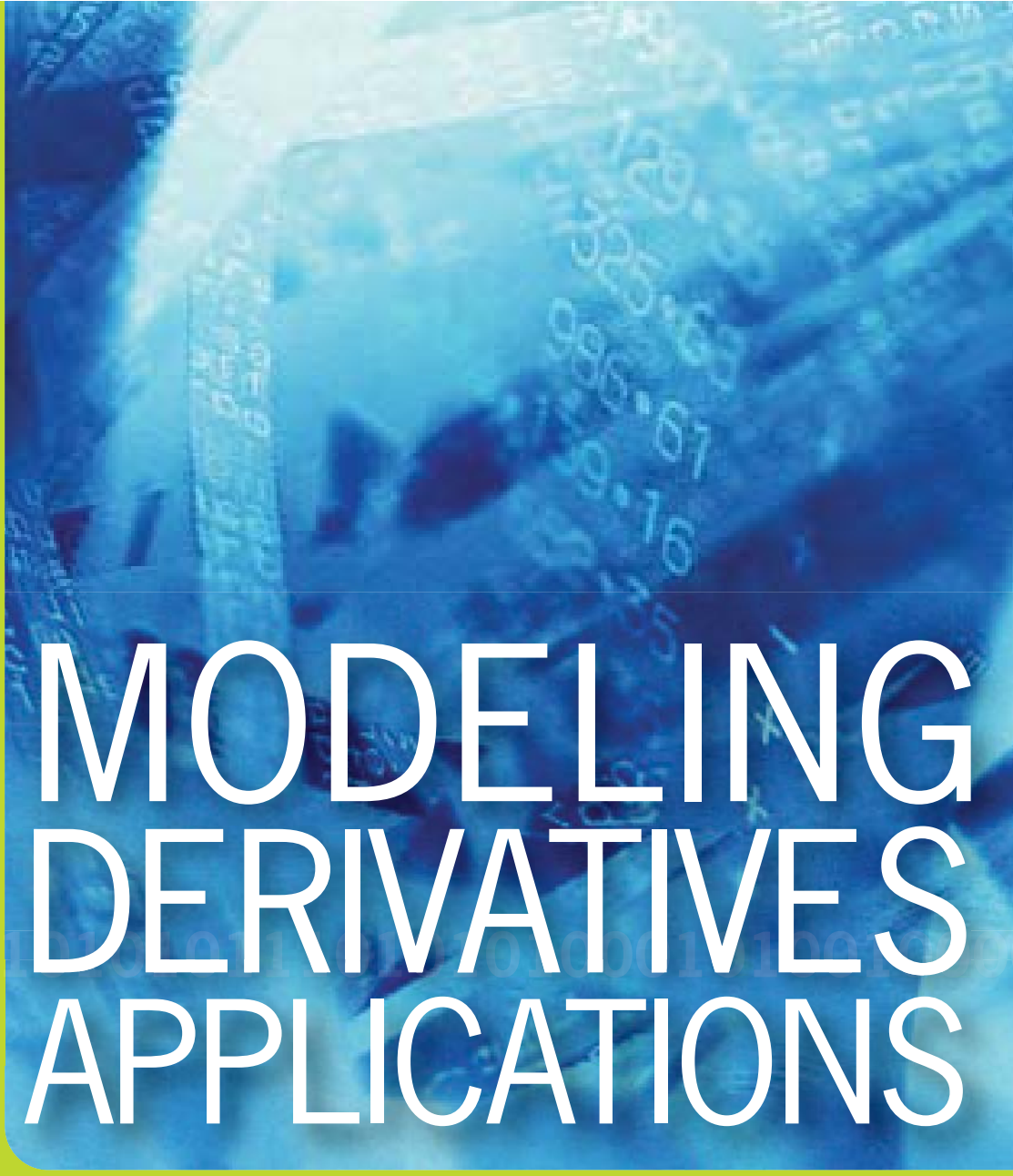


MODELING DERIVATIVES APPLICATIONS

IN MATLAB, C++, AND EXCEL

JUSTIN LONDON

DRAFT MANUSCRIPT

Books Available

November 2006

Modeling Derivatives Applications in Matlab, C++, and Excel

In an increasingly competitive world, it is quality of thinking that gives an edge—an idea that opens new doors, a technique that solves a problem, or an insight that simply helps make sense of it all.

We work with leading authors in the various arenas of business and finance to bring cutting-edge thinking and best-learning practices to a global market.

It is our goal to create world-class print publications and electronic products that give readers knowledge and understanding that can then be applied, whether studying or at work.

To find out more about our business products, you can visit us at www.ftpress.com.

Modeling Derivatives Applications in Matlab, C++, and Excel

Justin London

FT Press
FINANCIAL TIMES

Vice President, Editor-in-Chief: Tim Moore
Executive Editor: Jim Boyd
Editorial Assistant: Susie Abraham
Development Editor: Russ Hall
Associate Editor-in-Chief and Director of Marketing: Amy Neidlinger
Cover Designer: Chuti Prasertsith
Managing Editor: Gina Kanouse
Senior Project Editor: Lori Lyons
Copy Editor: Water Crest Publishing
Indexer: Christine Karpeles
Compositor: Lori Hughes
Manufacturing Buyer: Dan Uhrig

Publishing as Wharton School Publishing
Upper Saddle River, New Jersey 07458

FT Press offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales. For more information, please contact U.S. Corporate and Government Sales, 1-800- 382-3419, corpsalespearsontechgroup.com. For sales outside the U.S., please contact International Sales at internationalpearsoned.com.

Printed in the United States of America

First Printing October, 2006

ISBN: 0-13-196259-0

Pearson Education LTD.
Pearson Education Australia PTY, Limited.
Pearson Education Singapore, Pte. Ltd.
Pearson Education North Asia, Ltd.
Pearson Education Canada, Ltd.
Pearson Educatiòn de Mexico, S.A. de C.V.
Pearson Education—Japan
Pearson Education Malaysia, Pte. Ltd.

Library of Congress Cataloging-in-Publication Data

London, Justin, 1968-
Modeling Derivatives Applications in Matlab, C++, and Excel / Justin London.
p. cm.
ISBN 0-13-196259-0 (hardback : alk. paper)
1. Modeling Credit Derivatives. 2. Pricing Models in Matlab, C++, and Excel. I. Title.
HF5548.4.L692E45 2006
005.5'7–dc22

2006023492

To the memory of my grandparents, Milton and Evelyn London;
my parents, Leon and Leslie; and my sister, Joanna.

CONTENTS

PREFACE

Given the explosive growth in new financial derivatives such as credit derivatives, hundreds of financial institutions now market these complex instruments and employ thousands of financial and technical professionals needed to model them accurately and effectively. Moreover, the implementation of these models in C++ and Matlab (two widely used languages for implementing and building derivatives models) has made programming skills in these languages important for practitioners to have. In addition, the use of Excel is also important as many trading desks use Excel as a front-end trading application.

Modeling Derivatives Applications in Matlab, C++, and Excel is the first book to cover in detail important derivatives pricing models for credit derivatives (for example, credit default swaps and credit-linked notes), collateralized-debt obligations (CDOs), mortgage-backed securities (MBSs), asset-backed securities (ABSs), swaps, fixed income securities, and increasingly important weather, power, and energy derivatives using Matlab, C++, and Excel. Readers will benefit from both the mathematical derivations of the models, the theory underlying the models, as well as the code implementations.

Throughout this book, numerous examples are given using Matlab, C++, and Excel. Examples using actual real-time Bloomberg data show how these models work in practice. The purpose of the book is to teach readers how to properly develop and implement derivatives applications so that they can adapt the code for their own use as they develop their own applications. The best way to learn is to follow the examples and run the code. The chapters cover the following topics:

- Chapter 1: Swaps and fixed income securities
- Chapter 2: Monte Carlo and numerical techniques
- Chapter 3: Copulas and copula methodologies
- Chapter 4: Mortgage-backed securities
- Chapter 5: Collateralized-debt obligations
- Chapter 6: Credit derivatives
- Chapter 7: Single-factor interest rate models
- Chapter 8: The Heath-Jarrow-Morton (HJM) interest rate model
- Chapter 9: Weather derivatives
- Chapter 10: Energy and power derivatives

- Chapter 11: Also covers energy derivatives model implementation using MATLAB, but is written and based on the proprietary work of its author, Craig Pirrong, professor of finance and director of the Global Energy Management Institute at the University of Houston.

- Chapter 12: Commercial real-estate based securities (a type of asset-backed security), which is written and is based on the proprietary work of its author, Tien-Foo Sing, professor in the Department of Real Estate Finance at the National University of Singapore.

In order to provide different perspectives to readers and provide as much useful information as possible, the work and models developed and written by various leading practitioners and experts for certain topics are provided and incorporated throughout the book. Thus, not only does this book cover complex derivatives models and provide all of the code (which can be downloaded using a secure ID code from the companion Web site at *www.URL?????????????????.com*), but it also incorporates important work contributions from leading practitioners in the industry. For instance, the work of Galiani (2003) is discussed in the chapter on copulas and credit derivatives. The work of Picone (2004) is discussed in the chapter on collateralized-debt obligations. The work of Johnson (2004) is discussed in the chapters on fixed-income instruments and mortgage-backed securities. The valuable work for energy derivatives of Doerr (2002), Xiang (2004), and Xu (2004) is given. In Chapter 11, Craig Pirrong discusses the Pirrong-Jermayakan model, a two-dimensional alternating implicit difference (ADI) finite difference scheme for pricing energy derivatives. In Chapter 12, Tien-Foo Sing discusses using Monte Carlo to price asset-based securities. Moreover, numerous individuals named in the acknowledgments supplied useful code throughout the book.

The book emphasizes how to implement and code complex models for pricing, trading, and hedging using C++, Matlab, and Excel. The book does not focus on design patters or best coding practices (these issues may be discussed in subsequent editions of the book.) However, the book does provide some discussions and helpful tips for building efficient models. For instance, memory allocation for data structures is always an issue when developing a model that requires use and storage of multi-dimensional data. Use of a predefined two-dimensional array, for instance, is in not the most efficient way to allocate memory since it is fixed in size. A lot of memory may be unutilized and wasted if you do not know how large the structure needs to be to store the actual data. On the other hand, the predefined array sizes may turn out not to be large enough.

Although two-dimensional arrays are easy to define, use of array template classes (that can handle multiple dimensions) and vectors (of vectors) in the Standard Template Library in C++ are more efficient because they are dynamic and only use as much memory as is needed. Such structures are used in the book, although some two-dimensional arrays are used as well. Matlab, a matrix manipulation language, provides automatic memory allocation of memory as data is used if no array sizes are predefined. All data in Matlab are treated as matrix objects; e.g., a single number is treated as a 1 x 1 array. Data can be added or removed from an object and the object will dynamically expand or reduce the amount of memory space as needed.

Hopefully, this book will give readers the foundation to develop, build, and test their own models while saving readers a great deal of development time through use of pre-tested robust code.

ACKNOWLEDGMENTS

Special thanks to the following people for their code and work contributions to this book:
Ahsan Amin
Jim Carson
Uwe Doerr
Stefano Galiani
Chetan Jain
Stafford Johnson
Jochen Meyer
Dominic Picone
Craig Pirrong
Tien Foo Sing
Jan Vecer
Liuren Wu
Lei Xiong
James Xu

About the Author

Justin London has developed fixed-income and equity models for trading companies and his own quantitative consulting firm. He has analyzed and managed bank corporate loan portfolios using credit derivatives in the Asset Portfolio Group of a large bank in Chicago, Illinois, as well as advised several banks in their implementation of derivative trading systems. London is the founder of a global online trading and financial technology company. A graduate of the University of Michigan, London holds a BA in economics and mathematics, an MA in applied economics, and an MS in financial engineering, computer science, and mathematics, respectively.

CHAPTER 1

SWAPS AND FIXED INCOME INSTRUMENTS

SECTIONS

Swaps are often used to hedge interest rate risk exposure to balance sheets as well as for bond and loan portfolios. By matching the durations of balance sheet fixed income assets and liabilities (e.g., bond or loan instruments), swaps can immunize the balance sheet from interest rate risk. Ideally, the hedge should match both the duration and timing of cash flows of the fixed income portfolio with that of the swap as closely as possible. For instance, if a bank has a portfolio of floating-rate loans that have a duration of five years, the bank can enter a swap with a duration of five years (the duration of the swap is the difference between the fixed- and floating-leg durations of the swap) to receive the fixed rate and to pay floating (effectively changing the characterization of the loan from floating to fixed and thereby locking in a fixed rate of return.) Although in general, basis risk exists because there is not an exact match between the cash flows—e.g., interest rate payments on the loans and those of the swap do not match—the bank has reduced its exposure to shifts in the yield curve. Moreover, institutional money managers can also use a combination

of Chicago Board of Trade (CBOT) Treasury note futures and swap futures to structure hedges to protect a portfolio of corporate and Treasury securities from a rise in interest rates. Due to the liquidity and standardization of swap futures, swap futures are becoming a cheaper and more efficient way to hedge a fixed income portfolio than entering a swap.[1] This chapter discusses the details of hedging interest rate risk and bond portfolios using swaps and fixed income instruments (e.g., futures).

In §1.1, we discuss using Eurodollar futures to compute LIBOR swap rates. In §1.2, we discuss Treasury bills and Treasury bonds, including how they are quoted and priced. In §1.3, we discuss bootstrapping the yield curve to compute discount swap rates. In §1.4, we discuss hedging debt positions and interest rate movements using fixed income instruments. In §1.5, we discuss bond duration, modified duration, and DV01 (dollar value of a one-basis-point move) calculations, which are necessary for computing swap durations and modified durations because a fixed-for-floating interest rate swap is composed of two legs that are equivalent to a fixed-rate bond and a floating-rate bond. In §1.6, we discuss term structure of rates, and in §1.7, we discuss how to numerically bootstrap the yield curve. In §1.8, we discuss bootstrapping in Matlab and provide examples, while in §1.9, we discuss bootstrapping using Excel. In §1.10, we discuss general swap pricing in Matlab using the Black-Derman-Toy (BDT) and Heath-Jarrow-Morton (HJM) interest rate models, while in §1.11, we discuss swap pricing using term structures like the forward curve built from zero-coupon and coupon-bearing bond cash flows. In §1.12, we implement and price fixed-for-floating swaps, including calculations for duration and risk measures, using C++. Finally, in §1.13, a Bermudan swaption pricing implementation in Matlab is provided.

1.1 EURODOLLAR FUTURES

Eurodollar futures[2] contracts maturing in March, June, September, and December are sometimes used to calculate the LIBOR swap zero rates for swap maturities greater than one year. The Eurodollar futures interest rate can be used to compute forward rates for long-dated maturities. In the United States, spot LIBOR rates are usually used to define the LIBOR zero curve for maturities up to one year. Eurodollar futures are typically then used for maturities between one and two years and sometimes for maturities up to five, seven, and ten years. In addition, swap rates, which define par yield bonds, are used to calculate the zero curve for maturities longer than a year. Using a combination of spot LIBOR rates, Eurodollar futures, and swap rates, the LIBOR/swap zero curve can be generated using a bootstrap method procedure.

Typically, a convexity adjustment is made to convert Eurodollar futures rates into forward interest rates. For short maturities (up to one year), the Eurodollar futures interest rate can be assumed to be the same as the corresponding forward interest rate. But for longer maturities, the difference between futures and forward contracts becomes important when interest rates vary unpredictably.[3]

Suppose the quoted Eurodollar futures price is P; then the cash contract price is

$$10000\,(100 - 0.25(100 - P)) \tag{1.1}$$

which is equivalent to 10,000 times the cash futures price of $100 - 0.25(100 - P)$. Suppose $P = 96.7$; then the contract price is

$$10000\,(100 - 0.25(100 - 96.7)) = \$991,750$$

A Eurodollar contract is like a Treasury bill contract, but with some important differences. Both the Eurodollar and T-bill contracts have an underlying face value amount of $1 million. However, for a T-bill, the contract price converges at maturity to the price of a 91-day $1 million face-value Treasury bill, and if the contract held to maturity, this is the instrument delivered.[4] A Eurodollar futures contract is cash settled on the second London business day before the third Wednesday of the month.[5] The final marking to market sets the contract price equal to

$$f_0 = (\$1,000,000)\left(\frac{100 - 0.25R}{100}\right) = 10,000(100 - 0.25R)$$

where R is the quoted LIBOR Eurodollar rate at that time.[6] This quoted Eurodollar rate is the actual 90-day rate on Eurodollar deposits with quarterly compounding.[7] It is not a discount rate. As Hull states, "the Eurodollar futures contract is therefore a futures contract on an interest rate, whereas the Treasury bill futures contract is a futures contract on the price of a Treasury bill or a discount rate."[8]

1.2 Treasury Bills and Bonds

Treasury bills are short-term discount securities issued by the U.S. Treasury. At the time of sale, a percentage discount is applied to the face value. Treasury bill prices are quotes as a discount rate on a face value of $100. At maturity, the holder redeems the bill for full face value. The basis (or day counting convention) for the interest accrual is actual/360 so that interest accrues on the actual number of elapsed days between purchase and maturity, assuming each year has 360 days. The Treasury price quote is the annualized dollar return provided by the Treasury bill in 360 days expressed as a percentage of the face value

$$\frac{360}{n}(100 - P) \tag{1.2}$$

where P is the cash price of a Treasury bill that has a face value of $100 and n days to maturity.[9] The discount rate is not the rate of return earned on the Treasury bill. If the cash price of a 90-day Treasury bill is 99, the quoted price would be 1.00. The *rate of return* would be 1/99, or 1.01%, per 90 days. This is translates to

$$\frac{1}{99} \times \frac{360}{90} = 0.0404$$

or 4.04% per annum on an actual/360 basis. Alternatively, it is

$$\frac{1}{99} \times \frac{365}{90} = 0.04096$$

or approximately 4.10% per annum on an actual/365 basis. Both of these rates are expressed with a quarterly compounding period of 90 days. In order to directly compare Treasury yields with yields quoted on Treasury bonds, often semiannual compounding (a

compounding period of 180 days) is used on an actual/365 basis. The computed rate is known as the *bond equivalent yield*. In this case, the bond equivalent yield is

$$\frac{1}{99} \times \frac{365}{180} = 0.02048.$$

For short-term Treasury bills (fewer than 182 days to maturity), the *money-market yield* can be computed as 360/365 of the bond equivalent yield. In this example, it is 2.02%.

T-bill futures call for the delivery or purchase of a T-bill with a maturity of 91 days and a face value of $1,000,000. They are used for speculating and hedging short-term rates. Prices of T-bill futures are quoted in terms of the interbank money market (IMM) index or discount yield R_d :

$$\text{IMM} = 100 - R_d$$

Theoretical T-bill pricing is done with a carrying cost model

$$f_0 = S_0^M (1 + R_f)^T \tag{1.3}$$

where

$f_0 =$ price of T-bill futures.
$T =$ time to expiration on futures.
$S_0^M =$ price on spot T-bill with maturity of $M = 91 + T$.
$R_f =$ risk-free or repo rate.

Following Johnson (2004),[10] suppose the rate on a 161-day spot T-bill is 5.7% and the repo rate (or risk-free rate) for 70 days is 6.38%; then the price on a T-bill futures contract with an expiration of 70 days would be

$$f_0 = (97.5844)(1.0638)^{70/365} = 98.7487$$

where

$$S_0^{161} = \frac{100}{(1.057)^{161/365}} = 97.5844.$$

The future price is governed by arbitrage considerations. If the futures market price is above f^*, arbitrageurs would short the futures contract and go long the spot T-bill. For example, suppose the futures market price is $f^{M=70/365} = 99$. An arbitrageur would go short in the futures, agreeing to sell a 91-day T-bill at 99, 70 days later, and would go long the spot, borrowing 97.5844 at 6.38% for 70 days to finance the purchase of the 161-day T-bill that is trading at 97.5844. Seventy days later at expiration, the arbitrageur would sell the T-bill (which would now have a maturity of 91 days) on the futures for $f^{M=70/365} = 99$ and pay off his financing debt of $f^* = 98.74875$, realizing a cash flow (CF_T) of $2,513:

$$\begin{aligned} CF_T &= f_0^M - f_0^* \\ &= f_0^M - S_0^M (1 + R_f)^T \\ &= 99 - 97.5844(1.0638)^{70/365} \\ &= 99 - 98.7487 = 0.2513 \end{aligned}$$

so that the cash flow or profit is

$$CF_T = (\$1,000,000)\left(\frac{0.2513}{100}\right) = \$2,513.$$

Note that if $f^M = 99$, a money market manager planning to invest for 70 days in T-bills at 6.38% could earn a greater return by buying a 161-day bill and going short the 70-day T-bill futures to lock in the selling price. For example, using the preceding numbers, if a money manager was planning to invest 97.5844 for 70 days, she could buy a 161-day bill for that amount and go short in the futures at 99. Her return would be 7.8%, compared to only 6.38% from the 70-day T-bill:

$$R = \left(\frac{99}{97.5844}\right)^{365/70} - 1 = 0.078$$

If the market price is below f^*, then arbitrageurs would go long in the futures and short in the spot. Suppose $f^M = 98$. An arbitrageur would go long in the futures, agreeing to buy a 91-day T-bill for 98 seventy days later and would go short in the spot, borrowing the 161-day T-Bill, selling it for 97.5844 and investing the proceeds at 6.38% for 70 days. Seventy days later (expiration), the arbitrageur would buy the bill (which now would have a maturity of 91 days) on the futures for 98 (fm), use the bill to close his short position, and collect 98.74875 (f^*) from his investment, realizing a cash flow of \$7487.

$$\begin{aligned} CF_T &= f_0^* - f_0^M \\ &= S_0^M(1+R_f)^T - f_0^M \\ &= 97.5844(1.0638)^{70/365} - 98 \\ &= 98.7487 - 98 = 0.2513 = 0.7487 \end{aligned}$$

so that the cash flow or profit is

$$CF_T = (\$1,000,000)\left(\frac{0.7487}{100}\right) = \$7,487$$

If the carrying-cost model holds, then the spot rate on a 70-day bill (repo rate) will be equal to the synthetic rate (implied repo rate) formed by buying the 161-day bill and going short in the 70-day futures:

Buy 161-day T-bill $S_0^{161} = 97.5844$.

