WILEY FINANCE

WITH CD-ROM

Implementing models of financial derivatives

Object Orientated Applications with VBA

NICK WEBBER

Implementing Models of Financial Derivatives

Implementing Models of Financial Derivatives

Object Oriented Applications with VBA

Nick Webber

WILEY
A John Wiley and Sons, Ltd., Publication

Library of Congress Cataloging-in-Publication Data

Webber, Nick.
Implementing models of financial derivatives : object oriented applications with VBA / Nick Webber.
p. cm.
Includes bibliographical references and index.
ISBN 978-0-470-71220-7
1. Derivative securities – Mathematical models. 2. Microsoft Visual Basic for applications. I. Title.
HG6024.A3W43 2010
332.64′570285543 – dc22
2010022097

A catalogue record for this book is available from the British Library.

ISBN: 978-0-470-71220-7 (hardback), ISBN: 978-0-470-66251-9 (ebk),
ISBN: 978-0-470-66173-4 (ebk), ISBN: 978-0-470-66184-0 (ebk)

Typeset in 10/12pt Times by Laserwords Private Limited, Chennai, India
Printed in Great Britain by Antony Rowe Ltd, Chippenham, Wiltshire

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

This page intentionally left blank

Contents

This page intentionally left blank

Preface

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 π. 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

Part I
A Procedural Monte Carlo Method in VBA

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.

1
The Monte Carlo Method

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.

Standard references for option valuation and theory, at various levels, are Hull (2008), Joshi (2003), and Wilmott (1998). A much more advanced mathematical treatment is Musiela and Rutkowski (1997). Very good references for the Monte Carlo method are Glasserman (2004), Jäckel (2002), Dagpunar (2007) and McLeish (2005).

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)).

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

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, $P_t = \exp(\int_0^t r_s\, \mathrm{d}s)$, and the short rate $r_t \equiv r$ is constant, equation (1.1) reduces to $v_t = \mathrm{e}^{-r(T-t)}\mathbb{E}_t[H(S_T)]$. 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

$$\mathrm{d}S_t = rS_t\,\mathrm{d}t + \sigma S_t\,\mathrm{d}z_t \tag{1.2}$$

for a Wiener process $z = (z_t)_{t\geq 0}$, where we have also assumed that the volatility σ is constant. In this world the value v^c_t 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 Ω where we can regard $\omega \in \Omega$ as representing a sample path for S over an interval $[0, T_{\max}]$ for some maximum time $T_{\max}$. Write $S_t(\omega) = (S_t^q(\omega))_{q=1,\ldots,Q}$ for the value of S at time t in state ω. 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

$$v_t = \mathbb{E}^{\mathbb{P}}\left[H(\omega)\frac{P_t(\omega)}{P_T(\omega)} \mid \mathcal{F}_t\right] \tag{1.3}$$

$$= \int_\Omega H(\omega)\frac{P_t(\omega)}{P_T(\omega)}\,\mathrm{d}\mathbb{P} \tag{1.4}$$

where $\mathbb{P}$ is the risk-neutral measure on Ω 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 ω of $P_t(\omega)$. In practice, H and P_t will depend on ω only through a finite (and small) number of state variables observed at a discrete set of times $\mathcal{T} = \{t_i\}_{i=0,\ldots,N} \subseteq [0, T_{\max}]$ for some maximum time $T_{\max}$.

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 $g : X \to \mathbb{R}$, and that we want to compute the integral

$$G(X) = \int_X g(x)\,\mathrm{d}x. \tag{1.5}$$

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 $G(X)$,

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

where $\Delta^M x = \mathcal{B}(X)/M$ stands in for the volume element $\mathrm{d}x$. As $M \to \infty$, $\tilde{G}$ 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

$$G(X) = \int_X g(x)f(x)\,\mathrm{d}x \tag{1.7}$$

for some density f, and for a European option

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

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 $\mathbb{F}$ 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 $G^{\mathbb{F}}(X) = \mathbb{E}_X^{\mathbb{F}}[g(x)]$, the expected value of $g(x)$ under F for $x \in X$. Set $U = F^{-1}(X)$. Then

