1} & \theta l_{i,1} & 0 & \cdots & 0 \\
0 & \ddots & \ddots & \ddots & \ddots & \vdots \\
\vdots & \ddots & \ddots & \ddots & \ddots & \ddots \\
0 & \cdots & 0 & \theta u_{i,M} & 1 + \theta m_{i,M} & \theta l_{i,M} \\
0 & \cdots & 0 & 0 & 0 & 0
\end{pmatrix}
\begin{pmatrix}
\lambda_{\text{min}} \\
\vdots \\
\lambda_{\text{max}}
\end{pmatrix}
\]

\[ P_i, \quad \lambda = \begin{pmatrix}
\lambda_{\text{min}} \\
\vdots \\
\lambda_{\text{max}}
\end{pmatrix} \quad (G.19) \]
\[ Q_i = \begin{pmatrix} -1 & 1 & 0 & 0 & \cdots & 0 \\ -\theta u_{i,1} & 1 - \theta m_{i,1} & -\theta l_{i,1} & 0 & \cdots & 0 \\ 0 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \vdots & \cdot & \cdot & \cdot & \cdot & \cdot & 0 \\ 0 & \cdots & 0 & -\theta u_{i,M} & 1 - \theta m_{i,M} & -\theta l_{i,M} \\ 0 & \cdots & 0 & 0 & -1 & 1 \end{pmatrix}, \quad (G.20) \]

where \( \bar{\theta} = 1 - \theta \). Since \( \theta = \frac{1}{2} \) we have \( \bar{\theta} = \theta \).

Now equations (G.14) and (G.17) can be written compactly as

\[ P_i v_i + \lambda = Q_i v_{i-1} \quad (G.21) \]

and the boundary conditions are satisfied at times \( t_i \) for \( i = 0, \ldots, N \).

Equation (G.21) can be solved iteratively, from time \( t_N \) backwards to time \( t_0 \). At time \( t_N \) the value of \( v_N \) is computed directly from the payoff function and from equations (G.17). Set \( v_N \) to be

\[ v_{N,j} = H(S_j), \quad j = 1, \ldots, M \quad (G.22) \]

\[ v_{N,0} = v_{N,1} - \lambda_{\text{min}}, \quad (G.23) \]

\[ v_{N,M+1} = v_{N,M} + \lambda_{\text{max}}. \quad (G.24) \]

Suppose values \( v_i \) have been found for time \( t_i \) then the left-hand side of equation (G.21) is known and the right-hand side is a tridiagonal system of equations that can be solved explicitly by standard methods.\(^2\)

For an American option there is a further American comparison step. In a plain method it takes the following form. Write \( v_{i}^A \) for value of the American option at time \( t_i \). First one solves

\[ P_i v_{i}^A + \lambda = Q_i q_{i-1} \quad (G.25) \]

to find continuation values \( q_{i-1} \) for time \( t_{i-1} \). Then one sets

\[ v_{i-1}^A = \max(H_{i-1}, q_{i-1}) \quad (G.26) \]

where \( H_{i-1} \) is a vector of payoffs for time \( t_{i-1} \). As \( N \to \infty \) this procedure converges to the value of the American option but it converges slowly. A superior method is described in section G.5.

G.2.2 Transforming the PDE

The implementation is simplified, and the solutions that are found suffer less error, if the PDE can be simplified before it is solved. For instance, the Black–Scholes PDE (G.3) can be simplified in various

\(^2\) See numerics books like Press et al. (2007), and books in the derivatives literature such as Clewlow and Strickland (1998), Wilmott (1998), Travella and Randall (2000), and others.
ways. A simple transformation is to a new variable \( R_t = \ln(S_t) \). Write \( \hat{v}_t \) for the option value as a function of \( R_t \) so that \( \hat{v}_t(R) = v_t(e^R) \). The PDE in \( R_t \), equivalent to PDE (G.3), is

\[
\frac{\partial \hat{v}_t}{\partial t} + \left( r - \frac{1}{2} \sigma^2 \right) \frac{\partial \hat{v}_t}{\partial R} + \frac{1}{2} \sigma^2 \frac{\partial^2 \hat{v}_t}{\partial R^2} = r \hat{v}_t. \tag{G.27}
\]

The advantage of applying this transformation is that the coefficients no longer depend on \( S \) (or \( R \)) so that now

\[
u_{i,j} = u = \alpha \left( \sigma^2 + \left( r - \frac{1}{2} \sigma^2 \right) \Delta S \right), \tag{G.28}\]

\[
m_{i,j} = m = -r \Delta t - \alpha \sigma^2, \tag{G.29}\]

\[
l_{i,j} = l = \alpha \left( \sigma^2 - \left( r - \frac{1}{2} \sigma^2 \right) \Delta S \right), \tag{G.30}\]

are constants. The matrices \( P_i \) and \( Q_i \) for the PDE (G.27) are constant and are greatly simplified compared to their general forms given in equations (G.19) and (G.20). Corresponding simplifications and speed-ups can be made to the tridiagonal solvers used to find solutions to equation (G.21).

**G.3 IMPLEMENTING CRANK–NICOLSON**

The spreadsheet CN_pde_v1.xls contains a level 1 implementation of a Crank–Nicolson PDE solver. The program solves the Black–Scholes equation in the transformed form, equation (G.27). We go through the usual steps to create the application, following Chapter 3 and Appendix F. First we create a stub \( \text{main}() \), executed by a button on the front-end, then I/O is added in, and then, step by step, the main functionality.

**The interface and I/O**

The user interface is shown in Figure G.1. There are the usual groups of input and output cells. The grid parameters are the number of time steps, \( N \), and space steps, \( M \), and the values of the asset at the top and bottom of the grid, \( S_{\text{max}} \) and \( S_{\text{min}} \). The exercise type is either American or European (“a” or “e”) and the payoff type is either call or put (“c” or “p”).

Note that the user is advised to make the number of space steps, \( M \), a fixed multiple of the number of time steps, \( N \). We return to this later (in section G.4) when we look at the performance of the method.

Figure G.2 presents the basic I/O. It also gives the definitions of a pair of Enums, \( \text{ExerciseType} \) and \( \text{PayoffType} \), that are used in the code to represent payoff and exercise types. A number of procedures are defined. \( \text{ClearCells()} \) blanks out the output cells; \( \text{GetXtype()} \) and \( \text{GetPtype()} \) read in characters from the front-end and return the corresponding Enums; and \( \text{OutputValues()} \) outputs the option value and the computation time. Further code is needed at position (a), to initialize the arrays we need, at position (b), for the main loop, and at position (c) to extract the option value.

If you must have a Select statement instead of going polymorphic, Enums are safer than using raw characters. Of course it is better not to have structural Select statements in the code at all. An exercise in the body of the book asks the reader to make this application polymorphic. At that point the Enums may either disappear entirely, or be restricted just to the input side of the application. With a fully polymorphic factory there is no place at all for this use of Enums.

The outline of the main loop can now be added in. Option values are first found at time \( t_N \) from the payoff function. Then values are constructed backwards in time, starting with those at time \( t_{N-1} \) and...
implementing models of financial derivatives

\[ v_1 \text{ (Set } M = 10N) \]

<table>
<thead>
<tr>
<th>A Crank Nicolson pde solver</th>
<th>run</th>
</tr>
</thead>
</table>

scale \( M \) with \( N \): \( \text{eg set } M = 10N \)

<table>
<thead>
<tr>
<th>Process parameters</th>
<th>Grid parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S_0 = 100 )</td>
<td>( N = 12800 )</td>
</tr>
<tr>
<td>( r = 0.05 )</td>
<td>( M = 128000 )</td>
</tr>
<tr>
<td>( \sigma = 0.2 )</td>
<td>( S_{\text{max}} = 1000 )</td>
</tr>
<tr>
<td>( S_{\text{min}} = 10 )</td>
<td>( )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Option parameters</th>
<th>Option value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X = 100 )</td>
<td>( c = 10.45058 )</td>
</tr>
<tr>
<td>( T = 1 )</td>
<td>( \text{time} = 677.2969 )</td>
</tr>
<tr>
<td>( X_{\text{type}} = )</td>
<td>( )</td>
</tr>
<tr>
<td>( P_{\text{type}} = c )</td>
<td>( \text{counter} = 0 )</td>
</tr>
</tbody>
</table>

Figure G.1  The Crank–Nicolson user interface

iterating backwards to time \( t_0 \). Figure G.3 shows the outline loop to insert at position (c) in Figure G.2. It uses a \text{Sub, OutputCounter()}, to output a counter.

Data representation and initialization

Three arrays are required. An array of asset values, \( s_{\text{vals}} \), an array of option values found for the current step, \( o_{\text{vals}} \), and an array, \( n_{\text{vals}} \), to hold option values for the next step as they are being calculated. Here, unlike the lattice implemented in Appendix F, there is no need to roll back the asset values. They remain constant across every time step.

The arrays are constructed as follows. Let \( \overline{S}_j, j = 0, \ldots, M + 1 \), be the asset values at each level in the grid and let \( R_j, j = 0, \ldots, M + 1 \), be the log-values, \( R_j = \ln(\overline{S}_j) \). Set \( \Delta R = \ln(S_{\text{max}}/S_{\text{min}})/(M + 1) \). Note that if we set \( R_0 = \ln(S_{\text{min}}) \) and \( \overline{S}_j = \exp(R_0 + j\Delta R), j = 0, \ldots, M + 1 \), then there is no guarantee that the initial asset value, \( S_0 \), corresponds to any \( \overline{S}_j \). To ensure that \( S_0 \) lies on the grid we use the following procedure.

Set \( M_d = \lfloor \ln(S_0/S_{\text{min}})/\Delta R \rfloor \). \( M_d \) is the number of whole steps that one can go down starting from \( \ln(S_0) \) until just before one hits \( S_{\text{min}} \). Now set \( R_0 = \ln(S_0) - M_d\Delta R \) and \( \overline{S}_j = \exp(R_0 + j\Delta R), j = 0, \ldots, M + 1 \).

Although this ensures that \( S_0 = \overline{S}_{M_d} \) lies on the grid, the values of \( \overline{S}_j \) have been offset so that \( \overline{S}_0 \neq S_{\text{min}} \) and \( \overline{S}_{M+1} \neq S_{\text{max}} \). However, for practical values of \( M \), \( S_{\text{min}} \) and \( S_{\text{max}} \) the offset is slight.

The reification of \( \{\overline{S}_j\}_{j=0,...,M+1} \) in the application is \( s_{\text{vals}} \), indexed from 0 to \( M + 1 \). From \( s_{\text{vals}} \) the option values at the final time can be calculated and put into \( o_{\text{vals}} \). Code to initialize \( s_{\text{vals}} \) and \( o_{\text{vals}} \) is given in Figure G.4. This is inserted into Figure G.2 at position (a). \text{SetSvalues()} first sets the value of \( \overline{S}_0 \) from \( S_0 \) and then works up through each \( \overline{S}_j \) in turn. \text{SetOvalues()} calls \text{Compute_PO()} , also shown in the figure, to set the option values for the final time.

The array \( n_{\text{vals}} \) is declared here, but does not need to be initialized.
Appendix G: A Level 1 Crank–Nicolson PDE Implementation

The Enums

Public Enum ExerciseType
    AmericanExercise
    EuropeanExercise
End Enum

Public Enum PayoffType
    calltype
    puttype
End Enum

main(): I/O

Public Sub main()
    Call ClearCells
    Dim S_0 As Double: S_0 = Cells(9, 5).Value 'initial asset value
    Dim r As Double: r = Cells(10, 5).Value 'short rate
    Dim sigma As Double: sigma = Cells(11, 5).Value 'asset volatility
    Dim X As Double: X = Cells(14, 5).Value 'strike
    Dim T As Double: T = Cells(15, 5).Value 'maturity time
    Dim N As Long: N = Cells(9, 8).Value '# time steps
    Dim M As Long: M = Cells(10, 8).Value '# space steps
    Dim S_max As Double: S_max = Cells(11, 8).Value 'Max value of S
    Dim S_min As Double: S_min = Cells(12, 8).Value 'Min value of S
    Dim X_type As ExerciseType: X_type = GetXtype(18, 5) 'Exercise type
    Dim p_type As PayoffType: p_type = GetPtype(19, 5) 'Payoff type
    Dim e_time As Double: e_time = Timer
    'Position (a). Initialize arrays
    'Position (b). Initialization for tridiagonal inversion
    'Position (c). The main loop
    Dim o_value As Double
    'Position (d). Extract the option value
    e_time = Timer - e_time
    Call OutputValues(o_value, e_time)
End Sub

Private Sub OutputValues(o_value As Double, e_time As Double)
    Cells(17, 8).Value = o_value
    Cells(18, 8).Value = e_time
End Sub

Private Sub ClearCells()
    Cells(17, 8).Value = ""
    Cells(18, 8).Value = ""
    Cells(19, 8).Value = ""
End Sub

getters

Private Function GetPtype(X As Long, y As Long) As PayoffType
    Dim achar As String: achar = Cells(X, y).Value 'c or p
    Select Case achar
        Case "c": GetPtype = calltype
        Case "p": GetPtype = puttype
    End Select
End Function

Private Function GetXtype(X As Long, y As Long) As ExerciseType
    Dim achar As String: achar = Cells(X, y).Value 'a or e
    Select Case achar
        Case "a": GetXtype = AmericanExercise
        Case "e": GetXtype = EuropeanExercise
    End Select
End Function

Figure G.2  Crank–Nicolson: I/O
Dim interval As Long: interval = 10
Call OutputCounter(N, interval)
Dim i As Long
For i = N - 1 To 0 Step -1
    Call OutputCounter(i, interval)
    'Position (e). Iterate backwards
Next i

Public Sub OutputCounter(j As Long, interval As Long)
    If j / interval = j \ interval Then Cells(19, 8).Value = j
End Sub

Dim dt As Double: dt = T / N
Dim dx As Double: dx = Log(S_max / S_min) / (M + 1)
Dim S_ind As Long: S_ind = CLng(Log(S_0 / S_min) / dx) 'Index of S_0
Dim s_vals() As Double: Call SetSvalues(s_vals, M, S_ind, S_0, dx)
Dim o_vals() As Double: Call SetOvalues(o_vals, X, s_vals, p_type)
Dim n_vals() As Double: 'Declared but initialization not needed

Private Sub SetSvalues(ByRef s_vals() As Double, M As Long, S_ind As Long, _
S_0 As Double, dx As Double)
    ReDim s_vals(0 To M + 1) As Double
    Dim exp_dx As Double: exp_dx = Exp(dx)
s_vals(0) = S_0 * Exp(-S_ind * dx)
    Dim j As Long
    For j = 1 To M + 1 Step 1
        s_vals(j) = s_vals(j - 1) * exp_dx
    Next j
End Sub

Private Sub SetOvalues(ByRef o_vals() As Double, X As Double, _
s_vals() As Double, p_type As PayoffType)
    Dim lb As Long: lb = LBound(s_vals) '0
    Dim ub As Long: ub = UBound(s_vals) 'M+1
    ReDim o_vals(lb To ub) As Double
    Dim j As Long
    For j = lb To ub Step 1 'initialise payoff at maturity
        o_vals(j) = Compute_PO(p_type, s_vals(j), X)
    Next j
End Sub

Private Function Compute_PO(ptype As PayoffType, S As Double, X As Double) As Double
    Select Case ptype
        Case calltype: Compute_PO = my_max(0, S - X)
        Case puttype: Compute_PO = my_max(0, X - S)
    End Select
End Function

Figure G.3  Crank–Nicolson: outline main loop

Figure G.4  Crank–Nicolson: initializing s_vals and o_vals
Figure G.5 Crank–Nicolson: computing weights and boundary conditions

Two other segments of initialization are needed. The elements that form the diagonals of the matrix \( P \) and \( Q \) in equation (G.21) must be computed, and the boundary values \( \lambda_{\text{min}} \) and \( \lambda_{\text{max}} \) need to be set. The code for this, in Figure G.5, is added into Figure G.2 at position (c). The weights \( p_u \), \( p_m \) and \( p_d \) form the diagonals of \( P \). The variables \( \lambda_{\text{min}} \) and \( \lambda_{\text{max}} \) are used to represent the boundary values. \( \text{SetWeights()} \) and \( \text{SetBoundary()} \) implement equations (G.28), (G.29) and (G.30), and equations (G.18) respectively. \( \lambda_{\text{max}} \) and \( \lambda_{\text{min}} \) have to be computed from the actual increments in \( S \) at the top and bottom of the grid.

**The main loop**

The outline main loop is very simple. Expand the main loop to the form shown in Figure G.6 adding the indicated code at position (e) in Figure G.3. \( \text{EvolveBack()} \), shown in Figure G.7, constructs \( n_{\text{vals}} \) from \( o_{\text{vals}} \) by solving equation (G.21). The weights and boundary value arguments fully specify the tridiagonal matrix. \( \text{EvolveBack()} \) is a modified version of a pseudo-code program, \( \text{solve\_Crank\_Nicolson\_tridiagonal\_system()} \), appearing in Clewlow and Strickland (1998). This is a version of a general \( LU \) tridiagonal solver tailored for the Crank–Nicolson case with constant diagonal terms.
Dim i As Long
For i = N - 1 To 0 Step -1
    Call OutputCounter(i, interval)
    'Insert the following code
    Call EvolveBack(o_vals, n_vals, pu, pm, pd, lamb_min, lamb_max)
    o_vals = n_vals 'rollover
    If X_type = AmericanExercise Then
        Call AmericanComparison(s_vals, o_vals, X, p_type)
    End If
Next i

Figure G.6  Crank–Nicolson: expand the main loop

Private Sub EvolveBack(o_vals() As Double, n_vals() As Double, _
    pu As Double, pm As Double, pd As Double, _
    lamb_min As Double, lamb_max As Double)
    Dim lb As Long: lb = LBound(o_vals) '0
    Dim ub As Long: ub = UBound(o_vals) 'M + 1
    ReDim n_vals(lb To ub) As Double
    Dim y_vals() As Double: ReDim y_vals(lb To ub) As Double
    Dim i As Long
    For i = lb + 1 To ub - 1
        y_vals(i) = -pd * o_vals(i - 1) - (pm - 2) * o_vals(i) - pu * o_vals(i + 1)
    Next i
    Dim a() As Double: ReDim a(lb To ub) As Double
    Dim b() As Double: ReDim b(lb To ub) As Double
    a(ub - 1) = pm + pu
    b(ub - 1) = y_vals(ub - 1) + pu * lamb_max
    For i = ub - 2 To lb + 1 Step -1
        a(i) = pm - pd * pu / a(i + 1)
        b(i) = y_vals(i) - b(i + 1) * pu / a(i + 1)
    Next i
    n_vals(lb) = (b(lb + 1) + a(lb + 1) * lamb_min) / (pd + a(lb + 1))
    n_vals(lb + 1) = n_vals(lb) + lamb_min
    For i = lb + 2 To ub - 1 Step 1
        n_vals(i) = (b(i) - pd * n_vals(i - 1)) / a(i)
    Next i
    n_vals(ub) = n_vals(ub - 1) + lamb_max
End Sub

Figure G.7  Crank–Nicolson: EvolveBack()

In EvolveBack(), y_vals is an array of known values for the left-hand side of equation (G.21). For further description please see Clewlow and Strickland (1998).

After n_vals has been found the time step then rolls over, and finally, if required, an American comparison is made by the procedure AmericanComparison(). The code is given in Figure G.8. This step is identical to the corresponding step in the lattice implementation in Appendix F.

Finally extract the option value. Insert the line

\[ o_{\text{value}} = o_{\text{vals}}(S_{\text{ind}}) \]  \hspace{1cm} (G.32)

at position (d) in Figure G.2.

For completeness the full version of main() is given in Figure G.9.
We discuss the program design and then look at the Crank–Nicolson method itself. Singled out for
discussion are the American comparison step and the convergence properties of the method.

The design

This is a fairly simple level 1 procedural implementation. The largest number of lines of code in main() is devoted to input. This should be hived off into a separate procedure, aided by constructing a UDT. This and other changes would bring the application up to a level 2 design.

Considered as a level 1 program the code is reasonably clear. Most of the functionality is safely encapsulated in procedures leaving a structure that can be grasped without too much difficulty. Of course the program is level 1 so is single purpose and hard to generalize. It is cheap and cheerful and you get what you pay for.

It is not as fast as it could be. The reader may have noticed that there is a significant amount of duplicated computation that can be removed easily resulting in sizeable savings in execution time. Please refer to exercise 5 in Chapter 3.

The American comparison

The backwards evolution step is split into two separate parts.

1. A tridiagonal inversion to get the continuation values at time $t_{i-1}$ from option values at time $t_i$.
2. A direct comparison between the continuation values and the exercise values to obtain the option values at time $t_{i-1}$.

Unfortunately this method of comparison loses – throws away might be a more apt expression – the property of second-order convergence in $N$. A method converges only as fast as the slowest component and the comparison component is only first order in $N$. A better method, such as SOR or PSOR, is needed and we explore this possibility in section G.5.

Convergence results

How good is the method itself? We investigate its convergence properties and execution time.
Public Sub main()
Call ClearCells
Dim S_0 As Double: S_0 = Cells(9, 5).Value 'initial asset value
Dim r As Double: r = Cells(10, 5).Value 'short rate
Dim sigma As Double: sigma = Cells(11, 5).Value 'asset volatility
Dim X As Double: X = Cells(14, 5).Value 'strike
Dim T As Double: T = Cells(15, 5).Value 'maturity time
Dim N As Long: N = Cells(9, 8).Value '# time steps
Dim M As Long: M = Cells(10, 8).Value '# space steps
Dim S_max As Double: S_max = Cells(11, 8).Value 'Max value of S
Dim S_min As Double: S_min = Cells(12, 8).Value 'Min value of S
Dim X_type As ExerciseType: X_type = GetXtype(18, 5)
Dim p_type As PayoffType: p_type = GetPtype(19, 5)
Dim dt As Double: dt = T / N
Dim dx As Double: dx = Log(S_max / S_min) / (M + 1)
Dim S_ind As Long: S_ind = CLng(Log(S_0 / S_min) / dx) 'Index of S_0

'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
// Initialize
'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Dim s_vals() As Double: Call SetSvalues(s_vals, M, S_ind, S_0, dx)
Dim o_vals() As Double: Call SetOvalues(o_vals(), X, s_vals(), p_type)
Dim n_vals() As Double: 'Declared but initialization not needed

'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
// main loop
'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Dim e_time As Double: e_time = Timer
Dim interval As Long: interval = 10
Call OutputCounter(N, interval)
Dim i As Long
For i = N - 1 To 0 Step -1
    Call OutputCounter(i, interval)
    Call EvolveBack(o_vals, n_vals, pu, pm, pd, lambda_L, lambda_U)
o_vals = n_vals 'rollover
If X_type = AmericanExercise Then
    Call AmericanComparison(s_vals, o_vals, X, p_type)
End If
Next i
Dim o_value As Double: o_value = o_vals(S_ind)
e_time = Timer - e_time
Call OutputValues(o_value, e_time)
End Sub

Figure G.9  Crank-Nicolson: main()
Table G.1  Crank–Nicolson: Convergence in $N$ when $M = 10N$.
(Timings in seconds. Value to 5 decimal places)

<table>
<thead>
<tr>
<th>$N$</th>
<th>European call</th>
<th>American put</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Plain</td>
<td>Plain</td>
</tr>
<tr>
<td>10.45058</td>
<td>10.44876</td>
<td>6.07763</td>
</tr>
<tr>
<td></td>
<td>[0.02]</td>
<td>[0.02]</td>
</tr>
<tr>
<td>100</td>
<td>10.45013</td>
<td>6.08434</td>
</tr>
<tr>
<td></td>
<td>[0.03]</td>
<td>[0.11]</td>
</tr>
<tr>
<td>200</td>
<td>10.45047</td>
<td>6.08744</td>
</tr>
<tr>
<td></td>
<td>[0.16]</td>
<td>[0.41]</td>
</tr>
<tr>
<td>400</td>
<td>10.45056</td>
<td>6.08892</td>
</tr>
<tr>
<td></td>
<td>[0.61]</td>
<td>[1.59]</td>
</tr>
<tr>
<td>800</td>
<td>10.45058</td>
<td>6.08964</td>
</tr>
<tr>
<td></td>
<td>[2.28]</td>
<td>[6.42]</td>
</tr>
<tr>
<td>1600</td>
<td>10.45058</td>
<td>6.09001</td>
</tr>
<tr>
<td></td>
<td>[9.13]</td>
<td>[26]</td>
</tr>
<tr>
<td>3200</td>
<td>10.45058</td>
<td>6.09019</td>
</tr>
<tr>
<td></td>
<td>[36]</td>
<td>[103]</td>
</tr>
<tr>
<td>6400</td>
<td>10.45058</td>
<td>6.09028</td>
</tr>
<tr>
<td></td>
<td>[146]</td>
<td>[419]</td>
</tr>
<tr>
<td>12800</td>
<td>10.45058</td>
<td>6.09033</td>
</tr>
<tr>
<td></td>
<td>[676]</td>
<td>[1810]</td>
</tr>
<tr>
<td>25600</td>
<td>10.45058</td>
<td>6.09035</td>
</tr>
<tr>
<td></td>
<td>[2760]</td>
<td>[7410]</td>
</tr>
<tr>
<td>51200</td>
<td>10.45058</td>
<td>6.09036</td>
</tr>
<tr>
<td></td>
<td>[11160]</td>
<td>[30220]</td>
</tr>
</tbody>
</table>

The table has three columns of results. The first and second columns are for the European call and American put using the plain methods described in section G.3. The third column gives results for a PSOR algorithm for the American put described below in section G.5. For the moment we discuss only the results for the plain method.

The value of the European call is converging to its Black–Scholes value. That of the American put to a value of 6.09037 to 5 decimal places.

It is important that $M$ increases in proportion to $N$. This is a theoretical requirement and Crank–Nicolson must always be implemented this way. If $M$ is decoupled from $N$ the results can be alarming. Table G.2 plots $\ln(\text{error})$ for the European call in Table G.1 for various combinations of $N$ and $M$. The smallest value of $\ln(\text{error})$ in each column is given in bold. For fixed $N$ as $M$ increases, errors initially decline until they reach a minimum; then they begin to increase again. Similarly for fixed $M$ as $N$ increases, the error reduces but after reaching a minimum it levels out (and then begins to increase again).