Short position in T-bill futures at $f_0^M = f_0^* = 98.74875$.

$$R = \left(\frac{98.74875}{97.5844}\right)^{365/70} - 1 = 0.0638.$$

Furthermore, if the carrying-model holds, then the yield-to-maturity, *YTM*, of the futures will be equal to the implied forward rate F. Locking in the 91-day investment to be made 70 days from now, as follows:

1. Short 70-day T-bill at $S_0^{70} = 98.821$.
2. Buy $n = \frac{S_0^{70}}{S_0^{161}} = \frac{98.821}{97.5844} = 1.01267$ of 161-day bill at 97.5844.
3. End of 70 days, cover short bill for 100.
4. 90 days later, collect on the investment in the original 161-day bill: $1.01267(100) = 101.267$.

$$R = \left(\frac{101.267}{100}\right)^{365/91} - 1 = 0.0518 = F_{91,70}$$

which is equal to

$$YTM_f = \left(\frac{100}{98.74875}\right)^{365/91} - 1 = 0.0518.$$

Hedging with T-bill Futures

T-bill futures are often used for hedging by money managers. Suppose a money manager is expecting a \$5 million cash flow in June, which she plans to invest in a 91-day T-bill. With June T-bill futures trading at IMM of 91 (*June IMM* $= 91$ or $R_D = 9\%$), the manager could lock in a 9.56% rate by going long 5.115 June T-bill contracts:

$$f_0^{June} = (\$1,000,000)\left(\frac{100-(9)(0.25)}{100}\right) = \$977,500$$

$$YTM_f = \left(\frac{\$1,000,000}{\$977,500}\right)^{365/91} - 1 = 0.0956$$

$$n_f = \frac{CF_T}{f_0} = \frac{\$5,000,000}{\$977,500} = 5.115 \text{ long contracts}$$

Suppose in June, the spot 91-day T-bill rate is at 8%. The manager would find T-bill prices higher at \$980,995, but would realize a profit of \$17,877 from closing the futures position. Combining the profit with the \$5 million cash flow, the manager would be able to buy 5.115 T-bills[11] and earn a rate off the \$5 million investment of 9.56%:

At June contract maturity, rate on T-bill $= 8\%$.

$$\text{spot rate} = S_T^{91} = \frac{\$1,000,000}{(1.08)^{91/365}} = \$980,995.$$

$$\text{profit} = \pi_f = [\$980,995 - \$997,500](5.115) = \$17,877.$$

$$\text{hedge ratio} = n_{TB} = \frac{CF + \pi_f}{S_T^{91}} = \frac{(\$5,000,000 + \$17,877)}{\$980,995} = 5.115.$$

$$\text{contract price} = f_0^{June} = (\$1,000,000)\left(\frac{100-(9)(0.25)}{100}\right) = \$977,500.$$

$$\text{rate of return} = R = \left[\frac{(\$1,000,000)(5.115)}{\$5,000,000}\right]^{365/91} - 1 = 0.956 = 9.56\%.$$

Suppose instead that in June, the spot 91-day T-bill rate is at 10%. The manager would find T-bill prices lower at \$976,518, but would realize a loss of \$5,025 from closing the futures position. After paying the clearing house \$5,025, the manager would still be able to 5.115 T-bills given the lower T-bill prices, earning a rate of return from the \$5 million investment at 9.56%:

At June contract maturity, rate on T-bill $= 10\%$.

$$S_T^{91} = \frac{\$1,000,000}{(1.10)^{91/365}} = \$976,518.$$

$$\pi_f = [\$976,518 - \$997,500](5.115) = -\$5,025.$$

$$n_{TB} = \frac{CF + \pi_f}{S_T^{91}} = \frac{(\$5,000,000 - \$5,025)}{\$976,518} = 5.115.$$

$$f_0^{June} = (\$1,000,000)\left(\frac{100 - (9)(0.25)}{100}\right) = \$977,500.$$

$$R = \left[\frac{(\$1,000,000)(5.115)}{\$5,000,000}\right]^{365/91} - 1 = 0.956 = 9.56\%.$$

Note that at any rate, the money market manager earns a rate of return of 9.56%.

Long Futures Hedge: Hedging Synthetic Futures on 182-Day T-Bill

Suppose a money market manager is expecting a \$5 million cash flow in June, which she plans to invest in a 182-day T-bill. Because the T-bill underlying a futures contract has a maturity of 91 days, the manager would need to go long in both June T-bill futures and a September T-bill futures (note that there are approximately 91 days between the contract) in order to lock in a return on a 182-day T-bill instrument. If June T-bill futures were trading at IMM of 91 and September futures were trading at IMM of 91.4, then the manager could lock in a 9.3% rate on an instrument in 182-day T-bills by going long in 5.115 June T-bill futures and 5.11 September contracts:

June IMM $= 91$ or $R_D = 9\%$

Sept IMM $= 91.4$ or $R_D = 8.6\%$

$$f_0^{June} = (\$1,000,000)\left(\frac{100 - (9)(0.25)}{100}\right) = \$977,500$$

$$f_0^{Sept} = (\$1,000,000)\left(\frac{100 - (8.6)(0.25)}{100}\right) = \$978,500$$

$$YTM_f^{June} = \left[\frac{\$1,000,000}{\$977,500}\right]^{365/91} - 1 = 0.0956$$

$$YTM_f^{Sept} = \left[\frac{\$1,000,000}{\$978,500}\right]^{365/91} - 1 = 0.091$$

$$n_f^{June} = \frac{CF_T}{f_0} = \frac{\$5,000,000}{\$977,500} = 5.115 \text{ long contracts}$$

$$n_f^{Sept} = \frac{CF_T}{f_0} = \frac{\$5,000,000}{\$978,500} = 5.112 \text{ long contracts}$$

so that the return is

$$YTM_f^{182} = \left[(1.0956)^{91/35}(1.091)^{91/365}\right]^{365/182} - 1 = 0.093.$$

Suppose in June, the 91-day T-bill rate is at 8% and the spot 182-day T-bill rate is at 8.25%. At these rates, the price on the 91-day spot T-bill would be

$$S_T^{91} = \frac{\$1,000,000}{(1.08)^{91/365}} = \$980,995$$

and the price on the 182-day spot T-bill would be

$$S_T^{182} = \frac{\$1,000,000}{(1.08)^{182/365}} = \$961,245.$$

If the carrying-cost model holds, then the price on the September futures at the June date is

$$f_0^{Sept} = S_0^{182}(1 + R_f)^T = \$961,245(1.08)^{91/365} = \$979,865.$$

At these prices, the manager would be able to earn futures profits of

$$\text{June } \pi_f = [\$980,995 - \$977,500]5.115 = \$17,877$$
$$\text{Sept } \pi = [\$979,865 - \$978,500]5.11 = \$6,975$$

for a total profit of $24,852 from closing both futures contract (which offsets the higher T-bill futures prices) and would be able to buy

$$n_{TB} = \frac{\$5,000,000 + \$24,852}{\$961,245} = 5.227$$

182-day T-bills, yielding a rate of return of

$$R = \left[\frac{(5.227)(\$1,000,000)}{\$5,000,000}\right]^{365/182} - 1 = 0.093$$

or 9.3% from a $5 million investment. Readers can verify for themselves that the rate of return will still be 9.3% if the 91-day and 182-day T-bill spot rates rise. Thus, Treasury bond futures contracts call for the delivery or purchase of a T-bond with a face value of $100,000. The contract allows for the delivery of a number of T-bonds; there is a conversion factor used to determine the actual price of the futures given the bond that is delivered. In actuality, the cheapest-to-deliver T-bond is delivered. T-bond futures are quoted in terms of a T-bond with an 8% coupon, semiannual payments, maturity of 15 years, and face value of $100.

In Matlab, one can make Treasury bills directly comparable to Treasury notes and bonds by restating U.S. Treasury bill market parameters in U.S. Treasury bond form as zero-coupon bonds via the following function:

```
[TBondMatrix, Settle] = tbl2bond(TBillMatrix)
```

`TBillMatrix` are the Treasury bill parameters. An N-by-5 matrix is where each row describes a Treasury bill. N is the number of Treasury bills. Columns are `[Maturity DaysMaturity Bid Asked AskYield]`, as described in Table 1.1.

Table 1.1

`Maturity`	Maturity date, as a serial date number. Use `datenum` to convert date strings to serial date numbers.
`DaysMaturity`	Days to maturity, as an integer. Days to maturity is quoted on a skip-day basis; the actual number of days from settlement to maturity is `DaysMaturity + 1`.
`Bid`	Bid bank-discount rate: the percentage discount from face value at which the bill could be bought, annualized on a simple-interest basis. A decimal fraction.
`Asked`	Asked bank-discount rate, as a decimal fraction.
`AskYield`	Asked yield: the bond-equivalent yield from holding the bill to maturity, annualized on a simple-interest basis and assuming a 365-day year. A decimal fraction.

The output consists of the Treasury bond parameters given in **`TBondMatrix`**, an N-by-5 matrix where each row describes an equivalent Treasury (zero-coupon) bond. Columns are `[CouponRate Maturity Bid Asked AskYield]`, as described in Table 1.2

Table 1.2

`CouponRate`	Coupon rate, which is always 0.
`Maturity`	Maturity date, as a serial date number. This date is the same as the Treasury bill `Maturity` date.
`Bid`	Bid price based on $100 face value.
`Asked`	Asked price based on $100 face value.
`AskYield`	Asked yield to maturity: the effective return from holding the bond to maturity, annualized on a compound-interest basis.

Example 1

Given published Treasury bill market parameters for December 22, 1997:

```
TBill = [datenum('jan 02 1998') 10 0.0526 0.0522 0.0530
         datenum('feb 05 1998') 44 0.0537 0.0533 0.0544
         datenum('mar 05 1998') 72 0.0529 0.0527 0.0540];
```

Execute the function:

```
TBond = tbl2bond(TBill)

TBond =
             0 729760 99.854 99.855 0.053
             0 729790 99.344 99.349 0.0544
             0 729820 98.942 98.946 0.054
```

1.3 Computing Treasury Bill Prices and Yields in Matlab

In Matlab, you can specify T-bills yield as money-market or bond-equivalent yield.

Matlab Treasury bill functions all assume a face value of $100 for each Treasury bill. The price of a T-bill can be computed using the function

```
Price = prtbill(Settle, Maturity, Face, Discount)
```

where the arguments are as listed in Table 1.3:

Table 1.3

`Settle`	Enter as serial date number or date string. `Settle` must be earlier than or equal to `Maturity`.
`Maturity`	Enter as serial date number or date string.
`Face`	Redemption (par, face) value.
`Discount`	Discount rate of the Treasury bill. Enter as decimal fraction.

Example 2

```
Settle = '2/10/2005';
Maturity = '8/7/2005';
Face = 1000;
Discount = 0.0379;
Price = prtbill(Settle, Maturity, Face, Discount);

Price =
      981.2606
```

The yield to maturity of the T-bill can be calculated using the `yldtbill` function:

```
Yield = yldtbill(Settle, Maturity, Face, Price)
```

The yield of the T-bill in this example is:

```
Yield = 0.0386
```

The bond equivalent yield of the T-bill is

```
BEYield = beytbill(Settle, Maturity, Discount)
```

where `Discount` is the discount rate of the T-bill, which can be computed from the `discrate` function:

```
Discount = discrate(Settle, Maturity, Face, Price, Basis)
```

In this example, Basis = 2 (Actual/360 day-count convention) so that

```
Discount = 0.0379
```

as expected (as initially given) so that the bond equivalent yield is as follows:

```
BEYield = 0.0392
```

1.4 HEDGING DEBT POSITIONS

Hedging a Future 91-Day T-Bill Investment with T-Bill Call

Following Johnson (2004), suppose a treasurer expects higher short-term rates in June but is still concerned about the possibility of lower rates. To be able to gain from the higher rates and yet still hedge against lower rates, the treasurer could buy a June call option on a spot T-bill or a June option on a T-bill futures. For example, suppose there was a June T-bill futures option with an exercise price of 90 (strike price $X = 975,000$), price of 1.25 ($C = \$3,125$), and June expiration (on both underlying futures and option) occurring at the same time a $5,000,000 cash inflow is to be received. To hedge the 91-day investment with this call, the treasurer would need to buy 5.128205 calls (assume divisibility) at a cost of $16,025.64:

$$n_c = \frac{CF_T}{X} = \frac{\$5,000,000}{\$975,000} = 5.128205 \text{ contracts}$$
$$Cost = (5.128205)(\$3,125) = \$16,025.64$$
$$\pi_c = 5.1282085[Max(S_T - \$975,000, 0) - \$3,125]$$

If T-bill rates were lower at the June expiration, then the treasurer would profit from the calls that she would be able to use to defray part of the cost of the higher priced T-bills. As shown in Table 1.4, if the spot discount rate on T-bills is 10% or less, the treasurer would be able to buy 5.112 91-day spot T-bills with the $5 million cash inflow and profit from the calls, locking in a YTM of 9.3% on the $5 million investment. On the other hand, if T-bill rates were higher, then the treasurer would benefit from lower spot prices, while the losses on the call would be limited to just the $16,025.64 cost of the calls. In this case, for spot discount rates above 10%, the treasurer would be able to buy more T-bills the higher the rates, resulting in higher yields as rates increase. Thus, for the cost of the call options, the treasurer is able to lock in a minimum YTM on the $5 million June investment of 9.3%, with the chance to earn a higher rate if short-term rates increase.

Table 1.4 Hedging $5M *CF* in June with June T-Bill Futures Call

Call: X = 90 (975,000), C = 1.25 ($3,125), n = 5.1282051				
1	2	3	4	5
Spot Rate: R	Spot Price	Profit/Loss	nTB	YTM
8	980000	9615.38456	5.112	0.093
8.5	978750	3205.12819	5.112	0.093
9	977500	-3205.1282	5.112	0.093
9.5	976250	-9615.3846	5.112	0.093
9.75	975625	-12820.513	5.112	0.093
10	975000	-16025.641	5.112	0.093
10.25	974375	-16025.641	5.115	0.096
10.5	973750	-16025.641	5.118	0.098
10.75	973125	-16025.641	5.122	0.101
11	972500	-16025.641	5.125	0.104
11.25	971875	-16025.641	5.128	0.107

Source: Johnson, S. (2004)

Note that if the treasurer wanted to hedge a 182-day investment instead of 91-days with calls, then similar to the futures hedge, she would need to buy both June and September T-bill futures calls. At the June expiration, the manager would then close both positions and invest the $5,000,000 inflow plus (minus) the call profits (losses) in 182-day spot T-bills.

Short Hedge: Managing the Maturity Gap

Short hedges are used when corporations, municipal governments, financial institutions, dealers, and underwriters are planning to sell bonds or borrow funds at some future date and want to lock in the rate. The converse of the preceding example would be a money market manager who, instead of buying T-bills, was planning to sell her holdings of T-bills in June when the current bills would have maturities of 91 days or 182 days. To lock in a given revenue, the manager would go short in June T-bill futures (if she plans to sell 91-day bills) or June and September futures (if she planned to sell 182-day bills). If short-term rates increase (decrease), causing T-bill prices to decrease (increase), the money manager would receive less (more) revenue from selling the bills, but would gain (lose) when she closed the T-bill futures contracts by going long in the expiring June (and September) contract.

Another important use of short hedges is in locking in the rates on future debt positions. As an example, consider the case of a small bank with a maturity gap problem in which its short-term loan portfolio has an average maturity greater than the maturity of the CDs that it is using to finance the loans. Specifically, suppose in June, the bank makes loans of $1 million, all with maturities of 180 days. To finance the loan, though, suppose the bank's customers prefer 90-day CDs to 180-day, and as a result, the bank has to sell $1 million worth of 90-day CDs at a rate equal to the current LIBOR of 8.258%. Ninety days later (in September) the bank would owe $\$1,019,758 = (\$1,000,000)(1.08258)^{90/365}$; to finance this debt, the bank would have to sell $1,019,758 worth of 90-day CDs at the LIBOR at

that time. In the absence of a hedge, the bank would be subject to market risk. If short-term rates increase, the bank would have to pay higher interest on its planned September CD sale, lowering the interest spread it earns (the rate from $1 million 180-day loans minus interest paid on CDs to finance them); if rates decrease, the bank would increase its spread.

Suppose the bank is fearful of higher rates in September and decides to minimize its exposure to market risk by hedging its $1,019,758 CD sale in September with a September Eurodollar futures contract trading at IMM = 92.1. To hedge the liability, the bank would need to go short in 1.03951 September Eurodollar futures (assume perfect divisibility):

$$f_0^{Sept} = \frac{100 - (7.9)(0.25)}{100}(\$1,000,000) = \$981,000$$

$$n_f = \frac{\$1,019,758}{\$981,000} = 1.03951 \text{ Short Eurodollar Contracts}$$

At a futures price of $981,000, the bank would be able to lock in a rate on its September CDs of 8.1%.

$$YTM_f^{Sept} = \left(\frac{\$1,000,000}{\$981,000}\right)^{365/91} - 1 = 0.081$$

$$YTM_{182} = \left[(1.0825)^{90/365}(1.081)^{90/36}\right]^{365/180} - 1 = 0.0817.$$

With this rate and the 8.25% rate it pays on its first CDs, the bank would pays 8.17% on its CDs over the 180-day period: That is, when the first CDs mature in September, the bank will issue new 90-day CDs at the prevailing LIBOR to finance the $1,019,758 first CD debt plus (minus) any loss (profit) from closing its September Eurodollar futures position. If the LIBOR in September has increased, the bank will have to pay a greater interest on the new CD, but it will realize a profit from its futures that, in turn, will lower the amount of funds it needs to finance at the higher rate. On the other hand, if the LIBOR is lower, the bank will have lower interest payments on its new CDs, but it will also incur a loss on its futures position and therefore will have more funds that need to be financed at the lower rates. The impact that rates have on the amount of funds needed to be financed and the rate paid on them will exactly offset each other, leaving the bank with a fixed debt amount when the September CDs mature in December. As Table 1.5 shows, where the bank's December

Table 1.5

Sept LIBOR (R)	0.075	0.085
(2) $S_T^{CD} = f_T^{Sept} = \$1M/(1+R)^{90/365}$	$982,236	$979,640
(3) $\pi_t = [981,000 - f_T]1.0391$	-$1,378	$1,413
(4) Debt on June CD	$1,019,758	$1,019,758
(5) Total Funds to finance for next 90 days: Row (4) − Row (3)	$1,021,136	$1,018,345
(6) Debt at end of next 90 days: Row (5) $(1+R)^{90/365}$	$1,039,509	$1,039,509
(7) Rate for 180-day period: $R_{CD}^{180} = [\$1,039,509/\$1,000,000]^{365/180} - 1$	8.17%	8.17%

Source: Johnson, R.S.

liability (the liability at the end of the initial 180-day period) is shown to be $1,039,509 given September LIBOR rate scenarios of 7.5% and 8.7% (this will be true at any rate).

Note that the debt at the end of 180 days of $1,039,509 equates to a September 90-day rate of 8.1% and a 180-day rate for the period of 8.17%:

$$YTM_f^{Sept} = \left[\frac{\$1,039,509}{\$1,019,758}\right]^{365/90} - 1 = 0.081$$

$$R_{CD180} = \left[\frac{\$1,039,509}{\$1,000,000}\right]^{365/180} - 1 = 0.0817$$

Maturity Gap and the Carrying Cost Model

In the preceding example, we assumed the bank's maturity gap was created as a result of the bank's borrowers wanting 180-day loans and its investors/depositor wanting 90-day CDs. Suppose, though, that the bank does not have a maturity gap problem; that is, it can easily sell 180-day CDs to finance its 180-day loan assets and 90-day CD to finance its 90-day loans. However, suppose that the September Eurodollar futures price was above its carrying cost value. In this case, the bank would find that instead of financing with a 180-day spot CD, it would be cheaper if it financed its 180-day June loans with synthetic 180-day CDs formed by selling 90-day June CDs rolled over three months later with 90-day September CDs, with the September CD rate locked in with a short position in the September Eurodollar futures contract. For example, if the June spot 180-day CD were trading 96 to yield 8.63%, then the carrying cost value on the September Eurodollar contract would be 97.897 ($97.897 = 96(1.08258)^{90/365}$). If the September futures price were 98.1, then the bank would find it cheaper to finance the 180-day loans with synthetic 180-days CDs with an implied futures rate of 8.17% than with a 180-day spot CD at a rate of 8.63%. On the other hand, if the futures price is less than the carrying cost value, then the rate on the synthetic 180-day CDs would exceed the spot 180-day CD rate, and the bank would obtain a lower financing rate with the spot CD. Finally, if the carrying cost model governing Eurodollar futures prices holds, then the rate of the synthetic will be equal to rate on the spot; in this case, the bank would be indifferent to its choice of financing.

Managing the Maturity Gap with Eurodollar Put

Instead of hedging its future CD sale with Eurodollar futures, the bank could alternatively buy put options on either a Eurodollar or T-bill or put options on a Eurodollar futures or T-bill futures. In the preceding case, suppose the bank decides to hedge its September CD sale by buying a September T-bill futures put with an expiration coinciding with the maturity of its June CD, an exercise price of 90 ($X = \$975,000$), and a premium of.5 ($C = \$1,250$). With the September debt from the June CD of $1,019,758, the bank would need to buy 1.046 September T-bill futures puts at a total cost of $1,307 to hedge the rate it pays on its September CD:

$$n_p = \frac{CF_T}{X} = \frac{\$1,019,758}{\$975,000} = 1.0459056 \text{ puts}$$

$$Cost = (1.046)(\$1,250) = \$1,307$$

If rates at the September expiration are higher such that the discount rate on T-bills is greater than 10%, then the bank will profit from the puts. This profit would serve to reduce part of the $1,019,750 funds it would need to finance the maturing June CD that, in turn, would help to negate the higher rate it would have to pay on its September CD. As shown in Table 1.6, if the T-bill discount yield is 10% or higher and the bank's 90-day CD rate is 0.25% more than the yield on T-bills, then the bank would be able to lock in a debt obligation 90 days later of $1,047,500, for an effective 180-day rate of 9.876%.

On the other hand, if rates decrease such that the discount rate on a spot T-bill is less than 10%, then the bank would be able to finance its $1,047,500 debt at lower rates, while its losses on its T-bill futures puts would be limited to the premium of $1,307. As a result, for lower rates, the bank would realize a lower debt obligation 90 days later and therefore a lower rate paid over the 180-day period. Thus, for the cost of the puts, hedging the maturity gap with puts allows the bank to lock in a maximum rate paid on debt obligations with the possibility of paying lower rates if interest rates decrease.