$$G^{\mathbb{F}}(X) = \int_X g(x)\,\mathrm{d}\mathbb{F}, \tag{1.9}$$

$$= \int_X g(x) f(x)\,\mathrm{d}x, \tag{1.10}$$

$$= \int_U g\big(F^{-1}(u)\big)\,\mathrm{d}u. \tag{1.11}$$

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

Integral	**Approximation**	**Sampling distribution**	
$\int_X g(x)\,\mathrm{d}\mathbb{F},$	$\sim \frac{1}{M}\sum_{j=1}^{M} g(x_j),$	x_j sampled under F	(1.12a)
$\int_X g(x)f(x)\,\mathrm{d}x,$	$\sim \frac{\mathcal{B}(X)}{M}\sum_{j=1}^{M} g(x_j)f(x_j),$	x_j sampled uniformly on X	(1.12b)
$\int_U g\big(F^{-1}(u)\big)\,\mathrm{d}u,$	$\sim \frac{\mathcal{B}(U)}{M}\sum_{j=1}^{M} g\big(F^{-1}(u_j)\big),$	u_j sampled uniformly on U	(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 qth marginal distribution function,

$$F_q(u) = \Pr\big[x_q \leq u\big]. \tag{1.13}$$

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

$$\int_U g\big(F_1^{-1}(u_1), \ldots, F_Q^{-1}(u_Q)\big)\,\mathrm{d}C \tag{1.14}$$

where $\mathrm{d}u$ 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 $H(\omega)P_t(\omega)/P_T(\omega)$ under the measure $\mathbb{P}$. This means simulating M sample paths $\{w_j\}_{j=1,\ldots,M}$ for S (under $\mathbb{P}$), computing

$$v_j = H(\omega_j)\frac{P_t(\omega_j)}{P_T(\omega_j)}, \quad j = 1, \ldots, M, \tag{1.15}$$

and taking the average of the v_j.

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 $\mathcal{T} = \{t_i\}_{i=0,\dots,N}$, 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 $\tilde{S} = (\tilde{S}_{t_i})_{i=0,\dots,N}$ be a discrete Q-dimensional process, observed at times $t_i \in \mathcal{T}$, approximating S. The Monte Carlo method implicitly determines the process $\tilde{S}$ through its choice of discretization method, and the discrete approximations $\tilde{H}$ and $\tilde{P}$ to H and P.

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

$$\tilde{H} : \mathbb{R}^{Q\times(N+1)} \to \mathbb{R}, \tag{1.16}$$

$$\tilde{P} : \mathbb{R}^{Q\times(N+1)} \to \mathbb{R}, \tag{1.17}$$

to approximate H and P.

The Monte Carlo method generates a set of sample paths,

$$\left\{\hat{S}^j\right\}_{j=1,\dots,M} = \left\{\hat{S}_i^j\right\}_{i=0,\dots,N,\,j=1,\dots,M} \tag{1.18}$$

and approximates (1.4) by

$$\tilde{v}_t = \frac{1}{M}\sum_{j=1}^{M} \tilde{H}\left(\hat{S}^j\right)\frac{\tilde{P}_t\left(\hat{S}^j\right)}{\tilde{P}_T\left(\hat{S}^j\right)}. \tag{1.19}$$

This is a path-by-path approximation. The set $\hat{S}_i = \left(\hat{S}_i^1, \dots, \hat{S}_i^M\right)$ 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 $\tilde{v}_t$ in equation (1.19) has a distribution. The estimate $\tilde{v}_t$ should be unbiased, in that one hopes $\mathbb{E}[\tilde{v}_t] = v_t$, and efficient in the sense that, for any given M, $\text{var}[\tilde{v}_t]$ should be as small as possible. The standard deviation of $\tilde{v}_t$ (or its sample estimate) is called the method's standard error. Setting

$$v^j = \tilde{H}\left(\hat{S}^j\right)\frac{\tilde{P}_t\left(\hat{S}^j\right)}{\tilde{P}_T\left(\hat{S}^j\right)}, \tag{1.20}$$

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

$$se^2(\tilde{v}_t) = \frac{1}{M^2}\sum_{j=1}^{M}(v^j - \tilde{v}_t)^2. \tag{1.21}$$

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, $\tilde{P}_t(\hat{S}) = \mathrm{e}^{rt}$ so that

$$\frac{\tilde{P}_t(\hat{S}^j)}{\tilde{P}_T(\hat{S}^j)} = \mathrm{e}^{-r(T-t)}. \tag{1.22}$$