When $M$ is kept proportional to $N$, errors behave as they should. Figure G.10 plots $\ln(\text{error})$ against $\ln(N)$ for the European call from Table G.1. Convergence is uniform until the error becomes very small. The slope is $-2$, showing that the error is second order in $N$, and hence first order in computation time.

The situation is different for the American put. Figure G.11 plots $\ln(\text{error})$ against $\ln(N)$ for the American put (with plain backwards evolution) from Table G.1. Convergence is again uniform but now the slope...
## Table G.2  Crank–Nicolson: errors for a European call. \( \ln(\text{error}) \) against \( N \) and \( M \)

<table>
<thead>
<tr>
<th>( M )</th>
<th>100</th>
<th>300</th>
<th>500</th>
<th>700</th>
<th>900</th>
<th>1100</th>
<th>1300</th>
<th>1500</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>-7.70</td>
<td>-7.65</td>
<td>-7.65</td>
<td>-7.64</td>
<td>-7.64</td>
<td>-7.64</td>
<td>-7.64</td>
<td>-7.64</td>
</tr>
<tr>
<td>5000</td>
<td>-4.34</td>
<td>-8.73</td>
<td>-10.91</td>
<td>-10.89</td>
<td>-10.88</td>
<td>-10.88</td>
<td>-10.87</td>
<td>-10.87</td>
</tr>
<tr>
<td>9000</td>
<td>-4.05</td>
<td>-6.19</td>
<td>-8.77</td>
<td>-11.51</td>
<td>-12.08</td>
<td>-12.08</td>
<td>-12.07</td>
<td>-12.06</td>
</tr>
<tr>
<td>11000</td>
<td>-4.00</td>
<td>-5.80</td>
<td>-7.71</td>
<td>-10.07</td>
<td>-12.12</td>
<td>-12.48</td>
<td>-12.48</td>
<td>-12.47</td>
</tr>
<tr>
<td>13000</td>
<td>-3.98</td>
<td>-5.58</td>
<td>-7.09</td>
<td>-8.92</td>
<td>-11.03</td>
<td>-12.55</td>
<td>-12.81</td>
<td>-12.82</td>
</tr>
<tr>
<td>15000</td>
<td>-3.96</td>
<td>-5.44</td>
<td>-6.70</td>
<td>-8.16</td>
<td>-9.89</td>
<td>-11.74</td>
<td>-12.90</td>
<td>-13.10</td>
</tr>
</tbody>
</table>

![Figure G.10](image1.png)  
**Figure G.10**  European call: convergence, ln-ln scales

![Figure G.11](image2.png)  
**Figure G.11**  American put: convergence, ln-ln scales
is $-1$ so here the method is only first order in $N$ and order 0.5 in computation time. The errors are much greater than for the European case.

The reason for the worse performance of the American put is the way that the early exercise condition has been incorporated into the implementation. The direct comparison made by the procedure AmericanComparison() is accurate only to first order in $N$, not to second order, so the method as a whole is only first order. Second-order methods for the American comparison step exist. The SOR method described in section G.5 is one of these.

It takes 2 to 3 seconds for the value of the European call to become accurate to 5 decimal places, but the American put is still not accurate to 5 decimal places after over eight and a half hours of computation. By contrast the (fast) lattice method of Appendix F was accurate to 5 decimal places, for both the European call and the American put, within 2 or 3 minutes.

### G.5 SUCCESSIVE OVER-RELAXATION (SOR)

The problem solved by the backwards evolution algorithm given in Figure G.7 is to find a vector $x$ that satisfies

$$y = Mx \tag{G.33}$$

for a known vector $y$ and a tridiagonal matrix $M$. In our case on the $i$th time step we have $y = P_i v_i + \lambda$, $M = Q_i$, and $x$ are values found for $v_{i-1}$. With the transformed PDE (G.27), $M$ takes the form

\[
M = \begin{pmatrix}
-1 & 1 & 0 & 0 & \cdots & 0 \\
a & b & c & 0 & \cdots & 0 \\
0 & \ddots & \ddots & \ddots & \ddots & \vdots \\
\vdots & \ddots & \ddots & \ddots & \ddots & \ddots \\
0 & \cdots & 0 & a & b & c \\
0 & \cdots & 0 & 0 & -1 & 1
\end{pmatrix} \tag{G.34}
\]

where $a = -\overline{\theta}u$, $b = 1 - \overline{\theta}m$ and $c = -\overline{\theta}l$ is constant.

The $LU$ solver sets $M = LU$ for

\[
L = \begin{pmatrix}
1 & 0 & \cdots & \cdots & 0 \\
a & 1 & 0 & \cdots & 0 \\
0 & \ddots & \ddots & \ddots & \vdots \\
\vdots & \ddots & \ddots & \ddots & \ddots \\
0 & \cdots & 0 & a & 1 \\
0 & \cdots & 0 & -1 & 1
\end{pmatrix}, \tag{G.35}
\]

\[
U = \begin{pmatrix}
-1 & 1 & 0 & 0 & \cdots & 0 \\
0 & b & c & 0 & \cdots & 0 \\
\vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\
\vdots & \ddots & \ddots & \ddots & \ddots & \ddots \\
0 & \cdots & 0 & b & c \\
0 & \cdots & 0 & 0 & 1
\end{pmatrix}. \tag{G.36}
\]

The system $y = LUx$ can be solved in two steps. First solve $Lv = y$ for $v$ and then solve $Ux = v$. This is what EvolveBack() does.
So far so good. Difficulties arise when there is an American comparison step to perform alongside the backwards evolution.

American options satisfy the PDE (G.3) but with the condition

\[ v_t(S) \geq H(S) \]  

imposed at all times \( t \leq T \), not just at times \( t_i \). Equations (G.25) and (G.26) are not the correct equations to solve; instead one should solve

\[ P_i v^A_i + \lambda = Q_i \max(H_i - 1, v^A_{i-1}) \]  

which in practice means finding a best fit

\[ v^A_{i-1} = \arg \min_{v_i-1} |Q_i \max(H_i - 1, v_{i-1}) - P_i v_i - \lambda|. \]

This is what the SOR method does.

### G.5.1 The SOR method

References for the SOR method include, among others, Press et al. (2007) and Wilmott (1998), and most books on finite difference methods. See Duffy (2004) for a C++ perspective.

We start by investigating an alternative solution method for equation (G.33). One constructs a sequence of successive approximations \( x_k \) to \( x \), so that \( x_k \) converge to \( x \) as \( k \to \infty \). This is done in such a way that it is easy to impose the early exercise constraint.

Since \( M \) is tridiagonal we have \( M = U + D + L \) where

\[
U = \begin{pmatrix} 0 & 1 & 0 & \cdots & 0 \\ & \ddots & c & \ddots & \vdots \\ & & \ddots & \ddots & 0 \\ & & & \ddots & c \\ 0 & \cdots & 0 \\ \end{pmatrix},
\]

\[
D = \begin{pmatrix} -1 & 0 & \cdots & \cdots & 0 \\ 0 & b & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & b & 0 \\ 0 & \cdots & \cdots & 0 & 1 \\ \end{pmatrix},
\]

\[
L = \begin{pmatrix} 0 & \cdots & 0 \\ a & \ddots & \vdots \\ 0 & \ddots & \ddots & \vdots \\ \vdots & \ddots & a & \ddots \\ 0 & \cdots & 0 & -1 & 0 \\ \end{pmatrix}.
\]
Appendix G: A Level 1 Crank–Nicolson PDE Implementation 597

$D$ is the diagonal of $M$, $U$ is the super-diagonal of $M$, and $L$ is the sub-diagonal of $M$.

Write $x = (x_0, \ldots, x_{M+1})'$ then for each $j = 1, \ldots, M$ we require

$$U_{j,j+1}x_{j+1} + D_{j,j}x_j + L_{j,j-1}x_{j-1} = y_j, \quad j = 1, \ldots, M$$

(G.43)

(with slightly different conditions applied at $j = 0, M + 1$). This can be solved iteratively. From an initial value $x^0$ construct $x^{k+1}$ iteratively from $x^k$ by setting

$$x_{j}^{k+1} = \frac{1}{D_{j,j}}(y_j - U_{j,j+1}x_{j+1}^{k} - L_{j,j-1}x_{j-1}^{k}), \quad j = 1, \ldots, M$$

(G.44)

In matrix notation this is just

$$x^{k+1} = D^{-1}(y - (U + L)x^k).$$

(G.45)

$x^k$ converges to the solution $x$. One stops iterating when $|x^k - x^{k+1}|$ is sufficiently small.

Unfortunately convergence can be very slow. It can be improved a little by using the new values $x_j^{k+1}$ as soon as they are found. Set

$$x_{j}^{k+1} = \frac{1}{D_{j,j}}(y_j - U_{j,j+1}x_{j+1}^{k} - L_{j,j-1}x_{j-1}^{k+1})$$

(G.46)

$$= x_{j}^{k} + \frac{1}{D_{j,j}}(y_j - U_{j,j+1}x_{j+1}^{k} - D_{j,j}x_{j}^{k} - L_{j,j-1}x_{j-1}^{k+1})$$

(G.47)

Equation (G.46) is equivalent to

$$x^{k+1} = (D + L)^{-1}(y - Ux^k)$$

(G.48)

$$= x^k + (D + L)^{-1}(y - (U + D + L)x^k)$$

(G.49)

$$= x^k + (D + L)^{-1}\xi^k,$$

(G.50)

where

$$\xi^k = y - (D + U + L)x^k$$

(G.51)

is the error on the $k$th step. From the current value $x^k$ one computes the error $\xi^k$ and uses it to adjust the current values to get the next values. For a given tolerance level $\varepsilon_{\xi}$ on $\xi^k$ one iterates until $|\xi^{k+1} - \xi^k| < \varepsilon_{\xi}$.

Convergence is still too slow, but now it can be improved significantly by modifying the procedure implicit in equation (G.50). Choose $0 < \omega < 2$ and at each step set

$$x^{k+1} = x^k + \omega(D + L)^{-1}\xi^k.$$  

(G.52)

For $1 < \omega < 2$ this is the method of successive over-relaxation (SOR). At each step you over-correct. With a good choice of $\omega$ convergence is improved markedly. It can be implemented in practice by adapting equation (G.47), setting

$$x_{j}^{k+1} = x_{j}^{k} + \omega\frac{1}{D_{j,j}}(y_j - U_{j,j+1}x_{j+1}^{k} - D_{j,j}x_{j}^{k} - L_{j,j-1}x_{j-1}^{k+1}).$$

(G.53)
Implementing Models of Financial Derivatives

It turns out, of course, that the choice of $\omega$ is critical. One would like to choose it to make $x^k$ converge in the fewest iterations, but a poor choice can instead make convergence slower. Various methods are described in the literature (and reviewed in Press et al., 2007). Here we adopt an adaptive approach, letting the value of $\omega$ vary on each time step depending on whether the number of iterations taken for convergence is increasing or decreasing.

Suppose we are given an initial value $1 < \omega N < 2$, an initial direction $d_i = 1$ and fix an increment $\delta \omega > 0$. Let $K_i$ be the number of iterations it takes to converge on time step $t_i$. At time $t_i$ we have values $\omega_i$ and $d_i$. Then for time $t_{i-1}$ set

$$\omega_{i-1} = \omega_i + d_{i-1} \delta \omega$$  \hspace{1cm} (G.54)

where $d_{i-1} = d_i$ if $K_i \leq K_{i+1}$ and $d_{i-1} = -d_i$ if $K_i > K_{i+1}$.

If the number of iterations decreases then the value of $\omega$ changes in the same direction as it has previously just changed; if it increases then the next value of $\omega$ is changed in the direction opposite to the previous change.

Equation (G.53) is still solving only the basic European problem (G.33) but now it is easy to incorporate the American condition. On each step of the $k$th iteration compute the $j$th continuation value $q^k_{j+1}$,

$$q^k_{j+1} = x^k_j + \omega \frac{1}{D_{j,j}} (y_j - U_{j,j+1} x^k_{j+1} - D_{j,j} x^k_j - L_{j,j-1} x^k_{j-1})$$  \hspace{1cm} (G.55)

but then immediately impose the early exercise condition setting

$$x^{k+1}_j = \max(q^k_{j+1}, H_j), \ j = 1, \ldots, M$$  \hspace{1cm} (G.56)

This is called projected SOR. Using it allows the Crank–Nicolson method to convergence with order $O(\Delta t^2, \Delta S^2)$.

### G.5.2 Implementing the SOR method

To implement SOR one reconfigures the main loop in Figure G.6, replacing the procedure AmericanComparison() with a SOR-based method. While we are doing this we precompute the array $y$ and make other changes to improve the application a little. The spreadsheet CN_pde_SOR_v2.xls contains a level 2 version of a PDE solver for American and European options. The level 2 RunCN() is given in Figure G.12.

There are two new arrays: $h\_vals$ is an array of payoff values and $y\_vals$ corresponds to the RHS vector $y$ in equation (G.33). SetHvalues() computes the array $h\_vals$, $h\_vals$ is assigned to $o\_vals$ as the values at the final time. The Sub SetYvalues() constructs $y\_vals$ in the main loop on line G.12b.

There are several UDTs in the level 2 design: $tddata$ of type $TD\_data$ holds data for the tridiagonal matrix and $SOR\_data$ of type $SOR\_data$ has parameter values required for the SOR method. Please refer to the spreadsheet for their definitions.

At the start of each iteration of the main loop one has constructed a set of option values, $o\_vals$, from the previous time step. The first action is to compute the vector $y = P t v_i + \lambda$ on line G.12b. Line G.12c then calls SolverLU(). This is a slightly modified form of EvolveBack() from section G.3. It no longer calculates $y$ for itself and is passed its parameters in UDTs rather than individually. It returns a vector of values solving equation (G.21) in $n\_vals$. These solve the European backwards evolution problem.
Appendix G: A Level 1 Crank–Nicolson PDE Implementation

Private Sub RunCN()
    Call ClearCells
    Dim cn_data As CN_data: Call ReadInData(cn_data)
    '// Initialise
    'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
    Dim s_vals() As Double: Call SetSvalues(s_vals, cn_data)
    Dim h_vals() As Double: Call SetHvalues(h_vals, s_vals, cn_data)
    Dim n_vals() As Double: n_vals = h_vals
    Dim y_vals() As Double: 'RHS values. initialization not needed
    Dim tddata As TD_data: Call SetTDdata(s_vals, tddata, cn_data)
    Dim SORdata As SOR_data: Call SetSORdata(SORdata)
    '// main loop
    Dim e_time As Double: e_time = Timer
    Dim i As Long
    For i = cn_data.N - 1 To 0 Step -1
        Call OutputCounter(i, cn_data.interval)
        Call SetYvalues(y_vals, n_vals, tddata) 'b
        Call SolverLU(o_vals, y_vals, n_vals, tddata) 'c
        If cn_data.X_type = AmericanExercise Then
            Call MaxArrays(n_vals, h_vals) 'd
            Call SolverSOR(h_vals, y_vals, n_vals, _
                           tddata, SORdata, cn_data.N - i) 'e
        End If
        o_vals = n_vals 'f
    Next i
    '// end stuff
    'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
    Dim o_value As Double: o_value = GetOvalue(o_vals, cn_data)
    e_time = Timer - e_time
    Call OutputValues(o_value, e_time)
End Sub

Figure G.12 Crank–Nicolson: the level 2 SOR RunCN()

If the option is American, so that the Enum X_type in the UDT cn_data has value AmericanExercise,\(^4\) then the PSOR method is applied to perform an American update step.

First the library utility Sub MaxArrays() does a plain American comparison. After this call n_vals contains the option values produced by the AmericanComparison() procedure from section G.3. Here, though, n_vals is now fed to SolverSOR() to act as starting values in a PSOR iteration.

SolverSOR() is shown in Figure G.13. The main Do-loop constructs \(x^{k+1}\), denoted by x_new, from \(x^k\), denoted by x_old, implementing equation (G.55) on lines G.13f and G.13g. The updating step, equation (G.56), is performed on line G.13h (with minor modification at the boundaries). If the distance between x_new and x_old is less than a tolerance level sdata.TOLERANCE, then the iteration rolls over and ends.

\(^4\) Please refer to the spreadsheet for definitions of the Enums.
Private Sub SolverSOR(h_vals() As Double, y_vals() As Double, _
ByRef n_vals() As Double, _
tddata As TD_data, sdata As SOR_data, _
ii As Long)
Dim pu As Double: pu = tddata.pu
Dim pm As Double: pm = tddata.pm
Dim pd As Double: pd = tddata.pd
Dim omega As Double: omega = sdata.omega
Dim lb As Long: lb = LBound(n_vals) '0
Dim ub As Long: ub = UBound(n_vals) 'M + 1
Dim x_old() As Double: ReDim x_old(lb To ub) As Double
Dim x_new() As Double: ReDim x_new(lb To ub) As Double
x_old = n_vals
Dim not_converged As Boolean: not_converged = True 'a
Dim iter As Long: iter = 0 'b
Do While not_converged And iter < sdata.MAX_ITERATIONS
Dim error As Double: error = y_vals(lb) - (x_old(lb + 1) - x_old(lb)) 'b
x_new(lb) = x_old(lb) - omega * error 'c
x_new(lb) = my_max(h_vals(lb), x_new(lb)) 'd
Dim i As Long
For i = lb + 1 To ub - 1 Step 1 'e
error = y_vals(i) - (pd * x_new(i - 1) + pm * x_old(i) + pu * x_old(i + 1)) 'f
x_new(i) = x_old(i) + omega * error / pm 'g
x_new(i) = my_max(h_vals(i), x_new(i)) 'h
Next i
error = y_vals(ub) - (x_old(ub) - x_new(ub - 1)) 'i
x_new(ub) = x_old(ub) - omega * error 'j
x_new(ub) = my_max(h_vals(ub), x_new(ub)) 'k
Dim x_error As Double: x_error = RMSE(x_new, x_old) 'l
If x_error < sdata.TOLERANCE Then not_converged = False 'm
x_old = x_new 'n
iter = iter + 1
Loop
Call UpdateSORdata(iter, sdata) 'p
n_vals = x_old 'q
End Sub

Private Sub UpdateSORdata(iter As Long, ByRef sdata As SOR_data)
If iter > sdata.previous_iter Then sdata.dw = -sdata.dw
Dim omega As Double: omega = sdata.omega + sdata.dw
If omega > sdata.U_w Then omega = sdata.U_w
If omega < sdata.L_w Then omega = sdata.L_w
sdata.previous_iter = iter
sdata.omega = omega
End Sub

Figure G.13 Crank–Nicolson: the PSOR solver, SolverSOR()
Appendix G: A Level 1 Crank–Nicolson PDE Implementation

UpdateSORdata(), also shown in Figure G.13, implements an adaptive $\omega$, allowing $\omega$ to change depending on whether the number of iterations is increasing or decreasing from the previous step. The value of $\omega$ on the previous step, and the number of previous iterations, are kept in the UDT sdata. The Sub UpdateSORdata() takes care of the book-keeping.

G.5.3 Comparison of the methods

Values of the American put obtained using the PSOR method are given in the final column of Table G.1. The method is clearly slower, for a given value of $N$, than the plain method, but it appears to be more accurate. This is borne out by a convergence comparison.

Figure G.14 shows a comparison of the convergence of the plain and PSOR methods for the American put of Table G.1. The figure plots $\ln(error)$ against $\ln(N)$. The PSOR method is clearly much more accurate for each $N$ and is also converging more rapidly. The plain method is converging at a rate $-1$ in $N$, like Figure G.11, but the PSOR method is converging at rate $\sim -1.5$. This is far faster, but a little misleading. Figure G.15 plots $\ln(error)$ against $\ln(time)$.$^5$ This comparison is much more to the point, in practical terms.

The difference is still very significant, but much less. The plain method is converging $-0.5$ in time (as before) whereas the PSOR method is converging $\sim -0.7$ in time. The practical significance of this is that the PSOR method is correct to 5 decimal places after “only” four hours or so instead of the eight and a half hours it took previously.

This is still not as fast as the (faster) lattice method, and it is not the order 2 convergence in time we were promised, but we are on the right track.

\[ \begin{align*}
\text{ln-ln convergence in N: PSOR and plain methods} \\
\text{ln}(N) & \quad \text{ln}(\text{error}) \\
\text{PSOR} & \quad \text{plain}
\end{align*} \]

Figure G.14 Convergence for an American put in the plain and PSOR methods: Convergence in $N$

$^5$ Times are from a faster version of the plain method. See exercise 5, Chapter 3.
G.6 SUMMARY

The Crank–Nicolson method is a reliable stand-by. It is excellent for use with simple models with a single state variable; life gets hard with two or more state variables.

For European options it is not really amenable to speed-ups. It is take it or leave it. A comparison with the lattice method of Appendix F indicates that while Crank–Nicolson is much faster than the lattice for a European option, when using plain evolution it is much slower for an American option: a fast lattice method can sometimes beat a Crank–Nicolson method that does not use a second-order American comparison step.

---

Figure G.15 Convergence for an American put in the plain and PSOR methods: Convergence in execution time

---

6 Although it is susceptible to slow-downs.
This appendix briefly summarizes four root-finding algorithms (the bisection method, the method of false position, Ridders’ method, and the Newton-Raphson method) and one algorithm for finding the minimum of a function (the golden section search method). This reflects part of the coverage in Press et al. (2007). These methods are used by the implied volatility exercise stream. Level 2 implementations are given.

### H.1 ROOT FINDING ALGORITHMS

The four methods we review are simple methods; not necessarily particularly good, but simple. Press et al. (2007) review all of these and other, better, methods.

We assume that we are given an interval \( I = [L, U] \subseteq \mathbb{R} \) and a continuous function \( f : I \to \mathbb{R} \) and we want to find a root of \( f \) in the interval \( I \), that is, a value \( x_0 \) such that \( f(x_0) = 0 \). We assume that either \( f(L) < 0 \) and \( f(U) > 0 \) or \( f(L) > 0 \) and \( f(U) < 0 \). This ensures that there exists at least one solution to the equation in \( I \).

The algorithms execute a series of iterations. They exit if one of the following occurs:

1. A limit on the number of iterations, \( N_{\text{max}} \), is hit.
2. The new approximation to the root is within a tolerance level \( \varepsilon_x \) of the previous approximation.
3. A value of \( x \) such that \( |f(x)| < \varepsilon_f \), for some tolerance level \( \varepsilon_f \), has been found.

We look at each method in turn.

#### H.1.1 The bisection algorithm

The bisection algorithm is a cheerful method that easily finds a value \( x_0 \in I \) such that \( f(x_0) \sim 0 \). The method finds successive intervals \( [l_i, u_i], i \geq 0 \), with \( [l_0, u_0] = [L, U] \) and \( [l_{i+1}, u_{i+1}] \subset [l_i, u_i] \) for all \( i \), such that the root is guaranteed to lie in every interval. The bracketing intervals half in length at every iteration, converging in the limit to \( x_0 \).

**Description**

Choose a tolerance \( \varepsilon_x \) on the root, a tolerance \( \varepsilon_f \) on the closeness of \( f(x_0) \) to zero, and a maximum iteration count \( N_{\text{max}} \). The algorithm works as follows:

1. Set \( x_{\text{min}} = L \) and \( x_{\text{max}} = U \), and set \( i = 0 \).
2. Set \( x_{\text{mid}} = \frac{1}{2}(x_{\text{min}} + x_{\text{max}}) \), \( f_{\text{mid}} = f(x_{\text{mid}}) \), and \( i = i + 1 \);
   - if \( i > N_{\text{max}} \) or \( |x_{\text{max}} - x_{\text{min}}| < \varepsilon_x \) or \( |f_{\text{mid}}| < \varepsilon_f \) goto step 4.
3. If \( f(x_{\text{min}}) > 0 \) and \( f_{\text{mid}} < 0 \), or if \( f(x_{\text{min}}) < 0 \) and \( f_{\text{mid}} > 0 \), set \( x_{\text{max}} = x_{\text{mid}} \);
   - if \( f(x_{\text{min}}) > 0 \) and \( f_{\text{mid}} > 0 \), or if \( f(x_{\text{min}}) < 0 \) and \( f_{\text{mid}} < 0 \), set \( x_{\text{min}} = x_{\text{mid}} \). Goto step 2.
4. Return \( x_{\text{mid}} \).

At each step the root lies in the interval \( [x_{\text{min}}, x_{\text{max}}] \). The bisection algorithm is a very good root finder for a narrowly defined set of functions. It is safe (as long as the initial bounds bracket a root) but its rate
of convergence is slow. It does not need to calculate any derivatives \( f'(x) \). It is a simple method for finding implied volatilities from the Black–Scholes and other simple pricing formulae. In these well-controlled circumstances with functions approximately linear, close to their roots, it typically requires relatively few iterations to converge sufficiently.

**Implementation**

Let \( N \) be the cumulative normal distribution function for a normal variate with mean \( \mu \) and variance \( \sigma^2 \), and let \( \nu \) be a constant target value. app_roots_bisection.xls is a simple level 2 implementation of the bisection method that solves the equation

\[
N(x \mid \mu, \sigma) - \nu = 0. \tag{H.1}
\]

\( N \) is a smooth monotonic function and the bisection methods finds roots easily.

The key procedure in app_roots_bisection.xls is `Bisection()` (Figure H.1). There are a pair of `Types`, also shown in the figure, passed as arguments to `Bisection()`. They contain parameter values for the

```vba
Private Enum ExitT
    normalexit
    IterationsExceeded
End Enum
Public Type FnData
    mu As Double
    sig As Double
End Type
Private Type ProbData
    lb As Double
    ub As Double
    target As Double
End Type
Private Sub Bisection(ByRef root As Double, ByRef fval As Double, ByRef et As ExitT, _
    fdat As FnData, pdat As ProbData)
    Const MAX_ITERATIONS As Long = 100
    Const R_TOLERANCE As Double = 0.000001 'tolerance on the root
    Const F_TOLERANCE As Double = 0.00000000001 'tolerance on the target
    et = normalexit
    Dim lx As Double: lx = pdat.lb
    Dim ux As Double: ux = pdat.ub
    Dim lf As Double: lf = N(lx, fdat) 'N at lower bound
    Dim uf As Double: uf = N(ux, fdat) 'N at upper bound
    Dim target As Double: target = pdat.target
    Dim i As Long
    For i = 1 To MAX_ITERATIONS Step 1
        root = 0.5 * (ux + lx)
        fval = N(root, fdat)
        If Abs(ux - lx) < R_TOLERANCE Or Abs(fval - target) < F_TOLERANCE Then Exit Sub
        If (lf < target And fval > target) Or (lf > target And fval < target) Then
            ux = root
            uf = fval ' (not necessary)
        Else
            lx = root
            If fval ' (not necessary)
        End If
    Next i
    et = IterationsExceeded
End Sub
```

**Figure H.1** Bisection()
function being evaluated, in \( \text{FnData} \), and parameters for the bisection problem in \( \text{ProbData} \). The procedure \( N() \) computes values of the distribution function. It is defined in the module NormalCDF (whose details do not concern us here). \( \text{FnData} \) has to be declared to be Public since \( N() \) is in a separate module.