Table 1.6

Maturity Gap Hedged with T-Bill Puts

(1) R_D	(2) S_Y	(3) $Rate_{CD}$	(4) π_P	(5) *Debt on CD*	(6) *Funds Needed* (5) - (4)	(7) *Debt 90 Days Later* $(6)[1+(3)]^{90/345}$	(8) *Rate* $[(7)/1M]^{365/180}$
7%	$982,500	.07588	–1307	1,019,750	1,021,065	1,039,646	8.203%
8%	$980,000	.08690	–1307	1,019,750	1,021,065	1,042,262	8.756%
9%	$977,500	.09807	–1307	1,019,750	1,021,065	1,044,893	9.313%
10%	$975,000	.10940	–1307	1,019,750	1,021,065	1,047,500	9.867%
11%	$972,500	.12080	1307	1,019,750	1,018,451	1,047,500	9.867%
12%	$970,000	.13240	3922	1,019,750	1,015,836	1,047,500	9.867%

Assume 90-day CD rate is .25% greater than T-bill rate:

$$Rate_{CD} = \left[\frac{\$1M}{S_T}\right]^{365/91} + .0025 - 1$$

$$\pi_P = 1.4059056[Max[\$975{,}000 - S_T, 0] - \$1307$$

Source: Johnson, R.S.

Short Hedge: Hedging a Variable-Rate Loan

As a second example of a short naive hedge, consider the case of a corporation obtaining a one-year, $1 million variable-rate loan from a bank. In the loan agreement, suppose the loan starts on date 9/20 at a rate of 9.5% and then is reset on 12/20, 3/20, and 6/20 to equal the spot LIBOR (annual) plus 150 basis points (.015 or 1.5%) divided by four: $(\text{LIBOR} + .015)/4$.

To the bank, this loan represents a variable-rate asset, which it can hedge against interest rate changes by issuing 90-day CDs each quarter that are tied to the LIBOR. To the

corporation, though, the loan subjects them to interest rate risk (unless they are using the loan to finance a variable-rate asset). To hedge this variable-rate loan, though, the corporation could go short in a series of Eurodollar futures contracts—Eurodollar strip. For this case, suppose the company goes short in contracts expiring at 12/20, 3/20, and 6/20 and trading at the prices shown in Table 1.7.

Table 1.7

T	12/20	3/20	6/20
IMM Index	91.5	91.75	92
f_0(per \$100 Par)	97.875	97.9375	98

The locked-in rates obtained using Eurodollar futures contracts are equal to 100 minus the IMM index plus the basis points on the loans:

$$\begin{aligned}
&\textit{Locked-in Rate} = [100 - \textit{IMM}] + [\textit{BP}/100] \\
&12/20 : R_{12/20} = [100 - 91.5] + 1.5\% = 10\% \\
&\quad 3/20 : R_{3/20} = [100 - 91.75] + 1.5\% = 9.75\% \\
&\quad 6/20 : R_{6/20} = [100 - 92] + 1.5\% = 9.5\%
\end{aligned}$$

For example, suppose on date 12/20, the assumed spot LIBOR is 9%, yielding a settlement IMM index price of 91 and a closing futures price of 97.75 per \$100 face value. At that rate, the corporation would realize a profit of \$1,250 from having it short position on the 12/20 futures contract. That is:

$$\begin{aligned}
f_0 &= \frac{(100 - (100 - 91.5))(0.25)}{100}(\$1,000,000) = \$978,750 \\
f_T &= \frac{(100 - (100 - 91))(0.25)}{100}(\$1,000,000) = \$977,500 \\
&\text{Profit on 12/20 contract} = \$978,750 - \$977,500 = \$1,250
\end{aligned}$$

At the 12/20 date, though, the new interest that the corporation would have to pay for the next quarter would be set at \$26,250:

$$\begin{aligned}
&12/20 \text{ Interest} = [(\text{LIBOR} + .015)/4](\$1,000,000) \\
&12/20 \text{ Interest} = [(.09 + .015)/4](\$1,000,000) \\
&12/20 \text{ Interest} = \$26,250
\end{aligned}$$

Subtracting the futures profit from the \$26,250 interest payment (and ignoring the time value factor), the corporation's hedged interest payment for the next quarter is \$25,000. On an annualized basis, this equates to a 10% interest on a \$1 million loan, the same rate as the locked-in rate:

$$\text{Hedged Rate} = R^A = \frac{4(\$25,000)}{\$1,000,000} = 0.10$$

On the other hand, if the 12/20 LIBOR were 8%, then the quarterly interest payment would be only $\$23,750((.08+.015)/4)(\$1,000,000) = \$23,750$). This gain to the corporation, though, would be offset by a \$1,250 loss on the futures contract (i.e., at 8%, $f_T = \$980,000$, therefore, profit on the 12/20 contract is $\$978,750 - \$980,000 = -\$1,250$). As a result, the total quarterly debt of the company again would be $\$25,000(\$23,750 + \$1,250)$. Ignoring the time value factor, the annualized hedged rate the company pays would again be 10%. Thus, the corporation's short position in the 12/20 Eurodollar futures contract at 91.5 enables it to lock-in a quarterly debt obligation of \$25,000 and 10% annualized borrowing rate. If the LIBOR is at 9% on date 12/20, the company will have to pay \$26,250 on its loan the next quarter, but it will also have a profit on its 12/20 Eurodollar futures of \$1,250, which it can use to defray part of the interest expenses, yielding an effective hedged rate of 10%. The interest payments, futures profits, and effective interests are summarized:

$$
\begin{aligned}
12/20 : \textit{LIBOR} &= 9\% \\
\text{Futures:} & \\
\text{Settlementprice: } S_T &= 100 - \textit{LIBOR} \\
S_T &= 100 - 9(.25) = 97.75 \\
\pi_f &= \frac{97.875 - 97.75}{100}(\$1M) = \$1,250 \\
\text{Interest} &= \frac{\textit{LIBOR} + 150\textit{BP}}{4}(\$1M) \\
&= \frac{.09 + .015}{4}(\$1M) = \$26,250 \\
\text{EffectiveInterest} &= \$26,250 - \$1,250 = \$25,000 \\
\text{EffectiveRate} &= R^A = \frac{4(\$25,000)}{\$1M} = .10
\end{aligned}
$$

If the LIBOR is at 6% on date 12/20, the company will have to pay only \$18,750 on its loan the next quarter, but it will also have to cover a loss on its 12/20 Eurodollar futures of \$6,250. The payment of interest and the loss on the futures yields an effective hedged rate of 10%:

$$
\begin{aligned}
12/20 : \textit{LIBOR} &= 6\% \\
\text{Futures:} & \\
\text{Settlementprice} : S_T &= 100 - \textit{LIBOR} \\
S_T &= 100 - 6(.25) = 98.5 \\
\pi_f &= \frac{97.875 - 98.5}{100}(\$1M) = -\$6,250 \\
\text{Interest} &= \frac{\textit{LIBOR} + 150\textit{BP}}{4}(\$1M) \\
&= \frac{.06 + .015}{4}(\$1M) = \$18,750
\end{aligned}
$$

$$\text{EffectiveInterest} = \$18,750 + \$6,250 = \$25,000$$

$$\text{EffectiveRate} = R^A = \frac{4(\$25,000)}{\$1M} = .10$$

Given the other locked-in rates, the one-year fixed rate for the corporation on its variable-rate loan hedged with the Eurodollar futures contracts would therefore be 9.6873%:

$$\textit{Loan Rate} = \left[(1.095)^{.25}(1.10)^{.25}(1.0975)^{.25}(1.095)^{.25}\right]^1 - 1 = .096873.$$

Note, in practice, the corporation could have obtained a one-year fixed-rate loan. With futures contracts, though, the company now has a choice of taking either a fixed-rate loan or a synthetic fixed-rate loan formed with a variable-rate loan and short position in a Eurodollar futures contract, whichever is cheaper. Also, note that the corporation could have used a series of Eurodollar puts or futures puts to hedge its variable-rate loan. With a put hedge, each quarter the company would be able to lock in a maximum rate on its loan with the possibility of a lower rate if interest rates decrease.

1.5 Bond and Swap Duration, Modified Duration, and DV01

Duration (also known as Macauley's duration) is the present-value-weighted average time (in years) to maturity of the cash flow payments of a fixed income security. If c_i is the total cash flow payment at time t_i, then duration is computed as

$$\textit{Duration} = \frac{\sum_{i=1}^{n} \frac{t_i c_i}{(1+y_i)^{t_i}}}{B} \tag{1.4}$$

where n = number of cash flows, t = time to cash flow (in years), y_i is the simple-compound discount yield at time t_i, and $B = \sum_{i=1}^{n} \frac{c_i}{(1+y_i)^{t_i}}$ is the bond price.[12] At maturity, the cash flow includes both the principal face value and coupon payment. Yield modified duration[13] divides the duration number by $(1 + 1/YTM)$, where *YTM* is the yield to maturity of the bond. However, the *YTM* used to calculate the yield modified duration of a swap is the par swap rate (e.g., the swap rate that makes the present value of the swap zero).

To calculate the duration (modified duration) of a swap, one computes the modified duration of the long leg minus the modified duration of the short leg. Duration for the fixed leg is the present-value-weighted average maturity of the cash flows, whereas duration of the floating leg is the time to the next reset for the floating leg. The duration calculations (for both sides) divided by $1/(1+YTM)$ is the modified duration for each side. The modified duration of an interest rate swap leg can be calculated like a bond

$$MD = \frac{\textit{Leg DV01}}{\textit{Leg PV}} * 10,000 \tag{1.5}$$

where DV01 (sometimes denoted PV01 in the literature) is the dollar value of a basis point change of the leg[14] and PV is the present value of the leg. This calculation should always

be positive. An alternative, but equivalent, calculation to compute modified duration for each leg is to compute

$$MD = \frac{Leg\ DV01}{(Notional\ Value + Market\ Value)} * 10,000 \tag{1.6}$$

where the Market Value is the leg PV. This alternative computation is appropriate if the leg PV does not include the final exchange of principal. However, the leg DV01 must include the notional exchange to give a bond-like duration, so the leg DV01 must also include notional.

The difference between the modified durations of both legs, the net PV, is then taken to compute the modified duration of the swap:

$$\text{Swap MD} = \text{MD of Receive Leg} - \text{MD of Pay Leg} \tag{1.7}$$

Note that if the market value (leg PV) of the swap is zero, then duration is computed. Thus, for a par swap where the market value is zero, the duration (referred to as *deal risk* in Bloomberg) is equal to the modified duration.

For forward-starting swaps—e.g., a 10-year swap that starts in 7 years—typically a discounted swap modified duration is computed by adjusting equation (1.7) by a factor that is very close to the discount factor for the start date of the swap. An excellent approximation for this discount factor is provided by the NPV of the floating leg (including the final notional exchange), divided by the notional of the trade:

$$\text{Discounted Swap MD} = \text{Swap MD} \times \text{Floating Leg PV}/\text{Notional} \tag{1.8}$$

Note that quantitative differences arise between the yield modified duration (which assumes a flat discount rate) and the DV01 formulation in (1.5) or (1.6), based on various factors such as a sloping yield curve, instrument valuations away from par, instruments with long tenors, and forward-starting instruments. With an upward-sloping discount curve, the PV of the largest cash flow (at maturity) affects duration the most. Because the yield is lower than the discount rate at maturity, the value of the yield MD is higher than the value of the DV01 MD. The effect of the upward-sloping curve is amplified with a longer tenor, because the largest cash flow at maturity is even more different when discounted using the curve for DV01 MD and using the yield for yield MD; thus, the yield MD is greater than the DV01 MD. A downward-sloping discount curve has the opposite affect because the yield is higher than the discount rate at maturity.

Large NPVs during a tenor with an upward-sloping yield curve result from a coupon much different from the discount rate. The present value of the coupon cash flows have a larger weight in the bond price (PV). For DV01 MD, the PV of coupons in the long end decreases more when the curve is shifted by 1 basis point, compared to yield MD where coupons at the end are discounted at a lower rate (the yield) and are less sensitive to interest rate changes. Also, the DV01 MD is calculated by dividing by a higher value of bond PV so that the yield MD is greater than the DV01 MD. For a forward starting swap with an upward-sloping yield curve, yield MD is calculated as the sum of the yield MDs of two bonds with opposite signs—a long position in a bond starting today and maturing in T_2 years, and a short position in bond starting today and maturity in T_1 years, where

$0 < T_1 < T_2$. The cash flows affecting yield are those between $T_1 + 1$ and T_2 years, which are discounted at a yield lower than the discount rates at the long end of the curve. As a result, yield MD is higher than the DV01 MD.

Hedging Bond Portfolios

Consider the following bond portfolio example shown in Table 1.8.[15]

Of the Treasury bonds in the portfolio, the 5% of August 2011 was the current on-the-run 10-year, while the 6.5% of February 2010 became the cheapest-to-deliver (CTD) for the CBOT 10-year Treasury note futures when the data was recorded. The 5.625% of May 08 had recently been CTD. The coupons of the 11 corporate bonds in Table 1.8 range from a high of 9.375% to a low of 6.15% and have maturities from August 2007 to August 2011. With the exception of American Standard, all of these issues are investment grade credits where the credit ratings range from the S&P AA- of the Transamerica issue to the BBB- of the News America Holdings and Litton Industries issue. American Standard is a BB+ credit.

Table 1.8 shows weighted average durations for the two sectors and for the portfolio as a whole. The Treasury sector shows that this $262.34 million holding has a duration of 6.56 years. The $436.25 million corporate sector holding has a duration of 5.39 years, and the $698.59 million portfolio has a duration of 5.83 years.

Fixed income securities with different coupons and maturities respond differently to yield changes. This price sensitivity to yield change can be captured in terms of the dollar value of a basis point (DV01) for a given security. To find the DV01 for an individual fixed income security, given a modified duration and a full price, solve the following:

Table 1.8

Issuer	Coupon	Maturity	YTM	Modified Duration (years)	DV01 ($)	Par Amount ($ millions)	Full Price ($000s)	S&P Credit Rating
Treasury	6.5	2/15/10	4.55	6.56	0.0748	67	76,387	
Treasury	5.625	5/15/08	4.64	5.48	0.0588	92	98,762	
Treasury	5	8/15/11	4.57	7.77	0.0807	84	87,192	
Treasury Sector				6.56			262,341	
Time Warner Enterprises	8.18	8/15/07	5.47	4.72	0.0540	82	93,710	BBB+
Texas Utilities	6.375	1/1/08	6.19	5.06	0.0517	30	30,678	BBB
Rockwell International	6.15	1/15/08	6.14	5.13	0.0521	52	52,837	A
Transamerica Corporation	9.375	3/1/08	6.34	4.93	0.0573	15	17,445	AA-
Coastal Corporation	6.5	6/1/08	6.76	5.25	0.0528	30	30,153	BBB
United Airlines	6.831	9/1/08	5.99	5.51	0.0579	38	39,949	A-
Burlington Northern Santa Fe	7.34	9/24/08	5.67	5.36	0.0606	3	3,390	A+
News America Holdings	7.375	10/17/08	6.56	5.34	0.0575	30	32,292	BBB-
Litton Industries	8	10/15/09	5.70	5.81	0.0647	63	66,834	BBB-
American Standard Inc.	7.625	2/15/10	7.59	6.10	0.0615	41	41,332	BB+
Caterpillar Inc.	9.375	8/15/11	6.01	6.79	0.0853	22	27,628	A+
Corporate Sector				5.39			436,248	
Portfolio				5.83			698,589	

$$DV01 = \frac{(Duration/100) * Full\ \text{Price}}{100}$$

For instance, using this formula with the price and duration date in Table 1.8, one sees that a $98,762,000 position in the Treasury 5.625% of May 08, with its 5.48 duration, has a DV01 of $54,122, while an $87,192,000 position in the Treasury 5% of August 11, with its 7.77 duration, has a $67,748 DV01. These DV01's predict that 1 basis point rise in yield would drive the value of the 5.625% of May 08 down $54,122, while the same yield change would drive the value of the smaller holding in the 5% of August 11 down $67,748. Clearly, the 5% of August 11 is more sensitive to yield changes than the 5.625% of May 08 is.

The portfolio is subject to inflation, interest rate, and credit risk that could sharply erode the value of the holdings and make it difficult to liquidate them at acceptable prices. The structuring of hedge positions then requires that the futures position be ratioed to the positions, and then requires that the futures position be ratioed to the position in the security being hedged in order that the two positions will respond equally to a given yield change. Given that the DV01 of the 10-year Treasury note futures contract was $72.50 on the day these dates were recorded (the 10-year swap futures has a $77.00 DV01), one can solve for the optimal hedge ratio to see that it would take 747 contracts to hedge the 5.625% of May 08 and 934 contracts to hedge the 5% of August 11:

$$\frac{5.625\%\ \text{of May 08 DV01}}{\text{futures DV01}} = \frac{54,122}{72.50} = 747\ \text{contracts}$$

$$\frac{5\%\ \text{of August 11 DV01}}{\text{futures DV01}} = \frac{67,748}{72.50} = 934\ \text{contracts}$$

To structure a portfolio hedge, one can use a weighted average duration for the Treasury sector, the corporate sector, or the entire holding, and then use the total full price of the relevant sectors to find DV01s. Table 1.9 shows that, based on the data in Table 1.8, the Treasury sector has a $172,096 DV01, the corporate segment has a $235,138 DV01, and

Table 1.9

		DV01 ($)
Treasury Sector		172,096
Corporate Sector		235,138
Portfolio		402,277
10-year Treasury Futures		72.50
10-year Swap Futures		77.00
Hedge Ratios		
	To Hedge	*Hedge Ratio*
1	Full portfolio with 10-yr. T-note futures	5,618
2	Full portfolio with swap futures	5,289
3	Treasury sector with 10-yr. T-note futures	2,374
4	Corporate sector with swap futures	3,054

the entire portfolio has a $407,277 DV01 (note that the portfolio DV01 does not equal the sum of the two sector DV01s due to rounding error).

Plugging in these DV01s into the hedge ratio formulas yields the optimal hedge ratios. Hedging the entire portfolio with Treasury futures requires a short position in 5,618 contracts:

$$\frac{\text{Portfolio DV01 } 407,277}{\text{10-year Treasury Futures DV01 } 72.50} = 5,618 \text{ contracts}$$

Similarly, hedging the entire portfolio with swap futures requires a short position of 5,289 10-year swap futures contracts:

$$\frac{\text{Portfolio DV01 } 407,277}{\text{10-year Swap Futures DV01 } 77.00} = 5,289 \text{ contracts}$$

Finally, an optimal hedging strategy where the Treasury sector is hedged with Treasury futures and the corporate sector is hedged with swap futures requires a short position in 2,375 10-year Treasury futures contracts and a short position in 3,054 10-year swap futures contracts:

$$\frac{\text{Treasury Sector DV01 } 172,096}{\text{10-year Treasury Futures DV01 } 72.50} = 2,375 \text{ contracts}$$

$$\frac{\text{Corporate Sector DV01 } 235,138}{\text{10-year Swap Futures DV01 } 72.50} = 3,054 \text{ contracts}$$

It is important to note that these hedges are static in nature—they only apply at a given point in time based on current market data. Because Treasury, corporate yields, and swap rate cause shifts in the DV01s, these hedges should be monitored constantly and rebalanced based on the changes in interest rates and thus amount of risk exposure.

Hedging and portfolio performance can be measured by scenario and prediction analysis of yield shifts. During the summer of 2001, the 10-year Treasury-swap rate credit spread increased 40 basis points. Corporate yields generally followed the direction of the swap rate. Should that occur again where the swap rate and corporate yields rise 60 bps while Treasury yields rise only 20 bps, the underlying portfolio will incur an approximate $17.55 million loss. Table 1.10 shows that the sector DV01s predict that $3,441,920 of the loss will come from the Treasury sector and $14,108,280 of it will come from the corporate sector. The Treasury futures hedge (according to the prediction of $72.50 DV01) will respond to the 20 bp change in the Treasury yield and generate an $8,146,000 gain (see Table 1.10, Scenario 1). Based on this scenario, a hedge mismatch loss of $9,404,100 is incurred as a result of the fact that the Treasury futures hedge can be expected to offset less than half the portfolio loss.

Alternatively, swap futures can be used to hedge the entire portfolio. This strategy has the advantage of responding to the larger change in the swap rate and corporate yields. However, as Table 1.10 Scenario 2 shows, the relevant DV01s predict that this hedge will generate a futures gain much larger than the loss the underlying portfolio will incur. As Table 1.10, Scenario 3 shows, the relevant DV01s predict that these two hedge positions will generate a total futures gain of $17,551,780. Thus, Scenario 3 appears far more promising

Table 1.10

Underlying Portfolio Result

Sector	DV01 ($)	Yield Change (bps)	Number of Contracts	Gain/Loss ($)
Treasury	172,096	20		-3,441,920
Corporate	235,138	60		-14,108,280
Portfolio				-17,550,200

Scenario 1 - Hedge Entire Portfolio with Treasury Futures

Sector	DV01 ($)	Yield Change (bps)	Number of Contracts	Gain/Loss ($)
10-year T-note	72.50	20	5,618	8,146,100
Portfolio				-17,550,200
Hedge Mistatch				-9,404,100

Scenario 2 - Hedge Entire Portfolio with Swap Futures

Sector	DV01 ($)	Yield Change (bps)	Number of Contracts	Gain/Loss ($)
10-year swap	77.00	60	5,289	24,435,180
Portfolio				-17,550,200
Hedge Mistatch				6,884,980

Scenario 3 - Hedge Treasury Sector with Treasury Futures, Hedge Corporate Sector with Swap Futures

Sector	DV01 ($)	Yield Change (bps)	Number of Contracts	Gain/Loss ($)
10-year T-note	72.50	20	2,374	3,442,300
10-year swap	77.00	60	3,054	14,109,480
Total Hedge Result				17,550,780
Portfolio				-17,550,200
Portfolio Mismatch				1,380
Treasury to 10-year T-note Mismatch				380
Corporate to Swap Mismatch				1,200

than either of the first two hedges given the minimal hedge mismatch of $1,580 (which represents less than a basis point and is far less than normal bid-ask spreads, and so for practical purposes represents a good offset). It highlights the importance of using exact or like (highly correlated) sector hedges to hedge underlying sectors: hedge the Treasury sector with 10-year Treasury note futures and the corporate sector with 10-year swap futures.