A Monte Carlo method generates M sample paths, $\{\hat{S}_0^j, \ldots, \hat{S}_N^j\}_{j=1,\ldots,M}$, computes

$$\tilde{H}^j = \tilde{H}_X^c(\hat{S}^j) = (\hat{S}_N^j - X)^+, \tag{1.23}$$

and sets

$$\tilde{v}_t = \mathrm{e}^{-r(T-t)} \frac{1}{M} \sum_{j=1}^{M} \tilde{H}^j. \tag{1.24}$$

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

$$\tau_B(\omega) = \min\{t \geq 0 \mid S_t(\omega) \leq B\} \tag{1.25}$$

with value ∞ if B is never hit. Suppose $S_0 > B$ and let the payoff function be

$$H_B(\omega) = (S_T(\omega) - X)^+ 1_{\{\tau_B(\omega) \leq T\}} \tag{1.26}$$

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

In this case one could set[1]

$$\tilde{H}_B(\hat{S}) = (\hat{S}_N - X)^+ 1_{\{\tilde{\tau}_B(\hat{S}) \leq T\}} \tag{1.27}$$

where

$$\tilde{\tau}_B(\hat{S}) = \min_{i=0,\ldots,N}\{t_i \mid \hat{S}_i \leq B\}, \text{ or } \infty \text{ if } \hat{S}_i > B \text{ for all } i. \tag{1.28}$$

A Monte Carlo method generates $\{\hat{S}_0^j, \ldots, \hat{S}_N^j\}_{j=1,\ldots,M}$, computes

$$\tilde{H}^j = \tilde{H}_B(\hat{S}^j) = \begin{cases} (\hat{S}_N^j - X)^+, & \min\{\hat{S}_0^j, \ldots, \hat{S}_N^j\} \leq B, \\ 0, & \text{otherwise,} \end{cases} \tag{1.29}$$

and sets

$$\tilde{v}_t = \mathrm{e}^{-r(T-t)} \frac{1}{M} \sum_{j=1}^{M} \tilde{H}^j. \tag{1.30}$$

[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'étre* 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,

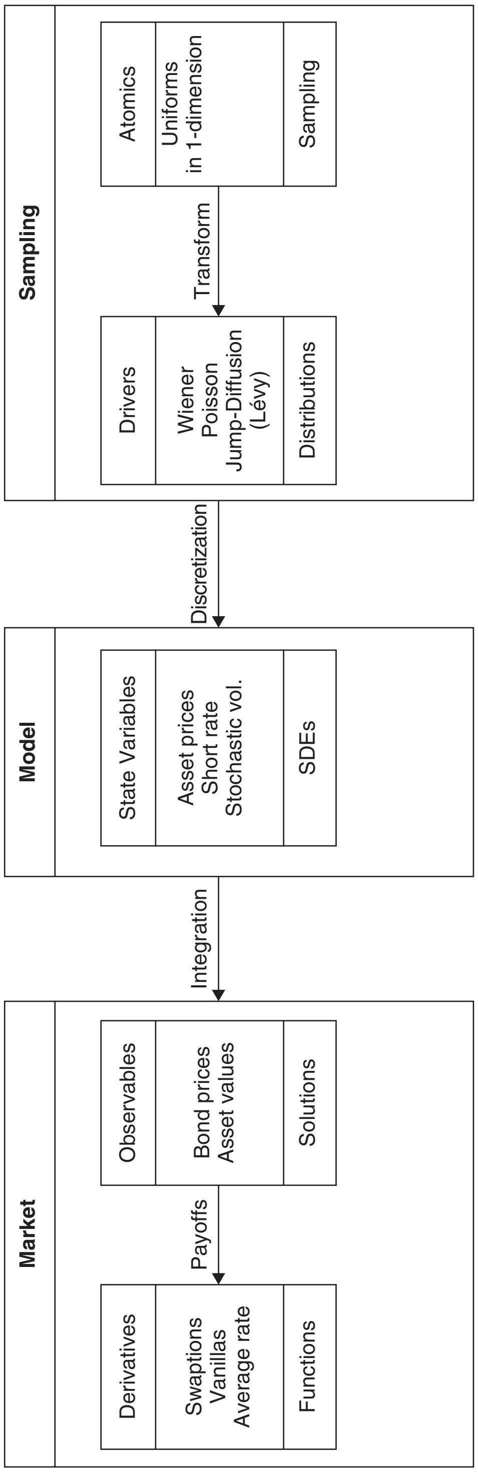

Figure 1.1 The structure of a Monte Carlo valuation scheme

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,[2] 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 Ω. 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 $\tilde{S}^j_{i+1} \mid \tilde{S}^j_i$. We denote this by δ, so that $\delta(\tilde{S}^j_i)$ is a draw from the distribution $\tilde{S}^j_{i+1} \mid \tilde{S}^j_i$. It is the method used to discretize S that determines δ, and hence determines the process $\tilde{S}$.