Steps 2 and 3 are implemented in a For-loop. An Enum, \( \text{et} \), tracks whether the algorithm has converged or whether it exits before it converges. \( \text{Bisection()} \) returns, ByRef, the value of the root, the function value at the root, and the exit condition, \( \text{et} \). It computes, but does not really need, the values of the function \( N() \) at the end-points of the current interval.

\( \text{Bisection()} \) is set up to solve equations of the form \( f(x) = \nu \). In the code \( \nu \) is represented by the variable \( \text{target} \).

The calling procedure, \( \text{main()} \), given in Figure H.2, is simple. It reads in the data, tests if the bracket condition required by \( \text{Bisection()} \) holds, runs \( \text{Bisection()} \) and finally outputs the values returned by \( \text{Bisection()} \).

**H.1.2 The method of false position**

This method also requires no knowledge of the first derivative of \( f \). Like the bisection method it finds successive intervals \( [l_i, u_i] \), \( i \geq 0 \), with \( [l_0, u_0] = [L, U] \) and \( [l_{i+1}, u_{i+1}] \subset [l_i, u_i] \) for all \( i \), such that at least one root always remains bracketed in each interval.

**Description**

Over each interval \( [l_i, u_i] \) the method approximates \( f \) as a linear function. Suppose we know the values \( f(l_i) \) and \( f(u_i) \). Were \( f \) linear, then the root would be located at

\[
x = l_i - \frac{f(l_i)}{f(u_i) - f(l_i)} (u_i - l_i).
\]

The algorithm computes \( x \) and \( f(x) \). If \( f(l_i) > 0 \) and \( f(x) < 0 \), or if \( f(l_i) < 0 \) and \( f(x) > 0 \), it sets \( [l_{i+1}, u_{i+1}] = [l_i, x] \). Otherwise it sets \( [l_{i+1}, u_{i+1}] = [x, u_i] \). Successive intervals lie strictly within one another, and always contain a root.

This method can converge faster than the bisection method but for unpleasant \( f \) can require rather more iterations.
Private Sub FalsePosition(ByRef root As Double, ByRef fval As Double, ByRef et As ExitT, _
    fdat As FnData, pdat As ProbData)
    Const MAX_ITERATIONS As Long = 100
    Const R_TOLERANCE As Double = 0.000001 'tolerance on the root
    Const F_TOLERANCE As Double = 0.00000000001 'tolerance on the target
    et = normalexit
    Dim lx As Double: lx = pdat.lb
    Dim ux As Double: ux = pdat.ub
    Dim lf As Double: lf = N(lx, fdat) 'N at lower bound
    Dim uf As Double: uf = N(ux, fdat) 'N at upper bound
    Dim target As Double: target = pdat.target
    Dim i As Long
    For i = 1 To MAX_ITERATIONS Step 1
        root = lx + (target - lf) * (ux - lx) / (uf - lf)
        fval = N(root, fdat)
        If Abs(ux - lx) < R_TOLERANCE Or Abs(fval - target) < F_TOLERANCE Then Exit Sub
        If (lf < target And fval > target) Or (lf > target And fval < target) Then
            ux = root
            uf = fval
        Else
            lx = root
            lf = fval
        End If
    Next i
    et = IterationsExceeded
End Sub

Figure H.3  FalsePosition()

Implementation

The method is implemented in app_roots_false.xls. Like app_roots_bisection.xls it solves equation (H.1). The procedure FalsePosition(), given in Figure H.3, is similar in form to Bisection(). For the method of false position the values of the function at the end-points of the current interval are used and have to be computed.

The only substantive change in this method from the bisection method is that root is now computed from equation (H.2) and not as a mid-point.

H.1.3 Ridders’ method

Ridders’ method finds a root by approximating \( f \) not as a linear function but with a functional form that tries to capture the curvature in \( f \). We present the variation described in Press et al. (2007).

Description

At the \( i \)th iteration one has a bracketing interval \([l_i, u_i]\). \( f \) is approximated on \([l_i, u_i]\) by a function \( \hat{f} \) of the form

\[
\hat{f}(x) = (a + bx)e^{-q(x-l_i)}.
\]  (H.3)

The analytic root of \( \hat{f} \) determines the next interval.
Appendix H: Root-Finding and Minimization Algorithms

Specifically set \( m_i = \frac{1}{2}(u_i + l_i) \) and compute \( f(m_i) \). Suppose that \( g(x) = f(x)e^{g(x-l_i)} \) is linear on \([l_i, u_i]\) in that

\[
\frac{g(l_i) - g(m_i)}{l_i - m_i} = \frac{g(m_i) - g(u_i)}{m_i - u_i}.
\]  

(H.4)

Since \( l_i - m_i = m_i - u_i = \Delta \), say, this requirement can be satisfied if

\[
f(l_i) - f(m_i) e^{g \Delta} = f(m_i) e^{g \Delta} - f(u_i) e^{2g \Delta}.
\]  

(H.5)

Solving for \( Q = e^{g \Delta} \) one obtains

\[
Q = \frac{f(m_i)}{f(u_i)} + \frac{\sqrt{f^2(m_i) - f(l_i) f(u_i)}}{|f(u_i)|}.
\]  

(H.6)

Since \( g \) is linear its root is at \( x \) where

\[
x = m_i - (m_i - l_i) \frac{g(m_i)}{g(m_i) - g(l_i)}.
\]  

(H.7)

Substituting for \( g \) and \( Q \) and simplifying, one obtains

\[
x = m_i + \text{sign}(f(l_i) - f(u_i)) (m_i - l_i) \frac{f(m_i)}{\sqrt{f^2(m_i) - f(l_i) f(u_i)}}.
\]  

(H.8)

In general \( x \) is likely to be closer to the true root of \( f \) than a root obtained by direct application of a false position, as illustrated in Figure H.4. By construction, \( g \) is linear such that

\[
g(l) = f(l) \quad \text{and} \quad \frac{g(u)}{f(u)} = \left( \frac{g(m)}{f(m)} \right)^2.
\]  

(H.9)

Figure H.4 The interpolating function \( g(x) \)
Interpolating with \( g \) can cancel out the curvature present in \( f \), so that the interpolated point \( x \) is much closer to the true root than either the point \( y \), resulting from the method of false position, or \( m \), from the bisection method, would yield.

**Implementation**

The spreadsheet app_roots_Ridders.xls contains an implementation of Ridders’ method. Figure H.5 shows the procedure `Ridders()`. Unlike the previous two methods, which found an \( x \) such that \( N(x \mid \mu, \sigma) = \nu \), in this case it is expedient to explicitly find a root \( x \) of \( f(x) = N(x \mid \mu, \sigma) - \nu = 0 \).

The method calculates \( f \) at two points per iteration, at \( m_i \) and at \( x \), but its rate of convergence is still strictly greater than bisection.

**H.1.4 The Newton–Raphson method**

The Newton–Raphson method uses knowledge about the first derivative of \( f \) to find the next approximation to \( x_0 \). If this value cannot easily be computed then the method cannot be used.

It is no longer the case that the new approximation to \( x_0 \) has to lie within the current interval. If it does not, then the method may be about to generate approximations to the root that head off to infinity.

We start with an initial interval \([L, U]\) known to bracket a root. The algorithm does not track successive intervals, but only the values of \( x \) that it generates.
Description

Suppose that \( f \) is a linear function. For \( x \in \mathbb{R} \) and from the values \( f(x) \) and \( \frac{\partial f}{\partial x}(x) = f'(x) \) the root of \( f \) would be at

\[
x_0 = x - \frac{f(x)}{f'(x)},
\]

(H.10)

This is used as the basis for an iteration. If \( f \) is almost linear then values of \( x \) generated from successive applications of equation (H.10) converge to a root of \( f \). Unfortunately if \( f \) is too far from linearity the method can diverge, with \( |x_i| \to \infty \).

Implementation

At the current value of \( x_i \) the algorithm computes \( f(x_i) \) and \( f'(x_i) \) and sets

\[
x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}.
\]

(H.11)

If \( x_{i+1} \notin [L, U] \), then the algorithm exits with an error condition.

```plaintext
Private Enum ExitT
    normalexit
    IterationsExceeded
    OutOfInterval
End Enum

Private Sub NewtonRaphson(ByRef root As Double, ByRef fval As Double, ByRef et As ExitT, _
    fdat As FnData, pdat As ProbData)
    Const MAX_ITERATIONS As Long = 100
    Const R_TOLERANCE As Double = 0.000001 'tolerance on the root
    Const F_TOLERANCE As Double = 0.00000000001 'tolerance on the target
    et = normalexit
    Dim lx As Double: lx = pdat.lb
    Dim ux As Double: ux = pdat.ub
    Dim target As Double: target = pdat.target
    root = 0.5 * (lx + ux) 'starting value
    Dim i As Long
    For i = 1 To MAX_ITERATIONS Step 1
        fval = N(root, fdat)
        If Abs(fval - target) < F_TOLERANCE Then Exit Sub
        Dim deriv As Double: deriv = dN(root, fdat)
        Dim dx As Double: dx = (target - fval) / deriv
        root = root + dx
        If root < lx Or root > ux Then
            et = OutOfInterval
            Exit Sub
        End If
        If Abs(dx) < R_TOLERANCE Then Exit Sub
    Next i
    et = IterationsExceeded
End Sub
```

Figure H.6  NewtonRaphson()
Figure H.6 shows the procedure `NewtonRaphson()` in the spreadsheet app_roots_NR.xls. Like the other three implementations it solves equation (H.1). The initial value of $x$ is chosen to be the mid-point of the bracketing interval. The `Enum, ExitT`, may now take an additional value, `OutOfInterval`, reflecting the new error condition.

**H.2 MINIMIZATION ALGORITHMS**

We have the following problem in mind. Suppose we have a set of market prices $V = \{v_k\}_{k=1,...,K}$ that we want to calibrate to with a model. Suppose the model has a set of parameters $P = \{p_q\}_{q=1,...,Q}$. Write $\hat{v}_k \equiv \hat{v}_k(P)$ for a model price and $\hat{V} \equiv \hat{V}(P) = \{\hat{v}_k(P)\}_{k=1,...,K}$ for the set of model prices. The problem is to find values of the parameters so that model prices are as close as possible, by some criterion, to the market prices. Given a criterion function $J(V, \hat{V}(P))$ the problem is to determine an optimal set $\hat{P}$,

$$\hat{P} = \arg \min_P J(V, \hat{V}(P)).$$  \hspace{1cm} (H.12)

One criterion function is weighted least squares. Let $\{w_k\}_{k=1,...,K}$, with $0 < w_k < 1$ and $\sum_{k=1}^K w_k = 1$, be a set of weights. Define $J^w$ as

$$J^w(V, \hat{V}) = \sqrt{\sum_{k=1}^K w_k(v_k - \hat{v}_k)^2}.$$  \hspace{1cm} (H.13)

We suppose that the weights are given in advance and reflect either the “goodness” of prices or their “importance”. A large value of $w_k$ means that we want to fit closely to $v_k$, a small value means that we are not so concerned about fitting closely.

If an option is not very liquid then its price may have a large bid–ask spread, or else it is unreliable in that if you attempted to trade at that price you might be able instead to trade only at a price not so close to it. Conversely, if an option is liquid the bid–ask spread is likely to be smaller and you are more likely to be able to trade closer to the given price.

It is important to fit closely to prices of (liquid) instruments that you use in a hedging portfolio. If you are hedging an exotic option there is likely to be a particular set of preferred hedging instruments. Different exotics may have different preferred sets, so if calibrating different exotics in isolation you would place greater weight to different sets of hedging instruments.

Equation (H.13) is a $Q$-dimensional optimization problem. Here we discuss a solution method for $Q = 1$. General solution methods for $Q > 1$ are discussed in Press et al. (2007).

When $Q = 1$ suppose that a model has a single parameter, which we now denote by $\sigma$, whose value we need to determine to fit the model to the market.\footnote{This is not likely to be a model that fits very well but we let that pass.} We want to solve

$$\hat{\sigma} = \arg \min_{\sigma} J(\sigma)$$  \hspace{1cm} (H.14)

for

$$J(\sigma) = \sqrt{\frac{1}{K} \sum_{k=1}^K (v_k - \hat{v}_k(\sigma))^2}.$$  \hspace{1cm} (H.15)
where for concreteness we have set all the weights to $1/K$. This can be solved easily by methods like the golden section search method, described in the next section. The accuracy of this method is linear in the number of iterations. Other methods, described in Press et al. (2007), converge more rapidly but are not as simple.

1-dimensional minimization problems require as a starting point a bracketed solution. Let $I = [a, b] \subset \mathbb{R}$ and $J : I \rightarrow \mathbb{R}$ be a continuous function. A bracketed solution for $J$ is a triplet, $\Sigma = (\sigma_L, \sigma_M, \sigma_U)$ with $\sigma_M \in (\sigma_L, \sigma_U) \subset I$ such that $J(\sigma_M) < J(\sigma_L)$ and $J(\sigma_M) < J(\sigma_U)$. The bracketing interval is $[\sigma_L, \sigma_U]$. This has width $\sigma_U - \sigma_L$.

The condition ensures that a minimum lies in the interior of the bracketing interval. On each iteration of the algorithm another, smaller, bracketing interval is found. The minimum is constrained to lie within intervals whose width tends to zero.

**H.2.1 The golden section search algorithm**

Let $J : I \rightarrow \mathbb{R}$ be a continuous function. We want to find a local minimum for $J$ in the interior of $I$. Suppose we are given an initial bracketed solution $(\sigma_L, \sigma_M, \sigma_U)$ for $J$. At each iteration we start with a bracketed solution of some width and from it find another bracketed solution whose width is strictly less.

Let $\varphi = (3 - \sqrt{5})/2 \approx 0.38197$. This is related to the golden ratio $\phi = 2 - \varphi$ from which the method gets its name. The method proceeds as follows.

Suppose on the $i$th iteration we are given a bracketing solution $\Sigma^i = (\sigma_L, \sigma_M, \sigma_U)$. We construct a new point $\sigma$. If $\sigma_M - \sigma_L > \sigma_U - \sigma_M$ set $\sigma$ to be

$$\sigma = \sigma_M - \varphi (\sigma_M - \sigma_L) \in [\sigma_L, \sigma_M]; \quad (H.16)$$

if $\sigma_U - \sigma_M > \sigma_M - \sigma_L$ set $\sigma$ to be

$$\sigma = \sigma_M + \varphi (\sigma_U - \sigma_M) \in [\sigma_M, \sigma_U]. \quad (H.17)$$

The new point $\sigma$ is a fraction $\varphi$ out from the point $\sigma_M$ towards the boundary of the larger interval.

If $J(\sigma) > J(\sigma_M)$ set the new bracketing solution $\Sigma^{i+1}$ to be $\Sigma^{i+1} = (\sigma_L, \sigma, \sigma_U)$. If $J(\sigma) < J(\sigma_M)$ set the new bracketing solution to be $\Sigma^{i+1} = (\sigma_M, \sigma, \sigma_U)$.

Each new interval is a fraction, $1 - \varphi = 1/\phi$, of the width of the previous interval. The choice of $\varphi$ as the reduction ratio is optimal in a best-worst-case sense.

**H.2.2 Implementation**

Code for the golden section search method is given in Figure H.7 (spreadsheet app_minimize_golden.xls). The procedure `GoldenSection()` is set up to find a minimum for the function

$$f(x) = (x - a)(xN(x) - b) \quad (H.18)$$

where $N(x)$ is the distribution function for the normal distribution mean $\mu$ and variance $\sigma^2$. It uses slightly modified but essentially the same Enums as the root-finding code. The method converges rapidly.

A procedure `CheckFtol()` is used to test the tolerance condition on the value of $f$. This is because if the tolerance condition is met, `GoldenSection()` has to make sure that it returns the value of $x$ for the minimum value of $f(x)$ it has found.
Private Sub GoldenSection(ByVal root As Double, ByVal fval As Double, ByVal et As ExitT, _
    fdat As FnData, pdat As ProbData)
    Const MAX_ITERATIONS As Long = 100
    Const R_TOLERANCE As Double = 0.000001 'tolerance on the root
    Const F_TOLERANCE As Double = 0.0000000000001 'tolerance on the target
    Const phi As Double = 0.381966011250105 '= 1 - 1/goldenratio
    et = normalexit
    Dim lx As Double: lx = pdat.lb 'Initial lower bound
    Dim mx As Double: mx = pdat.mb 'Initial mid point
    Dim ux As Double: ux = pdat.ub 'Initial upper bound
    Dim mf As Double: mf = fn(mx, fdat) 'fn value at mid point
    Dim i As Long
    For i = 1 To MAX_ITERATIONS Step 1
        If mx - lx > ux - mx Then 'LHS is largest interval
            x = mx - phi * (mx - lx)
            xf = fn(x, fdat)
            If CheckFtol(x, xf, mx, mf, F_TOLERANCE) Then Exit Sub
            If xf > mf Then 'mf is still smallest
                lx = x
                xf = mf
            Else 'Have new smallest
                ux = mx
                mx = x
                mf = xf
            End If
        Else 'RHS is largest interval
            x = mx + phi * (ux - mx)
            xf = fn(x, fdat)
            If CheckFtol(x, xf, mx, mf, F_TOLERANCE) Then Exit Sub
            If xf > mf Then 'mf is still smallest
                ux = x
                xf = mf
            Else 'Have new minimum
                lx = mx
                mx = x
                mf = xf
            End If
        End If
    Next i
    If Abs(lx - ux) < R_TOLERANCE Then Exit Sub
    et = IterationsExceeded
End Sub

Private Function CheckFtol(ByVal x As Double, ByVal xf As Double, _
    mx As Double, mf As Double, Ftol As Double) As Boolean
    CheckFtol = Abs(xf - mf) < Ftol
    If CheckFtol And xf > mf Then 'Need to reset x and xf
        x = mx
        xf = mf
    End If
End Function

Figure H.7 GoldenSection()
Adapter pattern  A design pattern in which a wrapper object exists just to alter the interface of the object it contains.

Aggregation  An object is an aggregate of another if it can be regarded as associated with it but its existence is independent of it.

Association  Two objects are associated if one is essentially an attribute of the other.

Attribute  An object is an attribute of another if there is a sense in which it forms part of its state.

Base class  In this book, the interface that a factory-creatable object is created as an instance of.

Bidirectional association  A situation in which two objects have references to one another, allowing them to communicate.

Binding  Resolving the type of a polymorphic object. Late binding occurs when the type cannot be determined until run-time; an object is early bound if its type can be determined at compile time.

Bodge  An expedient programming technique often encountered in practical applications.

Cast  To convert from one type to another.

Client  Someone or something, such as another application or procedure, that uses your code.

Cohesion  A situation in which an object has a tight well-defined responsibility that is its and its alone.

Collection  A VBA container that behaves like a set.

Composited object  An object contained in another as a data member.

Composition  A pattern in which one object contains another (whose lifetime is bounded by the lifetime of the containing object).

Concept code  Demonstration code that works, illustrating some functionality, but which is nowhere near industrial strength.

Conforming object  An object conforms to an interface if it Implements it.

Constructor  In VBA the nearest thing is the Class_Initialize() Sub that is called when an object is created.

Container  An entity that can store data, allowing access in ways that optimizes some aspect of performance.

Contract  The expectations an object places upon its clients for the object to be able to be used correctly.
Copy and conversion constructors  Special constructors uses to create an object from another of the same or different type. Not applicable to VBA.

Copy assignment  Overloading the assignment operator, “=”, to tailor what happens when one object is assigned to another. Not applicable in VBA.

Coupling  Different parts of an application requiring a knowledge of what the other parts do or how they do it. A coupled application is a poorly designed application.

Ctor  Shorthand for “constructor”.

Data member  A Private data member often forming part of an object’s state.

Decorator pattern  A design pattern where an object contains a composited object of the same type.

Decoupled  The state in which objects make no assumptions about functionality elsewhere in an application.

Derived objects  An object that inherits the functionality of another. Not applicable in VBA.

Design patterns  Standard ways of setting up relationships between objects to achieve some goal. VBA is limited in the extent that design patterns can be implemented.

Destructor  A special method called when an object goes out of scope or is deleted. In VBA, the closest thing is the Class_Terminate() Sub.

Dictionary  A VBA library object that behaves like an associative container, with items referable to with keys.

Dispatch  To forward on a request to another object that can deal with it.

Dtor  Shorthand for “destructor”.

Encapsulation  Enforcing a barrier between what happens inside an object and what the outside world sees of it.

Environment  The context in which an application is run; the settings that make up the context.

Evil  In programming, something that you must never never do, except when you decide you have to.

Exception safety  In VBA, making sure that your code exits gracefully even if code elsewhere goes wobbly.

Expose  For an object to make parts of itself available to clients.

Façade pattern  A design pattern in which an object exists to provide a unified interface for a set of composited objects.

Factory pattern  A design pattern that lets you create objects essentially polymorphically.

Friend  Data or procedures declared as Friend have scope limited to their project.

Front-end  Here, an Excel spreadsheet used as a user interface to a VBA application.

Functor  In C++ an object with overloaded operator(). In VBA we define a functor to be an object with a single application method, perhaps named run() (the object may also have structural methods such as a SetValues() method).

Getter  A member procedure or Property that returns the value of a (Private) data member.

GoTo  An evil command that should be used only in error trapping.

Handle  Used here to mean an identifier handed out by one object to another so that the receiving object can identify itself, or the particular request it made, in future transactions.
Honte  The proper way of doing something, even if it is worse.
Idiom  A standard mechanism, here a coding technique, for achieving some goal.
Implementation  Private methods and data.
Indirection  In this context, achieving a programming objective by creating an object to be responsible for achieving it.
Industrial strength  Code that is ready to go live, released to clients, usually as part of, or an extension to, a larger application, with a decent chance of not falling over now or in the future.
Inheritance  Broadly, the ability of an object to extend its functionality by taking on the attributes of another object. In VBA, the corresponding concept is that of conforming to an interface.
Initialization  Giving an object or variable an initial value at the point of its creation.
Instantiation  Creating an object as an instance of some type, properly in a Set-New statement.
Interface  An object’s Public and Friend member procedures.
Interfaces  The VBA mechanism that enables polymorphic code design.
Invariant  A property of the data defining the state of an object which must be satisfied for the object to be in a valid state.
Invocation chain  A series of objects that wrap the underlying application, fired when an application starts up.
Literal  A hard-wired in numerical or String value.
Magic number  A number hard-wired into code. It is better to replace magic numbers with Const variables.
Member function  A procedure belonging to an object.
Message  Calling one of an object’s interface functions is sometimes referred to as sending it a message or a request.
Meta-class data  Information common to all objects of a type.
Method  See Member function.
Mix-in interface  An interface design to endow conforming objects with specific types of pick-and-mix functionality.
Not-first-time idiom  See Static initialization idiom.
Object declaration  Declaring an object reference with a Dim statement (or similar).
Object definition  The definition of an object in a class module.
Object-oriented programming  A programming idiom to make effective use of objects.
Operator overloading  A way of extending the scope of primitive operators in a language so that they can act on user-defined objects. Not possible in VBA.
Path  In a Monte Carlo method, a set of states representing a single realization of a stochastic process through time.
Plain old data  An object whose data is Public that has no member functions.
Polymorphic hierarchy  An interface object and all the objects conforming to it.
Polymorphism  The ability of one object to be substitutable for another, in a manner transparent to the client.
**Primitive data or type** Types defined as part of the language; Double, Long, *et cetera*.

**Private** Data and procedures declared to be Private have scope confined to their module.

**Procedure** A Function or Sub.

**Profile** To establish quantitatively where code spends its execution time.

**Programming to an interface** Deciding beforehand what the interface to an object is to be, and only then programming it up.

**Property** In VBA a special category of member procedure that facilitates setting and getting of data members.

**Psychic connection** See *Telepathy*.

**Public** Data and procedures declared to be Public are accessible to all modules, including those outside their immediate project.

**RAII (resource acquisition is initialization)** Using a constructor to acquire resources that are later released in the corresponding destructor.

**Re-factor** To rewrite and restructure a code base after a period of maintenance to make it once more clean and elegant. You wish.

**Reference counting** Tracking how many references there are in an application to an instantiation of an object. In VBA, keeping track of how many instantiations there are.

**Registration and call-back** A design pattern in which an object makes itself known to another, and enables it to execute some part of its interface.

**Reification** (a) To depersonalize; to regard a worker solely as an economic object. (b) The concrete material representation in a computer program of an abstract concept or mathematical expression.

**Request** See *Message*.

**Scalable code** An application that requires no new coding when additional conforming classes are added to it.

**Scope** Where in a program a name is recognized.

**Scripting run-time library** A Microsoft library that allows access to certain objects such as *Dictionary* and the *FileSystemObject* object.

**Select** A statement that should be avoided whenever sensible and replaced by polymorphic code.

**Sentinel** A data member, perhaps a *Boolean*, whose value determines whether an operation is allowed or not.

**Serialization** Representing an object in a form that can be read to and from file.

**Setter** A member procedure of Property that sets the value of a (Private) data member.

**Singleton pattern** A design pattern that ensures that one and only one instantiation of an object can exist. Can be mimicked using meta-classes.

**Slice** In a Monte Carlo method, a collection of states of the underlying variables, all for the same time.

**Stack** The sequence of nested procedure calls in existence at any one time.

**State** The state of an object is defined by the (valid) values of its data members.

**Static initialization idiom** A pattern employing sentinels that ensures that Static variables in a procedure have their values set just once.
Strategic pattern  A design pattern that decouples knowledge of an algorithm from its clients.

Striding  A technique of indexing a 1-dimensional array so that it mimics the behaviour of a higher dimensional array.

Strong typing  In excess of 80 words a minute. Declaring all variables with as restrictive a type as possible.

Stub  A procedure containing only the signature and End lines.