The large variation in Scenario 1 and Scenario 2, compared with the small hedge mismatch in Scenario 3, can be explained by the impact of basis risk that cannot be captured by single sector hedges of an entire underlying fixed income portfolio with multiple sectors. A 10-year Treasury futures is not as correlated with the corporate sector as the 10-year swap futures. In other words, swaps correlate more closely with corporates than Treasuries do. For example, consider a corporate-swap and corporate-Treasury regression using the TransAmerica Corp. 9.375% March 2008 corporate bond in Table 1.8. The corporate-swap correlation regression coefficient R^2 is 0.9134, while the corporate-Treasury R^2 is 0.7966, as shown in Figure 1.1.

The 0.9134 R^2 of the corporate-swap regression suggests that the variability of the swap rate accounts for 91.34% of the variability in the corporate yield. In contrast, the 0.7966

Transamerica Corp. 9.375% of March 2008 vs. 10-Year Constant Maturity Treasury and 10-Year Constant Maturity Swap (Daily, October 2000 to September 2001)

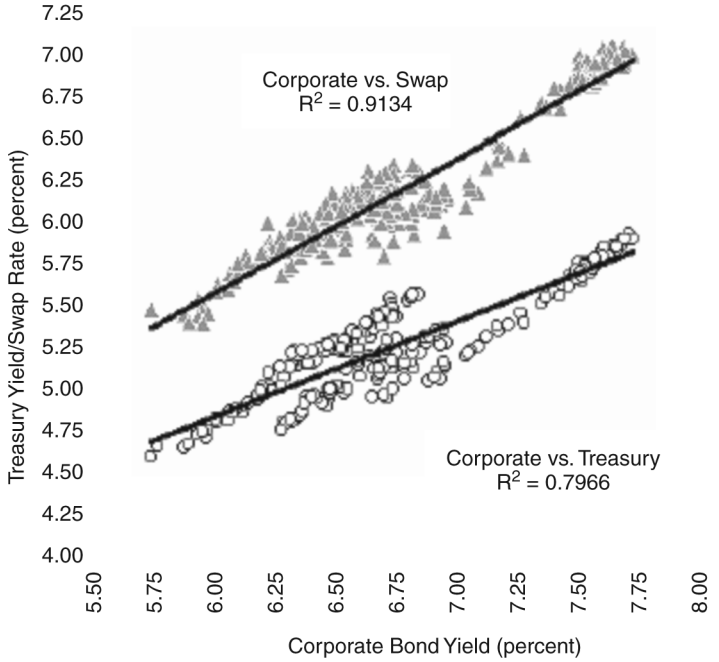


Figure 1.1 *Reprinted by permission of the Board of Trade of the City of Chicago, Inc., © 2001. All Rights Reserved.*

R^2 of the corporate-Treasury regression suggests that the variability of the Treasury yield accounts for slightly less than 80% of the variability of the yield of the corporate bond.

1.6 Term Structure of Rates

Term structure modeling is essential for valuation of fixed income securities and derivatives as a means for quantifying the relationship between either price or yield among a set of securities that differ only in the timing of their cash flows or their term until maturity. Term structure models describe the evolution of interest rates over time. The relationship expressed by the term structure is traditionally the par-coupon yield relationship, though in general, the term structure could be the discount function, the spot-yield curve, or some other price-yield relationship.

The set of securities that define a term structure is known as the *reference set*, which may be a set of U.S. Treasuries, agency debentures, off-the-run Treasury issues, interest rate swaps, or single-A rated corporate bonds, for instance. The n-year zero coupon yield, also known as the n-year spot rate, is the interest rate on an investment that is earned for a period of time starting today and lasting for n years. There is widespread usage of the par yield

curve for the Treasury market so that many market sectors are defined from a reference set derived from the Treasury market;[16] for example, "the reference set that defines the agency debenture market is a set of yield spreads on the on-the-run Treasuries, so that the five-year debenture issued by an agency may be priced at par to yield 15 basis points more than the current five-year Treasury issue."[17] Other important yield curves include the forward rate curve and swap curve. Forward interest rates (future spot rates) are the rates of interest implied by current spot rates for periods of time in the future. The swap curve shows the fixed rate that is to be paid for receiving a floating rate such as three-month Libor, for a given swap maturity.

There are typically three types of term structures. The (coupon) yield curve is the yield-to-maturity structure of coupon bonds. The zero-coupon yield or spot-rate curve is the term structure of discount rates of zero-coupon bonds. The forward rate curve is the term structure of forward rates implicit in zero-coupon discount rates.

1.7 BOOTSTRAP METHOD

To construct the zero-coupon yield curve when spot rates cannot be observed directly, use the *bootstrap method* to extract it from observable coupon-bearing bond (cash) prices, swap rates, and interest rate futures. The bootstrap method, an iterative numerical method for extracting spot rates from previously computed spot rates and observable bond prices, can be used to construct the discount rate curve. Denote y_n as the yield to maturity of an n-period bond of maturity and d_n as the spot (discount) rate of the n-period maturity bond. The method is based on the idea that a coupon bond can be decomposed into the sum of zero-coupon bonds:

$$\begin{aligned} P_n &= \frac{c/m}{1+y_n/2} + \frac{c/m}{(1+y_n/2)^2} + \ldots + \frac{F+c/m}{(1+y_n/2)^n} \\ &= \frac{c/m}{1+d_n/2} + \frac{c/m}{(1+d_n/2)^2} + \ldots + \frac{F+c/m}{(1+d_n/2)^n} \\ &= Z_1 + Z_2 + \ldots + Z_n \end{aligned}$$

The standard procedure for bootstrapping the Treasury yield curve, known as *recursive stripping*, is as follows:

1. Obtain the current price of U.S. Treasuries (and/or Eurodollar futures).
2. Solve for the yield on the one-period zero-coupon bond, z_1.
3. Solve for z_2, given the price of the two-period bond and z_1 calculated in step 2.
4. Solve for z_3, given the price of the three-period bond and z_1 and z_2.
5. Continue until all the spot rates have been calculated.

Bootstrapping is a process whereby one starts with known data points and then solves for unknown data points using an underlying arbitrage theory. Every coupon bond can be valued as a package of zero-coupon bonds that mimic its cash flow and risk characteristics.

By mapping yields-to-maturity for each theoretical zero-coupon bond, to the dates spanning the investment horizon, one can create a theoretical zero-rate curve.

The zero spot rates are typically quoted on a bond equivalent basis.

One can use the bootstrap method to extract the zero coupon curve, which can in turn be used to construct a discount rate curve to discount cash flows. The generation of these curves typically starts with a series of on-the-run and selected off-the-run issues as input. All cash flows are used to construct the spot curve, and rates between maturities (for these coupons) are linearly interpolated.

For each bond maturity, we solve for the current yield that equates the current bond maturity with the sum of its discounted cash flows based on the determination of the yields of all the shorter maturity bonds. For instance, to determine the short rate yield on a bond that matures in n-years, we need to solve for d_n, which can only be achieved if all the yields for bonds maturing at periods $i = 1, ..., n-1$ are determined. The method is called the bootstrap method because it determines yields (and thus the short rate curve) by building on top or "bootstrapping" from all previously determined yields:

$$P_n = \sum_{i=1}^{n-1} \frac{C_n}{(1+d_i)^i} + \frac{C_n + FV}{(1+d_n/2)^n}$$

For a zero-coupon bond, one can compute the (continuously computed) discount rate d_c from a discount spot rate with compounding n times per annum[18] via the following equation:

$$d_c = n \ln\left(1 + \frac{c_n}{n}\right)$$

Consider the Treasury bills and bonds listed in Table 1.11. We assume bond prices that mature every six months are available out to five years.

When computing the spot rates, on-the-run treasuries are typically used as they are more liquid and normally trade close to par, thereby mitigating any tax-biases associated with discount or premium bonds. Unfortunately, on-the-run securities are only available at 0.5-, 1-, 2-, 3-, 5-, and 10-year maturities. Of the methods used to obtain spot rates between

Table 1.11

Bond Price	Annual Coupon	Semiannual Period	Maturity (Years)	Period Coupon
102.2969	6.125	1	0.5	3.0625
104.0469	6.25	2	1.0	3.125
104.0000	5.25	3	1.5	2.625
103.5469	4.75	4	2.0	2.375
109.5156	7.25	5	2.5	3.625
111.1719	7.5	6	3.0	3.750
122.4844	10.75	7	3.5	5.375
119.6094	9.375	8	4.0	4.687
111.3281	7.0	9	4.5	3.500
108.7031	6.25	10	5.0	3.125

these values, exponential cubic splines[19] is the most common. However, recursive stripping is also common. We assume the maturity face value is 100.

To bootstrap the curve, we start by computing the discount spot rate that equates the cash flows to the 0.5-year maturity bond price of 102.27:

$$102.27 = \frac{(3.0625 + 100)}{(1 + d_1/2)}$$

or solving for d_1:

$$d_1 = (103.0625/102.27 - 1) * 2 = 0.014968$$

We compute the discount sum for semiannual period 1:

$$\textit{DiscountSum}1 = 1/(1 + d_1) = 1/(1.04968) = 0.9853$$

We next solve for the one-year spot rate that gives the bond price at semiannual period 2:

$$\begin{aligned} 104.05 &= \frac{3.125}{(1 + d_1)} + \frac{103.125}{(1 + d_2/2)^2} \\ &= 3.125 * \textit{DiscountSum}1 + 103.125/(1 + d_2/2)^2 \\ &= 2 * \left(\left(\frac{\textit{PeriodCoupon}2 + \textit{FaceValue}}{\textit{BondPrice}2 - \textit{PeriodCoupon}2 * \textit{DiscountSum}1} \right)^{1/2} - 1 \right) \end{aligned}$$

so that

$$d_2 = 2.1250\%.$$

We compute the discount sum for semiannual period 2:

$$\begin{aligned} \textit{DiscountSum}2 &= \textit{DiscountSum}1 + 1/(1 + d_2)^2 \\ &= 0.9853 + 1/(1.02125)^2 \\ &= 1.94407 \end{aligned}$$

We next solve for the sport rate that equates the cash flows to the 1.5-year maturity bond price (third semiannual period):

$$104 = 2.625 * \textit{DiscountSum}2 + \frac{102.625}{(1 + d_3/2)^3}$$

or solving

$$\begin{aligned} d_3 &= 2 * \left(\left(\frac{\textit{PeriodCoupon}3 + \textit{FaceValue}}{\textit{BondPrice}3 - \textit{PeriodCoupon}3 * \text{DiscountSum}2} \right)^{1/3} - 1 \right) \\ &= 2.4822\%. \end{aligned}$$

So that, in general, the bootstrapped spot rate for semiannual period n is given by

$$d_n = 2 * \left(\left(\frac{\textit{PeriodCoupon}(n) + \textit{FaceValue}}{\textit{BondPrice}(n) - \textit{PeriodCoupon}(n) * \textit{DiscountSum}(n-1)} \right)^{1/n} - 1 \right)$$

where

$$\textit{DiscountSum}(n-1) = \textit{DiscountSum}(n-2) + 1/(1+d_2)^{n-1}.$$

If we continue the process, we find the discount sums and spot rates shown in Table 1.12, which yield the term structure of rates shown in Figure 1.2.

In practice, bootstrapping requires the input of T-bills, Treasury bond, Treasury note futures, Eurodollar futures, or coupon-bearing bond prices. One must interpolate the spot rate yields for specific maturities not available from the available market prices. For instance, some bond tables list net yields for bonds in a sequence of one, three, and five years. Interpolation would be used to determine the yield for the second and fourth year. In effect, interpolation is a process of trial and error utilizing a numerical method like the Newton-Raphson method. What complicates the procedure is that day count conventions need to be taken into account for computing accrued interest on coupon-bearing bonds.

Table 1.12

Period	Discount Sum	Spot Rate
1	0.985252547	1.4968%
2	1.944068996	2.1250%
3	2.87315116	2.4822%
4	3.76649457	2.8597%
5	4.623301565	3.1391%
6	5.442647715	3.3766%
7	6.227073953	3.5295%
8	6.975043088	3.6966%
9	7.682592791	3.9187%
10	8.359670683	3.9767%

1.8 BOOTSTRAPPING IN MATLAB

There are three ways to import data into Matlab: (1) loading the data as a "flat file" in ASCII format and then converting it to a numerical matrix; (2) loading the data first into Excel and then using ExcelLink to pass numeric matrices to the Matlab workspace; or (3) using the Matlab Database Toolbox to pull the data from an ODBC-compliant database.

Matlab, via the Financial Toolbox, provides two bootstrapping functions: `zbtprice`, which bootstraps the zero curve from coupon-bond data-given prices and `zbtyield`, which bootstraps the zero curve from coupon-bond data-given yields. Suppose we want to bootstrap the yield curve in Matlab using the corporate bond data in Figure 1.1 to get the zero rates at the maturity rates of the bonds. Assume the settlement date is January 2, 2006.

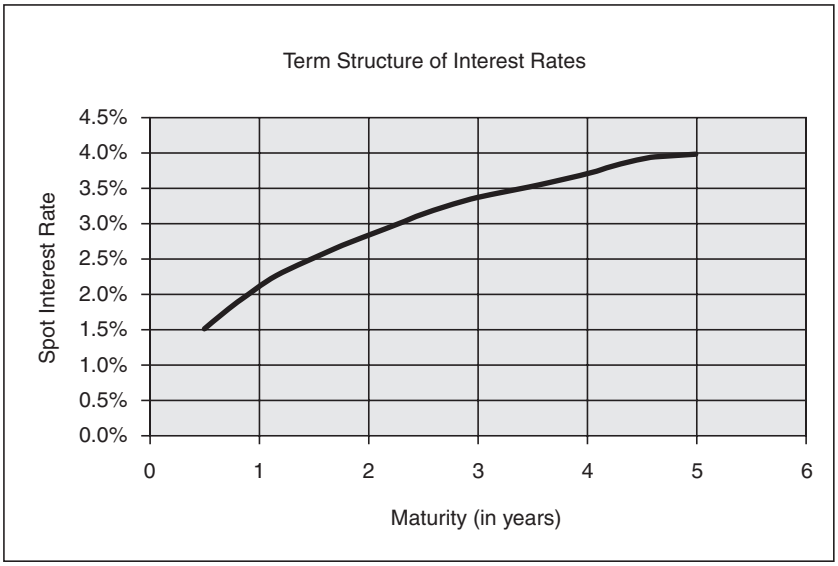


Figure 1.2 Term structure of rates

```
% Bonds = [Maturity CouponRate FaceValue]
Bonds = [datenum('15-Aug-2007')  0.08180   100;
         datenum('01-Jan-2008')  0.06375   100;
         datenum('15-Jan-2008')  0.06150   100;
         datenum('01-Mar-2008')  0.09375   100;
         datenum('01-Jun-2008')  0.06500   100;
         datenum('01-Sep-2008')  0.06831   100;
         datenum('24-Sep-2008')  0.07340   100;
         datenum('17-Oct-2008')  0.07375   100;
         datenum('15-Oct-2009')  0.08000   100;
         datenum('15-Feb-2010')  0.07625   100;
         datenum('15-Aug-2011')  0.09375   100];

Yields = [0.0547; 0.0619; 0.0604; 0.0634; 0.0676; 0.0599; 0.0567;
          0.0656; 0.0670; 0.0750; 0.0601];

Prices = [114.28; 102.26; 101.61; 116.30; 100.51; 105.13; 113.00;
          107.64; 111.39; 100.81; 125.58];

Settle = datenum('02-Jan-2006');

[ZeroRates, CurveDates] = zbtyield(Bonds,Yields,Settle)

ZeroRates =

    0.0547
    0.0623
```

```
    0.0607
    0.0641
    0.0683
    0.0600
    0.0565
    0.0661
    0.0676
    0.0769
    0.0586

datestr(CurveDates) =

15-Aug-2007
01-Jan-2008
15-Jan-2008
01-Mar-2008
01-Jun-2008
01-Sep-2008
24-Sep-2008
17-Oct-2008
15-Oct-2009
15-Feb-2010
15-Aug-2011
```

We could also bootstrap using the bond prices:

```
      [ZeroRates] = zbtprices(Bonds,Prices,Settle);

ZeroRates =

    0.0251
    0.0581
    0.0582
    0.0400
    0.0665
    0.0554
    0.0362
    0.0533
    0.0548
    0.0810
    0.0469
```

It should be noted that these zero rates are not risk-free discount rates because the coupon bonds are not risk free. To compute risk-free zero-rates, T-bills, Treasury notes, and Treasury bonds should be used.

1.9 BOOTSTRAPPING IN EXCEL

Consider the Excel spreadsheet (ZC.xls) shown in Figure 1.3 with a worksheet called "Bootstrap," which takes zero coupon rates as input and interpolates between maturity dates. To view the Visual Basic code, click on Tools > Macro > Visual Basic Editor.

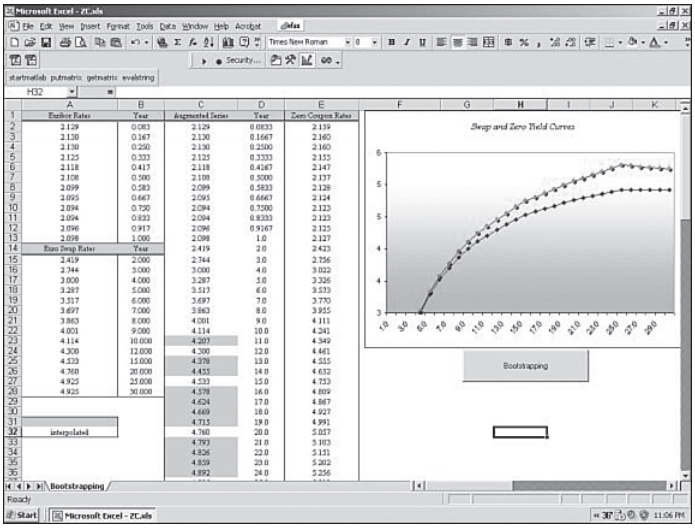


Figure 1.3 Interpolating zero coupon rates between maturity dates

```
Public ZC(1 To 100) As Double                // stores zero coupon rates
Public swaprate(1 To 100) As Double          // stores swap rates
Public dataswap(1 To 100) As Double
```

Sub_interpolation_swap.rtf

```
Sub interpolation_swap()

  ReadRates_over10y  'Load swap rates over 10 years

  For i = 1 To 12
    Sheets("Bootstrapping").Cells(i + 1, 3) =
      Sheets("Bootstrapping").Cells(i + 1, 1)
    Sheets("Bootstrapping").Cells(i + 1, 4) =
      Sheets("Bootstrapping").Cells(i + 1, 2)
  Next i

  For i = 1 To 9
    Sheets("Bootstrapping").Cells(i + 13, 3) =
      Sheets("Bootstrapping").Cells(i + 14, 1)
    Sheets("Bootstrapping").Cells(i + 13, 4) =
      Sheets("Bootstrapping").Cells(i + 14, 2)

  Next i

  Sheets("Bootstrapping").Cells(24, 3) = swaprate(12)
  Sheets("Bootstrapping").Cells(27, 3) = swaprate(15)
  Sheets("Bootstrapping").Cells(32, 3) = swaprate(20)
  Sheets("Bootstrapping").Cells(37, 3) = swaprate(25)
```

```
Sheets("Bootstrapping").Cells(42, 3) = swaprate(30)

'Start the interpolation procedure for each knot

swaprate(11) = (swaprate(10) + swaprate(12)) * 0.5
Sheets("Bootstrapping").Cells(23, 3) = swaprate(11)


For i = 13 To 14
  swaprate(i) = swaprate(12) + ((swaprate(15) - swaprate(12)) *
    ((i - dataswap(2)) / (dataswap(3)
- dataswap(2))))
  Sheets("Bootstrapping").Cells(i + 12, 3) = swaprate(i)
Next i


For i = 16 To 19
swaprate(i) = swaprate(15) + ((swaprate(20) - swaprate(15)) *
  ((i - dataswap(3)) / (dataswap(4) - dataswap(3))))
  Sheets("Bootstrapping").Cells(i + 12, 3) = swaprate(i)
Next i


For i = 21 To 24
  swaprate(i) = swaprate(20) + ((swaprate(25) - swaprate(20)) *
    ((i - dataswap(4)) / (dataswap(5)
- dataswap(4))))
  Sheets("Bootstrapping").Cells(i + 12, 3) = swaprate(i)
Next i


For i = 26 To 29
swaprate(i) = swaprate(25) + ((swaprate(30) - swaprate(25)) *
  ((i - dataswap(5)) / (dataswap(6)
- dataswap(5))))
  Sheets("Bootstrapping").Cells(i + 12, 3) = swaprate(i)
Next i

End Sub

Sub ReadRates_over10y()

Dim i As Integer
Dim j As Integer
Sheets("Bootstrapping").Select

For i = 1 To 6
  dataswap(i) = Sheets("Bootstrapping").Cells(i + 22, 2)
  swaprate(dataswap(i)) = Sheets("Bootstrapping").Cells(i + 22, 1)
Next i

End Sub

Sub ZC_Rates()

Rates_Load
```

```
  ' Uniform the day-count convention between spot and swap rates
    (both 30/360)

  For i = 1 To 12

    ZC(i) = Sheets("Bootstrapping").Cells(i + 1, 3) * 365 / 360

  Next i

  For j = 13 To 41

  dummy_sum = 0

    For i = 1 To j - 12
          dummy_sum = dummy_sum + (swaprate(j) / 100) /
            ((1 + (ZC(11 + i) / 100)) ^ dataswap(11 + i))
    Next i

    ZC(j) = ((((1 + (swaprate(j) / 100)) / (1 - dummy_sum)) ^
      (1 / (dataswap(11 + i)))) - 1) * 100

  Next j

  For i = 1 To 41
    Sheets("Bootstrapping").Cells(i + 1, 5) = ZC(i)
  Next i

End Sub

Sub Rates_Load()

  Dim i As Integer

  For i = 1 To 41
    dataswap(i) = Sheets("Bootstrapping").Cells(i + 1, 4)
              swaprate(i) = Sheets("Bootstrapping").Cells(i + 1, 3)
  Next i

End Sub
```

1.10 General Swap Pricing in Matlab

The Matlab Fixed-Income Toolbox[20] contains functions that perform swap pricing and portfolio hedging. The Fixed-Income Toolbox contains the function **`liborfloat2fixed`**, which computes a fixed-rate par yield that equates the floating-rate side of a swap to the fixed-rate side. The solver sets the present value of the fixed side to the present value of the floating side without having to line up and compare fixed and floating periods.

The following assumptions are used for floating-rate input:

- LIBOR rates are quarterly—for example, that of Eurodollar futures.
- Effective date is the first third Wednesday after the settlement date.
- All delivery dates are spaced three months apart.