Note that we do not assume that $\tilde{S}^j_{i+1} \mid \tilde{S}^j_i = S_{t_{i+1}} \mid \tilde{S}^j_i$ 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 $\mathrm{d}S_t = \mu(S_t)\,\mathrm{d}t + \sigma(S_t)\,\mathrm{d}z_t$ and a time step $\Delta t_i = t_{i+1} - t_i$ it sets

$$\hat{S}^j_{i+1} = \delta\big(\hat{S}^j_i\big) = \hat{S}^j_i + \mu\big(\hat{S}^j_i\big)\Delta t_i + \sigma\big(\hat{S}^j_i\big)\sqrt{\Delta t_i}\,\varepsilon_t \tag{1.31}$$

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

From a computational perspective in computing a set $\{\hat{S}^j_i\}$ 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 $\mathcal{S} = \mathbb{R}^Q$. Let $\mathcal{T} = \{t_i\}_{i=0,\ldots,N}$, $0 = t_0 < \cdots < t_N = T_{\max}$, be a set of discretization times and set $X_i \equiv X_{t_i}$ 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 $\hat{X}^j_i \in \mathbb{R}^Q$, $j = 1, \ldots, M$, $i = 1, \ldots, N$, for the value at time t_i of the discrete process along the jth sample path, and $\hat{X}^j_{i,q}$ for the value of its qth coordinate, then

$$\Xi = \big\{\hat{X}^j_{i,q}\big\}^{j=1,\ldots,M}_{i=0,\ldots,N;q=1,\ldots,Q} \subseteq \mathbb{R}^{M\times(N+1)\times Q} \tag{1.32}$$

is the entire set of reals generated in the simulation.

Ξ 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 Ξ is computed. A scheme determines how the set Ξ is sliced when constructing the Monte Carlo method: what is stored, what is evolved, what is returned.

For a (random) evolution operator $\delta : \mathbb{R}^Q \to \mathbb{R}^Q$, $\delta(\hat{X}^j_i) = \hat{X}^j_{i+1}$ on $\hat{X}$, set

$$\delta'_i : \mathbb{R}^{i\times Q} \to \mathbb{R}^{(i+1)\times Q},\ \big(\hat{X}_0, \ldots, \hat{X}_{i-1}\big) \mapsto \big(\hat{X}_0, \ldots, \hat{X}_{i-1}, \delta\big(\hat{X}_{i-1}\big)\big), \tag{1.33}$$

and define

$$\delta_{(n)} = \delta \circ \cdots \circ \delta, \tag{1.34}$$

to be the n-fold composition of δ, with $\delta_{(0)} = 1$,

$$\delta_N = \delta'_N \circ \cdots \circ \delta'_1 : \mathbb{R}^Q \rightarrow \mathbb{R}^{(N+1)\times Q}, \tag{1.35}$$

and

$$\delta^M = (\delta, \ldots, \delta) : \mathbb{R}^{M\times Q} \rightarrow \mathbb{R}^{M\times Q}, \tag{1.36}$$

to be the extension of δ to $\mathbb{R}^{M\times Q}$.

We have $\delta_{(n)}(\hat{X}_i^j) = \hat{X}_{i+n}^j$. δ_N generates a sample path of $\hat{X}$, and δ^M moves a slice forwards by one time step.

There are broadly four ways of constructing Ξ. These are to construct and return a sequence:

1. *Element-wise.* A single value at a time, $\hat{X}_0^1, \hat{X}_1^1, \ldots, \hat{X}_N^1, \hat{X}_0^2, \ldots, \hat{X}_N^M$,

$$\begin{array}{ccccccc} \hat{X}_0^1 & \longrightarrow & \hat{X}_1^1 & \longrightarrow & \cdots & \longrightarrow & \hat{X}_N^1, \\ \hat{X}_0^2 & \longrightarrow & \hat{X}_1^2 & \longrightarrow & \cdots & \longrightarrow & \hat{X}_N^2, \\ \vdots & & \vdots & & & & \vdots \\ \hat{X}_0^M & \longrightarrow & \hat{X}_1^M & \longrightarrow & \cdots & \longrightarrow & \hat{X}_N^M. \end{array} \tag{1.37}$$