Syntactic sugar  A feature that adds no additional functionality apart from increasing the readability of code. Anything that isn’t machine code.

Telepathy  The ability of one part of an application to know about another part, despite there being no contact between them. An ill-advised programming technique.

Template pattern  A design pattern in which parts of an algorithm are specified polymorphically in derived classes.

Tracking  Endowing each instantiation of an object with a unique identifier, and keeping track of how many instantiations are in existence.

Tricky  Code that tries to go faster by being more complicated than it should be.

Wrapper object  In VBA, an object whose role is to manage the functionality of a composite object.

Yukky  Worse even than yucky. A style of programming that fails to compensate in syntax and construction what it lacks in spelling.
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application programming interface</td>
</tr>
<tr>
<td>ATM</td>
<td>At the money</td>
</tr>
<tr>
<td>CEV</td>
<td>Constant elasticity of variance</td>
</tr>
<tr>
<td>CIR</td>
<td>Cox, Ingersoll and Ross</td>
</tr>
<tr>
<td>CN</td>
<td>Crank-Nicolson</td>
</tr>
<tr>
<td>CV</td>
<td>Control variate</td>
</tr>
<tr>
<td>DLL</td>
<td>Dynamic-link library</td>
</tr>
<tr>
<td>DP</td>
<td>Decimal point</td>
</tr>
<tr>
<td>EEB</td>
<td>Early exercise boundary</td>
</tr>
<tr>
<td>FD</td>
<td>Finite difference</td>
</tr>
<tr>
<td>GBM</td>
<td>Geometric Brownian motion</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated development environment</td>
</tr>
<tr>
<td>IID</td>
<td>Independent identically distributed</td>
</tr>
<tr>
<td>I/O</td>
<td>Input and output</td>
</tr>
<tr>
<td>IS</td>
<td>Importance sampling</td>
</tr>
<tr>
<td>ITM</td>
<td>In the money</td>
</tr>
<tr>
<td>LD</td>
<td>Low discrepancy</td>
</tr>
<tr>
<td>LHS</td>
<td>Left-hand side</td>
</tr>
<tr>
<td>LMM</td>
<td>Libor market model</td>
</tr>
<tr>
<td>LSLS</td>
<td>Longstaff and Schwartz least squares Monte Carlo</td>
</tr>
<tr>
<td>MC</td>
<td>Monte Carlo</td>
</tr>
<tr>
<td>NIG</td>
<td>Normal inverse Gaussian</td>
</tr>
<tr>
<td>O-U</td>
<td>Ornstein-Uhlenbeck</td>
</tr>
<tr>
<td>OLS</td>
<td>Ordinary least squares</td>
</tr>
<tr>
<td>OOP</td>
<td>Object oriented programming</td>
</tr>
<tr>
<td>OTM</td>
<td>Out of the money</td>
</tr>
<tr>
<td>PDB</td>
<td>Pure discount bond</td>
</tr>
<tr>
<td>PDE</td>
<td>Partial differential equation</td>
</tr>
<tr>
<td>POD</td>
<td>Plain old data</td>
</tr>
<tr>
<td>PSOR</td>
<td>Projected successive over-relaxation</td>
</tr>
<tr>
<td>RAIi</td>
<td>Resource acquisition is initialization</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Full Form</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------</td>
</tr>
<tr>
<td>RHS</td>
<td>Right-hand side</td>
</tr>
<tr>
<td>RMSE</td>
<td>Root mean square error</td>
</tr>
<tr>
<td>SD</td>
<td>Standard deviation</td>
</tr>
<tr>
<td>SDE</td>
<td>Stochastic differential equation</td>
</tr>
<tr>
<td>Se</td>
<td>Standard error</td>
</tr>
<tr>
<td>SF</td>
<td>Significant figures</td>
</tr>
<tr>
<td>SOR</td>
<td>Successive over-relaxation</td>
</tr>
<tr>
<td>SV</td>
<td>Stochastic volatility</td>
</tr>
<tr>
<td>SVD</td>
<td>Singular value decomposition</td>
</tr>
<tr>
<td>UDT</td>
<td>User Defined Type</td>
</tr>
<tr>
<td>VaR</td>
<td>Value at risk</td>
</tr>
<tr>
<td>VBA</td>
<td>Visual basic for applications</td>
</tr>
<tr>
<td>VBE</td>
<td>Visual basic editor</td>
</tr>
<tr>
<td>VBIDE</td>
<td>Visual basic integrated development environment</td>
</tr>
<tr>
<td>VG</td>
<td>Variance gamma</td>
</tr>
<tr>
<td>ZLH</td>
<td>Zero lock-out Heston</td>
</tr>
</tbody>
</table>
Coding, Notational, and Typographical Conventions

Here are gathered some of the typographical, notation, coding conventions and standards adopted throughout this book.

Notation

- $A:B$ $B$ implements the interface object $A$, or, $B$ is a method or data member of object type $A$.

Coding conventions

- Long/Double are used by default rather than Integer/Single.
- The insides of Functions, Subs, Type declarations, For-Next statements, If-Else-End If statements, et cetera, are indented.
- Variables are declared as close as possible to their point of first use. Strongest possible typing is always used. Where possible variables are declared and initialized on the same line.
- Subs are called using the key word Call.
- Names of objects used as interfaces are prefixed by “I”; objects implementing an interface have names prefixed by the name of the interface.
- Names of objects used to hold meta-data for an interface are prefixed by “M”.
- Data members will always be Private; the names of Private data members always end in underscores.
- The scope of methods is always explicit and as limited as possible; Friend is usually used instead of Public.
- When program logic dictates that a variable’s scope should be restricted to lie within a procedure, Static variables will not be avoided, even though they are expensive.
- Hungarian naming conventions are not used.

Typographical conventions

- A fixed space font like this is used for code; and a sans serif font like this for text quoted from the front-end.
- Code examples are frequently compacted to remove blank lines and other white space that, for clarity, would normally be present in code. This is purely for typographical reasons.
• When displayed in the book lines of code may include punctuation marks as end-of-line-comments.
• In every code module there is always an Option Explicit statement. However to save space it is often omitted from displayed code.
• References to text equations are in brackets, (1.1a) et cetera; references to lines in figures are not bracketed.
This index lists, chapter by chapter, the illustrative code used in this book, with a brief description of each application. The code can be found on the accompanying CD. On the CD each chapter has a directory containing the code that supports it. For compatibility, spreadsheets are Excel 2003 .xls files. For brevity, the .xls extension is dropped from workbook names listed below.

Disclaimer: The code is intended for illustrative use. It is presented “as is”. Although every effort has been made to remove errors it is inevitable that some may remain as yet undetected. No liability can be accepted for any errors or their consequences.

Chapter 3  Procedural Programming: Level 1
MC_example_v0_names: The same as MC_example_v0 but doing I/O to the front-end using range names.
Cpp_example_v0: A C++ Monte Carlo in the same style as MC_example_v0. Used in a timing comparison.
MC_example_v1: A version of MC_example_v0 using some functions.

Chapter 4  Validation and Error Handling: Level 2
MC_example_v2: Figures 4.1–4.10. Adds in error handling to MC_example_v1. Has a UDT. Pulls as much functionality as possible into functions.

Chapter 5  Introducing Objects: Level 3
9b_ref_counter: Figure 5.3. Illustrates how a form of reference counting can be implemented in VBA.
4_timer: Figure 5.5. A stopwatch class used to time code execution.
MC_example_v3a: Figures 5.7–5.14. A modification of MC_example_v2 where functionality has been transferred out of functions and into objects.
MC_example_v3b: Figures 5.15–5.17. Introduces into MC_example_v2 a wrapper object to manage instantiating and destroying objects.

Chapter 6  Polymorphism and Interfaces: Level 4
TimerTimer: Table 6.1. Times the execution of three different stopwatch objects.
**Run_timer**: Table 6.2. Times the execution of the application `Run()` function, comparing it to a plain `Function` call.

**Poly_Timer**: Figures 6.7–6.8. Implements three stopwatches polymorphically.

**MC_example_v4**: Figures 6.2–6.23. Makes polymorphic certain application objects in `MC_example_v3b`. Adds in a further layer to the invocation chain to separate out I/O from the Monte Carlo functionality.

**Chapter 7** A Slice-Based Monte Carlo

**MC_example_v4b**: Figures 7.1–7.11. An adaptation of `MC_example_v4` to turn it into a slice-based Monte Carlo.

**Chapter 8** An Embryonic Factory: Level 5

**MC_example_v5**: Figures 8.5–8.15. Constructs a non-polymorphic factory to handle object creation. Relocates I/O to integrate it in with the factory.

**Chapter 9** Input and Output to File in VBA

**MC_example_v5a**: Figures 9.1 and 9.2. Adds an object to `MC_example_v5` to output to a `TextStream` file object.

**1_File**: Figures 9.3–9.8. An illustration of input and output to random access and sequential files using VBA intrinsic file handling functions.

**Chapter 10** Valuing a Book of Options

**MC_example_v5b**: Figures 10.1–10.25. An application to read in sets of option specifications (from file or spreadsheet) and output results (to file or spreadsheet).

**MC_example_v5b_times**: Tables 10.1 and 10.2. Same as `MC_example_v5b`, but slightly modified to produce timings for I/O.

**Chapter 11** The VBA Object Library and a Simple Polymorphic Factory

**13_factory**: Figures 11.2–11.8. Demonstrates a simple fully polymorphic factory for an application to compute the value of $\pi$ by various methods.

**Chapter 12** A Fully Polymorphic Factory: Level 6

**MC_example_v6**: Figures 12.1–12.17. A fully polymorphic factory in a Monte Carlo application. It is able to create objects from a variety of interface types.

**Chapter 13** A Semi-Polymorphic Factory: Meta-Classes

**MC_example_v6a**: Figures 13.1–13.19. A non-polymorphic factory that could be used if the techniques used to construct the fully polymorphic factory in `MC_example_v6` are unavailable. Introduces meta-classes.

**Chapter 14** Performance and Cost in VBA

**Timing0_arithmetic**: Tables 14.2–14.14. Times arithmetic operations and low-level VBA statements and constructions

Timing2_assignment: Table 14.17. Times for casting from one primitive type to another.

Timing3_statics: Table 14.20. Timings for generating uniforms and normal variates by various methods.

Chapter 15  Level and Performance

MC_example_v1a: Figures 15.1–Table 15.4. A level 0/1 Monte Carlo test harness that times variations on a base case.

Barrier examples: Table 15.6. There are three spreadsheets in this set, listed in Table H.17.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MC_v0_barrier</td>
<td>Level 0 barrier option code</td>
</tr>
<tr>
<td>MC_v1_barrier</td>
<td>Level 1 barrier option code</td>
</tr>
<tr>
<td>MC_v2_barrier</td>
<td>Level 2 barrier option code</td>
</tr>
</tbody>
</table>

(H.17)

They adapt level 0, 1 and 2 spreadsheets to value barrier options.

Chapter 16  Evolution and Data Structures

Timing_state_evolution: Tables 16.2 and 16.3. Times the evolution of different state representations.

MC_v1_q1_V: Figure 16.8 and Table 16.5. Evolves and times different implementations of a 1-factor slice Monte Carlo.

MC_v1_q3_strideObj: Figure 16.6 and Table 16.7. A striding implementation of a 3-factor model with the striding array and its indexing wrapped in an object.

Structureless state: Table 16.6. A set of three spreadsheets used to implement and time various array-based 3-factor Monte Carlo implementations. The sheets are listed in Table H.18. The names are prefixed by “MC_v1_q3_”.

<table>
<thead>
<tr>
<th>Name (suffix)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3v</td>
<td>Three 1-dimensional arrays</td>
</tr>
<tr>
<td>array</td>
<td>A single 2-dimensional array</td>
</tr>
<tr>
<td>stride</td>
<td>A striding representation of a 2-dimensional array</td>
</tr>
</tbody>
</table>

(H.18)

Container structures: Tables 16.7–16.9. A set of spreadsheets used to time various slice-state container combinations. These are summarised in Table H.19. Names are prefixed by “MC_v1_q3_”; the table shows only the suffixes.

<table>
<thead>
<tr>
<th>Inner container</th>
<th>Outer container</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array:</td>
<td>Dictionary Collection Variant Array</td>
</tr>
<tr>
<td>UDT:</td>
<td>DictArray CollArray VarArray  ArrayUDT</td>
</tr>
<tr>
<td></td>
<td>– – –</td>
</tr>
<tr>
<td>POD</td>
<td>ArrayPODarray</td>
</tr>
<tr>
<td>UDT:</td>
<td>– – VarPODudt ArrayPODudt</td>
</tr>
<tr>
<td>Doubles:</td>
<td>DictPOD CollPOD VarPOD ArrayPOD</td>
</tr>
<tr>
<td></td>
<td>CollPODkeyed</td>
</tr>
</tbody>
</table>

(H.19)

MC_v1_q3_WrappedArrayUDT: Figure 16.9 and Table 16.10. Wraps an array slice with UDT states in an object. Provides times for this implementation.
Chapter 17  Wiener Sample Paths and Antithetic Variates

MC_AverageRate_anti: Figure 17.2 and 17.3. A simple antithetic variate implement-
   ation.

MC_AverageRate_anti_timings: Figure 17.1; Tables 17.1 and 19.2. Timings and effi-
   ciency gains for the antithetic variate method.

Chapter 18  The Wiener Process and Stratified Sampling

MC_AverageRate_strat: Figures 18.1–18.6. Fully stratified sampling with an average
   rate option.

MC_AverageRate_strat_timings: Tables 18.1 and 18.3. Timings and efficiency gains
   for fully stratified sampling.

Chapter 19  Low-Discrepancy Sampling

sobol_40: Figure 19.1. Generates and plots Sobol’ sequence numbers.

MC_AverageRate_LD: Figure 19.2–Table 19.3. Using low-discrepancy sampling with
   stratification. Includes timings.

Chapter 20  Variance Reduction with Control Variates

HestonExplicit: Section 20.3.2. Exact solutions for vanilla options in the Heston model,
   computed by quadrature.

Chapter 21  Implementing Control Variates

MC_AverageRate_CV: A basic CV illustration. Valuing an arithmetical average rate
   option with up to three control variates. The option itself computes CV values.

MC_AverageRate_benchmark: Benchmarks an explicit formula for a part-way geo-
   metric average rate option against an MC implementation. (Appendix B.)

MC_AverageRate_bank_CV: Figures 21.1–21.6. Illustrates CVs computed in a bank,
   outside of the objects that may or may not choose to use them.

MC_AverageRate_CV_timings: Table 21.1. A variation on MC_AverageRate_CV used
   to compute efficiency gains.

MC_delta_GBM: Table 21.2. Illustrates the principle of using a delta control variate
   with an average rate option in GBM.

MC_exotic_CV: Table 21.3. A control variate valuation of an exotic European option
   with a quadratic payoff. The option computes its CV values.

Chapter 22  Extreme Options and Importance Sampling

MC_AverageRate_IS: Figures 22.1–22.10. An implementation of importance sampling.

MC_AverageRate_IS_timings: Tables 22.1–22.7. A spreadsheet used to compute
   timings for the tables in the chapter.

Chapter 23  Combining Variance Reduction Methods

MC_speedups_all: Tables 18.2 and 18.4; Figures 23.1–23.4; Tables 23.1–23.9. A
   spreadsheet combining all the main variance reduction methods previously met.

Chapter 25  Discretization Methods

MC_convergence_GBM: Table 25.1 panel (a); Figure 25.1. Benchmarking discretization
   schemes in the GBM model.
MC_convergence_OU: Table 25.1 panel (b). Benchmarking discretization schemes in the Vasicek model.

Chapter 26 Applications to Models
MC_convergence_CIR_simul: Figures 26.1–26.3; Table 26.1. Exploring the effectiveness of discretization schemes in the CIR model.
MC_convergence_bond_simul: Table 26.2; Figure 26.4. Comparisons of methods to compute discount factors in the Vasicek model.

Chapter 27 Valuation in the Heston Model
MC_convergence_Heston_simul: Testing weak convergence in the Heston model with mean and variance functions.
MC_convergence_Heston_simul_call: Figure 27.1. Investigating weak convergence with a European call option.
MC_Heston_auxiliary_CV: Tables 27.2–27.9; Figures 27.2 and 27.3. Using GBM auxiliary model CVs with Heston.
MC_Heston.CV_factor: Slice-based Heston CVs with an OOP factory.

Chapter 28 Valuing American and Bermudan Options

Chapter 29 Estimating the Early Exercise Boundary
MC_American_basis: Figure 29.1. Produces plots of basis functions.
MC_American.CV.EExB: Figures 29.3 and 29.4; Tables 29.2 and 29.3; Figures 31.5, 31.6, 31.8 and 31.9, panels (a), (b) and (c); Tables 31.2 and 31.3, panel (a). Computes estimates of the early exercise boundary.

Chapter 30 The Plain LSLS Method
MC_American: Figure 29.2; Tables 30.1–30.5; Figures 30.1–30.7. Implements plain LSLS.

Chapter 31 Control Variates and the LSLS Method
MC_American.CV_one_pass_set_barrier: Figure 28.4; Tables 29.4 and 29.5; Tables 31.1 and 31.5; Figure 31.10. Computes values of mimicking barrier options.
MC_American.CV_one_pass: Figures 31.1–31.4, 31.7 and 31.9 panels (d), (e) and (f); tables 31.3 panels (b) and (c), and 31.4. Full implementation of the CV enhanced LSLS method.
MC_American Richardson: Uses Richardson extrapolation in an American option object containing three composited Bermudan option objects. Has an improved CV manager that constructs a bank of CVs from references supplied by option objects.

Appendix B Some Option Formulae
MC_AverageRate.benchmark: Benchmarks a Monte Carlo application against explicit solutions for geometric average rate options.
**MC_exotic_benchmark**: Implements an explicit valuation formula for the exotic option described in Chapter 20.

**CompoundOptionExplicit**: An implementation of a compound option formula for the value of a Bermudan option used as a CV for the American put.

---

**Appendix C**  
The Utility Code Modules

**LibraryProcedures**: A workbook containing a complete set of utility library modules.

**complex**: An implementation of a complex number class

**quadrature**: Implementations of the trapezium, extended trapezium and Simpson quadrature methods.

---

**Appendix D**  
Running DLLs from VBA

**DLLExample**: A folder containing a workbook, dll_test.xls, that runs functions from a .dll file, testdll.dll, created in C++. The complete set of files in the folder are listed in Table H.20.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dll_test.xls</td>
<td>The workbook using the dll</td>
</tr>
<tr>
<td>testdll.dll</td>
<td>The .dll itself</td>
</tr>
<tr>
<td>dllmain.cpp</td>
<td>The C++ implementation behind the .dll</td>
</tr>
<tr>
<td>dllmain.h</td>
<td>The C++ header file accompanying dllmain.cpp</td>
</tr>
<tr>
<td>testdll.dev</td>
<td>A DevCpp project file used to compile dllmain.cpp</td>
</tr>
<tr>
<td>libtestdll.def</td>
<td>The .dll .def file</td>
</tr>
</tbody>
</table>

---

**Appendix E**  
Object-Oriented Programming

**OOP_example**: Figures E.1–E.4. Four ways to implement a stopwatch. Motivates the introduction of objects.

**PolyFunct**: Figures E.5 and E.6. Implements a functor interface as an illustration of polymorphism.

**9c_singleton**: Figures E.11 and E.12. Illustrates a module based singleton pattern.

---

**Appendix F**  
A Yukky Level 0 Monolithic Lattice Implementation

**Lattice_application_v0**: Figure F.1–Table F.1, and Figure F.11. An implementation of a basic trinomial lattice method. Does not incorporate pruning.

**Lattice_application_v0_faster**: Figure F.10. A trinomial lattice method that incorporates pruning.

---

**Appendix G**  
A Level 1 Crank–Nicolson PDE Implementation


---

**Appendix H**  
Root-Finding and Minimization Algorithms

**app_roots_bisection**: Figures H.1 and H.2. Finds a root of a function using the bisection algorithm.
**app_roots_false:** Figure H.3. Implements the method of false position to find a root of function.

**app_roots_NR:** Figure H.6. An implementation of the Newton–Raphson root-finding method.

**app_roots_Ridders:** Figure H.5. An implementation of Ridders’ method for finding a root of a univariate function.

**app_minimize_golden:** Figure H.7. Finds the minimum of a function of one variable by the golden section search method.
This page intentionally left blank
Index to Spreadsheets

1_file.xls, 145, 624
4_timer.xls, 623
9b_ref_counter.xls, 623
9c_singleton.xls, 563, 565, 628
13_factory.xls, 184, 624

app_minimize_golden.xls, 611, 629
app_roots_bisection.xls, 604, 606, 628
app_roots_false.xls, 606, 629
app_roots_NR.xls, 610, 629
app_roots_Ridders.xls, 608, 629

CN_pde_SOR_v2.xls, 598, 628
CN_pde_v1.xls, 585, 628
complex.xls, 531, 540, 628
CompoundOptionExplicit.xls, 529, 628
Cpp_example_v0.xls, 36, 623
dll_test.xls, 545, 628

HestonExplicit.xls, 329, 626

Lattice_application_v0.xls, 571, 628
Lattice_application_v0_faster.xls, 578, 628
Lattice_Bermudan_benchmark.xls, 452, 627
LibraryProcedures.xls, 17, 22, 531, 628

MC_American.xls, 475, 477, 479, 627
MC_American_basis.xls, 627
MC_American_CV_EExB.xls, 470, 499, 503, 505, 627
MC_American_CV_one_pass.xls, 494, 499, 503, 505, 627
MC_American_CV_one_pass_set_barrier.xls, 454, 473, 497, 507, 509, 627
MC_American_Richardson.xls, 627
MC_AverageRate_anti.xls, 285, 626
MC_AverageRate_anti_timings.xls, 285, 626
MC_AverageRate_bank_CV.xls, 333, 626
MC_AverageRate_benchmark.xls, 526, 627
MC_AverageRate_CV.xls, 343, 626
MC_AverageRate_CV_timings.xls, 339, 626
MC_AverageRate_IS.xls, 353, 626
MC_AverageRate_IS_timings.xls, 362, 626
MC_AverageRate_LD.xls, 312, 333, 626
MC_AverageRate_Strat.xls, 298, 626
MC_AverageRate_Strat_timings.xls, 626
MC_convergence_Bond_simul.xls, 627
MC_convergence_CIR_simul.xls, 420, 627
MC_convergence_GBM.xls, 413, 626
MC_convergence_Heston_simul.xls, 627
MC_convergence_Heston_standard.xls, 436, 627
MC_convergence_OU.xls, 414, 627
MC_delta_GBM.xls, 342, 626
MC_example_v0.xls, 36, 623
MC_example_v0_names.xls, 29, 623
MC_example_v1.xls, 25, 623
MC_example_v1a.xls, 249, 625
MC_example_v2.xls, 39, 48, 623
MC_example_v3a.xls, 67, 623
MC_example_v3b.xls, 76, 623
MC_example_v4.xls, 81, 102, 105, 624
MC_example_v4b.xls, 107, 624
MC_example_v5.xls, 133, 624
MC_example_v5a.xls, 624
MC_example_v5b.xls, 153, 624
MC_example_v5b_times.xls, 174, 624
MC_example_v6.xls, 198 et cetera, 624
MC_example_v6a.xls, 211, 624
MC_exotic_benchmark.xls, 528, 628
MC_exotic_CV.xls, 343, 626
MC_Heston_auxiliary_CV.xls, 438, 627
MC_Heston_CV_factory.xls, 627
MC_speedups_all.xls, 302, 372, 377, 494, 626
MC_v0_barrier.xls, 255, 625
MC_v1_barrier.xls, 255, 625
MC_v1_q1_V.xls, 625
MC_v1_q3_3v.xls, 625
MC_v1_q3_array.xls, 625
MC_v1_q3_ArrayPOD.xls, 625
MC_v1_q3_ArrayPODarray.xls, 625
<table>
<thead>
<tr>
<th>Spreadsheet Name</th>
<th>Page Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArrayPODudt.xls</td>
<td>625</td>
</tr>
<tr>
<td>ArrayUDT.xls</td>
<td>625</td>
</tr>
<tr>
<td>CollArray.xls</td>
<td>625</td>
</tr>
<tr>
<td>CollPOD.xls</td>
<td>625</td>
</tr>
<tr>
<td>CollPODkeyed.xls</td>
<td>625</td>
</tr>
<tr>
<td>DictArray.xls</td>
<td>625</td>
</tr>
<tr>
<td>DictPOD.xls</td>
<td>625</td>
</tr>
<tr>
<td>stride.xls</td>
<td>625</td>
</tr>
<tr>
<td>strideObj.xls</td>
<td>625</td>
</tr>
<tr>
<td>VarArray.xls</td>
<td>625</td>
</tr>
<tr>
<td>VarPOD.xls</td>
<td>625</td>
</tr>
<tr>
<td>VarPODudt.xls</td>
<td>625</td>
</tr>
<tr>
<td>WrappedArrayUDT.xls</td>
<td>625</td>
</tr>
<tr>
<td>OOP_example.xls</td>
<td>628</td>
</tr>
<tr>
<td>PolyFunct.xls</td>
<td>628</td>
</tr>
<tr>
<td>Poly_Timer.xls</td>
<td>624</td>
</tr>
<tr>
<td>quadrature.xls</td>
<td>531, 542, 628</td>
</tr>
<tr>
<td>Run_timer.xls</td>
<td>624</td>
</tr>
<tr>
<td>sobol_40.xls</td>
<td>310, 626</td>
</tr>
<tr>
<td>TimerTimer.xls</td>
<td>81, 623</td>
</tr>
<tr>
<td>Timing_state_evolution.xls</td>
<td>267, 625</td>
</tr>
<tr>
<td>Timing0_arithmetic.xls</td>
<td>624</td>
</tr>
<tr>
<td>Timing1_procedures.xls</td>
<td>624</td>
</tr>
<tr>
<td>Timing2_assignment.xls</td>
<td>625</td>
</tr>
<tr>
<td>Timing3Statics.xls</td>
<td>625</td>
</tr>
</tbody>
</table>
### Index to Implementations

in C++
- DDLs, 545–547, 628
  - plain, yukky application, Cpp_example.c0, 36

American put (GBM)
- Monte Carlo
  - LSLS with control variates
    - MC_American.CV_one_pass.xls, 494, 499, 627
  - with Richardson extrapolation
    - MC_American.Richardson.xls, 627
  - PDE method. See PDE method
  - trinomial lattice. See Lattice, trinomial