- All periods start on the third Wednesday of delivery months.
- All periods end on the same dates of delivery months, three months after the start dates.
- Accrual basis of floating rates is actual/360.
- Applicable forward rates are estimated by interpolation in months when forward-rate data in not available.

The following assumptions are used for floating-rate output:

- Design allows you to create a bond of any coupon, basis, or frequency, based upon the floating-rate input.
- The start date is a valuation date—that is, a date when an agreement to enter into a contract by the settlement date is made.
- Settlement can be on or after the start date. If it is after, a forward fixed-rate contract results.
- Effective date is assumed to be the first third Wednesday after settlement—the same date as that of the floating rate.
- The end date of the bond is a designated number of years away, on the same day and month as the effective date.
- Coupon payments occur on anniversary dates. The frequency is determined by the period of the bond.
- Fixed rates are not interpolated. A fixed-rate bond of the same present value as that of the floating-rate payments is created.

To compute par fixed-rate of swap given three-month LIBOR data, the following function is used:

```
[FixedSpec, ForwardDates, ForwardRates] =
       liborfloat2fixed(ThreeMonthRates, Settle, Tenor, StartDate,
       Interpolation, ConvexAdj, RateParam, InArrears, Sigma,
       FixedCompound, FixedBasis)
```

The input arguments are shown in Table 1.13.

The output is as follows:

`FixedBasis` computes forward rates, dates, and the swap fixed rate.

`FixedSpec` specifies the structure of the fixed-rate side of the swap:

- `Coupon`: Par-swap rate.
- `Settle`: Start date.
- `Maturity`: End date.

Table 1.13

`ThreeMonthRates`	Three-month Eurodollar futures data or forward rate agreement data. (A forward rate agreement stipulates that a certain interest rate applies to a certain principal amount for a given future time period.) An n-by-3 matrix in the form of `[month year IMMQuote]`. The floating rate is assumed to compound quarterly and to accrue on an actual/360 basis.
`Settle`	Settlement date of swap. Scalar.
`Tenor`	Life of the swap. Scalar.
`StartDate`	(Optional) Scalar value to denote reference date for valuation of (forward) swap. This in effect allows forward swap valuation. Default = `Settle`.
`Interpolation`	(Optional) Interpolation method to determine applicable forward rate for months when no Eurodollar data is available. Default is `'linear'` or 1. Other possible values are `'Nearest'` or 0, and `'Cubic'` or 2.
`ConvexAdj`	(Optional) Default = 0 (off). 1 = on. Denotes whether futures/-forward convexity adjustment is required. Pertains to forward rate adjustments when those rates are taken from Eurodollar futures data.
`RateParam`	(Optional) Short-rate model's parameters (Hull-White) `[a S]`, where the short-rate process is: $dr = [\theta(t) - ar]dt + Sdz$ Default = `[0.05 0.015]`.
`InArrears`	(Optional) Default = 0 (off). Set to 1 for on. If on, the routine does an automatic convexity adjustment to forward rates.
`Sigma`	(Optional) Overall annual volatility of caplets.
`FixedCompound`	(Optional) Scalar value. Compounding or frequency of payment on the fixed side. Also, the reset frequency. Default = 4 (quarterly). Other values are 1, 2, and 12.
`FixedBasis`	(Optional). Scalar value. Basis of the fixed side. 0 = actual/actual, 1 = 30/360 (SIA, default), 2 = actual/360, 3 = actual/365, 4 = 30/360 (PSA), 5 = 30/360 (ISDA), 6 = 30/360 (European), 7 = act/365 (Japanese).

- `Period`: Frequency of payment.
- `Basis`: Accrual basis.

`ForwardDates` are dates corresponding to `ForwardRates` (all third Wednesdays of the month, spread three months apart). The first element is the third Wednesday immediately after `Settle`.

`ForwardRates` are forward rates corresponding to the forward dates, quarterly compounded, and on the actual/360 basis. To preserve input integrity, tenor is rounded upward to the closest integer. Currently traded tenors are 2, 5, and 10 years. The function assumes

that floating-rate observations occur quarterly on the third Wednesday of a delivery month. The first delivery month is the month of the first third Wednesday after `Settle`. Floating-side payments occur on the third-month anniversaries of observation dates.

Example 3

Use the supplied `EDdata.xls` file as input to a `liborfloat2fixed` computation.

```
[EDFutData, textdata] = xlsread('EDdata.xls');
Settle                = datenum('15-Oct-2002');
Tenor                 = 2;

[FixedSpec, ForwardDates, ForwardRates] = ...
liborfloat2fixed(EDFutData, Settle, Tenor)

FixedSpec =

      Coupon: 0.0222
      Settle: '16-Oct-2002'
    Maturity: '16-Oct-2004'
      Period: 4
       Basis: 1

ForwardDates =

      731505  (16-Oct-2002)
      731596  (15-Jan-2003)
      731687  (16-Apr-2003)
      731778  (16-Jul-2003)
      731869  (15-Oct-2003)
      731967  (21-Jan-2004)
      732058  (21-Apr-2004)
      732149  (21-Jul-2004)

ForwardRates =

    0.0177
    0.0166
    0.0170
    0.0188
    0.0214
    0.0248
    0.0279
    0.0305
```

Table 1.14 shows Eurodollar data on Friday 11, 2002 that we will used for bootstrapping the yield curve and for computing swap rates in MATLAB.

Using this data, you can compute 1-, 2-, 3-, 4-, 5-, 7-, and 10-year swap rates with the toolbox function `liborfloat2fixed`. The function requires you to input only Eurodollar data, the settlement date, and tenor of the swap. Matlab then performs the required computations.

To illustrate how this function works, first load the data contained in the supplied Excel worksheet, `EDdata.xls`.

Table 1.14

Eurodollar Data on Friday 11, 2002					
Month	**Year**	**Settle**	**Month**	**Year**	**Settle**
10	2002	98.21	6	2007	94.88
11	2002	98.26	9	2007	94.74
12	2002	98.3	12	2007	94.595
1	2003	98.3	3	2008	94.48
2	2003	98.31	6	2008	94.375
3	2003	98.275	9	2008	94.28
6	2003	98.12	12	2008	94.185
9	2003	97.87	3	2009	94.1
12	2003	97.575	6	2009	94.005
3	2004	97.26	9	2009	93.925
6	2004	96.98	12	2009	93.865
9	2004	96.745	3	2010	93.82
12	2004	96.515	6	2010	93.755
3	2005	96.33	9	2010	93.7
6	2005	96.135	12	2010	93.645
9	2005	95.955	3	2011	93.61
12	2005	95.78	6	2011	93.56
3	2006	95.63	9	2011	93.515
6	2006	95.465	12	2011	93.47
9	2006	95.315	3	2012	93.445
12	2006	95.16	6	2012	93.41
3	2007	95.025	9	2012	93.39

Source: Matlab

```
[EDRawData, textdata] = xlsread('EDdata.xls');
```

Extract the month from the first column and the year from the second column. The rate used as proxy is the arithmetic average of rates on opening and closing.

```
Month = EDRawData(:,1);
Year = EDRawData(:,2);
IMMData = (EDRawData(:,3);
EDFutData = [Month, Year, IMMData];
```

Next, input the current date.

```
Settle = datenum('11-Oct-2002');
```

To compute for the two-year swap rate, set the tenor to 2.

```
Tenor = 2;
```

Finally, compute the swap rate with `liborfloat2fixed`.

```
[FixedSpec, ForwardDates, ForwardRates] =...
     liborfloat2fixed(EDFutData, Settle, Tenor)
```

Matlab returns a par-swap rate of 2.23% using the default setting (quarterly compounding and 30/360 accrual), and forward dates and rates data (quarterly compounded), comparable to 2.17% of Friday's average broker data in Table H15 of *Federal Reserve Statistical Release* (http://www.federalreserve.gov/releases/h15/update/).

```
FixedSpec =

       Coupon: 0.0223
       Settle: '16-Oct-2002'
     Maturity: '16-Oct-2004'
        Period: 4
         Basis: 1

ForwardDates =

      731505
      731596
      731687
      731778
      731869
      731967
      732058
      732149

ForwardRates =

    0.0179
    0.0170
    0.0177
    0.0196
    0.0222
    0.0255
    0.0285
    0.0311
```

In the `FixedSpec` output, note that the swap rate actually goes forward from the third Wednesday of October 2002 (October 16, 2002), five days after the original `Settle` input (October 11, 2002). This, however, is still the best proxy for the swap rate on `Settle`, as the assumption merely starts the swap's effective period and does not affect its valuation method or its length.

The correction suggested by Hull and White improves the result by turning on convexity adjustment as part of the input to `liborfloat2fixed`. (See Hull, J., *Options, Futures, and Other Derivatives*, 4th Edition, Prentice Hall, 2000.) For a long swap—e.g., five years or more—this correction could prove to be substantial.

The adjustment requires additional parameters:

- `StartDate`, which you make the same as `Settle` (the default) by providing an empty matrix `[ ]` as input.

- `ConvexAdj` to tell `liborfloat2fixed` to perform the adjustment.

- `RateParam`, which provides the parameters `a` and `S` as input to the Hull-White short-rate process.

- Optional parameters `InArrears` and `Sigma`, for which you can use empty matrices `[]` to accept the Matlab defaults.

- `FixedCompound`, with which you can facilitate comparison with values cited in Table H15 of *Federal Reserve Statistical Release* by turning the default quarterly compounding into semiannual compounding, with the (default) basis of 30/360.

```
StartDate = [];
Interpolation = [];
ConvexAdj = 1;
RateParam = [0.03; 0.017];
FixedCompound = 2;
[FixedSpec, ForwardDates, ForwardRates] =...
liborfloat2fixed(EDFutData, Settle, Tenor, StartDate, Interpolation,
ConvexAdj, RateParam, [], [], FixedCompound)
```

This returns 2.21% as the two-year swap rate, quite close to the reported swap rate for that date. Analogously, Table 1.15 summarizes the solutions for 1-, 3-, 5-, 7-, and 10-year swap rates (convexity-adjusted and unadjusted).

Table 1.15

Calculated and Market Average Data of Swap Rates on Friday, October 11, 2002

Swap Length (Years)	Unadjusted	Adjusted	Table H15	Adjusted Error (Basis Points)
1	1.80%	1.79%	1.80%	-1
2	2.24%	2.21%	2.22%	-1
3	2.70%	2.66%	2.66%	0
4	3.12%	3.03%	3.04%	-1
5	3.50%	3.37%	3.36%	+1
7	4.16%	3.92%	3.89%	+3
10	4.87%	4.42%	4.39%	+3

Source: Matlab

To compute the duration of a LIBOR-based interest rate swap, we use the **`liborduration`** function:

```
[PayFixDuration GetFixDuration] = liborduration(SwapFixRate, Tenor,
Settle)
```

The input arguments are shown in Table 1.16.

Table 1.16

`SwapFixRate`	Scalar or column vector of swap fixed rates in decimal.
`Tenor`	Scalar or column vector indicating life of the swap in years. Fractional numbers are rounded upward.
`Settle`	Scalar or column vector of settlement dates.

The output arguments are as follows:

- `PayFixDuration` is the modified duration, in years, realized when entering pay-fix side of the swap.
- `GetFixDuration` is the modified duration, in years, realized when entering receive-fix side of the swap.

Example 4

Given the following data

```
SwapFixRate = 0.0383;
Tenor = 7;
Settle = datenum('11-Oct-2002');
```

compute the swap durations.

```
[PayFixDuration GetFixDuration] = liborduration(SwapFixRate,...
    Tenor, Settle)

PayFixDuration =

   -4.7567

GetFixDuration =

    4.7567
```

A swap can be valued in Matlab using a Black-Derman-Toy (BDT) tree or an HJM tree. The syntax is as follows:

```
[Price, PriceTree, CFTree, SwapRate] = swapbybdt(BDTTree, LegRate,
   Settle, Maturity, LegReset, Basis, Principal, LegType, Options)
```

The input arguments are shown Table 1.17.
The outputs are as follows:

- `Price` is number of instruments (`NINST`)-by-1 expected prices of the swap at time 0.
- `PriceTree` is the tree structure with a vector of the swap values at each node.

Table 1.17

`BDTTree`	Interest rate tree structure created by `bdttree`.
`LegRate`	Number of instruments (`NINST`)-by-2 matrix, with each row defined as: `[CouponRate Spread]` or `[Spread CouponRate]` `CouponRate` is the decimal annual rate. `Spread` is the number of basis points over the reference rate. The first column represents the receiving leg, while the second column represents the paying leg.
`Settle`	Settlement date. `NINST`-by-`1` vector of serial date numbers or date strings. `Settle` must be earlier than or equal to `Maturity`.
`Maturity`	Maturity date. `NINST`-by-`1` vector of dates representing the maturity date for each swap.
`LegReset`	(Optional) `NINST`-by-`2` matrix representing the reset frequency per year for each swap. Default = `[1 1]`.
`Basis`	(Optional) `NINST`-by-`1` vector representing the basis used when annualizing the input forward rate tree. Default = 0 (actual/actual).
`Principal`	(Optional) `NINST`-by-`1` vector of the notional principal amounts. Default = `100`.
`LegType`	(Optional) `NINST`-by-`2` matrix. Each row represents an instrument. Each column indicates if the corresponding leg is fixed (`1`) or floating (`0`). This matrix defines the interpretation of the values entered in `LegRate`. Default is `[1 0]`for each instrument.
`Options`	(Optional) Derivatives pricing options structure created with `derivset`.

- `CFTree` is the tree structure with a vector of the swap cash flows at each node.
- `SwapRate` is a `NINST`-by-`1` vector of rates applicable to the fixed leg such that the swaps' values are zero at time 0. This rate is used in calculating the swaps' prices when the rate specified for the fixed leg in `LegRate` is `NaN`. `SwapRate` is padded with `NaN` for those instruments in which `CouponRate` is not set to `NaN`.

Example 5

To price an interest rate swap with a fixed receiving leg and a floating paying leg, payments are made once a year, and the notional principal amount is $1,000,000. The values for the remaining parameters are as follows:

- Coupon rate for fixed leg: 0.15 (15%)
- Spread for floating leg: 10 basis points
- Swap settlement date: Jan. 01, 2000
- Swap maturity date: Jan. 01, 2003

Based on the preceding information, set the required parameters and build the `LegRate`, `LegType`, and `LegReset` matrices.

```
Settle = '01-Jan-2000';
Maturity = '01-Jan-2003';
Basis = 0;
Principal = 1000000;
LegRate = [0.15 10]; % [CouponRate Spread]
LegType = [1 0]; % [Fixed Float]
LegReset = [1 1]; % Payments once per year
```

We price the swap using the `BDTTree` included in the MAT-file `deriv.mat`. `BDTTree` contains the time and forward rate information needed to price the instrument.

```
load deriv;
```

Use `swapbybdt` to compute the price of the swap.

```
Price = swapbybdt(BDTTree, LegRate, Settle, Maturity,...
     LegReset, Basis, Principal, LegType)

Price = 73032
```

Example 6

Using the previous data, calculate the swap rate, the coupon rate for the fixed leg such that the swap price at time = 0 is zero.

```
LegRate = [NaN 20];

[Price, PriceTree, CFTree, SwapRate] = swapbybdt(BDTTree,...
    LegRate, Settle, Maturity, LegReset, Basis, Principal, LegType)

Price =

 -2.8422e-014

PriceTree =

    FinObj: 'BDTPriceTree'
     tObs: [0 1 2 3 4]
     PTree: {1x5 cell}

CFTree =

    FinObj: 'BDTCFTree'
     tObs: [0 1 2 3 4]
     CFTree: {1x5 cell}

SwapRate =
             0.1210
```

A swap can also be valued using an HJM tree.

```
[Price, PriceTree, CFTree, SwapRate] = swapbyhjm(HJMTree, LegRate,
     Settle, Maturity, LegReset, Basis, Principal, LegType, Options)
```

Table 1.18

`HJMTree`	Forward rate tree structure created by `hjmtree`.
`LegRate`	Number of instruments (`NINST`)-by-2 matrix, with each row defined as: `[CouponRate Spread]` or `[Spread CouponRate]` `CouponRate` is the decimal annual rate. `Spread` is the number of basis points over the reference rate. The first column represents the receiving leg, while the second column represents the paying leg.
`Settle`	Settlement date. `NINST`-by-1 vector of serial date numbers or date strings. `Settle` must be earlier than or equal to `Maturity`.
`Maturity`	Maturity date. `NINST`-by-1 vector of dates representing the maturity date for each swap.
`LegReset`	(Optional) `NINST`-by-2 matrix representing the reset frequency per year for each swap. Default = `[1 1]`.
`Basis`	(Optional) `NINST`-by-1 vector representing the basis used when annualizing the input forward rate tree. Default = 0 (actual/actual).
`Principal`	(Optional) `NINST`-by-1 vector of the notional principal amounts. Default = `100`.
`LegType`	(Optional) `NINST`-by-2 matrix. Each row represents an instrument. Each column indicates if the corresponding leg is fixed (`1`) or floating (`0`). This matrix defines the interpretation of the values entered in `LegRate`. Default is `[1 0]` for each instrument.
`Options`	(Optional) Derivatives pricing options structure created with `derivset`.

The arguments of the `swapbyhjm` function are shown in Table 1.18.

The `Settle` date for every swap is set to the `ValuationDate` of the HJM tree. The swap argument `Settle` is ignored.

This function also calculates the `SwapRate` (fixed rate) so that the value of the swap is initially zero. To do this, enter `CouponRate` as `NaN`.

Description

`[Price, PriceTree, CFTree, SwapRate] = swapbyhjm(HJMTree, LegRate, Settle, Maturity, LegReset, Basis, Principal, LegType)` computes the price of a swap instrument from an HJM interest rate tree.

`Price` is the number of instruments (`NINST`)-by-1 expected prices of the swap at time 0.

`PriceTree` is the tree structure with a vector of the swap values at each node.

`CFTree` is the tree structure with a vector of the swap cash flows at each node.

`SwapRate` is a `NINST`-by-1 vector of rates applicable to the fixed leg such that the swaps' values are zero at time 0. This rate is used in calculating the swaps' prices when the rate specified for the fixed leg in `LegRate` is `NaN`. `SwapRate` is padded with `NaN` for those instruments in which `CouponRate` is not set to `NaN`.

Example 7

Price an interest rate swap with a fixed receiving leg and a floating paying leg. Payments are made once a year, and the notional principal amount is $100. The values for the remaining parameters are as follows:

- Coupon rate for fixed leg: 0.06 (6%)
- Spread for floating leg: 20 basis points
- Swap settlement date: Jan. 01, 2000
- Swap maturity date: Jan. 01, 2003

Based on the preceding information, set the required parameters and build the `LegRate`, `LegType`, and `LegReset` matrices.

```
Settle = '01-Jan-2000';
Maturity = '01-Jan-2003';
Basis = 0;
Principal = 100;
LegRate = [0.06 20]; % [CouponRate Spread]
LegType = [1 0]; % [Fixed Float]
LegReset = [1 1]; % Payments once per year
```

Price the swap using the `HJMTree` included in the MAT-file `deriv.mat`. `HJMTree` contains the time and forward rate information needed to price the instrument.

```
load deriv;
```

Use `swapbyhjm` to compute the price of the swap.

```
[Price, PriceTree, CFTree] = swapbyhjm(HJMTree, LegRate,...
     Settle, Maturity, LegReset, Basis, Principal, LegType)

Price =

   3.6923

PriceTree =

    FinObj: 'HJMPriceTree'
      tObs: [0 1 2 3 4]
     PBush: {1x5 cell}

CFTree =

    FinObj: 'HJMCFTree'
      tObs: [0 1 2 3 4]
    CFBush: {[0] [1x1x2 double] [1x2x2 double] ... [1x8 double]}
```

1.11 Swap Pricing in Matlab Using Term Structure Analysis

This example illustrates some of the term structure analysis functions found in the Financial Toolbox. Specifically, it illustrates how to derive implied zero (*spot*) and forward curves from the observed market prices of coupon-bearing bonds. The zero and forward curves implied from the market data are then used to price an interest rate swap agreement. In an interest rate swap, two parties agree to a periodic exchange of cash flows. One of the cash flows is based on a fixed interest rate held constant throughout the life of the swap. The other cash flow stream is tied to some variable index rate. Pricing a swap at inception amounts to finding the fixed rate of the swap agreement. This fixed rate, appropriately scaled by the notional principle of the swap agreement, determines the periodic sequence of fixed cash flows. In general, interest rate swaps are priced from the forward curve such that the variable cash flows implied from the series of forward rates and the periodic sequence of fixed-rate cash flows have the same present value. Thus, interest rate swap pricing and term structure analysis are closely related.

Step 1. Specify values for the settlement date, maturity dates, coupon rates, and market prices for 10 U.S. Treasury bonds. This data allows us to price a five-year swap with net cash flow payments exchanged every six months. For simplicity, accept default values for the end-of-month payment rule (rule in effect) and day-count basis (actual/actual). To avoid issues of accrued interest, assume that all Treasury bonds pay semiannual coupons and that settlement occurs on a coupon payment date.

```
Settle = datenum('15-Jan-1999');

BondData = {'15-Jul-1999' 0.06000 99.93
            '15-Jan-2000' 0.06125 99.72
            '15-Jul-2000' 0.06375 99.70
            '15-Jan-2001' 0.06500 99.40
            '15-Jul-2001' 0.06875 99.73
            '15-Jan-2002' 0.07000 99.42
            '15-Jul-2002' 0.07250 99.32
            '15-Jan-2003' 0.07375 98.45
            '15-Jul-2003' 0.07500 97.71
            '15-Jan-2004' 0.08000 98.15};
```

`BondData` is an instance of a Matlab *cell array*, indicated by the curly braces ({}).

Next assign the date stored in the cell array to `Maturity`, `CouponRate`, and `Prices` vectors for further processing.

```
Maturity   = datenum(strvcat(BondData{:,1}));
CouponRate = [BondData{:,2}]';
Prices     = [BondData{:,3}]';
Period     = 2; % semiannual coupons
```

Step 2. Now that the data has been specified, use the term structure function **`zbtprice`** to bootstrap the zero curve implied from the prices of the coupon-bearing bonds. This implied zero curve represents the series of zero-coupon Treasury rates consistent with the prices of the coupon-bearing bonds such that arbitrage opportunities will not exist.

```
ZeroRates = zbtprice([Maturity CouponRate], Prices, Settle);
```

The zero curve, stored in `ZeroRates`, is quoted on a semiannual bond basis (the periodic, six-month, interest rate is simply doubled to annualize). The first element of `ZeroRates` is the annualized rate over the next six months, the second element is the annualized rate over the next 12 months, and so on.