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 jth application of δ_N to $\hat{X}_0$,

$$\begin{array}{ccc} \hat{X}_0^1 & \longrightarrow & \left(\hat{X}_0^1, \hat{X}_1^1, \ldots, \hat{X}_N^1\right), \\ \hat{X}_0^2 & \longrightarrow & \left(\hat{X}_0^2, \hat{X}_1^2, \ldots, \hat{X}_N^2\right), \\ \vdots & & \vdots \\ \hat{X}_0^M & \longrightarrow & \left(\hat{X}_0^M, \hat{X}_1^M, \ldots, \hat{X}_N^M\right). \end{array} \tag{1.38}$$

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

$$\begin{pmatrix} \hat{X}_0^1 \\ \hat{X}_0^2 \\ \vdots \\ \hat{X}_0^M \end{pmatrix} \longrightarrow \begin{pmatrix} \hat{X}_1^1 \\ \hat{X}_1^2 \\ \vdots \\ \hat{X}_1^M \end{pmatrix} \longrightarrow \cdots \longrightarrow \begin{pmatrix} \hat{X}_N^1 \\ \hat{X}_N^2 \\ \vdots \\ \hat{X}_N^M \end{pmatrix}. \tag{1.39}$$

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

$$\hat{X}_0 \longrightarrow \begin{pmatrix} \hat{X}_0^1, & \hat{X}_1^1, & \ldots & \hat{X}_N^1 \\ \hat{X}_0^2, & \hat{X}_2^2, & \ldots & \hat{X}_N^2 \\ \vdots & \vdots & & \vdots \\ \hat{X}_0^M, & \hat{X}_1^M, & \ldots & \hat{X}_N^M \end{pmatrix}. \tag{1.40}$$

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 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 $M = 50\,000$, $N = 100$ and $Q = 3$ this is 15×10^6 `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, σ, 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.[4]

(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 σ and riskless rate r. Let $\mathrm{Pv}(x) = \mathrm{e}^{-r(T-t)}x$ and $\nu = (2r/\sigma^2) - 1$. For $H < X$ the value DOC_t of the option at time $t < T$ is given by

$$\mathrm{DOC}_t = S_t N(d_1(X)) - \mathrm{Pv}(X)N(d_2(X)) - \left(\frac{H}{S_t}\right)^{\nu+2} S_t N\left(d_1\left(\frac{XS_t^2}{H^2}\right)\right) + \left(\frac{H}{S_t}\right)^{\nu} \mathrm{Pv}(X) N\left(d_2\left(\frac{XS_t^2}{H^2}\right)\right), \tag{1.41}$$

and for $H \geq X$ by

$$\begin{aligned}\mathrm{DOC}_t = {} & S_t N(d_1(H)) - \mathrm{Pv}(X)N(d_2(H)) \\ & -\left(\frac{H}{S_t}\right)^{\nu+2} S_t N\left(d_1\left(\frac{S_t^2}{H}\right)\right) \\ & +\left(\frac{H}{S_t}\right)^{\nu} \mathrm{Pv}(X) N\left(d_2\left(\frac{S_t^2}{H}\right)\right),\end{aligned} \tag{1.42}$$

where N is the standard normal distribution function and

$$d_1(x) = \frac{1}{\sigma\sqrt{T-t}} \ln\left(\frac{S_t}{\mathrm{Pv}(x)}\right) + \frac{1}{2}\sigma\sqrt{T-t}, \tag{1.43}$$

$$d_2(x) = d_1(x) - \sigma\sqrt{T-t}, \tag{1.44}$$

(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, σ, 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_∞ 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

$$r_t(T) = r_\infty + (r_t - r_\infty)\frac{1-\mathrm{e}^{-\alpha\tau}}{\alpha\tau} + \frac{\sigma^2\tau}{4\alpha}\left(\frac{1-\mathrm{e}^{-\alpha\tau}}{\alpha\tau}\right)^2 \tag{1.45}$$

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

[4] The spreadsheet LibraryProcedures.xls contains a `Function normal_cdf()` that computes the standard normal distribution function much more efficiently. See Appendix C.