Arithmetic average rate option (GBM), Monte Carlo
- antithetic variates
  - MC_AverageRate_anti.xls, 285, 626
  - combined speed-ups
    - MC_speedups_all.xls, 372, 626
  - control variates
    - MC_AverageRate_bank_CV.xls, 333, 626
    - MC_AverageRate_CV.xls, 343, 626
    - MC_delta_GBM.xls, 342, 626
  - importance sampling
    - MC_AverageRate_IS.xls, 353, 626
  - LD sampling, MC_AverageRate_LD.xls, 312, 626
  - stratified sampling
    - MC_AverageRate_Strat.xls, 298, 626

Arithmetic average rate option (Heston), Monte Carlo
- with control variates
  - MC_Heston_auxiliary_CV.xls, 438, 627
  - with an OOP factory
    - MC_Heston_CV_factory.xls, 627

Barrier options (GBM), Monte Carlo
- level 0, MC_x0_barrier.xls, 255, 625
- level 1, MC_x1_barrier.xls, 255, 625
- level 2, MC_x2_barrier.xls, 255, 625

Bermudan put (GBM)
- Monte Carlo, plain LSLS
  - MC_American.xls, 477, 627
  - trinomial lattice, level 0
    - Lattice_Bermudan_benchmark.xls, 452, 627

Compound option (GBM), explicit solution
- CompoundOptionExplicit.xls, 529, 628

Counter, step
- with a function, 33, 48
- via a method call, 71, 92–93, 131
- the OutputCounter object, 286
- performance, 249–251, 576

DDL example, dll_test.xls, 545, 628

Discretization methods
- CIR process
  - MC_convergence_CIR_simul.xls, 420, 627
- GBM, MC_convergence_GBM.xls, 413, 626
- Heston, weak
  - MC_convergence_Heston_simul.xls, 627
  - Heston, weak, with a call
    - MC_convergence_Heston_simul_call.xls, 436, 627
- O-U process, MC_convergence OU.xls, 414, 627
- Vasicek discount factors
  - MC_convergence_bond_simul.xls, 627

European call (GBM)
- See Lattice, trinomial
- See PDE method
- See Plain Monte Carlo

European call (Heston)
- explicit solution, HestonExplicit.xls, 329, 626
- Monte Carlo
  - with control variates
    - MC_Heston_auxiliary_CV.xls, 438, 627
  - with an OOP factory
    - MC_Heston_CV_factory.xls, 627

Explicit solutions
- compound option (GBM)
  - CompoundOptionExplicit.xls, 529, 628
Explicit solutions (continued)

European call option (Heston)

HestonExplicit.xls, 329, 626

geometric average rate option (GBM)

MC_AverageRateBenchmark.xls, 526, 627

quadratic payoff option (GBM)

MC_exotic_Benchmark.xls, 528, 628

File I/O examples

TextStream object, MC_example_v5a.xls, 624

VBA intrinsic functions example

1_File.xls, 145–151, 624

Geometric average rate option (GBM)

explicit solution

MC_AverageRateBenchmark.xls, 526, 627

Monte Carlo, LD sampling

MC_AverageRateBenchmark.xls, 526, 627

Lattice, trinomial, level 0

American put and European call

Lattice_application_v0.xls, 570–575, 628

performance, 575–578

Bermudan put

Lattice_Bermudan_Benchmark.xls, 452, 627

main()

and the application wrapper object, 76–78, 123
calling procedure, 19, 21

Crank-Nicolson, level 1, 592

file test harness, 146

crto Invocation chain, 101

and Monte Carlo

level 1, 35–36

level 2, 48–49

level 3, 67–68

level 3a, 76

and the pi factory, 187

polymorphic Stopwatch harness, 56
timing stub, 235

trinomial lattice, level 0, 579

Minimization algorithms, golden section search

GoldenSection(), 612

Monte Carlo

in C++, plain, yukky application

Cpp_example_c0, 36

plain Monte Carlo (European call). See Plain Monte Carlo

American put (GBM). See American put (GBM)
arithmetic average rate option (GBM). See

Arithmetic average rate option (GBM)
arithmetic average rate option (Heston). See

Arithmetic average rate option (Heston)

barrier options (GBM). See Barrier options (GBM)

Bermudan put (GBM), plain LSLS

MC_American.xls, 477, 627

European call. See Plain Monte Carlo
geometric average rate option (GBM), LD sampling

MC_AverageRateBenchmark.xls, 526, 627

mimicking barrier option

MC_American_CV_one_pass_set_barrier.xls, 454, 473, 497, 627

quadratic payoff option (GBM), control variates

MC_exotic_CV.xls, 343, 626

Mimicking barrier option

MC_American_CV_one_pass_set_barrier.xls, 454, 473, 497, 627

π series expansions, 13_factory.xls, 184, 624

PDE method, Crank-Nicolson, level 1 (American put and European call)

LU solver, CN_pde_y1.xls, 585–590, 628

performance, 592–595

SOR solver, CN_pdeSOR_y2.xls, 598, 628

Plain Monte Carlo (European call)

level 0, element-wise

MC_example_v0.xls, 36

MC_example_v0_names.xls, 29

level 1, element-wise, MC_example_v1.xls, 25–34

level 2, element-wise, MC_example_v2.xls, 45–50

level 3, path-wise, MC_example_v3a.xls, 67–76

level 3b, path-wise, MC_example_v3b.xls, 76–77

level 4

path-wise, MC_example_v4.xls, 88–100

slice-wise, MC_example_v4b.xls, 107–116

level 5, slice-wise, MC_example_v5.xls, 122–133

level 6, slice-wise, MC_example_v6.xls, 197–206

level 6a, slice-wise, MC_example_v6a.xls, 211–228

performance

barrier option, 255–258

European call option, 254–255

Quadratic payoff option (GBM)

explicit solution

MC_exotic_Benchmark.xls, 528, 628

Monte Carlo, control variates

MC_exotic_CV.xls, 343, 626

Quadrature, quadrature.xls, 531, 542, 628

Random number generators

normal variate generators

from 12 uniforms, 23

inverse transform, cndev(), 251

performance comparisons, 246, 249–251

polar rejection

GetNormal(), 32–34

normal(), 22, 535

uniform variate generators

ran0(), 74, 238, 246–247
ran2(), 246, 247, 535
See also: Index, VBA intrinsic functions, Rnd()

Reference counting example
9b_ref_counter.xls, 62–63, 623

Root finding algorithms, 603–610
  bisection method, Bisection(), 604
  method of false position, FalsePosition(), 606
  Newton-Raphson, NewtonRaphson(), 609
  Ridder’s method, Ridders(), 608

Singleton pattern example
9c_singleton.xls, 563–565, 628

Stop watches
Getz and Gilbert stop watch, 65, 82, 86–88
high resolution stop watch, Green et al., 82
module stop-watch, OOP_example.xls, 550–552
plain StopWatch
  4_timer.xls, 56–57, 63–65, 550–552, 623
  benchmarking, 235–236
Statistics stop-watch, OOP_example.xls, 550
yukky timer, OOP_example.xls, 549–550

Utility code modules, LibraryProcedures.xls, 17, 22, 531, 628
  LibAverageRate, 538
  LibBrownianBridge, 533
  LibCopula, 538
  LibFactory, 538
  LibMatrix, 533
  LibRandom, 533
  LibSobol, 536
  LibStats, 536
  LibTrig, 536
  LibUtility, 531
  LibValidators, 533
  LibVec, 536

User defined objects
  complex, 540
    ComplexHelpers module, 540
  IntegratorExtended, 542
  IntegratorSimpson, 542
  IntegratorTrapezium, 542
  OnceOnly, 216, 221–222, 225
This page intentionally left blank
ACos(), 538
ACosH(), 538
ACot(), 538
ACotH(), 538
ACsc(), 538
ACscH(), 538
add(), 542
AddArrays(), 539
AddChar(), 532
anon(), 195–196, 532
ArithmeticAverage(), 539
Arch_AMH_copula(), 540
Arch_AMH_generator(), 540
Arch_Frank_copula(), 540
Arch_Frank_generator(), 540
Arch_GB_copula(), 540
Arch_GB_generator(), 540
Arch_GH_copula(), 540
Arch_GH_generator(), 540
Arch_Pareto_copula(), 540
Arch_Pareto_generator(), 540
ASec(), 538
ASEcH(), 538
ASin(), 538
ASinH(), 538
ATanH(), 538

BB_hitting_time_dist(), 535
BB_max_dist(), 535
BB_max_draw(), 535
BB_max_draw_vec(), 535
BB_min_dist(), 535
BB_min_draw(), 535
BB_min_draw_vec(), 535
beta() (Numerical Recipes), 537
beta_inverse() (Moshtier), 537
betacf() (Numerical Recipes), 537
betai() (Numerical Recipes), 537

bi_variate_n_dist() (Genz), 537
bi_variate_n_dist_upper() (Genz and Ge), 537
bi_variate_t_dist(), 537
Brownian_bridge(), 535
BScall(), 439, 541
BScalldelta(), 541
BScalldigital(), 541
BSput(), 496, 501, 541
BSputdelta(), 541

CastCreatable(), 201, 540
CastPara(), 540
CastReusable(), 195–196, 208 et cetera, 540
CastSyn(), 540
Cauchy_one_sided(), 537
cExp(), 542
check_bool(), 534
check_char(), 127, 129 et cetera, 216, 534
check_double(), 40–44, 534
check_even(), 534
check_in_range(), 167, 168, 534
check_long(), 40–44, 534
check_path(), 534
Check_positive(), 168, 534
Check_strictly_positive(), 128, 165, 220, 534
Chi_squ_dist(), 537
cndev() (Moro), 242, 243, 246–247, 251, 298, 537
Cody_erf() (Cody), 537
Comonotonic_copula(), 540
CompVec(), 536, 539
conj(), 542
Correlation(), 539
638 Index to Library Functions

CosH(), 538
Cot(), 538
CotH(), 538
Countermonotonic_copula(), 540
CscH(), 538
Csc(), 538
CtsGeoAvCall(), 538, 541
CtsGeoAvPut(), 538, 541
DisGeoAvCall(), 538, 541
DisGeoAvPut(), 538, 541
DisStnGeoAvCall(), 538, 541
DisStnGeoAvPut(), 538, 541
divide(), 542

Equals(), 532
EqualsZero(), 532

FindIndex(), 336, 532
ForwardError(), 43–44, 531–532

gamma_inverse() (Morris), 537
gammln() (Numerical Recipes), 537
gamma() (Numerical Recipes), 537
Gaussian_copula(), 540
gecf() (Numerical Recipes), 537
GeometricAverage(), 539
get_bool(), 534
get_bool_wk(), 534
get_char(), 92, 127, 128 et cetera, 534
get_char_force(), 533, 534
get_char_wk(), 533, 534
get_double(), 40–44, 46 et cetera, 534
get_double_in_range(), 534
get_double_wk(), 165, 168, 534
get_long(), 40–44, 46 et cetera, 534
get_long_in_range(), 533, 534
get_long_wk(), 165, 534
get_path_wk(), 175, 534
get_string(), 534
get_string_wk(), 534
GetRow(), 539
GetTwoNormals(), 535
GEV_dist(), 537
GEV_inv_dist(), 537
gser() (Numerical Recipes), 537

Im(), 542
Independent_copula(), 540
InnerProduct(), 539
Invert_LT_matrix(), 533
IsEven(), 532
IsOdd(), 532
LessThanOrEquals(), 532
make(), 542
MakeVecStep(), 539
MaskSetToScalar(), 539
MaxArrays(), 539, 599
MinArrays(), 539
multiply(), 542
MultiplyByScalar(), 500, 539
my_atan2(), 538
my_max(), 30 31, 48, 70 et cetera, 532, 574
my_max_long(), 532
my_min(), 420, 532
my_min_long(), 532

n_probs() (Genz), 537
neg(), 542
NegateVec() , 285 et cetera, 536, 539
norm(), 542
normal_cdf(), 17, 537
normal_inverse() (Acklam), 537
normal1(), 22, 535
NotVec(), 539

ols_r(), 533
PadOut(), 532
PadSpaces(), 532
Pow(), 542
Powc(), 542

RaiseError(), 42, 111 et cetera, 531, 532
ran0(), 74, 251, 254
performance, 238, 242, 243, 246–247
See also: stat_ran0()
ran2() (Numerical Recipes), 246, 247, 535
rand_beta(), 535
rand_beta_A_and_W(), 535
rand_beta_cheng(), 535
rand_beta_johnk(), 535
rand_chi2(), 535
rand_exp_VN(), 535
rand_gamma() (Numerical Recipes), 535
rand_ig(), 535
rand_Poisson(), 535
Re(), 542
ReadColumn(), 534
RescaleToZeroMean(), 536, 539
RMSE(), 539, 600

Sec(), 538
SecH(), 538
SetToBool(), 500, 539
Index to Library Functions

SetToScalar(), 539
SetToZero(), 539
SinH(), 538
sobseq() (Numerical Recipes), 251, 310, 313, 536
SpaceOutChars(), 532
Sq(), 542
Sqrt(), 542
StandardDeviation(), 539
stat_ran0() (Numerical Recipes), 33–34, 47–48, 535
Student_copula(), 540
student0(), 535
subtract(), 542
swap(), 217

t_inv_dist(), 537
t_dist() (Numerical Recipes), 537
TanH(), 538
ValidateIncreasing(), 534
ValidateStrictlyIncreasing(), 534
VecAnd(), 539
VecMax(), 539
VecMin(), 539
VecMod(), 539
VecSize(), 539
VecSum(), 539

WB_max_draw(), 535
WB_max_draw_vec(), 535
WB_min_draw(), 535
WB_min_draw_vec(), 535
Weibull_dist(), 537
Weibull_inv_dist(), 537
WriteColumn(), 171, 534
WriteColumnIndexes(), 534
WriteColumnL(), 534


Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1994) *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.


This page intentionally left blank
Index

: (statement separator), 28
label delimiter, 28
# (cast), 33, 236, 237, 238
file functions, 143–145
:: (scope qualifier), 58, 621
in C++, 58
& (concatenation), 241
+ (concatenation), 241
_ (continuation line character), 44
in implementing events, 121
in interface implementation, 84–85
and naming conventions. See Names, naming conventions

See Comments
.
(name qualifier). See Names, qualified names
and UDTs, 244
= (assignment operator), 244
^ (power operator), 237
/ (division), 236, 237, 238
\ (integer division), 238

0.5 strong Itô-Taylor discretization. See Itô-Taylor discretization schemes, 0.5 strong
1.0 strong Itô-Taylor discretization. See Itô-Taylor discretization schemes, 1.0 strong
1.5 strong Itô-Taylor discretization. See Itô-Taylor discretization schemes, 1.5 strong
2.0 weak Itô-Taylor discretization. See Itô-Taylor discretization schemes, 2.0 weak
2-dimensional arrays. See Arrays

Accumulator account numeraire, 325, 424, 581
Accuracy. See Convergence
Acklam. See Index to library functions,
normal_inverse()
ActiveX controls, buttons, 27, 571–572
Adapter pattern, 559, 560–561, 613
See also: Design patterns
Aggregation, of objects, 613

American options. See also: Bermudan options
American put. See American put
American straddle, 450, 462, 475
See also: Bermudan option valuation
American put, 451
See also: Option valuation, American put
on a knock-in barrier option, 489
on a knock-out barrier option, 489
See also: Index to implementations, American put
American straddle, 450, 462, 475
Antithetic variates, (Chapter 17), 283–290
average rate options (GBM), effectiveness with,
288–289, 379–380, 381–382, 382–383, 384
and bias, 285
correlation, 284, 289
and the decorator pattern, 285, 304
with the Heston model, 290
and long-step Monte Carlo, 380, 382
and moment matching, 291
and slice-wise evolution, 285
and the uniform distribution, 284
with Wiener processes, 284
See also: Variance reduction

API. See Application programming interface
Application methods
Application.Max(), 30, 243, 252
Application.NormSDist(), 17
Application.NormSInv(), 22, 243, 246
Application.Pi(), 38, 185
Application.Run(), 83–84
Application object, 242
Application programming interface (API), 56, 81, 545
GetTickCount(), 88
Arbitrage, 10
and violation of put-call parity, 393
Array expansion idiom, 147, 166, 174
Arrays
2-dimensional arrays, 268–270
as separate 1-dimensional arrays, 268
Arrays (continued)
  striding, 268–270, 617
  performance, 273–274
array utility functions, 536
  FindIndex(), 532
passing ByRef, 107
and DLLs, 545–547
dynamic array, 266
and file I/O, 146–148
holistic Monte Carlo, 296
and level 0 lattice implementation, 573
and level 1 Crank-Nicolson implementation, 586
and multidimensional state, 263, 265–268
  performance, 266–267
path-wise Monte Carlo, 157, 257
performance
  as a container, 273–277
  higher dimensional arrays, 245, 296
  looping, 245, 275
  and pre-extraction, 272, 273
re-sizing. See Array expansion idiom
  See also: Memory manager
as return types, 107
slice-wise Monte Carlo, 107–116, 255
  performance, 272–273
static array, 261
Variant array. See Variant arrays
  and wrapper objects, 263, 261–267, 270–271
Assignment
  See also: Copy assignment
  arrays, 245, 272
casting, 244
  performance, 238, 241, 243, 244, 245
references, 556
UDTs, 244, 554
Association, of objects, 613
Associative container, 222, 262, 614
At the money (ATM) options. See also: Out of the
  money options; In the money options
  and antithetic variates, 288
  and Heston examples, 435
ATM. See At the money options
Average rate options
  See also: Option valuation, average rate options
  binary chop evolution, 13
  continuously averaged geometrically averaged
    explicit solution, European call, 18
discretely reset arithmetically averaged, 117
  See also: Index to implementations, Arithmetic
    average rate option (GBM)
  See also: Index to implementations, Arithmetic
    average rate option (Heston)
  See also: Option valuation, average rate options benchmark values, 281
  and valuation
    in a book with control variates, 333, 377
    in a book with importance sampling, 352
    with control variates, 324–325, 338–342
    in Heston, 330
discretely reset geometrically averaged, 117
  as a control variate, 317, 324–325
  explicit solutions, European call, 117, 523–526
  and LD sampling, 315
  library functions, 538
See also: Index to implementations, Geometric
  average rate option
  fixed strike, 117
  state information, 107
B-splines, 464, 465–466, 479–482
  See also: Basis functions
Backwards evolution. See Evolution, direction, backwards
Bad practice, 58, 233, 243, 517
  See also: Evil
Barrier options
  See also: Option valuation, barrier options
  barrier bond options, 398
  and bias, 102, 255, 396–397
double barrier options, 95, 102, 255–258
  flexible barrier option, 488
general formulation, 450–452
mimicking barrier options. See Mimicking
  barrier options
  and plain Monte Carlo, 102
  state, 97, 107, 113
  vanilla barrier options
    knock-in, 7, 97, 98, 113, 258, 316
    knock-out, 16, 95–97, 98, 113, 290, 316, 398
  See also: Index to implementations,
    Barrier options
Base class, 613
  in C++, 86, 558
  VBA version, 193
Basis functions, in LSLS MC, 463–470
  See also: Continuation values; LSLS method
B-splines, 464, 465–466, 479–482
Chebyshev polynomials, 1st and 2nd kind, 464, 475
  comparison between, 466–468
Gegenbauer polynomials, 464
Laguerre polynomials, 464–465, 479–483
Legendre polynomials, 464–465, 479–482
  natural basis functions, 464, 465, 479–482
  and OLS, 468
  power functions, 464–465, 466–468, 479–482
  scaled basis functions, 465
  and SVD, 468
Bermudan-2 put, 528–529
Bermudan-early control variate, 493
Bermudan option valuation, by Monte Carlo
  bundling algorithms, 457
  continuation values. See Continuation values
early exercise boundary. See Early exercise boundary.
error
foresight error, 392, 452, 459
sub-optimal exercise error, 449, 454–455, 471, 474, 487
LSLS method. See LSLS method
stochastic mesh method, 457
Bermudan options
See also: Bermudan option valuation
Bermudan butterfly, 462
Bermudan compound option, 209
Bermudan-early put, 493
as a control variate, 497–498, 505
Bermudan put option. See Bermudan put option
Bermudan straddle, 462
Bermudan swaption, 449, 450, 465
Bermudan put control variates, 492
Bermudan put option
See also: Option valuation, Bermudan puts
Bermudan-2 put, 528–529
convergence to American put, 452–453
See also: Compound option
See also: Index to implementations, Bermudan puts
Bermudan-T1 control variate, 493
Bermudan terminal control variate, 493
Beta distribution, 11, 233, 536
Bias, in Monte Carlo. See also: Discretization, convergence
and antithetic variates, 285
and barrier option valuation, 7, 102
bias reduction methods. See Bias reduction methods
in bond option valuation, CIR, 390–392
in the drift, 391, 393
and predictor-corrector methods, 411–412
and early exercise
in American put valuation, 452, 453, 455, 503
high-bias estimator, 457
See also: Bias, foresight bias
low bias estimator, 457, 487
in LSLS method, 473–474
and basis functions, 468, 471, 479–482, 487, 509
in exponents, 424, 426–427
foresight bias, 392, 452, 459
in Heston, 436
and LD sampling, 314–315
measure of, 474
sources of bias, 389–390
Bias reduction methods
for barrier options, 396–397
martingale correction, 393
martingale simulation, 393
moment matching, 394
Bidirectional association, of objects, 613
Binary chop evolution. See Evolution, direction, binary chop
Binary files, 137
Binding
early bound, 139
late bound, 139, 244, 262, 263
Binomial lattice, 210, 567
Bisection method, root finding, 603
in exercises, 38, 79, 104
Bisection(), 604
Bit-arithmetic, 310
Bi-variate normal distribution, 529
Black-Scholes framework, 3, 7, 25, 324
See also: Black-Scholes model
Black-Scholes model
See also: Black-Scholes framework
See also: Black-Scholes PDE
See also: Geometric Brownian motion
See also: Implied volatility
See also: Simulation, geometric Brownian motion
Black-Scholes call control variate, 338–339
Black-Scholes formula, 25
explicit solutions
European call, 25
discretely reset geometrically averaged average rate option, 523–525
library implementation procedures, 538–540
quadratic payoff option, 526–528
2-reset Bermudan option, 528–529
Monte Carlo applications, benchmarking, 254–255
parameter validation, 40
Black-Scholes PDE, 581
implementation, 585–590, 598–601
numerical assessment, 593–595, 601–602
Block comments, 26, 571
Blow up, discretizations, 391, 420
Bodge, 613
and overflow, 420
and rectification functions, 407, 416
CIR, 410, 417, 420
Heston, 432, 446
and weak design, 155
Bond options
barrier bond options, 398
in CIR, 389–391
and early exercise, 460–461
in Fong and Vasicek, 325–327
in Vasicek, 326, 327–328
Book of options, 153–176
benefits of scale, 174
and control variates, 333, 377
and importance sampling, 352, 368
and knock-out options, 16, 258
Boolean
and IIF(), 239
in masks, 335, 494
Boolean (continued)
  as state, 97, 107, 111–113, 256
  and the Static initialization idiom, 47, 247
to toggle functionality, 361
Borel measure, 4
Bracketing interval, 603
Branching probabilities. See Lattice methods,
branching probabilities
Bridge density, 292
  bridge density of a Wiener process, 293
  of the maximum from a bridge, 397
  sampling from, 292
Bridge distribution, GBM, sampling from the
maximum, 397
Broadie and Kaya, 432, 446
Brownian bridge, 118, 293
and evolution, 294–297
and the LSLS method, 449, 482
and stratified sampling, 292–293
See also: Bridge density
Bullet, bite, 16
Bundling algorithms, Bermudan option valuation, 457
Buttons, 19–20, 25–26, 34
  ActiveX controls, 27, 571–572
  and invocation chain, 101
See also: Calling procedure
ByRef, 555
  and DLLs, 245–247
  and error codes, 550
  passing, examples
    arrays, 107, 285, 376, 479, 494, 536
    Doubles, 45
    Enums, 181
    UTDs, 146
ByVal, 555
  and DLLs, 545–546
  performance, 242–243
C++, 39, 40, 47, 58
  clone(), 216
  comparison with VBA, xv, 36, 55, 556, 557, 565
  delete, 58
  DevCpp, 36, 241, 628
  DLLs, 545–547, 628
  functors, 71, 614
  inheritance, 86, 558
  initialization, 28, 554
  meta-class data, 559
  objects in, 55, 56, 58, 60, 549, 555, 556
  operator(), 71, 614
  operator overloading, 556, 557, 615
  performance, xv, 36, 235, 241
  polymorphic factory, 179, 183, 216
  polymorphism in, 558
  std::cout(), 138
  std::exp(), 241
  streams, 137, 138
  swap(), 217
  this, 66
Calibration, models, 10, 12, 18, 610
  exact, 392
  moment matching, 394–396
Call, 621
Call-back, 559, 562
  call-back procedure, 100
See Registration and call-back
Calling procedure, 19, 119, 605
See also: Invoker; main()
Casts. See also: Type conversion
  between interfaces, 195–196
  and file input, 148
  and validation, 185, 201
Catch and re-throw idiom, 44, 76, 531
Catching errors. See Error handling, trapping errors
Cells(). See also: Front-end
  clearing contents, 45, 72, 130, 585
  and range names, 29
  validating input from, 28, 533
Central moments
  of log-normal distribution, 418
  of normal distribution, 23
  of uniform distribution, 22
CEV (Constant elasticity of variance) model, 102–103, 114
Channels. See also: Streams, I/O
  and the Environment object, 172
  error channel, 19, 21
  and the factory, 198, 212, 226
  file number for, 144
  I/O, 19, 78, 122, 137, 174
  costs, 174–176
  input channel, 20, 78
  monitor channel, 130
  output channel, 78, 127, 130
  sequential, 137
Chebyshev polynomials, 1st and 2nd kind, 464, 475
See also: Basis functions
Chirayukool, Pokpong, xvii, 447
CIR. See Cox, Ingersoll and Ross model
Class modules
  and events, 119–121
  and factory registration, 188, 197
  and fixed length Strings, 263
  and interfaces, 84–86
  and object definition, 56, 59–63, 556
Private scope, 58
  and re-use, 57
  and state, 40
  structure of, 59
  and the VBComponents object, 180–183
See also: Index to implementations, Utility code modules
Class_Initialize(), 59–60, 120, 557, 613
and encapsulation, 555
and initialization, 72
Class_Terminate(), 59–60, 120, 557, 614
Cleanliness
  clean recompile, 190, 518–521
  code, 57
  interface mechanism, 102
  and re-factoring, 616
Client, 613
  client code, 35, 56, 62, 81, 557
  communication with, 33, 47, 91, 166, 169, 214, 244
  contract with, 613
  decoupling, 29, 46
  and encapsulation, 551, 560
  and events, 121
  expose to, 614
  of objects, 56, 59, 83
  transparency, 83, 271, 557, 615
See also: Front-end
Client code, 35, 56, 62, 81, 557
close(), 216
Clustering
  LD sampling, 307
  stratified sampling, 292, 297–298
CN. See Crank-Nicolson
Code modules. See Standard modules
CodeModule object, 180–183
Cody. See Index to library functions, Cody_erf()
Cohesion, 39, 613
Collection object, 262–263, 613
  and containers, 270, 275
  and FileSystemObject, 139–140
  performance, 274, 275–276
  looping with For-Each, 263, 275
  looping on keys, 275
  and states, 276
Comma-delimited files, 144
Comments
  block comments, 26, 571
  and CodeModuleProperties, 181–182, 183
  in-line comments, 622
  toggling, 249, 256
  and tricky code, 248
Complex numbers,
  the complex number exercise, 79
  and Heston, 329
  library object, 540
Compiler, See VBA compiler
Compile-time errors, 40, 42, 194, 223, 244
Composited object, 72, 76, 613
Composition, 559, 613
  and the adapter pattern, 560
  composited object, 72, 76, 613
  and the decorator pattern, 124, 562, 614
  and the façade pattern, 560
  and the factory, 163, 127, 218
  and polymorphism, 153
See also: Design patterns
Compound option, 483–484, 529
See also: Option valuation, compound options
Concept code, 565, 613
Conforming object, 84–86, 557–558
  factory creation. See Instantiation
  in the level 4 plain Monte Carlo, 88–100
  and meta-class data, 558–559
See also: Meta-classes
  nomenclature, 85
  over-ride, 85
See also: Derived objects; Interfaces
Console, 164
Const
  and arithmetic operations, 236
  Const expression, 48, 179
  data members, 247
  honte, 245
  and initialization, 554
  and loop bounds, 245
  and magic numbers, 615
  performance, 236, 245, 247
  and the Static initialisation idiom, 48, 254, 257
Const variables. See Const
Constant elasticity of variance. See CEV
Constant GBM IS density, 350–351
See also: Importance sampling
Constructors, 55, 60–61, 613
  and Class_Initialize(), 72, 557
  conversion constructor, 55, 554, 556, 614
  copy constructors, 55, 553, 614
  default constructor, 60, 179, 553, 554, 556
  and events, 120
  and the factory
    embryonic factory, 123, 130
    polymorphic factory, 198–200
    semi-polymorphic factory, 212, 214, 218
    and initializing, 310, 335
    inputting from, 69, 92
    limitations of in VBA, 60, 69
    and outputting, 72, 141
    and POD-like objects, 263
    RAII. See RAII idiom
    and reference counting, 61–63
    and registration, 183, 184
    and side-effects, 200
    and the singleton pattern 563
Containers. See also: VBA containers
  associative container, 222, 262, 614
  queue, 263
  set, 222, 263
Continuation lines, 44
Continuation values
basis functions, in LSLS MC. See Basis functions and control option benchmark, 483
and control variates, 498–505
European put control variate, 498
implementation, 499
performance, 499–503
in LSLS MC, 458
notation, 450
slice of, 477
Continuously reset geometrically averaged average rate options. See Average rate options
Control, option, 10
Contract, programming, 40, 91, 223, 613
Control statements
Do-Exit-Loop, performance, 240–241
Do-Until. See Do-Until
Do-While. See Do-While
End, 493
Exit, 42, 256, 495
For-Each. See For-Each
For-Next. See For-Next
GoTo. See GoTo
If-Then. See If-Then
If-Then-Else. See If-Then
On Error, 42–43, 50, 517
Resume, 42, 44, 50
Resume Next, 42, 517
Select-Case. See Select-Case
While-Wend, 240
Control variates, (Chapters 20 and 21), 317–332, 333–344
asset control variates, 320–321
auxiliary model control variates, 325–330
and Fong and Vasicek, 325–328
and Heston, 328–330
average rate options (GBM), effectiveness with. See Option valuation, average rate options, control variates
OTM options, 339, 342
bank of, 333, 335
Bermudan put. See LSLS method, using control variates
using rollback control variates, 499–505
using valuation control variates, 497
call control variate, 338
combined with importance sampling, 371–372
correlation. See Correlation, and control variates
delta control variates, 321–322
and Heston, 327, 330
with Heston. See Heston model, variance reduction, control variates
payoff matching control variates, 323–324
quadratic option, effectiveness with. See Option valuation, quadratic option, control variates
registration of, 335, 373, 377
tailored control variates, average rate options, 324–325, 371
See also: Variance reduction
Convergence
of discretization schemes. See Discretization, convergence
of numerical methods
Monte Carlo. See Option valuation
See also Discretization, convergence
See Lattice methods, convergence
See PDE methods, convergence
See Root finding algorithms, convergence
See Simulation
Convergence criteria, of discretization schemes
strong, 399–400
test function, 400, 413, 435
weak, 399–400
Conversion constructor, 55, 554, 556, 614
Convertible bond, 450, 461
Copula distributions, 5, 538
Copy assignment, 55, 553, 556
Copy constructor, 55, 553, 614
Correlation
and antithetic variates, 284, 289
and control variates, 317, 318
asset control variates, 320
auxiliary model control variates, 330
and the Bermudan put, 497–498
delta control variates, 342
in the Heston model, 436–438
payoff approximating control variates, 324
and discretization, 404
and the Heston process, 432–434
generating correlated normals, 279
and stratification, 441
Coupling, 104, 208, 356, 614
and Enums, 153
and the factory, 119, 133, 179
psychic, 228
Cox, Ingersoll and Ross (CIR) model, 390–392, 400–401
Feller condition, 408, 417–424, 429
and linear growth condition, 408
and Lipschitz condition, 402, 408, 410, 432
option valuation
and barrier options, 398
bond option (by Monte Carlo), 390–392, 460–461
pure discount bond (explicit solution), 430
simulation. See Simulation, Cox, Ingersoll and Ross and skewness, 429
strong solution to the SDE, absence of, 417
See also: Heston model
Crank-Nicolson. See PDE methods, Crank-Nicolson
Ctor. See Constructors
CV. See Control variates
Data. See also: Object oriented programming; Objects
arrays. See Arrays
assignment. See Assignment
casts. See Casts
Const variables. See Const
containers. See Containers
data section, 59
See also: Declaration section
declaration. See Declaration
files. See Files
Friend data, 31, 50, 55, 58, 614, 621
global variables. See Global variables
handles. See Handles
initialization. See Initialization
input. See Input
invariant, 615
lifetime. See Lifetime
literals. See Literals
local variables, 39, 47, 50
magic number, 615
memory manager, 174
names. See Names
objects. See Objects
output. See Output
plain old data (POD). See Plain old data
pre-extraction, 272, 273
Private data. See Private data
Public data. See Public data
references. See References
scope. See Scope
Static variables. See Static variables
storage. See Storage
type conversion. See Type conversion
user defined type (UDT). See User defined types
VBA types. See VBA primitive types
visibility. See Visibility
Data members, 614
See also: Sentinel; Plain old data; Composited object
accessing, 59
See also: Properties; Data members, restricting access
Const data members, 247
and the decorator pattern, 124, 562
and the factory, 127–130
and globals, 62
initialization, 60, 84. See also: Constructors
and interfaces, 84–86, 557–558
and meta-class data, 558–559
and Properties, 60–61, 69, 72, 92, 614, 616
restricting access to, 157, 159
and serialization, 127
and state, 40, 553–554, 616
Data section, 59
See also: Declaration section
Date, 56, 145, 148