Step 3. From the implied zero curve, find the corresponding series of implied forward rates using the term structure function **`zero2fwd`**.

```
ForwardRates = zero2fwd(ZeroRates, Maturity, Settle);
```

The forward curve, stored in `ForwardRates`, is also quoted on a semiannual bond basis. The first element of `ForwardRates` is the annualized rate applied to the interval between settlement and six months after settlement, the second element is the annualized rate applied to the interval from six months to 12 months after settlement, and so on. This implied forward curve is also consistent with the observed market prices such that arbitrage activities will be unprofitable. Because the first forward rate is also a zero rate, the first element of `ZeroRates` and `ForwardRates` are the same.

Step 4. Now that you have derived the zero curve, convert it to a sequence of discount factors with the term structure function **`zero2disc`**.

```
DiscountFactors = zero2disc(ZeroRates, Maturity, Settle);
```

Step 5. From the discount factors, compute the present value of the variable cash flows derived from the implied forward rates. For plain interest rate swaps, the notional principle remains constant for each payment date and cancels out of each side of the present value equation. The next line assumes unit notional principle.

```
PresentValue = sum((ForwardRates/Period) .* DiscountFactors);
```

Step 6. Compute the swap's price (the fixed rate) by equating the present value of the fixed cash flows with the present value of the cash flows derived from the implied forward rates. Again, because the notional principle cancels out of each side of the equation, it is simply assumed to be 1.

```
SwapFixedRate = Period * PresentValue / sum(DiscountFactors);
```

The output would be as follows:

```
 Zero Rates Forward Rates
   0.0614 0.0614
   0.0642 0.0670
   0.0660 0.0695
   0.0684 0.0758
   0.0702 0.0774
   0.0726 0.0846
   0.0754 0.0925
   0.0795 0.1077
   0.0827 0.1089
```

```
   0.0868 0.1239

Swap Price (Fixed Rate) = 0.0845
```

All rates are in decimal format. The swap price, 8.45%, would likely be the midpoint between a market-maker's bid/ask quotes.

Example 8

Consider a nine-year fixed for floating swap with a notional of $3.3 million and fixed coupon rate of 3.969%, a floating rate based on three-month T-bill, with a start date of March 28, 2004, effective date of March 29, 2004, and a maturity of March 28, 2013. Assume the fixed side is on a 30/360 day count while the floating leg is on an actual/360 day count with quarterly resets. Figure 1.4 displays the Bloomberg screen with the swap data terms.

Figures 1.5–1.7 show the swap cash flow payments and payment schedule. Figures 1.8 provides the swap curve. Figure 1.9, shows the risk measures, DV01 and duration, for both the pay and receive legs.

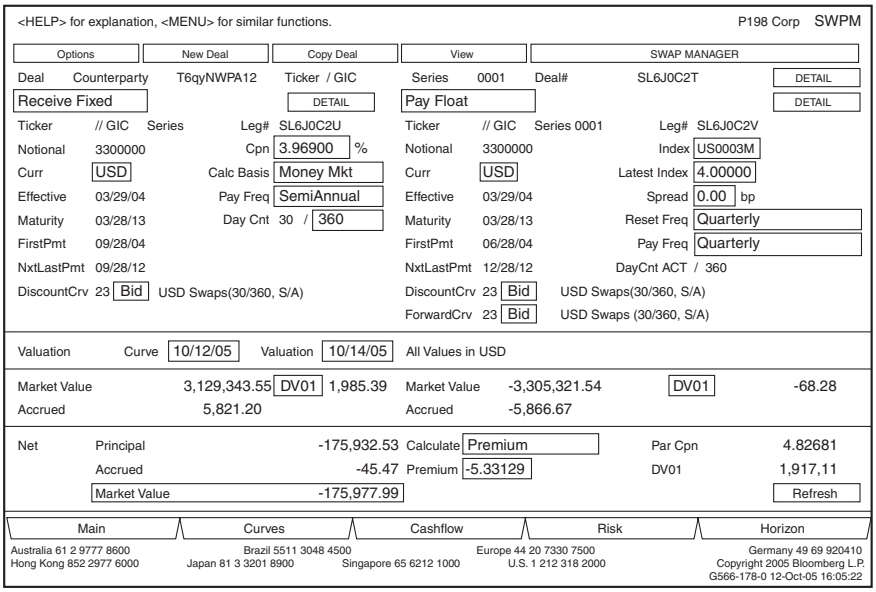
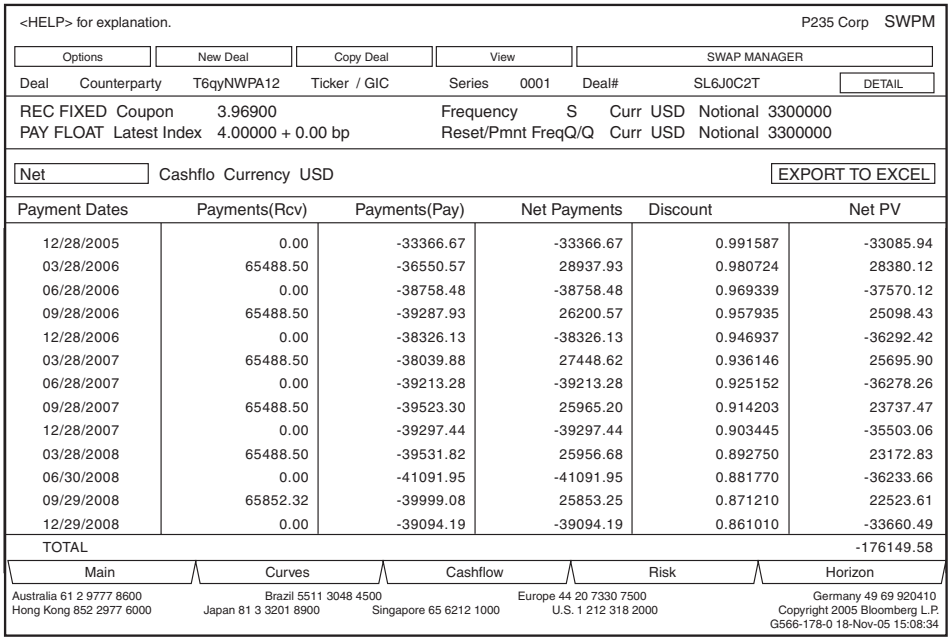


Figure 1.4



<HELP> for explanation. P235 Corp SWPM

Options | New Deal | Copy Deal | View | SWAP MANAGER

Deal Counterparty T6qyNWPA12 Ticker / GIC Series 0001 Deal# SL6J0C2T DETAIL

REC FIXED Coupon 3.96900 Frequency S Curr USD Notional 3300000
PAY FLOAT Latest Index 4.00000 + 0.00 bp Reset/Pmnt FreqQ/Q Curr USD Notional 3300000

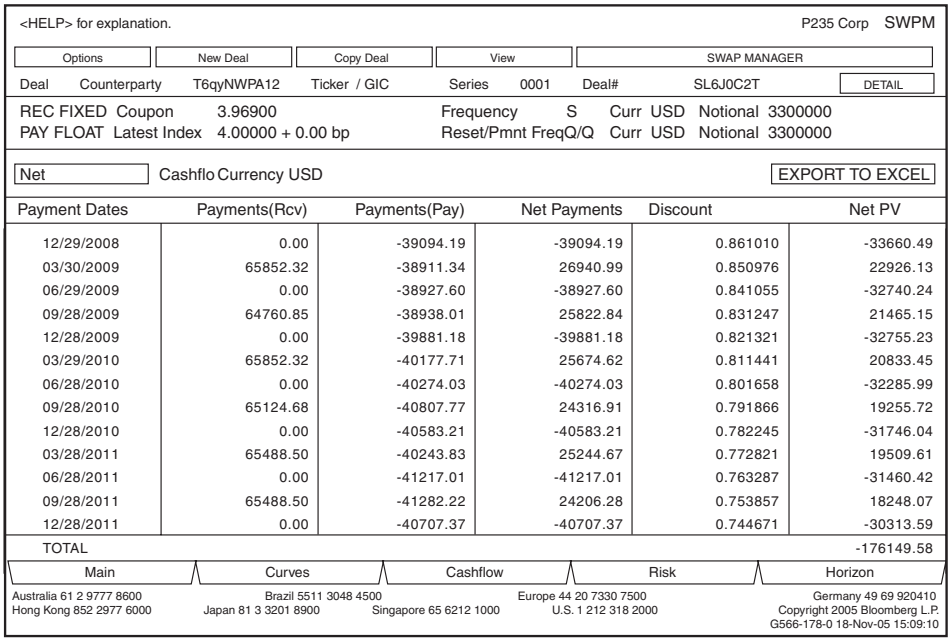
Net Cashflo Currency USD EXPORT TO EXCEL

Payment Dates	Payments(Rcv)	Payments(Pay)	Net Payments	Discount	Net PV
12/28/2005	0.00	-33366.67	-33366.67	0.991587	-33085.94
03/28/2006	65488.50	-36550.57	28937.93	0.980724	28380.12
06/28/2006	0.00	-38758.48	-38758.48	0.969339	-37570.12
09/28/2006	65488.50	-39287.93	26200.57	0.957935	25098.43
12/28/2006	0.00	-38326.13	-38326.13	0.946937	-36292.42
03/28/2007	65488.50	-38039.88	27448.62	0.936146	25695.90
06/28/2007	0.00	-39213.28	-39213.28	0.925152	-36278.26
09/28/2007	65488.50	-39523.30	25965.20	0.914203	23737.47
12/28/2007	0.00	-39297.44	-39297.44	0.903445	-35503.06
03/28/2008	65488.50	-39531.82	25956.68	0.892750	23172.83
06/30/2008	0.00	-41091.95	-41091.95	0.881770	-36233.66
09/29/2008	65852.32	-39999.08	25853.25	0.871210	22523.61
12/29/2008	0.00	-39094.19	-39094.19	0.861010	-33660.49
TOTAL					-176149.58

Main | Curves | Cashflow | Risk | Horizon

Australia 61 2 9777 8600 Brazil 5511 3048 4500 Europe 44 20 7330 7500 Germany 49 69 920410
Hong Kong 852 2977 6000 Japan 81 3 3201 8900 Singapore 65 6212 1000 U.S. 1 212 318 2000
Copyright 2005 Bloomberg L.P.
G566-178-0 18-Nov-05 15:08:34

Figure 1.5



<HELP> for explanation. P235 Corp SWPM

Options | New Deal | Copy Deal | View | SWAP MANAGER

Deal Counterparty T6qyNWPA12 Ticker / GIC Series 0001 Deal# SL6J0C2T DETAIL

REC FIXED Coupon 3.96900 Frequency S Curr USD Notional 3300000
PAY FLOAT Latest Index 4.00000 + 0.00 bp Reset/Pmnt FreqQ/Q Curr USD Notional 3300000

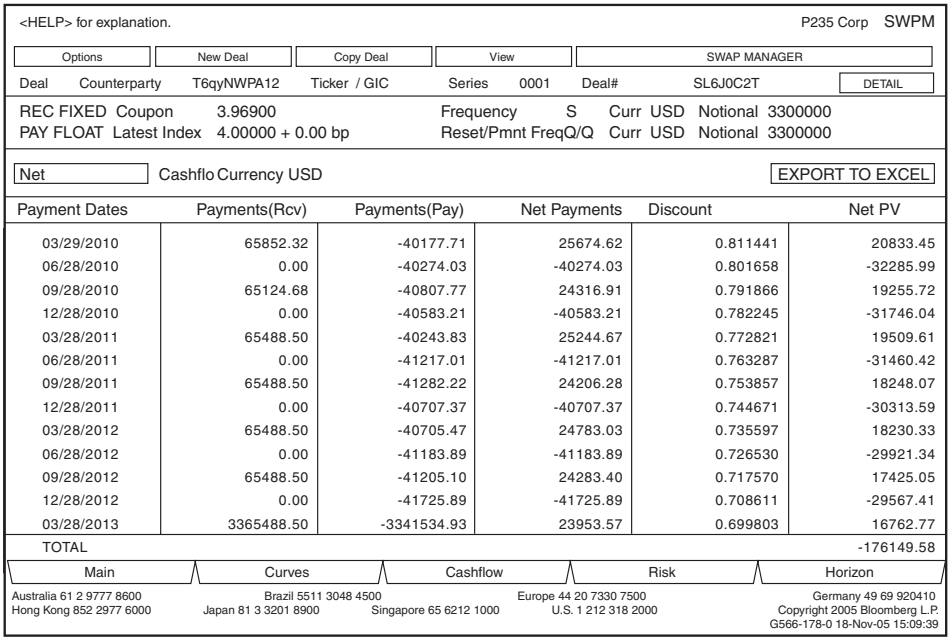
Net Cashflo Currency USD EXPORT TO EXCEL

Payment Dates	Payments(Rcv)	Payments(Pay)	Net Payments	Discount	Net PV
12/29/2008	0.00	-39094.19	-39094.19	0.861010	-33660.49
03/30/2009	65852.32	-38911.34	26940.99	0.850976	22926.13
06/29/2009	0.00	-38927.60	-38927.60	0.841055	-32740.24
09/28/2009	64760.85	-38938.01	25822.84	0.831247	21465.15
12/28/2009	0.00	-39881.18	-39881.18	0.821321	-32755.23
03/29/2010	65852.32	-40177.71	25674.62	0.811441	20833.45
06/28/2010	0.00	-40274.03	-40274.03	0.801658	-32285.99
09/28/2010	65124.68	-40807.77	24316.91	0.791866	19255.72
12/28/2010	0.00	-40583.21	-40583.21	0.782245	-31746.04
03/28/2011	65488.50	-40243.83	25244.67	0.772821	19509.61
06/28/2011	0.00	-41217.01	-41217.01	0.763287	-31460.42
09/28/2011	65488.50	-41282.22	24206.28	0.753857	18248.07
12/28/2011	0.00	-40707.37	-40707.37	0.744671	-30313.59
TOTAL					-176149.58

Main | Curves | Cashflow | Risk | Horizon

Australia 61 2 9777 8600 Brazil 5511 3048 4500 Europe 44 20 7330 7500 Germany 49 69 920410
Hong Kong 852 2977 6000 Japan 81 3 3201 8900 Singapore 65 6212 1000 U.S. 1 212 318 2000
Copyright 2005 Bloomberg L.P.
G566-178-0 18-Nov-05 15:09:10

Figure 1.6



<HELP> for explanation. P235 Corp SWPM

Options | New Deal | Copy Deal | View | SWAP MANAGER

Deal Counterparty T6qyNWPA12 Ticker / GIC Series 0001 Deal# SL6J0C2T DETAIL

REC FIXED Coupon 3.96900 Frequency S Curr USD Notional 3300000
PAY FLOAT Latest Index 4.00000 + 0.00 bp Reset/Pmnt FreqQ/Q Curr USD Notional 3300000

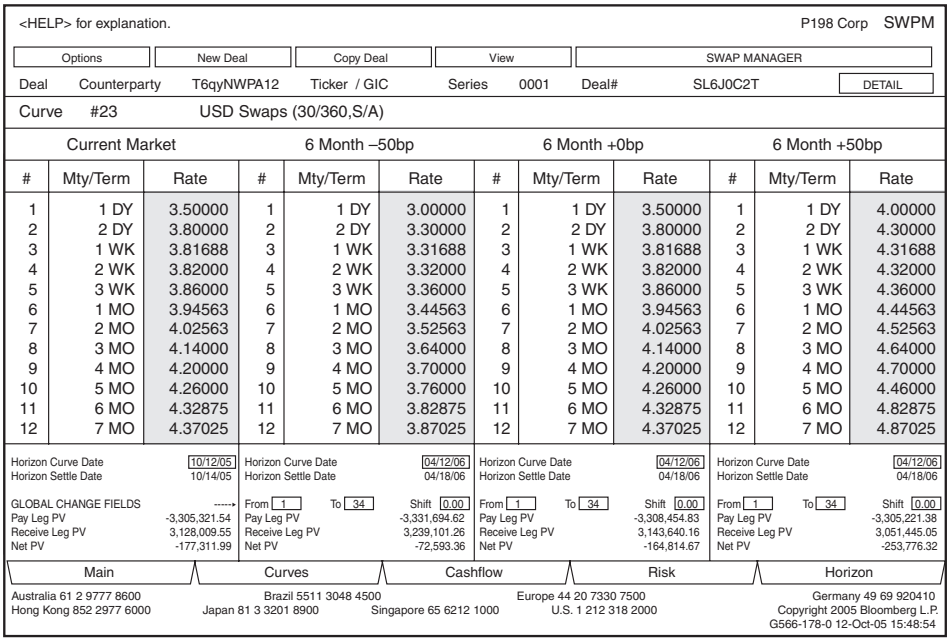
Net Cashflo Currency USD EXPORT TO EXCEL

Payment Dates	Payments(Rcv)	Payments(Pay)	Net Payments	Discount	Net PV
03/29/2010	65852.32	-40177.71	25674.62	0.811441	20833.45
06/28/2010	0.00	-40274.03	-40274.03	0.801658	-32285.99
09/28/2010	65124.68	-40807.77	24316.91	0.791866	19255.72
12/28/2010	0.00	-40583.21	-40583.21	0.782245	-31746.04
03/28/2011	65488.50	-40243.83	25244.67	0.772821	19509.61
06/28/2011	0.00	-41217.01	-41217.01	0.763287	-31460.42
09/28/2011	65488.50	-41282.22	24206.28	0.753857	18248.07
12/28/2011	0.00	-40707.37	-40707.37	0.744671	-30313.59
03/28/2012	65488.50	-40705.47	24783.03	0.735597	18230.33
06/28/2012	0.00	-41183.89	-41183.89	0.726530	-29921.34
09/28/2012	65488.50	-41205.10	24283.40	0.717570	17425.05
12/28/2012	0.00	-41725.89	-41725.89	0.708611	-29567.41
03/28/2013	3365488.50	-3341534.93	23953.57	0.699803	16762.77
TOTAL					-176149.58

Main | Curves | Cashflow | Risk | Horizon

Australia 61 2 9777 8600 Brazil 5511 3048 4500 Europe 44 20 7330 7500 Germany 49 69 920410
Hong Kong 852 2977 6000 Japan 81 3 3201 8900 Singapore 65 6212 1000 U.S. 1 212 318 2000 Copyright 2005 Bloomberg L.P.
G566-178-0 18-Nov-05 15:09:39

Figure 1.7



<HELP> for explanation. P198 Corp SWPM

Options | New Deal | Copy Deal | View | SWAP MANAGER

Deal Counterparty T6qyNWPA12 Ticker / GIC Series 0001 Deal# SL6J0C2T DETAIL

Curve #23 USD Swaps (30/360,S/A)

Current Market			6 Month –50bp			6 Month +0bp			6 Month +50bp		
#	Mty/Term	Rate	#	Mty/Term	Rate	#	Mty/Term	Rate	#	Mty/Term	Rate
1	1 DY	3.50000	1	1 DY	3.00000	1	1 DY	3.50000	1	1 DY	4.00000
2	2 DY	3.80000	2	2 DY	3.30000	2	2 DY	3.80000	2	2 DY	4.30000
3	1 WK	3.81688	3	1 WK	3.31688	3	1 WK	3.81688	3	1 WK	4.31688
4	2 WK	3.82000	4	2 WK	3.32000	4	2 WK	3.82000	4	2 WK	4.32000
5	3 WK	3.86000	5	3 WK	3.36000	5	3 WK	3.86000	5	3 WK	4.36000
6	1 MO	3.94563	6	1 MO	3.44563	6	1 MO	3.94563	6	1 MO	4.44563
7	2 MO	4.02563	7	2 MO	3.52563	7	2 MO	4.02563	7	2 MO	4.52563
8	3 MO	4.14000	8	3 MO	3.64000	8	3 MO	4.14000	8	3 MO	4.64000
9	4 MO	4.20000	9	4 MO	3.70000	9	4 MO	4.20000	9	4 MO	4.70000
10	5 MO	4.26000	10	5 MO	3.76000	10	5 MO	4.26000	10	5 MO	4.46000
11	6 MO	4.32875	11	6 MO	3.82875	11	6 MO	4.32875	11	6 MO	4.82875
12	7 MO	4.37025	12	7 MO	3.87025	12	7 MO	4.37025	12	7 MO	4.87025

Horizon Curve Date	10/12/05		Horizon Curve Date	04/12/06		Horizon Curve Date	04/12/06		Horizon Curve Date	04/12/06	
Horizon Settle Date	10/14/05		Horizon Settle Date	04/18/06		Horizon Settle Date	04/18/06		Horizon Settle Date	04/18/06	
GLOBAL CHANGE FIELDS	---->		From 1	To 34	Shift 0.00	From 1	To 34	Shift 0.00	From 1	To 34	Shift 0.00
Pay Leg PV	-3,305,321.54		Pay Leg PV	-3,331,694.62		Pay Leg PV	-3,308,454.83		Pay Leg PV	-3,305,221.38	
Receive Leg PV	3,128,009.55		Receive Leg PV	3,239,101.26		Receive Leg PV	3,143,640.16		Receive Leg PV	3,051,445.05	
Net PV	-177,311.99		Net PV	-72,593.36		Net PV	-164,814.67		Net PV	-253,776.32	

Main | Curves | Cashflow | Risk | Horizon

Australia 61 2 9777 8600 Brazil 5511 3048 4500 Europe 44 20 7330 7500 Germany 49 69 920410
Hong Kong 852 2977 6000 Japan 81 3 3201 8900 Singapore 65 6212 1000 U.S. 1 212 318 2000 Copyright 2005 Bloomberg L.P.
G566-178-0 12-Oct-05 15:48:54

Figure 1.8

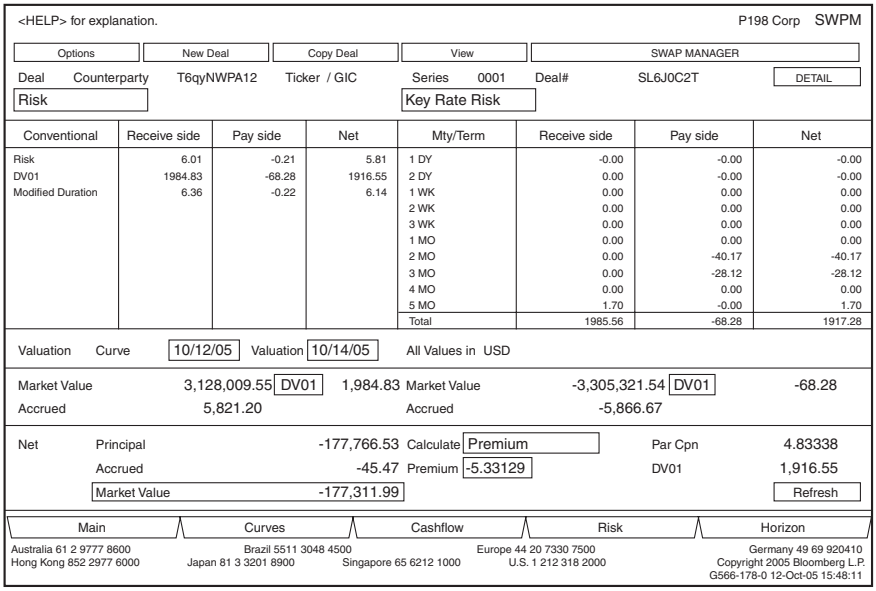


Figure 1.9

1.12 SWAP VALUATION IN C++

To price a vanilla fixed-for-floating swap in C++, we define a *Swap* class. The *Swap* class is composed of two legs—a *FloatingLeg* class, floatLeg, and a *FixedLeg* class, fixedLeg, which represent the two sides of the swap. Because the calculation of payment dates based on the start date, effective date, and maturity date is required, the *Swap* class utilizes a *Date* class[21] that contains methods for computing date operations. The *Date* class is defined in the "datecl.h" file and will be used throughout this book to compute payment dates.