You have an application that for some reason needs to compute equation (1.45) very frequently.[5] Write a `Function`, taking r_t, α, σ and μ 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 σ, where the average a_t is computed geometrically,

$$a_t = \exp\left(\frac{1}{T-t}\int_t^T \ln(S_u)\,\mathrm{d}u\right), \tag{1.46}$$

has value A_t,

$$A_t = \mathrm{e}^{-\delta(T-t)} S_t \mathrm{N}(d_1) - \mathrm{e}^{-r(T-t)} X \mathrm{N}(d_2), \tag{1.47}$$

where

$$\delta = \frac{1}{2}\left(r + \frac{1}{2}\hat{\sigma}^2\right), \tag{1.48}$$

$$\hat{\sigma} = \frac{1}{\sqrt{3}}\sigma, \tag{1.49}$$

$$d_1 = \frac{1}{\hat{\sigma}\sqrt{T-t}} \ln\left(\frac{\mathrm{e}^{-\delta(T-t)} S_t}{\mathrm{e}^{-r(T-t)} X}\right) + \frac{1}{2}\hat{\sigma}\sqrt{T-t}, \tag{1.50}$$

$$d_2 = d_1 - \hat{\sigma}\sqrt{T-t}. \tag{1.51}$$

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

```
GetChar(X As Long, Y As Long, valids As String) As String ',        (1.52)
```

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 valids`. Test it for the case when acceptable characters are either "p" or "c" so that `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.

2 Levels of Programming Sophistication

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, `main()`, that fires the application proper. It comes equipped with an error channel. In our case clicking a button on an Excel spreadsheet causes `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 *polymorphism* and *encapsulation* are discussed later in the book.

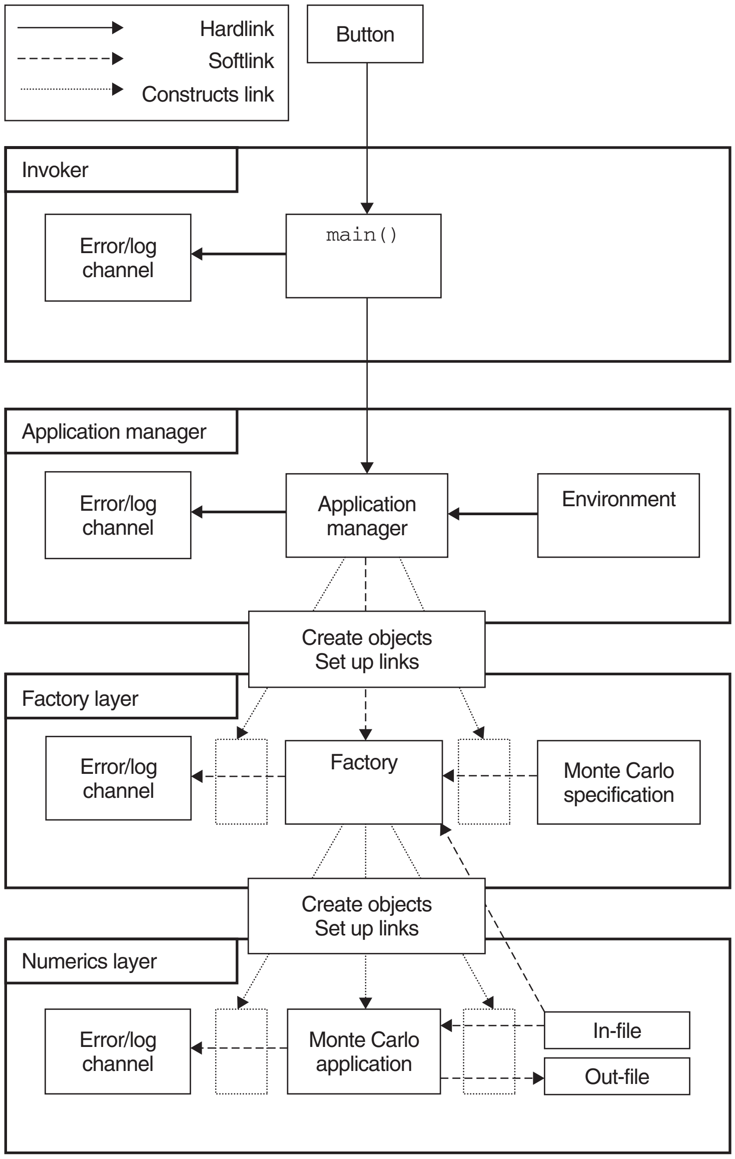

Figure 2.1 Outline of a Platonic application

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.