Debug menu, 518
Declaration. See also: Dim
and constants, 245
and constructors, 59
See also: RAII idiom
declaration section, 59, 75, 181–183, 201
Declare, 88, 546
and DLLs, 545–547
of Enums, 153
of events, 39, 59, 120–121
and initialization, 553
in-line, 28, 29
location of
and Dim-New, 57–58, 68
at point of first use, 28, 29
at top of a block, 28
and modules, 554
with narrowest scope, 31, 39
Object, 65
and the Dictionary object, 262
and the FileSystemObject object, 139
and objects. See Objects, declaration
and Option Explicit, 27, 517
order of, 235
of references, 56, 57–59, 556–557, 615
required, 517
of Types, 46
undeclared arguments or return types, 243
Declaration section, 59, 75, 181–183, 201
Decorator pattern, 562, 614
with antithetic variates, 285, 304
in the Monte Carlo application object, 124
and self-references, 61
See also: Design patterns
Decoupling, 19, 21, 116, 258, 285, 617
and Cells(), 29
and the factory, 119, 127, 179, 211
and input/output, 78, 90–91, 174, 228
and the path object, 153
and Statics, 247
and StopWatch, 56
Default constructor, 60, 179, 553, 554, 556
Delegation, 21, 48, 77, 78, 285
in the factory, 188, 216, 218
in the template pattern, 563
delete, 58
Densities. See Distributions
Deprecate, 155, 298
Derived objects, 614
in C++, 86, 558
in VBA, 214
See also: Conforming object
Design flaw, 38, 45, 50, 153, 591
Design patterns
adapter pattern, 560–561, 613
composition. See Composition
Design patterns (continued)
decorator pattern. See Decorator pattern
façade pattern. See Façade pattern
factory pattern. See Factory pattern
gang of four, 559
polymorphic factory. See Factory pattern
prototype pattern, 223
registration. See Registration
registration and call-back. See Registration and
call-back
singleton pattern. See Singleton pattern
strategy pattern. See Strategy pattern
template pattern, 562–563, 617
wrapper objects. See Wrapper objects
See also: Invocation chain
Destructor, 614
automatic execution, 60, 61
Class_Terminate(), 59–60, 120, 557
and End, 495
and files, 141
and object definition, 59
and RAI. See RAI idiom
and reference counting, 61–63
side-effects, 200
DevCcpp, 36, 241, 628
Developer tab, 180, 517, 571
Dictionary object, 262–263, 614
and containers, 270, 275
looping through with keys, 262
and OnceOnly, 222
and parameter values, 194, 203
performance, 275–276
and registration, 184, 188, 200–203
Digital option, importance sampling example, 346–348
Dim, 615
and declaring object references, 57
and object instantiation, Dim-New, 57–58, 68,
72, 76
See also: Redim
Disambiguation, 44, 58, 181
Discount factors, simulation of. See Simulation, discount factors
Discretely reset arithmetically averaged average rate
options. See Average rate options
Discretely reset geometrically averaged average rate
options. See Average rate options
Discretization. See also: Monte Carlo; Simulation
blowing up, 391, 420
convergence
correlation criteria
strong, 399–400
test function, 400, 413, 435
weak, 399–400
empirical rates, 413–416
and the CIR process, 420–424
and discount factors, 428–429
and GBM, 413
and the Heston process, 435–436
and Vasicek, 414
theoretical rates
0.5 strong, 402
1.0 strong, 402–403
1.0 weak, 402
1.5 strong, 404–406
2.0 weak, 410–411
and transforming the SDE, 407–408
exact simulation. See Simulation, exact simulation
and negative values
with the CIR process, 407–408
in the Euler scheme, 407
and rectification functions, 327, 407, 420
schemes
Euler scheme. See Euler discretization scheme
Itô-Taylor discretizations. See Itô-Taylor
discretization schemes
log-Euler, with CIR. See Log-Euler discretization
schemes
log-normal approximation. See Log-normal
approximation discretization schemes
Milstein scheme. See Milstein discretization
scheme
moment freezing. See Moment freezing
discretization schemes
moment matching approximations
See Log-normal approximation
See Moment freezing
See Normal approximation
normal approximation, CIR, 418
predictor-corrector method. See
Predictor-Corrector method
of a transformed SDE, 407–408
Discretization error. See Discretization, convergence
discretization schemes. See Discretization, schemes
Dispatch, 90, 95, 614
double dispatch, 157
and the façade pattern, 113
and the factory, 188, 198, 218, 222, 223
Distributions
See also: Importance sampling
See also: Moments
See also: Simulation
beta distribution, 11, 233, 536
and bias. See Bias
bi-variate normal distribution, 529
bridge density. See Bridge density
density of CIR process, 391, 417
closure under convolution, 10
copula distributions, 5, 538
library functions, 533, 536, 538
log-normal density. See Log-normal density
density, 293, 536
normal distribution, 22, 25, 251, 536
Index

Ornstein-Uhlenbeck process.

See Ornstein-Uhlenbeck process
t-distribution, 536
uniform distribution. See Uniform distribution
Division by zero, 475
DLL. See Dynamic linked library
Do-Exit-Loop, performance, 240–241
Do-Until
and multiple input, 147, 167, 168
performance, 240–241
Do-While
performance, 240–241
in the PSOR solver, 600
Double, 621
and ∼, 237
and arithmetic operations, 236–241
casting, 244
casting to Double, 33
casting to Long, 238
and comparison tests, 239
and DLLs, 545
and explicit typing, 243
input, 43, 200, 201, 203, 533
as loop counter, 245
and my_max(), 30
and POD objects, 263
and state, 265–268, 273–277
validation, 41, 533
Double barrier options, 95, 102, 255–258
Double dispatch, 157
Dtor. See Destructor
Dummy variables, 28
Dynamic linked library (DLL), 545–547
and the Err object, 40
“Bad DLL” error message, 519, 521
implementation, 628

Early bound, 139
Early exercise boundary
See also: Barrier options, mimicking barrier option
benchmark, from lattice method, 452–453
construction of, in the LSLS method, 463
using CV-rollback, 498–503
plain LSLS, 470–471
in two-pass method, 505–509
convergence to, in the LSLS method
with CV-rollback, single pass, 499–505
with CV-rollback, two pass, 505–509
plain LSLS, 470–471
EEB. See Early exercise boundary
Effective dimension, and path dependency, 107
Efficiency. See Efficiency gain
See also: Performance; Variance reduction
Efficiency gain, 282
Elegance
and the decorator pattern, 285
layout, 68
polymorphism, 102
and re-factoring, 616
Select statement, 240, 574
structure, 78
Element-wise evolution. See Evolution, types, element-wise
Encapsulation, 554, 555, 614
breaking encapsulation, 55, 61, 216, 557
and error handling, 39
and functions, 21, 31, 45–48, 209, 244, 252
and local variables, 47
and Tactics, 245–247, 254
and states, 265, 272
and UDTs, 244
and VBA objects, 55
See also: Façade pattern
See also: Wrapper objects
End, 493

Enums
and coupling, 153
declaration of, 153
with importance sampling, 352–353
with root finding, 605, 610
and Select, 153, 177, 585
and telepathy, 155, 166
and the VBComponent object, 180–181
Environment, 614
environment file, 21, 22
Environment object, 164, 172–174
and the fully polymorphic factor, 198
and the semi-polymorphic factory, 211–212, 218
environmental data, 19, 21, 22

Equivalent martingale measure, 424, 567
Err object, 39, 40–45
and ErrorHandler, 68
and events, 119
Error handling, 39–45
See also: On Error; Exception safe; Err object;
Resume
catching errors. See Error handling, trapping errors
compile-time errors, 40, 42, 194, 223, 244
ear error messages, 151, 166
and files, 138, 145, 147–149, 166
idioms for, 42, 44, 76, 531
raising errors, 40–45. See also: Error handling,
throwing errors
and input, 43–45, 92, 166
and inverting near-singular matrices, 445
by OnceOnly(), 221
and the singleton pattern, 564
and standard error, bad, 111
and telepathy, 127, 223
testing existence
and Class_Initialize(), 63
and Class_Terminate(), 60
Error handling (continued)
by the factory, 183, 185–188, 200–201
of files, 149
of keys in a Collection, 263
run-time errors, 40, 85, 194–195, 217, 244
throwing errors, 39, 40–43, 50
trapping errors
  cleaning up, 42, 44
  in main(), 42, 49, 68, 76
  and validation, 222, 533
See also: Validation
Error handling idiom, 42, 50
Error messages
  error handling, 151, 166
  Excel, 263, 519–521
Error trapping. See Error handling
Euler discretization scheme, 14, 37, 402
  in 1-dimension, 406–407
  and bias, 328, 407
  with CEV, 103
  with CIR, 327, 390, 406
    blowing up, 391
      order of convergence, 420–424
        with transformed SDE, 407–408
    convergence, theoretical order, 402
    and delta control variates, 328
    with exotic processes, 416
    floored, 327, 329
    with Fong and Vasicek, 327
    with GBM, benchmark, 37, 406
      order of convergence, 413–414
    with Heston, 329, 432
      order of convergence, 435–436
    negative values, 407
    and predictor-corrector schemes, 411–412
      Euler predictor, 412, 413, 425
    transforming the SDE, 407–408
    with Vasicek, benchmark, 406
      order of convergence, 414
European call options, 7
  Black-Scholes, 25
  in the Heston model, 328–329, 439–442
  general European style options, 3–7, 386
  in Merton jump-diffusion, 23
See also: Exercise streams, implied volatility
See also: Implied volatility
See also: Index to implementations, European call
See also: Lattice methods, convergence
See also: Option valuation, European call
See also: PDE methods, convergence
European put options
  and basis functions, 465
  as control variate for Bermudan puts, 491–492, 498
Event, 120
Events, 39, 56, 59, 101, 119–121, 133

See also: Workbook events
Catching events, 121
Class_Initialize(), 59–60, 120, 557, 613
  and encapsulation, 555
  and initialization, 72
Class_Terminate(), 59–60, 120, 557, 614
defining events, 120
Event, 120
firing events. See Events, raising events
RaiseEvent, 120
raising events, 39, 55, 119, 120, 121
WithEvents, 39, 120, 121
Evil, programming, 614
inappropriate use of End, 495, 511
inappropriate use of globals, 62
inappropriate use of GoTo, 43, 614
inappropriate use of names, 60
inappropriate use of Object, 65
inappropriate use of Variants, 262
lame Case-Else, 223
raising class events, 120
See also: Telepathy; Psychic connection
Evolution, direction
  backwards, 294–295
  and American options. See LSLS method
  binary chop, 294
    implementation, 298, 312
    partial binary chop, 294–295
See also: Stratified sampling
  forward, 13, 294
  and barrier options, 97
  and plain Monte Carlo. See Index to implementations, Plain Monte Carlo
Evolution, types, 15–16
  element-wise
    level 1, (Chapter 3), 29–30
    level 2, (Chapter 4), 48–49
    level 4, (Chapter 6), 93–100
  holistic, storage constraints, 296
  path-wise
    level 3, (Chapter 5), 67–76, 76–77
    level 5, (Chapter 10), 153–174
    and moment matching, 394, 395
  performance comparisons, 254–258
  slice-wise, 107, 261
    and American option valuation, 449
    and antithetic variates, 285
    and bridge methods, 293–295
    level 4, (Chapter 7), 107–116
    level 5, (Chapter 8), 122–133
    level 6, (Chapter 12), 193
    level 6a, (Chapter 13), 211
    and moment matching, 394, 395
    and storage constraints, 486
Exact simulation. See Simulation, exact simulation
Files (continued)
output to, 137, 138
costs of, 174–176
for an option book, 166–170
and random access files, 148–150
and sequential files, 66, 170
and the TextStream object, 66, 141
and VBA intrinsic file functions, 144–145
serialization, 127, 616
the TextStream object, 140–141
and UDTs, 145
VBA file objects
See FileSystemObject object
See TextStream object
See also: VBA intrinsic file functions and statements
FileSystemObject object, 66, 138–140, 616
and VBA intrinsic functions, 143
Filtration, 3, 4
Finite difference methods (FD). See PDE methods
Fixed length Strings, 59, 263, 545–546
Flexible barrier option, 488
Fong and Vasicek model, 325
and an auxiliary model, 325–326
and a delta control variate 326–327
simulation. See Simulation, Fong and Vasicek
process
For-Each, looping through
a Collection, 263, 275–276
the VBComponents object, 189, 198
a Dictionary, illegality of, 262
For-Next
and application building, 29–30, 574–575
default value of loop increment, 29
Exiting from a For loop, 256
and type of loop counter, 245
performance, 240–241
and looping through POD-like objects, 263
and Step, 29, 247
Foresight bias, 392, 452, 459
Form, 164
Forward evolution. See Evolution, direction, forward
Friend data, 31, 50, 55, 58, 614, 621
Front-end, Excel
application building, 25–26, 570–572
for Crank-Nicolson, 586
and the Environment object, 218
input from, 533
in the Crank-Nicolson application, 586
in the lattice application, 570–572
in the Monte Carlo application, 28, 29, 69, 92, 122
and validation, 28, 43–45, 533
for the Monte Carlo applications
level 1, 26
level 4, 89
level 5, 120
level 6a, 213
output to
in the Crank-Nicolson application, 585–586
in the lattice application, 571, 572
in the Monte Carlo application, 26, 45
and utility functions, 533
for the pi application, 184
for the trinomial lattice, 571
Fudge, 420
Functions, 35, 45, 47, 48–50
See also: Procedures; Subs
Functor, 71, 79, 191, 553, 557, 565, 614
Fundamental pricing equation, 3, 567, 581
Gandhi and Hunt, 426, 428
Gang of four, 559
Gaussian affine models
2-factor Gaussian, 278
explicit solutions, 10, 426
and observables, 12
GBM. See Geometric Brownian motion
Gegenbauer polynomials, 464
See also: Basis functions
Genz. See Index to library functions, n_probs(); bi_variate_t_dist()
Genz and Ge. See Index to library functions,
bi_variate_n_dist_upper()
Geometric Brownian motion
bridge distribution, of the maximum, sampling from, 397
exact solution to SDE, 32, 118
simulation. See Simulation, Geometric Brownian motion
See also: Black-Scholes model
Getters, 60–61, 64, 556, 614
and encapsulation, 554
and the factories, 127, 198–200, 218, 222
and input, 69, 533
and POD-like objects, 263, 266
and Properties, 60–61
for results, 109, 111, 124, 184
Glasserman, P., xvi, 3, 281, 291, 307, 348, 368, 393, 399, 457
Global variables, 21
evil, 62
and state, 40, 50, 550, 553–554
Go-faster stripes, 247
Golden section search. See Minimization algorithms, golden section search
GoTo, 28, 43, 614
and error handling, 42
Gubbins, 42
HACK, 50
Handles, 614
to files, 144, 147
to objects, 356, 361
Heath, Jarrow and Morton (HJM) model, 8
Hedging, 10–11
calibration to hedging instruments, 10, 393–396, 610
calibration, 12
computing hedge ratios, path-wise differentiation, 331–332
delta control variate, 321–322, 324, 342–343, 445
Heptanomial lattice, 105, 152
Heston model, 277–278, 328–330, 401, 431
calibration, 12
explicit solution, European call, 328
Feller condition, 435–436, 445
See also: Zero lock-out Heston
GBM, as perturbation of, 328
implementation
delta control variate, 445
explicit solution, 328
Monte Carlo, 438–439
the little Heston trap, 329
option valuation, 436–446
arithmetic average rate option, 443–444
and barrier options, 446
European call, benchmark, 439
quadratic option, 442–443
simulation. See Simulation, Heston process
variance reduction
and antithetic variates, 290
control variates
and auxiliary instrument, 319
auxiliary model, 319, 330, 438–445
delta control variate, 330, 445
and stratification, 316, 441
zero lock-out Heston (ZLH), 435–436, 438–445
See also: Cox, Ingersoll and Ross (CIR) model;
Black-Scholes model
Hitting time. See also: Stopping time
to a barrier, 454, 511
reification of, 97, 113
distribution, 396
to the early exercise boundary, 492, 498, 511
Holistic evolution. See Evolution, types, holistic evolution
Honte, 245, 615
Horror, 62
House-keeping, 101
Hubris. See Platonic application
I/O. See also: Files; Input; Output
channels. See Channels
See also: Streams, I/O
console, 164
form, 164
streams. See Streams, I/O
IDE, 121, 517
Idioms, 615
array expansion idiom, 147, 166, 174
catch and re-throw idiom, 44, 76, 531
error handling idiom, 42, 50
Not first time. See Static initialisation idiom
RAI idiom, 72, 76, 95, 141, 559, 616
singleton idiom, 63, 216, 563
Static initialization idiom. See Static initialization idiom
See also: Design patterns
If-Then, If-Then-Else
block If, 239
compound If, comparison with Select, 240
in-line If, 239
Ignorance, 517, 559
IID, 14, 32
Implements, 84–86, 557–558
and registration, 197, 201
Implied volatility, 38
See also: Exercises, implied volatility stream
See also: Root finding algorithms
volatility surface, 10, 12
Importance sampling, (Chapter 22), 345–369
average rate option, (GBM), 364, 366, 379–380, 381–382
bank of, 352, 368
combined with control variates, 371–372
digital option, standard error analysis, 346–348
IS densities
constant GBM density, 350–351
terminally modified density, 351–352
ITM options, 345–369
OTM options, 345–369
quadratic options, 363, 366, 379, 381
registration of, 352–356, 373, 377
See also: Variance reduction
In the money (ITM) options
and importance sampling, 345–369
and the LSLS method, exercise in, 479
and problems for Monte Carlo, 282, 315
and stratified sampling, 302
Indenting, 28, 621
Independent identically distributed. See IID
Indirection, 67, 81, 122, 266, 615
and wrapper objects, 271
See also: Dispatch
Industrial strength, 21, 101, 613, 616
Ingratiation, xvii, 29, 102
Inheritance, 614
in C++, 558
in VBA, 55
Initialization, 615
in application building, 28–30, 572–574, 585–589
default, 28, 553–554, 557
in-line, 28, 29
of objects. See Initialization of objects
and the Static initialization idiom. See Static initialization idiom
Initialization of objects, 553–556

*Class.Initialize*(). See *Events*, *Class.Initialize*
and *Dim-New*, 58
in the factory
embryonic, 127–130
polymorphic
  *ICreatable*, 194
  *IReuseable*, 195
and POD-like objects, 263
and *Properties*, 60–61
and RAII. See *RAII idiom with a SetValues() method*, 68–73
with a *SetValues()* method, 84–85, 91–97, 127
in VBA, 556–557