The *Swap* class contains the following *Date* data members:

```
Date maturity_;              // swap maturity date
Date fixedAccruedDate_;      // fixed interest accrual date
Date floatAccruedDate_;      // floating interest accrual date
Date effectiveDate_;         // effective date
Date settlementDate_;        // settlement date
Date valuationDate_;         // valuation date
```

The *Date* methods are defined in the "datecl.cpp" source file. The *Swap* class is defined with various inline methods to add in the computation of the swap payments, netting, and pricing:

```
double getNotional()         // return notional amount
double calcDV01()            // compute swap DV01
void netPayments()           // net fixed and floating payments
void calcPayDates(Date tradeDate, Date endDate, Date valuation)
```

```
  // calculate pay dates
void setDiscountRates(std::map<double,double> rate)  // set discount rates
```

The *Swap* class utilizes the *floatLeg* and *fixedLeg* data class members to perform many of the operations in these methods. The *Swap* class contains overloaded constructors to receive and store important data for pricing: the maturity date, effective date, settlement date, valuation date, floating (libor) rates, discount rates, the fixed swap rate, and the valuation type (receive fixed-pay floating or receive floating-pay fixed). The *Swap* class definition is:

SWAP.h

```
#ifndef _SWAP_H__
#define _SWAP_H__

#include "TNT\TNT.h"
#include "datecl.h"
#include <string>
#include <vector>
#include <map>
#define NUM_DATES        100
#define THIRTY            30
#define THREE_SIXTY      360
#define THREE_SIXTY_FIVE 365
#define NOTIONAL         1000000

static std::vector<Date> payDates_;

static double interpolate(double rate1, double rate2, double t1, double t2,
  double x) {
   double dy = rate2 - rate1;
   double dt = t2 - t1;
   double slope = dy/dt;

   return rate1 + slope*x;
}

class FloatingLeg
{
 public:
    FloatingLeg(std::map<double,double> floatLegRate, double floatLegBasis,
      int payFrequency)
                : floatLegRate_(floatLegRate),floatLegBasis_(floatLegBasis),
                  payFrequency_(payFrequency) {}
    FloatingLeg() {}
    virtual ~FloatingLeg() {}
    inline double calcDuration() {
      double duration = 0.0;
      duration = (double) (payDates_[0] - valuationDate_ + 1)/
        THREE_SIXTY_FIVE; //(maturityDate_ - valuationDate_ + 1);
        //(paysum/val;
      return duration;
    }
    inline void setEffectiveDate(Date date) { startDate_ = date; }
    inline void setValuationDate(Date date) { valuationDate_ = date; }
    inline void setNotional(double notional) { notional_ = notional; }
```

```
inline Date getEffectiveDate() { return startDate_; }
inline void setFrequency(int frequency) { payFrequency_ = frequency; }
inline void setMaturityDate(Date mat) { maturityDate_ = mat; }
inline std::vector<double> getPayFloat() { return floatRates; }
inline void setFloatValue(double value) { value_ = value; }
inline double getFloatValue() {
    return value_;
}
inline void setFloatRate(std::map<double,double> rate) {
    floatLegRate_ = rate;
}
inline double calcDV01() {

    double duration = calcDuration();
    double val = getFloatValue();
    double DV = 0.0;

    DV = -(duration*notional_)*((double)1/10000);
    cout << "float DV01 = " << DV << endl;

    return -DV;
}
inline double calcModifiedDuration() {

    double val = calcDV01();
    double marketValue = getFloatValue();
    double MD = (val/(notional_ + marketValue))*10000;

    cout << "float modified duration = " << MD << endl;

    return MD;
 }
 inline void calcPayFloat() {

    std::vector<Date>::iterator iter;
    double val = 0;
    double diff = 0.0;
    int diff1 = 0.0;
    int d = 0.0;
    int d1 = 0.0;
    Date dateDiff;
    int cnt = 0;
    double floatVal = 0.0;

    for (iter = payDates_.begin(); iter != payDates_.end(); iter++)
    {
         d = payDates_[cnt+1] - payDates_[0] + 1;
         d1 = payDates_[cnt+1] - payDates_[cnt] + 1;

         diff = payDates_[cnt+1] - payDates_[0]+1;
         diff = (double) diff/THREE_SIXTY_FIVE;
         floatVal = interpolate(floatLegRate_[floor(diff)],
                     floatLegRate_[ceil(diff)],floor(diff),
                     ceil(diff),diff);

         if (payFrequency_ == 1)
            val = notional_*(THIRTY/THREE_SIXTY)*floatVal;
```

```
            else
               val = notional_*((double)d1/THREE_SIXTY_FIVE)*floatVal;

            floatRates.push_back(val);

            cnt++;
         }
      }
   private:
        double floatLegBasis_;
        std::vector<double> floatRates;
        std::map<double,double> floatLegRate_;
        std::map<double,double> payfloatLeg_;
        Date startDate_;
        Date maturityDate_;
        Date valuationDate_;
        double value_;
        double spread_;
        double notional_;
        double duration_;
        double accrual_;
        int payFrequency_;
};

class FixedLeg
{
   public:
        FixedLeg(double payfixedLeg, double fixedLegRate,
          double fixedLegBasis, int payFrequency)
                : payfixedLeg_(payfixedLeg), fixedLegRate_(fixedLegRate),
                  fixedLegBasis_(fixedLegBasis),
                payFrequency_(payFrequency) { }
        FixedLeg() {}
        virtual ~FixedLeg() {}
        inline double calcDV01() {

          double duration = calcDuration();
          double val = getFixedValue();
          double DV = 0.0;

          DV = (duration*notional_)*((double)1/10000);
          cout << "fixed DV01 = " << DV << endl;

          return DV;
        }
        inline void setEffectiveDate(Date date) { effectiveDate_ = date; }
        inline double calcDuration() {

          double sum = 0;
          double val = getFixedValue();
          double duration = 0.0;
          double dis = 0.0;

          for (int i = 0; i < payfixedLeg_.size(); i++)
             sum = sum + payfixedLeg_[i]*((double)(payDates_[i+1] -
                valuationDate_ + 1)/
                (maturityDate_ - valuationDate_ + 1));
```

```
      sum = sum + notional_;
      duration = sum/val;
      cout << "fixed duration = " << duration << endl;

      return duration;
   }
   inline double calcModifiedDuration() {

      double val = calcDV01();
      double marketValue = getFixedValue();
      double MD = (val/(notional_ + marketValue))*10000;

       //cout << "fixed modified duration = " << MD << endl;

      return MD;
   }
   inline void setFixedValue(double val) { value_ = val; }
   inline void setValuationDate(Date date) { valuationDate_ = date; }
   inline double getFixedValue() {
      return value_;
   }
   inline void setNotional(double notional) { notional_ = notional; }
   inline void setFrequency(int frequency) { payFrequency_
     = frequency; }
   inline void setFixedRate(double rate) { fixedLegRate_ = rate; }
   inline void setMaturityDate(Date mat) { maturityDate_ = mat; }
   std::vector<double> getPayFixed() { return payfixedLeg_; }
   inline void calcPayFixed()
   {
      std::vector<Date>::iterator iter;
      double val = 0;
      int diff = 0;
      Date dateDiff;
      int cnt = 0;

      for (iter = payDates_.begin(); iter != payDates_.end(); iter++)
      {
         if (payFrequency_ == 1)
         {
              if (cnt <= payDates_.size())
              {
                   if (cnt + payFrequency_ < payDates_.size())
                      diff = payDates_[cnt+payFrequency_] -
                                payDates_[cnt]+1;
                   else
                      diff = 0;

                   if ((cnt != 0) && (cnt % payFrequency_ == 0))
                       val = notional_*(THIRTY/THREE_SIXTY)*
                                fixedLegRate_;
                   else
                       val = 0;
              }
              else
                 val = notional_*(THIRTY/THREE_SIXTY)*fixedLegRate_;
         }
```

```
              else
              {
                if (cnt <= payDates_.size())
                {
                   if (cnt + payFrequency_ <= payDates_.size())
                   {
                      // subtract five because there are 5 less days in a
                      // 360 day year
                        diff = (payDates_[cnt+payFrequency_] -
                                 payDates_[cnt] + 1) - 5;
                        //cout << "diff = " << diff << endl;
                   }
                   else
                      diff = 0;

                   if ((cnt > 0) && ((cnt-1) % payFrequency_ == 0))
                      val = notional_*((double)diff/THREE_SIXTY)*
                               fixedLegRate_;
                   else
                      val = 0;
                }
                else
                {
                   //diff = (maturityDate_ - payDates_[cnt] + 1) - 5;
                   //val = notional_*((double)diff/THREE_SIXTY)*
                             fixedLegRate_;
                   val = 0;
                }
                //cout << "fixed val = " << val << "cnt = " << cnt << endl;
              }
              cnt++;
              payfixedLeg_.push_back(val);
          } // for
        }
   private:
        std::vector<double> payfixedLeg_;
        double fixedLegRate_;
        double fixedLegBasis_;
        double duration_;
        double value_;
        double accrual_;
        double notional_;
        double basis_;
        int payFrequency_;
        Date effectiveDate_;
        Date maturityDate_;
        Date valuationDate_;
};

class Swap
{
    public:
        Swap() : notional_(NOTIONAL), maturity_("12/31/2010"),
          swapType(0) {}
        Swap(double notional, Date maturity, Date effectiveDate,
          Date settlementDate, Date valuation,
        std::map<double,double> liborRate, std::map<double,double> disc,
```

```
          double fixedRate, int type)
           : notional_(notional), maturity_(maturity),
             effectiveDate_(effectiveDate), settlementDate_(settlementDate),
             valuationDate_(valuation), floatRates_(liborRate),
             fixedRate_(fixedRate), swapType(type)
        {
           getNotional();
           calcPayDates(effectiveDate_,maturity_,valuationDate_);
           fixedLeg.setNotional(notional_);
           fixedLeg.setValuationDate(valuationDate_);
           fixedLeg.setFrequency(2);
           fixedLeg.setFixedRate(fixedRate);
           fixedLeg.setMaturityDate(maturity);
           fixedLeg.calcPayFixed();
           fixedLeg.setEffectiveDate(effectiveDate_);
           floatLeg.setMaturityDate(maturity);
           floatLeg.setFrequency(4);
           floatLeg.setNotional(notional_);
           floatLeg.setFloatRate(liborRate);
           floatLeg.setValuationDate(valuationDate_);
           setDiscountRates(disc);
           floatLeg.calcPayFloat();
           netPayments();
           //fixedLeg.calcModifiedDuration();
           //floatLeg.calcModifiedDuration();
           calcDV01();                  }
           virtual ~Swap() {}
           inline double getNotional() {
                return notional_;
           }
           inline void setDiscountRates(std::map<double,double> rate) {
              discRates_ = rate;
           }
           inline double calcDV01() {

              double val = 0;

              if (swapType == 0)
                 val = fixedLeg.calcDV01() - floatLeg.calcDV01();
              else
                val = floatLeg.calcDV01() - fixedLeg.calcDV01();

              cout << "Swap DV01 = " << val << endl << endl;

              return val;
           }
           inline void calcPayDates(Date tradeDate, Date endDate,
             Date valuation)
           {
              effectiveDate_ = tradeDate-1;

              if (effectiveDate_ == Date::SATURDAY)
                 effectiveDate_ = effectiveDate_ + 2;
              else if (effectiveDate_ == Date::SUNDAY)
                 effectiveDate_ = effectiveDate_ + 1;

              Date currDate = effectiveDate_;
```

```
    int cnt = 0;

    while (currDate <= endDate)
    {
         currDate.AddMonths(3);
         while (currDate.day > effectiveDate_.day)
              currDate = currDate - 1;

         if (currDate <= valuation)
         {
              if (currDate.day_of_week == Date::SATURDAY)
                 currDate = currDate + 2;
              else if (currDate.day_of_week == Date::SUNDAY)
                 currDate = currDate + 1;
              else if (currDate ==
                currDate.ChristmasDay(currDate.year))
                 currDate = currDate + 1;

              fixedAccruedDate_ = currDate;
          }
          if ((currDate <= endDate) && (currDate >= valuation))
          {
              if (currDate.day_of_week == Date::SATURDAY)
                 currDate = currDate + 2;
              else if (currDate.day_of_week == Date::SUNDAY)
                 currDate = currDate + 1;
              else if (currDate ==
                currDate.ChristmasDay(currDate.year))
                 currDate = currDate + 1;

              payDates_.push_back(currDate);

              //cout << "payDate = " << payDates_[cnt] << endl;
              cnt++;
           }
      }
  }
  inline void netPayments()
  {
      double val = 0.0;
      double y = 0.0;
      std::vector<double> fixed = fixedLeg.getPayFixed();
      std::vector<double> fl = floatLeg.getPayFloat();
      double x = 0;
      double sum = 0.0;
      double sumfix = 0.0;
      double sumfloat = 0.0;

      if (swapType == 0)
      {
         fixedAccrued_ =  notional_*fixedRate_*((valuationDate_ -
                             fixedAccruedDate_))/THREE_SIXTY;
         floatAccrued_ = -notional_*floatRates_[0]*
                             ((valuationDate_ -
                             fixedAccruedDate_))/THREE_SIXTY_FIVE;
      }
      else
```

```
            {
               fixedAccrued_ = -notional_*fixedRate_*((valuationDate_ -
                                fixedAccruedDate_))/THREE_SIXTY;
               floatAccrued_ =  notional_*floatRates_[0]*
                                 ((valuationDate_ -
                                 fixedAccruedDate_))/THREE_SIXTY_FIVE;
            }

            int cnt = 0;
            for (int i = 0; i < payDates_.size(); i++)
            {
               x = (double) (payDates_[i+1] - payDates_[0] + 1)/
                     THREE_SIXTY_FIVE;
               //if (x > 1)
               y = interpolate(discRates_[floor(x)],discRates_[ceil(x)],
                     floor(x),ceil(x),x);
               //else
               //   y = interpolate(discRates_[floor(x)],
                           discRates_[ceil(x)-0.5],floor(x),
                           ceil(x)-0.5,x);
               //cout << "y = " << y << endl;

               if (swapType == 0)
               {
                  val = (fixed[i] - fl[i])*y;
                  sumfix = sumfix + fixed[i]*y;
                  sumfloat = sumfloat - fl[i]*y;
               }
               else
               {
                  val = (fl[i] - fixed[i])*y;
                  sumfix = sumfix - fixed[i]*y;
                  sumfloat = sumfloat + fl[i]*y;
               }
               sum = sum + val;

               //cout << "payDates = " << payDates_[i] << " " <<
                                        "net PV = " << val << endl;
            }
            fixedLeg.setFixedValue(sumfix);
            floatLeg.setFloatValue(sumfloat);

            cout << "Fixed Accrued = " << fixedAccrued_ << endl;
            cout << "Float Accrued = " << floatAccrued_ << endl;
            cout << "Accrued = " << fixedAccrued_
                                 + floatAccrued_ << endl;
            cout << "Principal = " << sum << endl;
            cout << "Market Value = " << sum + (fixedAccrued_ +
                                        floatAccrued_) << endl;
         }
      private:
        int swapType;
        double notional_;
        double fixedAccrued_;
        double floatAccrued_;
        double fixedRate_;
        FloatingLeg floatLeg;
```

```
        FixedLeg fixedLeg;
        Date maturity_;
        Date fixedAccruedDate_;
        Date floatAccruedDate_;
        Date effectiveDate_;
        Date settlementDate_;
        Date valuationDate_;
        std::string index;
        std::map<double,double> discRates_;
        std::map<double,double> floatRates_;
        double value;
};

#endif _SWAP_H__
```

Consider the preceding swap with the following characteristics:

```
Notional = $3,300,000
Maturity = March 28, 2013
Effective Date = March 29, 2004
Valuation Date = October 14, 2005
Swap Rate = 3.969%
Floating Rate = 3 Mo. T-Bill
Basis Spread = 0.00
```

We read the following (Table 1.19) in the 3-Mo T-Bill data from a file (taken from Bloomberg).

Table 1.19

Maturity	3-Mo T-Bill	Discount	Maturity	3-Mo T-Bill	Discount
1 DY	0.0400000	0.999708	9 MO	0.0445188	0.967342
2 DY 0000	0.0380000	0.999708	10 MO	0.0448525	0.963507
1 WK	0.0381688	0.999258	11 MO	0.0451725	0.959660
2 WK	0.0382000	0.999258	1 YR	0.0454563	0.955712
3 WK	0.0386000	0.998517	2 YR	0.0463900	0.912185
1 MO	0.0394583	0.996614	3 YR	0.0471200	0.869450
2 MO	0.0402583	0.993225	4 YR	0.0472200	0.829536
3 MO	0.0414000	0.989193	5 YR	0.0475900	0.790131
4 MO	0.0420000	0.985853	6 YR	0.0479100	0.752230
5 MO	0.0426000	0.982445	7 YR	0.0481800	0.715816
6 MO	0.0432875	0.978239	8 YR	0.0486000	0.679864
7 MO	0.0437025	0.974794	9 YR	0.0487000	0.647234
8 MO	0.0441225	0.971079	10 YR	0.0490200	0.614542

To discount the cash flows on the scheduled payment dates, the floating (3-Mo T-Bill) and discount rates are linearly interpolated using the following globally defined function:

```
static double interpolate(double rate1, double rate2, double t1, double
t2, double x)
{
```

```
    double dy = rate2 - rate1;
    double dt = t2 - t1;
    double slope = dy/dt;

    return rate1 + slope*x;

}
```

We interpolate using the **floor** and **ceil** built-in math routines of the rate we want to interpolate. For instance, in the `FloatLeg` function `calcPayFloat`, we call

```
floatVal = interpolate(floatLegRate_[floor(diff)],floatLegRate_[ceil(diff)],
  floor(diff),ceil(diff),diff);
```

It is important to note that this is a simple form of interpolation. In actual practice to get more accurate interpolated values, we would want to bootstrap the yield curve using liquid instruments like T-bill futures and Eurodollar futures or use a numerical technique like cubic splines.