In-line comments, 622

Input. See also: Files, input from; Front-end, input from; I/O, channels
and decoupling, 90–91, 93
and encapsulation, 69, 75
and environmental data. See *Environment*
and the factory 20, 21
  the embryonic factory, 123, 127–130
  the polymorphic factory, 187, 197–206
  the semi-polymorphic factory, 212, 216, 218, 222, 225, 226
formatted input, 145, 148, 151
and the *InputManager* object, 69, 92, 127
and UDTs, 39, 46–47, 145, 155, 591

Instantiation, 56, 57–58, 615
and the *Dictionary* object, 262
and *Dim-New*, 57–58
and the factory
  embryonic, 127–130
  and an option book, 162–163
  polymorphic, 184–188, 198–203
  semi-polymorphic, 216, 222–225
and the *FileSystemObject* object, 139
on the fly, 185, 196, 207, 532, 544
pop an object, 196, 532, 544
and RAII. See *RAII idiom with a SetValues()* method, 84–85, 91–97, 127
and side effects, 183, 200
and the singleton pattern, 563–565
as a structural property, 553–556
in VBA, 556–557

*Integer*, 621
  casting, 244
  and DLLs, 545
  literals, 33

Integrated development environment. See IDE

Interest rate models
  See Cox, Ingersoll and Ross (CIR) model
  See Fong and Vasicek model
  See Gaussian affine models
  See *HJM* model

See Libor market model (LMM) model
See Vasicek model

Interfaces, 84–86
See also: Polymorphism
  application interface, 193, 228
  conforming object. See *Conforming object mix-in interfaces*, 193, 477, 615
  *ICreatable*, 194–195
  *IReuseable*, 195

Invariant, 615

Inverse transform method. See *Normal variate generators, inverse transform*

Invocation chain, 155, 183, 184, 615
Invoker, 19, 187

IS. See Importance sampling

ITM. See In the money options

Iterated Itô integrals, 403–406, 425
  moments of, 403, 405, 425
  simulation of, 404, 405, 406, 425
See also: Iterated Itô-Taylor discretization schemes

Itô-Taylor discretization schemes.
  0.5 strong. See *Euler discretization scheme*
  1.0 strong. See *Milstein discretization scheme*
  1.5 strong, 404–406
  in 1-dimension, 103, 408–410
  with CIR, 410
    bodges, 420
    order of convergence, 420–424
    with exotic processes, 416
    with GBM, benchmark, 409
    order of convergence, 413–414
    with Vasicek, benchmark, 410
    order of convergence, 414
  2.0 weak, in 1-dimension, 410–411
  with CIR, 411
    bodges, 420
    order of convergence, 420–424
  and exotic processes, 416
  with GBM, benchmark, 410
  order of convergence, 413–414
  with Vasicek, benchmark, 411
  order of convergence, 414
  relationship with predictor-corrector, 413

Jäckel, P., xvi, 3, 281, 307, 312, 399, 417

Jamshidian formula, 326

Jump-diffusion models, 319
See also: Merton jump-diffusion model

Knock-in barrier option, 7, 97, 98, 113, 255, 316
Knock-out barrier option, 16, 95–97, 98, 113, 255, 290, 316, 396

Labels
  and “:”, 28
Index

and error trapping, 42–45, 50
and GoTo, 28
Laguerre polynomials, 464–465, 479–483
See also: Basis functions
Late bound, 139, 244, 262, 263
Lattice methods
binomial, 210, 567
branching probabilities, 105, 210, 457, 567, 569
and moment matching, 570
convergence, 575–576
American put, 452–453, 576
continuation values, 450, 568, 575
early exercise boundary, 452–453
comparison with PDE method, 593
European call, 576
heptanomial, 105, 152
moment matching, 570
pruning, 104, 576
Richardson extrapolation, 452
trinomial, 567–570
exercise stream, 38, 104–105, 152
implementation, 570–575
performance, 575–576
Layout, 68
indenting, 28, 621
Laziness, 275, 298, 495, 531, 550
LD. See Low discrepancy sampling
Legendre polynomials, 464–465, 466, 479–482
See also: Basis functions
Level, programming. See Programming level
Libor market model (LMM)
calibration, 12
martingale correction, 393
state variables, 12, 268, 449
Lifetime
of composit ed objects, 559, 613
of function arguments, 185
of Statics, 47
Linear equation methods
ordinary least squares regression (OLS), 445, 463, 466, 468
singular value decomposition (SVD), 445, 468
tri-diagonal solvers
See also: PDE methods, SOR; PDE methods, PSOR
LU tri-diagonal solver, 589, 595
Linear growth condition, 401–402
and CIR, 408
Lipschitz condition, 401–402, 416
and CIR, 402, 408, 410, 433
and Heston, 432
Liquid prices, and calibration, 10, 11, 610
Literals, 33, 554, 615
and performance, 236, 237
and telepathy, 194
LMM. See Libor market model
Local variables, 39, 47, 50
Log-Euler discretization schemes, with CIR, 419
blows up, 420, 421, 424
order of convergence, 420–424
Log-normal approximation discretization schemes, CIR, 418–419
convergence, 420–424
with Heston, convergence, 435–436
Log-normal density, 331, 526
central moments, 418
moments of, 331
Long, 621
and ^, 237
casting, 244
casting to Long, 238, 554
and comparison tests, 239
and Err.Raise, 39, 40
as a handle, 361
and input, 44, 203
as keys (in a Dictionary), 275
as loop counter, 245
and validation, 41, 44, 533
Long-step Monte Carlo, 26, 418, 429
with barrier options, 257
and efficiency, 378, 383, 384
and step frequency, 112, 261
Longstaff and Schwartz least squares (LSLS) Monte Carlo method. See LSLS method
Lookback call option, discretely reset, 305
Low discrepancy (LD) sampling, (Chapter 19), 307–317
See also: Stratified sampling
average rate options, 314–315, 315–316
geometrically averaged, 315
Bermudan puts, 482–483, 487–488
and bias, 314–315
and standard error, 314–315
See also: Sobol’ sequences
See also: Variance reduction methods
LSLS method
2-pass method, 505–509
American option, general, 456–457, 458
American put options, 459–460
See also, Option valuation, American put bias in, 474
bond options, 460–461
cash-flow formulation, 459
error in EEB estimate, 491
error in mimicking barrier option value, 491
using control variates, for early exercise boundary, 498–505
using control variates, valuation
Bermudan-early control variate, 493
Bermudan put control variate, 492
Bermudan-T1 control variate, 493
Bermudan terminal control variate, 493
LSLS method (continued)
   European put control variate, 492
   performance of, 497–498
   stock control variate, 492
   stopping time control variates, 491
See also: Continuation values; Early exercise boundary
LU tri-diagonal solver, 589, 595

Magic number, 615
main(), 19–21, 26
See also: Calling procedure
See also: Index to implementations, main()

Maintenance
   and comments, 26, 28
   and data representation, 265
   and objects, 549
   and re-factoring, 616
   and references, 58
   and tricky code, 234, 253
   and VBA compiler problems, 518
   and yukky code, 39, 578

Market
   computing values of observables, 389
   market value, 8, 38, 392
   money market, 449

Market component, of models, 8, 10
Martingale, in no-arbitrage pricing framework, 3, 424, 567
Martingale correction
   Libor market model, 393
   simulation and, 393–394
Martingale simulation, 393
MC. See Monte Carlo
Mé, 66, 92, 158
Measure
   Borel measure, 4
   equivalent martingale measure, 424, 567
   pricing measure. See Pricing measure
   risk neutral measure. See Pricing measure
Member functions. See Methods, of an object
Memory manager, 174
Mersenne twister, 251
Merton jump-diffusion model, 23
   explicit solution, European call, 23
Message
   See also: Request
   calling an object’s interface method, 556, 559–560, 615
Meta-classes, (Chapter 13), 211–229, 558–559, 615
   in C++, 559
   performance, 254–255
   reference counting, 212
   role of, 212–214
Method of false position, 605
   in exercises, 80, 104
   FalsePosition(), 606

Methods, of an object
   and objects’ implementation, 615
      Private methods, 59, 555
   and objects’ interface, 555, 615
      Public interface, 65
      See also: Request
   and POD-like objects, 263–266
Milstein discretization scheme, 37, 402–404
   in 1-dimension, 406–407
   in 2-dimension, 404
   with CEV, 103
   with CIR, 406
      order of convergence, 420–424
      positive values, 408
   convergence, theoretical order, 404
   with exotic processes, 416
   with GBM, benchmark, 37, 406
      order of convergence, 413–414
   with Heston, 432–433
      order of convergence, 435–436
   and iterated Itô integral, 403
   with Vasicek, benchmark, 406
      order of convergence, 414
Mimicking barrier options, 454–455
   and sub-optimal exercise, 473–474
   in two-pass method, 505–509
See also: Index to implementations, Mimicking barrier option
Minimization algorithms, 610–612
   golden section search, 152, 611
   exercises, 152
   GoldenSection(), 612
Mix-in interfaces. See Interfaces, mix-in interfaces
Models. See also: Stochastic differential equations;
   Discretization; Simulation
See also: Calibration
Black-Scholes framework, 3, 7, 25, 324
   See Black-Scholes model
CEV (Constant elasticity of variance) model, 102–103, 114
Cox, Ingersoll and Ross model. See Cox, Ingersoll and Ross model
Fong and Vasicek model. See Fong and Vasicek model
Gaussian affine models. See Gaussian affine models
Heath, Jarrow and Morton (HJM) model, 8
Heston model. See Heston model
interest rate models
   See Cox, Ingersoll and Ross model
   See Fong and Vasicek model
   See Gaussian affine models
   See Heath, Jarrow and Morton model
   See Libor market model
   See Vasicek model
jump-diffusion models. See Jump-diffusion models
Libor market model (LMM). See Libor market model
market component of models, 8, 10
Merton jump-diffusion. See Merton jump-diffusion
SABR, 8, 12
Shimko, Tejima and Van Deventer model, 277, 279
stochastic volatility models
See Heston model
See SABR model
See also: Fong and Vasicek model
stock models
See Black-Scholes model
See CEV model
See Jump-diffusion models
See Merton jump-diffusion model
See Shimko, Tejima and Van Deventer model
Vasicek model. See Vasicek model

Modules
class modules. See Class modules
code modules. See Standard modules
standard modules. See Standard modules

Moment freezing discretization schemes, 327, 329
with CIR, 419–420
order of convergence, 420–424
with Heston, 434
order of convergence, 435–436
Moment matching, lattices, 570
Moment matching correction, 394–396
and antithetic variates, 291
and calibration, 394
and holistic evolution, 16, 294
and put-call parity, 398
and stratified sampling, 291
and storage, 394, 395

Moment matching discretization schemes
See Log-normal approximation discretization schemes
See Moment freezing discretization schemes
See Normal approximation discretization scheme

Moments
central moments. See Central moments
of CIR process, 418
of iterated Itô integrals, 403, 405, 425
and weak discretization schemes, 411
Money market account numeraire, 3
See also: Accumulator account numeraire
Monitor, 72, 91
See also: Index to implementation, Counter, step

Monte Carlo. See also: Simulation; Index to implementations, Monte Carlo
and American options. See Option valuation,
American put
See also: Bermudan option valuation, by Monte Carlo
Bermudan option valuation. See Bermudan option valuation, by Monte Carlo
bias. See Bias
See also: Discretization, convergence discretization. See Discretization

efficiency gain, 282
evolution, direction. See Evolution, direction
evolution, types. See Evolution, types
long-step. See Long-step Monte Carlo
option valuation. See Option valuation
plain Monte Carlo. See Index to implementations,
Plain Monte Carlo
Richardson extrapolation, and the American put.
See Richardson extrapolation
sample paths. See Sample paths
short step, 112, 254, 431
simulation. See Simulation
slices. See Slices
standard error. See Standard error

variance reduction
antithetic variates. See Antithetic variates
control variates. See Control variates
in the Heston model. See Heston model, variance reduction
importance sampling. See Importance sampling
low discrepancy sampling. See Low discrepancy sampling
spectral decomposition, 294
stratified sampling. See Stratified sampling

Moro. See Index to library functions, cndev()
Morris. See Index to library functions,
gamma_inverse()
Moshier. See Index to library functions,
beta_inverse()

Names
disambiguation, 44, 58, 181
inappropriate names, 60
of loop counter, 468
name clash, 44, 531, 546
See also: Names, disambiguation

naming conventions, Private data members, 59
qualified names, 44, 130
analogy with object syntax, 550, 556, 565
and library functions, 531
and With, 66
range names, 59

Namespace, 183, 531
Natural basis functions, 464, 465, 479–482
See also: Basis functions
New, 57–58, 61, 556
and Dim, 57–58, 68, 72, 76
and polymorphism, 84–86, 558
Newton-Raphson method, 608–610
in exercises, 209
NewtonRaphson(), 609
NIG. See Normal inverse Gaussian process
Nitty-gritty, coding, 13, 233, 234
Non-central $\chi^2$ distribution, 391, 417
Non-polymorphic interface, 153
Non-virtual interface, in VBA. See Non-polymorphic interface
Normal approximation discretization scheme, CIR, 418
Normal density, 293, 536
Normal distribution, 22, 25, 251, 536
Normal inverse Gaussian (NIG) process, 292
Normal variate generators
  from 12 uniforms, 23
  inverse transform, 6, 246, 251
  and the bridge distribution, 391
  performance, 246, 249–251, 254–255
  and stratified sampling, 292, 312, 441
See also: Index to library functions, cndev()
See also: Application methods, Application.NormSInv()
polar rejection, 33
  and ~, 238
  GetNormal(), 33
  normal1(), 22, 535
  performance, 246, 249–251, 254–255
Not first time idiom. See Static initialisation idiom
Nothing, 57–58, 60, 556
  and anon(), 196
  and reference counting, 61–62
Numeraire, 3, 567
  accumulator account numeraire, 325, 424, 581
  money market account numeraire, 3
See also: Accumulator account numeraire
Numerical integration
  extended trapezium rule. See Extended trapezium rule
  and LD sampling, 307
  library quadrature objects, 542
  and the Monte Carlo method, 4–6, 11, 14, 389, 390
  and option pricing in the Heston model, 317, 328–329
Simpson’s rule. See Simpson’s rule
  trapezium rule. See Trapezium rule
Numerical methods. See also: Index to implementations
bodge. See Bodge
convergence, of numerical methods
  Monte Carlo. See Option valuation
  See also: Discretization, convergence
  See Lattice methods, convergence
  See PDE methods, convergence
  See Root finding algorithms, convergence
  See also Simulation
discretization error. See Discretization, convergence
efficiency. See Efficiency gain
See also: Performance; Variance reduction
lattice methods. See Lattice methods
  linear equation methods. See Linear equation methods
  minimization algorithms. See Minimization algorithms
Monte Carlo. See Monte Carlo

See also: Simulation
  numerical integration. See Numerical integration
  PDE methods. See PDE methods
  pre-computing, 233, 234
  quadrature. See Numerical integration
  random number generators
    See Normal variate generators
    See Uniform variate generators
Richardson extrapolation, and the American put
  See Richardson extrapolation
  root finding algorithms. See Root finding algorithms
  simulation. See Simulation
  See also: Discretization; Random number generators
speed. See Speed
Numerical Recipes
  library functions, 531, 533, 536
  low discrepancy sampling (LD) sampling, 307, 310
  minimization, 610, 611
  quadrature methods, 543
  random number generators, 33, 74, 246, 251
  singular value decomposition (SVD), 468
  tri-diagonal solvers, 584, 596, 598
  root finding algorithms, 603, 606
  vade mecum, xvi
NVI. See Non-virtual interface

Object, 65
  and anon(), 196
  and the factory, 223
Object declaration, references, 56, 57–59, 556–557, 615
Object definition, in VBA, 56, 59–61
  StopWatch example, 63–65
See also: Class modules
Object organization. See also Design patterns
  aggregation, 613
  association, 613
  bidirectional association, 613
Object oriented programming. See also: Programming
  concepts; Object organization
dispatch. See Dispatch
  encapsulation. See Encapsulation
  exception safety, 188, 217, 614
  functor, 71, 79, 191, 553, 557, 565, 614
  interfaces. See Interfaces
  message. See Message
  meta-classes. See Meta-classes
  object declaration, references, 56, 57–59, 556–557, 615
  object definition. See Object definition
  object organization. See Object organization
  objects. See Objects
  plain old data (POD). See Plain old data
polymorphism. See Polymorphism
reference counting. See Reference counting
reification. See Reification
request. See Request
Simula 67, 549
strong typing. See Strong typing
tracking. See Tracking
in VBA, 55–102
See also: Class modules
See also: VBA intrinsic objects
See also: VBA OOP reserved words
See also: VBE object library
Objects. See also: Excel; VBA intrinsic objects; VBA
OOP reserved words
binding. See Binding
casting, 196
constructors. See Constructors
copy assignment, 55, 553, 556
data members. See Data members
declaration, references, 56, 57–59, 556–557, 615
definition. See Object definition
destructor. See Destructor
expose. See Expose
getters. See Getters
initialization. See Initialization
instantiation. See Instantiation
member functions. See Methods
methods. See Methods
references. See References
setters. See Setters
state. See State, of an object
See also: Object oriented programming
OLS. See Ordinary least squares regression
On Error, 42–43, 50, 517
Ontology, 95
OOP. See Object oriented programming
operator(), 71, 614
Operator overloading, 556, 557, 615
Optimization
compiler optimization
of loop contents, 235
of stub procedures, 242
and OLS regression, 463, 468
premature, 233
See also: Minimization algorithms
Option Explicit, 27, 40, 622
Option pricing formulae
in the Black-Scholes model
barrier down and out call, 17
Bermudan-2 put, 528–529
compound call option, 529
continuously reset geometrically averaged average rate option, 18
discretely reset geometrically averaged average rate option, 117, 523–526
European call, 25
quadratic payoff option, 526–528
in the CIR model
pure discount bond, 430
in the Heston model
European call, 328–329
in the Merton jump-diffusion model
European call option, 23
in the Vasicek model
bond call option, 326
pure discount bond, 17
Option Private, 58
Option valuation, by Monte Carlo, convergence
American put, 486–488
in basis functions, 487
extrapolation from Bermudan put value, 509
in sample paths, 486
and stratification, 487
average rate options
antithetics, 288–289
combined methods
GBM, 379, 381–382
Heston, 443–446
control variates, 338–342
importance sampling, 364, 366
low discrepancy sampling, 314–315
stratified sampling, 298–304
barrier options, 102
Bermudan puts, by LSLS, 473–474, 484–486
2-pass method, 505–509
in basis functions, 473–474
using rollback control variates, 499–505
in sample paths, 473
and stratification and LD sampling, 482–483, 487–488
using valuation control variates, 497
compound option (2 resets)
in basis functions, 483–484
European call
convergence, Heston, 435–436
importance sampling, 362, 364–366
variance reduction, combined methods
GBM, 378, 380–381, 384–386
Heston, 435–445
out of the money (OTM) options
average rate options
antithetics, 289
combined methods, 381–382
control variates, 339, 342
importance sampling, 366
low discrepancy sampling, 315
stratified sampling, 302
European call
combined methods 380–381
importance sampling, 364–366
Option valuation (continued)

quadratic option
combined methods, 381
importance sampling, 366
quadratic option, 323, 526–528
combined methods
GBM, 379, 381
Heston, 442–443, 445
cost variates
delta control variate, 343
payoff matching control variate, 323–324, 343–344
performance, 342–343
importance sampling, 363, 366

Optional sampling theorem, 492

Options. See also: Numerical methods; Option pricing formula; Option valuation
American options. See American options average rate options. See Average rate options
barrier options. See Barrier options
Bermudan options. See Bermudan options bond options. See Bond options
book of options, (Chapter 10), 153–176
compound option, 483–484, 529
contract, option, 10
convertible bond, 450, 461
digital option, importance sampling example, 346–348
European call options. See European call options
European put options. See European put options
extreme options, 315, 352, 368
in the money (ITM) options. See In the money (ITM) options
lookback call option, discretely reset, 305
out of the money (OTM) options. See Out of the money (OTM) options
quadratic option, 323, 526–528
See also: Option valuation
redeemable American bond option, 461
redeemable, callable, convertible bond, 461
reset dates. See Reset dates
See also: Put-call parity

Ordinary least squares regression (OLS), 445, 463, 466, 468

Ornstein-Uhlenbeck process
See also: Vasicek model
bridge distribution of integrated process, 391
exact solution to SDE, 427
sampling from, 327
simulation. See Simulation, Ornstein-Uhlenbeck process

OTM. See Out of the money (OTM) options

Out of the money (OTM) options
and combined variance reduction methods, 380–382
and importance sampling, (Chapter 22), 345–369
and problems for Monte Carlo, 282

See also: Option valuation, out of the money options

Output, 22
See also: Files, output to; Front-end, output to
See also: I/O, channels; Index to implementations,
Counter, step
to the client, 18, 123, 130
and decoupling, 75, 90–91, 93, 100
from the destructor, 72, 76, 92, 169
and encapsulation, 39, 45, 67, 75
and environmental data. See Environment
and events, 119
and the factory, 20–21
the embryonic factory, 122, 123, 127–133
the polymorphic factory, 198–200
the semi-polymorphic factory, 212, 217, 226
formatting output, 65, 137, 144, 151
and the OutputManager object, 71–72, 92
and registration and call-back, 100

Parameter classes, 194
Partial binary chop evolution, 294–295
See also: Evolution, direction, binary chop
Partial differential equation. See PDE methods
Path-wise differentiation, and hedge ratios, 331–332
Path-wise evolution. See Evolution, types, path-wise evolution

PDB. See Pure discount bond

PDE methods
comparison with lattice method, 592, 602
convergence, 582, 591–595, 601–602
for American put, 591, 593, 601
continuation values, 584, 591, 598
for European call, 593
higher order convergence
projective SOR (PSOR), 598
successive over-relaxation (SOR), 595–602
Crank-Nicolson, (Appendix G), 581–602
performance, 592–595
and early exercise, 449, 584, 591, 593
See also: PDE methods, SOR; PDE methods, PSOR
problems in higher dimensions, 11
PSOR, 598, 599, 601
SOR, 595–602
transforming the PDE, 584–585

Performance
arithmetic functions, 241
arithmetic operators, 236–238
arrays. See Arrays, performance
assignment, 238, 241, 243, 244, 245
C++, xv, 36, 235, 241
casts, 244, 248
comparison with C++, 36
concatenation, Strings, 241
Const variables, 236, 245, 247
control statements, 239–241, 245
data structures, (Chapter 16), 261–277
  slice representations, 268–277
  state representations, 266–268
  1-factor, 272–273
  n-factor, 273–277
  plain state, 273–275
  structured state, 275–277
  and the factory pattern, 254–258
  global variables, and Statics, comparison, 238, 245–247
I/O, 174–176
if statement, 239
implicit type conversion, 244
lattice, trinomial, 575–578
literals, 236, 237
loops, 240–241
  and meta-classes, 254–255
  and Monte Carlo
    evolution method. See Evolution, types, performance comparisons
    option valuation. See Option valuation
    programming level. See Index to implementations, Plain Monte Carlo
  numerical integration, discount factors, trapezium rule, 428
  objects in VBA
    in plain Monte Carlo. See Index to implementations, Plain Monte Carlo
  See also: VBA intrinsic objects
  PDE method, Crank-Nicolson, 592–595
plain old data, 266–268, 273–277
  power operator, 237
  procedure calls
    cost of a call, 242
    typing arguments, 243
    typing return values, 243
programming level. See Index to implementations, Plain Monte Carlo
  random number generators
    See Normal variate generators
    See Uniform variate generators
  rounding a division, 238
  rounding a double, 238
Select statement, 240
simulation
  of CIR process, 420–424
  of GBM, 413–414
  of Heston process, 435–436
  of Ornstein-Uhlenbeck process, 413–414
Statics, 245–247, 254
  Statics and globals, comparison, 238, 245–247
  Static initialization idiom, 247
  of a step counter, 250–251, 248, 576
UDTs, 244
Variant arrays, 275–276
VBA library functions. See Index to library functions
VBA primitive types. See VBA primitive types
VBA reserved words. See VBA reserved words
Pi series expansions
  Bailey, Borwein and Plouffe, 183
  Beeler et al., 37
  Euler, 183
  Gosper, 37
  Leibniz, 183
  miscellaneous, 37, 191, 209
  Ramanujan, 209
  Schellbach, 183
Plain Monte Carlo. See Index of implementations, Plain Monte Carlo
Plain old data (POD), 263, 555, 615
  See also: User defined types
  performance, 266–268, 273–277
  POD-like objects, 263–265
  slice representation, 270, 273–277
  state representation, 265–266
  and VBA containers, 270, 273
Polymorphic applications. See Programming level, level 4
Polymorphic factory. See Factory pattern, polymorphic factory
Polymorphic factory applications. See Programming level, level 6
Polymorphic hierarchy, 615
Polymorphism. See also: Interfaces
  base class. See Base classes
  derived objects. See Derived objects
  inheritance. See Inheritance
  polymorphic hierarchy, 615
  See also: Programming to an interface
Power functions, 464–465, 466–468, 479–482
  See also: Basis functions
Pre-computing, 233, 234
Predictor-corrector method, 411–413
  with CIR, 413
    order of convergence, 420–424
    with exotic processes, 416
    with GBM, benchmark, 412
    order of convergence, 413–414
    with Heston, 433–434
    order of convergence, 435–436
  partial predictor-corrector, 435
Predictor-corrector method (continued)
with Vasicek, benchmark, 413
order of convergence, 414
Pre-extraction, 272, 273
Prettiness, 26, 28
Pricing measure, 3
and importance sampling, 345
Private. See Private data
Private data, 31, 50, 616
and composition, 559
and decoupling, 75
and module scope, 550
naming convention, 59
and object implementation, 58, 59, 554–555
and POD-like objects, 263
and Properties, 60–61
and visibility, 216
Procedural applications. See Programming level, level 2
Procedural mindset, 153
Procedures. See also: Index to library functions
class events. See Events
Functions. See Functions
Properties. See Properties
signature. See Signature
stubs. See Stubs
Subs. See Subs
Profile, 235, 249, 616
Programming concepts. See also: Data; Object oriented programming
calling procedure. See Calling procedure
client. See Client
cohesion, 39, 613
comments. See Comments
concept code, 565, 613
continuation lines, 44
coupling, programming, 40, 91, 223, 613
decoupling. See Decoupling
decoupling. See Decoupling
delegation. See Delegation
environment. See Environment
error handling. See Error handling
error trapping. See Error handling
evil. See Evil
gubbins, 42
honte, 245, 615
idioms. See Idioms
indirection. See Indirection
industrial strength, 21, 101, 383, 613, 616
invocation chain, 155, 183, 184, 615
invoker, 19, 187
labels. See Labels
maintenance. See Maintenance
monitor. See Monitor
objects. See Objects
Platonic application, 19–20
polymorphic applications. See Programming level, level 4
polymorphic factory applications. See Programming level, level 6
procedural applications. See Programming level, level 2
procedures. See Procedures
profile, 235, 249, 616
programming level. See Programming level
psychic connection. See Psychic connection
re-factor, 616
references. See References
re-usability. See Re-usability
sentinel. See Sentinel
stack. See Stack
syntactic sugar, 42, 531, 556, 617
telepathy. See Telepathy
telepathic guarantee, 166
tricky code, 234, 248, 253, 272, 273, 617
validation. See Validation
yukky applications. See Programming level, level 0, yukky
Programming level
level 0, yukky, (Appendix F), 567–579
level 1, basic procedural, (Chapter 3), 25–38
level 2, full procedural, (Chapter 4), 39–51
level 3, non-polymorphic objects, (Chapter 5), 55–76
level 3b, non-polymorphic objects (invocation chain), (Chapter 5), 76–78
level 4, polymorphic objects, (Chapter 6), 81–105
level 5, embryonic factory, (Chapter 8), 119–133
level 6, design patterns (polymorphic factory), (Chapters 11 and 12), 179–210
level 6a, design patterns (semi-polymorphic factory), (Chapter 13), 211–229
Programming to an interface, 30, 89, 616
See also: Interfaces
Projective successive over-relaxation (PSOR), 598, 599, 601
See also: PDE methods
Proper
code, 298
compiler problem solution, 521
factory, 92
help, 517
lattice method, 576
random number generator, 34
See also: Honte
Properties, 60–61, 556, 616
See Properties
Properties
See also: Getters; Setters
and data members, 60–61, 69, 72, 92, 614, 616
restricting access to, 157, 159
and initialization, 60–61
and polymorphism, 85–86
and Private data, 60–61
and scope. See Private data
and SetValues(), 93, 95, 127
Prototype pattern, 223
See also: Design patterns
Pruning. See Lattice methods, pruning
PSOR. See Projected successive over-relaxation
Psychic connection, 225, 228, 616
See also: Telepathy
Public. See Public data
Public data, 31, 39, 50, 55, 56, 616
and fixed length Strings, 263
and library procedures, 40, 44
and object interfaces, 58–61, 75, 84, 550
and Option Private, 58
and UDTs, 554, 555
See also: Plain old data
Pure discount bond (PDB)
in the CIR model, 430
in the Fong and Vasicek model, 325–326
in the Shimko, Tejima and Van Deventer model, 279
in the Vasicek model, 17
See also: Bond options
Put-call parity, 393
martingale correction, 393–394
in Monte Carlo method, violation of, 392, 393, 398
Quadratic option, 323, 526–528
See also: Option valuation, quadratic option
See also: Index to implementations, Quadratic option
Quadrature. See Numerical integration
Qualified names. See Names, qualified names
Quantiles
and importance sampling, 347
and stratified sampling, 291–292
Queues, 263
Quick access toolbar, 517
RAII idiom, 72, 76–77, 141, 559, 616
RaiseEvent, 120
Raising errors. See Error handling, raising errors
Random access files, 137, 145, 148, 150
Random number generators
See Normal variate generators
See Uniform variate generators
Range names, 59
Rare events, 315
Rasmussen, N., 447, 459, 470, 494, 498, 509
Rectification functions, 327, 407, 420, 432, 446
Redeemable American bond option, 461
Redeemable, callable, convertible bond, 461
ReDim, 265, 266
ReDim Preserve, 147, 166, 174
See also: Memory management
Re-factor, 616
Reference counting, 61–63, 616
and meta-classes, 212, 214, 225
References, 61–63, 556
assignment, 556
dangling references, 60, 61
declaration of, 56, 57–59, 556–557
and maintenance, 58
reference counting. See Reference counting
self-references, 61
Registration. See also: Registration and call-back
automatic, 188, 197–203
in C++, 183
for control variates, 335, 373, 377
and the factory, 183, 184–188, 562
for importance sampling, 352–356, 373, 377
of objects, 179
for output, 100
side-effects, 183, 184, 200
and the singleton pattern, 563–565
of values, 74, 95, 97, 109
and the VBIDE object, 559
work-arounds, 211–229
See also: Design patterns
Registration and call-back, 55, 559, 616
and the factory, 562
for output, 100
See also: Design patterns
Reification, 616
of a grid, 586
of input data, 46
of options, 97, 109, 305, 562
of a slice, 261
of a state, 266
Request, 21, 556, 616
See also: Message
and the adapter pattern, 560
and control variates, 335, 337, 373, 439
and events, 121
to execute, 90, 107, 113, 159, 285
to the factory
embryonic factor, 123, 130, 162, 172
polymorphic factory, 183, 187–188, 200, 206
semi-polymorphic factory, 214, 216, 218,
221–223, 225, 228
and importance sampling, 352, 353, 356, 358, 361
for output, 92, 166–170
for results, 73, 95
Reset, VBA project, 257
Reset dates, 109–110, 116
and time steps, 110, 116, 261
See also: Options, average rate options; Options,
Bermudan put options
Resource acquisition is initialization. See RAII idiom
Resume, 42, 44, 50
Resume Next, 42, 517
Re-usability. See also: Index to library functions and files, 138
and functions, 21, 39, 242

REusable mix-in interface, 195
lack of, 35, 39, 549
and modules, 48
and objects, 56–57, 63, 75, 551

Ribbons, Excel, 180, 517
Richardson extrapolation, and the American put for the early exercise boundary, 453
with a lattice methods, 452
with Monte Carlo, 489, 491, 627

Ridder’s method, 606–608
in exercises, 80, 104
Ridders(), 608

Risk neutral measure. See Pricing measure
RMSE, 458, 471, 473–474, 499–505, 539

Root finding algorithms, 603–610
bisection method. See Bisection method
bracketing interval, 603
convergence, 604, 608
method of false position. See Method of false position

Newton-Raphson method. See Newton-Raphson method
Ridder’s method. See Ridder’s method

Root mean square error. See RMSE
Routine. See Procedures; Numerical Recipes
Run-time errors, 40, 85, 194–195, 217, 223, 244

SABR model, 8, 12
Sample paths, 4, 6, 7

See Evolution, types, path-wise; Simulation
Sample space, 4, 11
Sanity check, 92, 266
Scaled basis functions, 465

See also: Basis functions
Schiller, 66
Scope, 616
file scope, in C++, 179
module scope, 58, 554

See also: Private data
object scope. See Data members
procedure scope. See Local variables
project scope. See Friend data
of Properties. See Private data
public scope. See Public data
explicit destruction, 58, 60, 62, 101
implicit destruction, 58, 556
statement scope, 196, 544
of variables in blocks, 47

Scripting runtime library, 139, 262, 616

Scripting.Dictionary 262
Scripting.FileSystemObject, 139
SD. See Standard deviation
SDE. See Stochastic differential equations

SE. See Standard error
Select-Case, 616
and “;”, 29
and Enums, 153, 177, 585
and level 1 design, 551, 574, 585
non-polymorphic object creation, 83, 92, 102, 127, 153
performance, 240
polymorphic object creation, 184, 188
semi-polymorphic object creation, 216, 223, 226
Sentinel, 616

and the Static initialisation idiom, 47, 50, 616
Sequential files, 137, 144, 147
Serialization, 127, 153–155, 352–353, 360, 616
Set, 615
and Collection elements, 263
and events, 121
and interfaces, 85, 558
and object instantiation, 56, 57–58, 61, 556

See also: Nothing; Property; RAI idiom
Sets, 222, 263

Setters, 616

See also: Initialization
and encapsulation, 55, 554, 557
and POD-like objects, 263–265
and Properties, 60–61, 556

Shimko, Tegima and Van Deventer model, 278, 279
Short rate models
See Cox, Ingersoll and Ross model
See Fong and Vasicke model
See Gaussian affine models
See Vasicke model

Short step Monte Carlo, 112, 254, 431
Signature. See also: Procedures, stubs
of class events, 59
and DLLs, 545
and events, 121
and interfaces, 84–86
of library functions, 531–542
and option definition, 95

Simpson’s rule, 543
library object, IntegratorSimpson, 542

See also: Numerical integration
Simula 67, 549

Simulation. See also: Discretization; Monte Carlo;
Random number generators
Cox, Ingersoll and Ross process, 417–424
1.5 Itô strong, 410, 421
2.0 Itô weak, 411, 421
convergence rates, 420–424
Euler, 406, 421
exact simulation, 417–418
log-Euler, 419, 421
log-normal approximation, 418–419, 421, 434
Milstein, 406, 408, 421
moment freezing, 419–420
and negative values, 407–408
normal approximation, 418–419, 421
performance, 420–424
predictor-corrector, 413, 421
and rectification functions, 327, 420
of a transformed SDE, 407–408
discount factors, 424–429
convergence rates, 428–429
direct approximation
bias correction, 426
Gandhi and Hunt, 426
integral approximations
histogram, 424
trapezium, 425
Itô-Taylor approximations
0.5, Euler, 425
2.0 weak, 425–426
exact simulation
CIR process, 417–418
GBM, 32
Heston process, Broadie and Kaya, 432, 446
and moment freezing, 434
Ornstein-Uhlenbeck process, 427
exact simulation, absence of
and bias, 390
CEV, 103
and discretization, 401, 407
Fong and Vasicek process
Euler, 328
moment freezing, 328
geometric Brownian motion (GBM)
convergence rates, 413–414
exact simulation, 26, 32, 118, 413
Heston process, (Chapter 27), 431–446
1.5 Itô strong, 433
Broadie and Kaya, 432, 446
convergence rates, 435–436
Euler, 279, 432
Milstein, 432–433
moment freezing, 329, 434
performance, 435–436
predictor-corrector, 433–434, 435
and rectification functions, 432, 446
of iterated Itô integrals, 404, 405
martingale correction, 393–394
martingale simulation, 393
moment matching correction. See Moment matching correction
Ornstein-Uhlenbeck process. See also: Vasicek model
1.5 Itô strong, 409
2.0 Itô weak, 411
convergence rate, 414
Euler, 406
exact simulation, 413, 427
Milstein, 406
performance, 413–414
predictor-corrector, 413
Singular value decomposition (SVD), 445, 468
Slice-wise evolution. See Evolution, types, slice-wise
Slices, 6, 15
See also: Evolution, types, slice-wise and containers, 268–271
performance
1-factor state, 271–273
3-factor state, 273–277
slice object, 265, 277
Sobol’ sequences, 307–310
Bratley and Fox, 307
direction numbers, 307
SOR. See Successive over-relaxation
Space-delimited files, 144
Spectral decomposition, 294
Speed
of data structures, (Chapter 16), 261–279
of programming level, (Chapter 15), 249–259
of VBA, (Chapter 14), 233–248
See also: Convergence, of numerical methods;
Variance reduction; Performance
Squeaky, 216
Stack
communication, up, 39
errors and, 39, 42–44, 60
events and, 39
depth, 101, 296
exiting from 40, 495, 511
unwinding, 21
Standard deviation. See also: Normal distribution
and bias, 474, 485
and pruning, 577–578
and standard error, 6, 119, 396
with LSLS, 483, 487, 491, 498
with stratified sampling, 301, 304, 310
Standard error, 6, 31
See also: Variance reduction
and the accumulator object, 72–74
and bias, 508
and error trapping, 111
estimated as standard deviation, 119
Standard error (continued)
and importance sampling, 346–348
and LD sampling, 314–315
and residual error, 422, 428, 436
stratified sampling, 301
Standard modules
and events, 120
globals and persistent state, 40, 50, 551–553
as proto-objects, 553–556
as a singleton object, 228, 564–565
splitting code between, 39, 44, 45, 48
State
arrays and multidimensional state, 263, 265–268
Booleans as state, 97, 107, 111–113, 256
class modules and state, 40
the Collection object and states, 261, 262–263
Doubles state, 265–268, 273–277
and encapsulation, 265, 272
globals and persistent state, 40, 50, 551–553
of an object, 40, 55, 553–554, 616
and slice representations, 268–277
1-factor, 272–273
n-factor, 273–277
plain state, 273–275
structured state, 275–277
state representations, 265–268
state space, 11, 14, 450
and Static variables, 40, 550, 554
user defined types and state, 265–266
wrapping a state, 270
See also: State variables
State space, 11, 14, 450
State variables
and market values, 12
and modelling, 8–10
number of, 11–12, 582, 602
short rate models
See Cox, Ingersoll and Ross model
See Fong and Vasicek model
See Gaussian affine models
See Vasicek model
stochastic volatility, as a state variable
See Heston model; Fong and Vasicek model
traded asset, as a state variables
See Black-Scholes model; Heston model; Merton jump-diffusion model
Static. See Static variables
Static initialization idiom, 47–48, 50, 555, 616
in stat_ran0(), 33
performance, 244, 245–247
problems with, 48, 254, 257
Static variables
and globals, 33, 245–247
lifetimes, 47, 257
Reset, 257
performance, 245–247, 254
and state, 40, 550, 554
and the Static initialization idiom, 33, 47–48
std::cout(), 138
std::exp(), 241
Step, 29, 241, 247, 576
Stochastic differential equations (SDE)
See also: Discretization; Simulation
CEV (Constant elasticity of variance). See CEV model
Correlation. See Correlation
Cox, Ingersoll and Ross (CIR) process. See Cox, Ingersoll and Ross model
densities. See Distributions
distributions. See Distributions
exact solutions. See Exact solutions, to SDE
exotic processes, 416
Feller condition. See Feller condition
Feynman-Kac representation, 581
Filtration, 3, 4
Fong and Vasicek, 325
geometric Brownian motion. See Geometric Brownian motion
Heston process. See Heston model
hitting time. See Hitting time
iterated Itô integral. See Iterated Itô integral
linear growth condition. See Linear growth condition
Lipschitz condition. See Lipschitz condition
martingale, in no-arbitrage pricing framework, 10
measure. See Measure
Merton jump-diffusion. See Merton jump-diffusion model
moments. See Moments
numeraire. See Numeraire
optional sampling theorem, 492
Ornstein-Uhlenbeck process. See Ornstein-Uhlenbeck process
Poisson process. See Poisson process
sample space, 4, 11
state space, 11, 14, 450
state variables. See State variables
stochastic process. See Stochastic process
stopping time. See Stopping time
Wiener process. See Wiener process
Stochastic mesh method, Bermudan option valuation, 457
Stochastic process
for asset values, 3, 14
discretization of, 389
and evolution, 261
Lévy processes, 8
See also: Discretization; Models; Simulation; Stochastic differential equations
Stochastic volatility models
See Heston model; SABR model
See also: Fong and Vasicek model
Stock models
See Black-Scholes model
See CEV (Constant elasticity of variance) model
See Jump-diffusion models
See Merton jump-diffusion model
See Shimko, Tejima and Van Deventer model

Stopping time
for an American put, 454
sampling at stopping times, 491
stopping time control variates, 491–494

Storage, 13, 14
and evolution, 294–295, 296
moment matching, 394, 395
See also: Files; VBA containers

Strategy pattern, 562–563, 617
and StopLogs, 83
See also: Design patterns

Stratified sampling, (Chapter 18), 291–305
See also: Low discrepancy sampling
American put, 487–488
average rate options, 298–304, 443–444
Bermudan put, 482–483, 487–488
Brownian bridge, 292–293
combined with antithetic sampling, 304
evolution, direction of, 294–295
and Heston, 316, 441
ITM options, 302
and moment matching, 291
of normal variates, 292, 312, 441
correlated normals, 441
quadratic option, 442–443
and standard error, 301
stratification times, 302–304
of uniform variates, 291–292, 297–298
Wiener process, 295–298
See also: Variance reduction
Stratonovich integral, 406

Straw man, 176
Streams, I/O, 19, 78, 91, 218–222
in C++, 137, 138
and the TextStream object, 138–141
See also: Channels

Striding, 268–270, 617
performance, 273–274

String
casting
and Dates, 145, 148
from String, 41
to String, Cstr(), 170
concatenation, 241
and DLLs, 545
and Err.Raise, 39, 40
fixed length Strings, 263, 546
input, 18, 92, 127, 164, 200
from file, Input(), 144
as keys

Subs, 26, 45, 621
See also: Functions; Procedures

Successive over-relaxation (SOR), 595–602
See also: PDE methods

SV. See Stochastic volatility models
SVD. See Singular value decomposition
swap(), 217–218

Syntonym classes, 195–196
Syntactic sugar, 42, 531, 556, 617

t-distribution, 536
Tab-delimited files, 144
Telepathic guarantee, 166

Telepathy, 617
and application building, 25–29, 570–572, 585
and interfaces, 84–86, 130, 557
and procedure call performance, 242
and the VBA IDE, 121, 557

Sub-optimal exercise error, 449, 454–455, 471, 474, 487

Subs, 26, 45, 621
See also: Functions; Procedures

Subclassing, 176

Template pattern, 562–563, 617
See also: Design patterns

Terminally modified IS density, 351–352
See also: Importance sampling

TextStream object, 140–141
file output, 141
and the FileSystemobject object, 138–139
and streams, 138, 139
this, 66

ThisWorkbook
factory code, 183–188, 197–203
I/O, 166, 170. See also: Front-end
VBProject object, 180–183

Tracking, 62–63, 617
meta-class data, 214, 225
Traded asset, models with
  See Black-Scholes model; Heston model; Merton
  jump-diffusion model
Transformed SDE, discretization of, 407–408
Transforming a PDE, 584–585
Transparency, 271, 557, 615
Trapezium rule, 543
  and discount factors, 391, 425
  performance, 428
  library object, IntegratorTrapezium, 542
  See also: Numerical integration
Tricky Code, 234, 248, 253, 272, 273, 617
Tri-diagonal solvers. See Linear equation methods,
  tri-diagonal solvers
Trinomial lattice. See Lattice methods, trinomial
Type. See User defined types (UDTs)
Type conversion
  to Date, 148
  to Long, 238
  of objects, 553–556
  See also: Casts; Conversion constructor
UDT. See User defined types (UDTs)
Uniform distribution
  and antithetic sampling, 284
  and stratified sampling, 291–292, 297–298
Uniform variate generators
  Mersenne twister, 251
  performance, 246, 249–251
  ran0(), 74, 238, 246–247
  ran2(), 246, 247, 535
  See also: Rnd()
Unit hypercube
  and antithetic variates, 283, 284
  and LD sampling, 307, 310
  and stratified sampling, 297, 298
User defined types (UDTs)
  and complex numbers, 540
  as data members, 39
  and encapsulation, 45
  and file I/O, 145
  random access files, 145
  serialization, 153–155, 352–353, 360
  and input, 46–47
  naked UDTs, 265, 267
  and objects, 55, 57, 549, 553–556
  performance, 266–268, 275–277
  and states, 265–266
  and VBA containers, 261–263, 270–271
  wrapped UDTs, 267
User interface. See Front-end

Validation, 40–45
  absence of, 28, 35, 39. See also: Programming level,
  level 1
  by casting, 185, 201
  and input, 43–44, 46, 92, 203
  and state, 40
  validation functions, 41–44, 127, 216, 533
  See also: Index to implementations, Utility code
  modules
  See also: Programming level, level 2; Error handling
Variable length Strings, 145, 149
Variance gamma (VG), 292
Variance reduction methods
  antithetic variates. See Antithetic variates
  control variates. See Control variates
  in the Heston model. See Heston model, variance
  reduction
  importance sampling. See Importance sampling
  low discrepancy sampling. See Low discrepancy
  sampling
  spectral decomposition, 294
  stratified sampling. See Stratified sampling
Variant
  and Line Input #, 144
  performance
    casting, 244
    and typing, in procedures, 243
    sub-types, 56, 244
    and UDTs, 262
    and validation, 41, 533
  See also: Variant arrays
  Variant arrays, 262, 264, 270
  and Dictionary object keys, 262
  performance, 275–276
Vasicek model, 400
  as auxiliary model, 326
  and bias correction for CIR, 392
  conditional discount factor, 426, 428
  exact solution of SDE, 427
  explicit solutions
    bond options, 326
    term structure, 17
  Fong and Vasicek, as perturbation of, 325
  long rate, 427
  simulation. See Simulation, Ornstein-Uhlenbeck
  process
VB2TheMax, 536
VBA. See also: Excel
  comparison with C++. See C++, comparison with
  VBA
  compiler. See VBA compiler
  constants. See VBA constants
  containers. See VBA containers
  control statements. See Control statements
  enumerative types. See Enums
  See also: VBA enumerative types
  error messages, 263, 519, 521
  events. See Events
functions
See Functions; Subs
See VBA intrinsic file functions and statements
See VBA intrinsic functions
See also: Index to library functions
See also: Procedures
modules. See Modules
OOP reserved words. See VBA OOP reserved words
See also: VBA reserved words
performance. See Performance
reserved words. See VBA reserved words
See also: Procedures
Scripting runtime library. See Scripting runtime library
settings, 517
VBA compiler
clean recompile, 190, 518–521
compile-time errors, 40, 42, 194, 223, 244
compiler problems, 518–521
VBA constants
vbCrLf, 4
vbObjectError, 41
See also: Err object
vbTab, 141, 148
See also: Tab-delimited files
VBA containers
arrays. See Arrays
Collection object. See Collection object
Dictionary object. See Dictionary object
plain old data (POD). See Plain old data
user defined types (UDTs). See User defined types
Variant arrays. See Variant arrays
VBA enumerative types
vbext_ComponentType, 181
vbext_ProcKind, 182
VBA intrinsic file functions and statements, 143–145
ChDir, 143
ChDrive, 143
Close, 143
CurDrive(), 143
Dir(), 143, 148, 149
EOF(), 143
and file input, 147–148
FileAttr(), 144
FileDialog, 143
FileCopy, 143
FileDateTime(), 144
FileLen(), 144
FreeFile(), 144, 147
Get, 138, 144
and random access files, 145, 150, 151
GetAttr, 144
Input(), 144
Input #, 144, 145, 148
Kill, 143, 149
Line Input #, 144, 148
Loc(), 143
LOF(), 144
and random access files, 150
MkDir, 143
Name, 143
Open, 130, 143
and random access files, 149, 150
and sequential files, 147
Print #, 144
Put, 144
and random access files, 145, 150, 151
Reset, 143
RmDir, 143
Seek(), 143
SetAttr, 143
Write #, 144
VBA intrinsic functions
Abs(), performance, 241
Atn(), 536
CDate(), 148
CDbl() and casting, 554
and file input, 148
and validation, 41
Chr(), 45, 140, 144
See also: VBA, constants
CLng() and bit arithmetic, 310
performance, 238, 247
and validation, 41
Cos(), performance, 241
CreateObject() and the Dictionary object, 262
and FileSystemObject, 139
CStr(), casting, 170
Exp(), performance, 238, 241
Fix(), performance, 238
Format(), 144
IIF(), performance, 239
Int(), performance, 238, 247
IsNumeric(), and validation, 41
IsObject(), 66
Len(), and files, 149
Log(), performance, 237–238, 241
MsgBox(), 44 et cetera
Now(), 56
Randomize(), 34
Rnd(), 22, 34
Sgn(), performance, 241
Sin(), performance, 241
Split(), and file input, 148
Sqr(), performance, 237
VBA intrinsic functions (continued)
Round(), performance, 238
Tan(), 536
Time(), 56
Timer(), 56
and Stopwatch, 65, 88
VBA intrinsic objects
Collection object. See Collection object
Dictionary object. See Dictionary object
Err object. See Err object
FileSystemObject object. See FileSystemObject object
TextStream object. See TextStream object
VBA OOP reserved words. See also: VBA reserved words
Event, 120
Implements. See Implements
Is, 66
Me, 66, 92, 158
New. See New
Nothing. See Nothing
Object. See Object
Property. See Property
RaiseEvent, 120
Set. See Set
With, 66
WithEvents, 39, 120, 121
VBA primitive types
Boolean. See Boolean
Date, 56, 145, 148
Double. See Double
Integer. See Integer
Long. See Long
Single. See Single
String. See String
Variant. See Variant
VBA project, Reset, 257
VBA reserved words. See also: VBA OOP reserved words
ByRef. See ByRef
ByVal. See ByVal
Call, 621
Const. See Const
Dim. See Dim
Enum, 153–155, 166, 352–353, 605, 610
Function. See Functions
Option Explicit, 27, 40, 622
Option Private, 58
Private. See Private data
Property. See Property
Public. See Public data
ReDim, 265, 266
ReDim Preserve, 147, 166, 174
See also: Memory management
Static. See Static variables
Step, 29, 241, 247, 576
Sub. See Subs
Type. See User defined types (UDTs)
VBA settings, 517
VBCComponents object, 180
vbCrLf, 44
VBE object library, 179–183
access to the library, none, 211, 228
accessing the library, 179, 517
CodeModule object, 180–183
VBCOMPONENTS object, 180
VBProject object, 180, 182
vbext_ComponentType, 181
vbext_ProcKind, 182
VBIDE object, 179, 180, 188, 559, 562
vbObjectError, 41
See also: Err object
VBProject object, 180, 182
vbTab, 141, 148
VG. See Variance gamma
Virus checkers, 190, 211
Visibility, 31, 58, 216
See also: Scope
Visual basic for applications. See VBA
Visual basic editor. See VBE object library
Visual basic integrated development environment. See VBIDE object
Weak convergence criteria, 399–400
Weak will, 67
While-Wend, 240
Wiener process
and antithetic variates, 284
and bridge distribution, 292–293
and geometric Brownian motion, 3, 25
and stratified sampling, 295–298
With, 66
WithEvents, 39, 120, 121
Workbook events, Workbook_Open() and the factory, 183, 188, 193
and initialization, 554
Wrapper objects, 617
and containers, 263, 271
and the decorator pattern, 124
and DLLs, 546
option wrapper object example, 494
and polymorphism, 153
and RAII, 76
wrapping a state, 263, 270
See also: Design patterns
“X”. See Comments
Yucky. See Yukky application
Yukky application. See Programming level, level 0, yukky
Zero lock-out Heston (ZLH), 435–436, 438–445
Zhang, Kai, xvii, 387
ZLH. See Zero lock-out Heston