The main function is

MAIN_ch01.cpp

```
#include <strstrea.h>
#include <fstream.h>
#include <stdlib.h>
#include <iostream.h>
#include <string.h>
#include <math.h>
#include <map>
#include "Swap.h"
#define SIZE_X 100

void main()
{
    cout.setf(ios::showpoint);
    cout.precision(8);

    cout << "Swap Pricing Pay Fixed  " << endl << endl;
    Date start = "3/29/2004";
    cout << "Start date = " << start << endl;
    Date maturity = "3/28/2013";
    cout << "Maturity = " << maturity << endl;
    Date valuation = "10/14/2005"; //today";
    cout << "Valuation = " << valuation << endl;
    Date effectiveDate = "today";
    double notional = 3300000;
    cout << "Notional = " << notional << endl;
    double swapRate = 0.03969;
    cout << "Swap Rate = " << swapRate << endl;

    std::vector<double> mat;
    std::map<double,double> libor;
    std::map<double,double> discRate;
```

```
    char buffer[SIZE_X];
    char dataBuffer[SIZE_X];
    char* str = NULL;
    double yr = 0.0;
    double rate = 0.0;
    int swapType = 1;  // receive fixed-pay float ; 1 = pay fixed-receive
                       // float


    const char* file = "c:\\swapData.txt";
    ifstream fin;              // input file stream
    fin.clear();
    fin.open(file);

    if (fin.good())
    {
        while (!fin.eof())
        {
            fin.getline(buffer,sizeof(buffer)/sizeof(buffer[0]));
            //cout << buffer << endl;
            istrstream str(buffer);
            // Get data
            str >> dataBuffer;
            yr = atof(dataBuffer);

            str >> dataBuffer;
            if (strcmp(dataBuffer,"MO") == 0)
                yr = (double) yr/12;
            else if (strcmp(dataBuffer,"WK") == 0)
                yr = (double) yr/52;
            else if (strcmp(dataBuffer,"DY") == 0)
                yr = (double) yr/365;
            mat.push_back(yr);

            str >> dataBuffer;
            rate = atof(dataBuffer);
            libor[yr] = rate;

            str >> dataBuffer;
            rate = atof(dataBuffer);
            discRate[yr] = rate;
        }
     }
     else
        cout << "File not good!" << "\n";

     fin.close();

     Swap s(notional,maturity,start,start+1,valuation,libor,discRate,
       swapRate,swapType);
}
```

We get the following results:

```
Fixed Accrued = 5821.2000
Float Accrued = -5867.3096
```

```
Accrued = -46.109589
Principal = -176178.34
Market Value = -176224.45
```

The risk measures are as follows:

```
fixed duration = 5.8713815
fixed DV01 = 1937.5559
float duration = 0.20821918
float DV01 = -68.712329
Swap DV01 = 1868.8436
```

1.13 BERMUDAN SWAPTION PRICING IN MATLAB

Interest rate swaps have the characteristics of futures contracts. As such, they are used to lock in future interest rate positions, usually for longer periods than can be obtained with exchange-traded futures. Financial managers, though, who want downside protection for their position with the potential for gains if conditions become favorable, can also take a position in *swaptions* or options on swaps. Swaptions give the holder the right, but not the obligation, to enter into a swap at maturity: for example, a fixed-rate payer's position (or a floating-rate payer's position), with the exercise price set by the fixed rate on the swap. Bermudan swaptions, however, give the holder the right to enter on discrete prespecified dates throughout the life of the swaption. The following Matlab implementation values a Bermudan swaption:

bermudan_swaption.m

```
function []=bermudan_swaption()

%This program can work for arbitrary no. of factors. You have to specify no.
%of factors as well as volatility structure for each factor. The volatility
%structure can be obtained from principal component analysis of correlation
%matrix and adjusting to calibrated volatilities as done in excellent paper
%by Rebonato. See my web page for the references
%(http://www.geocities.com/anan2999). It does not take correlation
%structure as input. You can also specify CEV constant alpha for skew.
%Remember changing this constant changes effective volatility.

%randn('state',[1541045451;4027226640]) % add a good random number seed
%here if you wish.
%if you don't matlab will choose its own seed.

delta=.25; %Tenor spacing. usually .25 or .5

P=5000;  % No. of paths, do not try more than 5000 paths unless you are
         % very patient
T_e1=6.0;               %maturity of underlying swap in years(must be an
                        %exact multiple of delta)
T_x1=5.75;              %last exercise date of the swaption (must be an
                        %exact multiple of delta)
T_s1=3.0;               %lockout date (must be an exact multiple of delta)
```

```
T_e=T_e1/delta+1;
T_x=T_x1/delta+1;
T_s=T_s1/delta+1;
N=T_e;

F=2;   % number of factors. If you change this line also change volatility
       % structure appropriately
alpha=1.0;%CEV constant alpha for skew.Remember changing this value changes
         %effective volatility
%It is 1.0 for lognormal model.
k=.1; % strike, fixed coupon
pr_flag=+1; %payer receiver flag; assumes value of +1 for a payer swaption
%and a value of -1 for a receiver swaption.


n_spot=2;
L=repmat(.10,[P,T_e+1]);
vol=repmat(0,[T_e,F]);
for n=1:N,
   for f=1:F,
      if(f==1)
         vol(n,f)=.15; %volatility of first factor
      end
      if(f==2)
         vol(n,f)= (.15-(.009*(n)*.25).^.5); %volatility of second factor
      end
   end
end
%You can add more vaolatility factors in the above line but please also
%change F accordingly
%drift=repmat(0,[P,F]);
money_market=repmat(1,[T_x,P]);
swap=repmat(0,[T_x,P]);
B=repmat(1,[P,T_e]);


money_market(2,:)=money_market(1,:).*(1+delta*L(:,1))';
increment=repmat(0,[P,1]);
drift=repmat(0,[P,F]);

for t= 2 : T_x,

   t


  normal_matrix=randn([P,F]);
  drift(:,:)=0;
   for n= t : T_e,
      increment(:,1)=0;

                %      n
      for f=1:F,

         drift(:,f)=drift(:,f)+ delta*vol(n-n_spot+1,f).*
                      ((L(:,n).^alpha)./(1+delta.*L(:,n))); %
```

```
        increment(:,1)=increment(:,1)+vol(n-n_spot+1,f).*
                        (L(:,n).^alpha)./L(:,n)...
           .*(normal_matrix(:,f).*sqrt(delta)-.5.*vol(n-n_spot+1,f).*
             (L(:,n).^alpha)./L(:,n)...
           .*delta+drift(:,f).*delta);
     end

     L(:,n)=L(:,n).*exp(increment(:,1));
     L(L(:,n)<.00001,n)=.00001;
  end

  B(:,t)=1.0;
  for n=t+1:T_e,
     B(:,n)=B(:,n-1)./(1+delta.*L(:,n-1));
  end



  money_market(t+1,:)=money_market(t,:).*(1+delta*L(:,n_spot))';

  if((t>= T_s) & (t <=T_x))
     for n=t:(T_e-1), %//the swap leg is determined one date before
                      %//the end

                swap(t,:)=swap(t,:)+  (B(:,n+1).*
                            (L(:,n)-k).*pr_flag*delta)' ;
     end
  end
  n_spot=n_spot+1;

end



value=repmat(0,[P,1]);
stop_rule=repmat(T_x,[P,1]);

value(swap(T_x,:)>0,1) = (swap(T_x,swap(T_x,:)>0))';
coeff=repmat(0,[T_x,6]);

for t=(T_x-1):-1:T_s,
   i=0;
   a=0;
   y=0;
   for p=1:P,
      if (swap(t,p)> 0.0)
         i=i+1;
         a(i,1)=1;
         a(i,2)=swap(t,p);
         a(i,3)=swap(t,p)*swap(t,p);
         a(i,4)=money_market(t,p);
         a(i,5)=money_market(t,p)*money_market(t,p);
         a(i,6)=money_market(t,p)*swap(t,p);

         y(i,1)= money_market(t,p)/money_market(stop_rule(p,1),p) *
                  value(p,1);
```

```
      end

   end



   temp=inv(a'*a)*(a'*y);
   coeff(t,:)=temp';

   expec_cont_value=repmat(0,[P,1]);
   exer_value=repmat(0,[P,1]);

   expec_cont_value(:,1)=(coeff(t,1)+coeff(t,2).*swap(t,:)+
                          coeff(t,3).*swap(t,:)...
      .*swap(t,:)+coeff(t,4).*money_market(t,:)+
        coeff(t,5).*money_market(t,:)...
      .*money_market(t,:)+coeff(t,6).*money_market(t,:).*swap(t,:))';


exer_value(swap(t,:)>0,1)=(swap(t,swap(t,:)>0))';


 value((exer_value(:,1)>expec_cont_value(:,1))&(swap(t,:)>0)',1)...
    =exer_value((exer_value(:,1)> expec_cont_value(:,1))&(swap(t,:)>0)',1);

 stop_rule((exer_value(:,1)>expec_cont_value(:,1))&(swap(t,:)>0)',1)=t;


end

   price=0;
   for p=1:P,
      price=price+ (value(p,1)/(money_market(stop_rule(p,1),p)))/P;

   end

   price
```

ENDNOTES

1 This is the case for a plain vanilla fixed-for-floating. A hedge on a bond portfolio with an amortization of the principal or a hedge on the total return on bond portfolio would require use of constant maturity swaps (or index amortization swaps) and total return swaps, which do not have swap futures equivalents (as of today).

2 A *Eurodollar* is a dollar deposited in a U.S. or foreign bank outside the United States. The Eurodollar interest rate is the rate of interest earned on Eurodollars deposited by one bank with another bank.

3 It can be shown that if an underlying asset of a long futures contract is strongly positively correlated with interest rates, futures prices will be higher than forward prices because futures are settled daily, and the gain can be invested at a higher-than-average rate of interest since the positive correlation will make it more likely rates will increase. Similarly, when the underlying asset is strongly negatively correlated with interest rates, the futures position will incur an immediate loss, and this loss will tend to be financed at a lower-than-average rate of interest. An investor with a long position in a forward contract rather than a futures contract is not affected in this way by rate movements.

4 Hull, J. (1997), *99*.

5 Id. 99.

6 Id. 100.

7 Id. 100.

8 Id. 100.

9 Hull, J. (1997), 97.

10 See *http://www.academ.xu.edu/johnson/*.

11 In actuality, the manager would only be able to purchase five contracts because one can only purchase an integer multiple of contracts. Thus, she would be underhedged by 0.115 T-bill units, but if she bought six contracts, she would be overhedged by 0.985 T-bill units.

12 We can also compute the duration using a continuously compounded yield, y_i, $1 \leq i \leq n$, as

$$D = \sum_{i=1}^{n} t_i \left[\frac{c_i e^{-y_i t_i}}{B} \right]$$

where $B = \sum_{i=1}^{n} c_i e^{-y_i t_i}$.

13 The term "yield modified duration" refers to the traditional analytic formulation for modified duration using a flat discount rate.

14 To compute DV01, shift the yield curve up by 1 basis point, recompute the duration using yield + 1bp, and then subtract the initial duration before the 1 basis point shift.

15 Reproduced with permission from "Hedging a Fixed-Income Portfolio with Swap Futures," CBOT Interest Rate Swap Complex White Paper.

16 Audley, D., Chin, R, and Ramamuthy, S., "Term Structure Modeling" in *Interest Rate, Term Structure, and Valuation Modeling*, edited by Fabozzi, F., Wiley (2002)., pg. 95.

17 Ibid., pg. 95.

18 Typically, $n = 2$ as most bonds use semiannual compounding.

19 A cubic "spline" is a piecewise polynomial function, made up of individual polynomial sections or segments that are joined together at (user-selected) points known as *knot points*. A cubic spline is a function of order three, and a piecewise cubic polynomial that is twice differentiable at each knot point. At each knot point, the slope and curvature of the curve on either side must match. An exponential cubic function would fit an exponential curve through the discount points. Thus, cubic and exponential splines are used to fit a smooth curve to bond prices (yields) given the term discount factors.

See James and Webber (2000), *Interest Rate Modeling*, Wiley, pp. 430-432; Waggoner, D. (1997); Pienaar, R. and Choudhry., M. article in Fabozzi (2002), "Fitting the Term Structure of Interest Rates Using the Cubic Spline Methodology," pg. 157-185; O. De la Grandville (2001), *Bond Pricing and Portfolio Analysis*, MIT Press, pp. 248-252; Vasicek, O. and Fong, H. (1982), "Term Structure Modeling Using Exponential Splines," *Journal of Finance* 37, 1982, pp. 339-361.

20 See MATLAB Fixed-Income Toolkit User's Guide (2002), The MathWorks.

21 The *Date* class is an open source library written by James M. Curran (1994).

CHAPTER 2

MONTE CARLO AND NUMERICAL METHODS

SECTIONS

In this chapter, we discuss Monte Carlo simulation, a technique for pricing many types of derivatives when closed-form analytical solutions are not available, as well as for pricing (complex) path-dependent derivatives and for simulating multi-factor stochastic diffusion

processes. The technique was first used by Boyle (1977). In its basic form, Monte Carlo simulation is computationally inefficient. A large number of simulations—i.e., 100,000—generally are required to achieve a high degree of pricing accuracy. However, its efficiency can be improved using control variates and quasi-random numbers (deterministic sequences).

In §2.1, we describe the general Monte Carlo framework. In §2.2, we discuss simulating sample paths and how to generate normal deviates to simulate Brownian motion. In §2.3, correlated deviates and how to generate them is discussed. In §2.4, quasi-random sequences are reviewed as an improvement over pseudo-random number generators. In §2.4, we discuss importance sampling as an efficient method to reduce variance of Monte Carlo estimates. In §2.5, a Monte Carlo using important sampling implementation in Matlab is provided. In §2.6, we discuss quasi-random sequences to reduce variance of Monte Carlo estimates. In §2.7, we review variance reduction techniques. In §2.8, a Monte Carlo variance reduction technique using antithetics in Matlab is provided. In §2.9, a Monte Carlo implementation in C++ is given. In §2.10, we discuss the Fast Fourier Transform (FFT) method to value derivatives like spread options. In §2.11, an implementation of the FFT method is given in Matlab. In §2.12, we discuss Monte Carlo simulation for valuation of path-dependent securities such as Asian options. In §2.13, a Monte Carlo implementation in Matlab to price Asian currency options is given. In §2.14 and §2.15, we discuss explicit finite difference methods where the value at any time instant can be explicitly determined from its previous values in different states (up, down, middle) at the previous time instant. In §2.16, an explicit difference method implementation in C++ is given. In §2.17, the implicit difference method is discussed, where the derivative value at any time instant is determined implicitly from its values in different states (up, down, middle) at the next time instant. In §2.18, the LU decomposition is discussed for use in solving linear systems of implicit difference equations. In §2.19, an LU decomposition to solve implicit difference equations in Matlab is provided. In §2.20, an implicit difference scheme is given in Matlab to price a European call or put. In §2.21, the Crank-Nicolson scheme, a scheme that combines both explicit and implicit scheme features, is discussed. In §2.22, an implementation for pricing an Asian option in Matlab using the Crank-Nicolson scheme is given.

2.1 THE MONTE CARLO METHOD

Suppose we wish to simulate a sample path of geometric Brownian motion process for the stock price. We divide the time interval $T-t$ into N equal time steps $\Delta t = \frac{T-t}{N}$ and simulate a path $\{S(t_i), i = 0, 1, \ldots, N\}$ starting at the known state (initial price) S_0 at time t_0. Over a time step Δt the stock price changes according to:

$$\begin{aligned} S_{i+1} &= S_i \exp\left\{\left(m - \frac{\sigma^2}{2}\right)\Delta t + \sigma\sqrt{\Delta t}\varepsilon_{i+1}\right\} \\ &= S_{i+1} = S_i \exp\left\{\mu\Delta t + \sigma\sqrt{\Delta t}\varepsilon_{i+1}\right\} \end{aligned} \tag{2.1}$$

where ε_{i+1} is a standard normal deviate and $\mu = m - q - \sigma^2/2$. Note that the term in the drift coefficient, $\sigma^2\tau/2$, came from the square of the Wiener increment, $\frac{\sigma^2}{2}\Delta t\varepsilon_{i+1}^2$. We know that the variance of this random variable is of the second order in Δt, and we can

assume that it is a deterministic quantity equal to its mean. If the stock pays a dividend, then the discretization becomes

$$S_{i+1} = S_i \exp\left\{\left(m - q - \frac{\sigma^2}{2}\right)\Delta t + \sigma\sqrt{\Delta t}\varepsilon_{i+1}\right\} \tag{2.2}$$

where q is the dividend yield.

It is important to note that we cannot use equation (2.2) directly because m is unobservable in the real world as it depends on the risk-preferences of investors. Thus, we let $m = r$, the risk-free rate, so that $\mu = r - q - \sigma^2/2$, and we are now pricing in the risk-neutral world. Equation (2.1) holds if we assume the log of the stock price follows arithmetic Brownian motion. This is an exact approximation to the continuous-time process. This approximation matches the mean and variance of the lognormal distribution exactly. Indeed,

$$\begin{aligned}\mathrm{E}[S_{i+1}|S_i] &= \mathrm{E}[S_i \exp(\mu\tau + \sigma\sqrt{\tau}\varepsilon_{i+1})] \\ &= S_i e^{\mu\Delta t}\mathrm{E}[e^{\sigma\varepsilon_{i+1}\sqrt{\Delta t}}]\end{aligned}$$

The term inside the expectation operator is the moment-generating function of the standard normal. Thus,

$$\begin{aligned}\mathrm{E}\left[S_{i+1}|S_i\right] &= S_i e^{\mu\Delta t} e^{\frac{\sigma^2\Delta t}{2}} = S_i e^{(r-\frac{\sigma^2}{2})\Delta t + \frac{\sigma^2\Delta t}{2}} \\ &= S_i e^{r\Delta t}\end{aligned} \tag{2.3}$$

which is the same mean as the lognormal distribution in continuous time. The same holds true for the variance. (The exercise is left to the reader.) The only problem with the exact solution is that there is some computational overhead because one needs to call the exponential function every time. Note that

$$\mathrm{E}\left[e^{\sigma\varepsilon_{i+1}\sqrt{\Delta t}}\right] = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\infty} e^{\sigma x\sqrt{\Delta t} - x^2/2}dx = e^{\frac{\sigma^2\Delta t}{2}}$$

In many cases, an exact solution to an SDE cannot be found, and a first or second order Euler approximation can be used. If we expand (2.1) into a Taylor series and keep only the terms of the first order in Δt, we have:

$$S_{i+1} = S_i\left(1 + r\Delta t + \sigma\sqrt{\Delta t}\varepsilon_{i+1}\right) \tag{2.4}$$

The differences between using the exact simulation in (2.1) and the first order approximation in (2.4) is $O(\Delta t^2)$.

We could also use a higher order approximation scheme for SDEs, such as a Milshtein approximation. For simplicity, assume that the drift and the diffusion coefficients depend on the state variable only and not on time—i.e., $dx_i = \mu(x_i)dt + \sigma(x_i)dz_t$. The Milshtein approximation is given by

$$\begin{aligned}x_{i+1} = x_i &+ \left[\mu(x_i) - \frac{1}{2}\sigma(x_i)\sigma'(x_i)\right]\Delta t + \sigma(x_i)\sqrt{\Delta t}\varepsilon_{i+1} \\ &+ \frac{1}{2}\sigma(x_i)\sigma'(x_i)\varepsilon_{i+1}^2\Delta t + v(x_i)\varepsilon_{i+1}(\Delta t)^{\frac{3}{2}} + \eta(x_i)(\Delta t)^2\end{aligned} \tag{2.5}$$

where $v(x_i) = \frac{1}{2}\mu(x)\sigma'(x) + \frac{1}{2}\mu'(x)\sigma(x) + \frac{1}{4}\sigma(x)^2\sigma''(x)$ and $\eta(x) = \frac{1}{2}\mu(x)\mu'(x) + \frac{1}{4}\mu''(x)\sigma(x)^2$.

2.2 Generating Sample Paths and Normal Deviates

To generate sample paths, you need to generate a sequence of standard normal deviates $\{\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_N\}$. First, it is necessary to generate uniform random numbers from 0 to 1, $\{\xi_1, \xi_2, \ldots, \xi_N\}$ and then transform them into standard normal deviates. The following graph shows a plot of some simulated asset price paths using Monte Carlo. The paths are computed by (2.1) and are driven by the random standard normal deviates.[1]

No deterministic random number generators built into computer compilers are capable of producing true random numbers. These algorithms produce pseudo-random numbers usually generated from the internal computer clock. The algorithms are based on huge deterministic sequences of numbers, although you provide a seed to tell the algorithm where in the sequence to start.

There are two serious problems that can occur with pseudo-random number generators: (1) the number of trials in simulation performed is larger than the size of the sequence or cycle of the random number generator and (2) serial correlation between the numbers exists. Thus, in practice, pseudo-random number generators are not good enough for simulations for many runs because they have cycles not long enough and/or may produce "random" numbers with serial correlation.

To solve this problem and generate uniform random numbers, we will use the random number generator, *ran1*, found in Press et al. (1992). The function uses a Box-Muller transformation to ensure that a randomly generated number, using the *ran1* function, will lie in a unit circle. *gasdev* generates a Gaussian (normal) deviate from the uniform random number generated in *ran1*. *gasdev* takes a pointer to a long data type that is the address of an arbitrary seed number.

In the early days of simulation, one approach to generating a standard normal, $N(0,1)$, deviate was to use the central limit theorem. Note that

$$\frac{\sum_{i=1}^{n} U_i - (n/2)}{\sqrt{n/12}} \to N(0,1) \text{ as } n \to \infty,$$

provided $U_1, \ldots, U_n$ are independent uniform (0,1) random variables. Setting $n = 12$ yields

$$\sum_{i=1}^{n} U_i - 6 \xrightarrow{\text{D}} \text{N}(0,1)$$

This convolution method is quite fast, but is not exact.

The most commonly used exact algorithms for generating normal random variables is to generate them in pairs. The reason is that the bivariate normal density for two independent normal random variables having mean zero and unit variance

$$N(a,b) = \int_{-\infty}^{a}\int_{-\infty}^{b} \frac{1}{2\pi}\exp\left(-\frac{1}{2}\left(x^2+y^2\right)\right)dxdy$$

has a particularly nice structure in polar coordinates. Specifically, suppose $(N1, N2)$ is such a pair of normal random variables. Then, $(N1, N2)$ can be expressed in polar coordinates as

$$(N_1, N_2) = (R\cos\theta, R\sin\theta)$$

where $\theta(0 \le \theta \le 2\pi)$ is the angular component (in radians) and R is the radial component. Due to the spherical symmetry of such a bivariate normal density, θ is normally distributed on $[0, 2\pi]$ and independent of R. Furthermore,

$$R = \sqrt{N_1^2 + N_2^2} = \sqrt{\chi^2(2)}$$

where $\chi^2(2)$ is a chi-square random variable with two degrees of freedom. Because a $\chi^2(2)$ random variable has the same distribution as $2X$, where X is exponential with parameter 1—i.e., $X \sim e^{-(x-1)}$—we can utilize the following algorithm, known as the Box-Muller algorithm:

1. Generate two independent uniform (0,1) random variates, U_1 and U_2.
2. Set $N_1 = \sqrt{-2\log U_1}\cos(2\pi U_2)$ and $N_2 = \sqrt{-2\log U_2}\sin(2\pi U_1)$.

This can be a bit slow, because of the cosine and sine calculations that need to be performed. A variant (that is typically fast) is the polar rejection (transformation) method. This method also involves an acceptance-rejection procedure. Generate two independent uniform (0,1) random variates U_1 and U_2:

1. Set $V_1 = 2U_1 - 1$ and $V_2 = 2U_2 - 1$.
2. Compute $W = V_1^2 + V_2^2$.
3. If $W > 1$, return to step 1. Otherwise, set:

$$N_1 = \sqrt{\frac{(-2\log W)}{W}}V_1 \text{ and } N_2 = \sqrt{\frac{(-2\log W)}{W}}V_2.$$

These algorithms generate pairs of independent N(0,1) random variates. To generate $N(\mu, \sigma^2)$ random variates, use the following relationship:

$$N\left(\mu, \sigma^2\right) \sim \mu + \sigma N(0,1)$$

2.3 Generating Correlated Normal Random Variables

In many Monte Carlo simulations, especially in simulations of multivariate (multifactor) diffusion processes and multidimensional stochastic simulations—i.e., spread option

models—correlation between the variates exists and must be considered because the underlying factors themselves are correlated. For example, in a stochastic volatility model, the underlying asset and its stochastic volatility are correlated, and this correlation must be captured in the correlation between variates driving the diffusion process of the underlying asset and the diffusion process of its volatility. In general, any model with multivariate normal random variables has a correlation/covariance matrix that exists. Such correlation/-covariance matrix can be used to generate the joint probability distributions between the random variables.

Suppose that we want to generate a random variable X that is multivariate normal with mean vector $\boldsymbol{\mu}$ and covariance $\boldsymbol{\Sigma}$. Suppose, furthermore, that X is a two-dimensional vector, with mean vector and covariance matrix:

$$\boldsymbol{\mu} = \begin{pmatrix} \mu_1 \\ \mu_2 \end{pmatrix} \text{ and } \boldsymbol{\Sigma} = \begin{pmatrix} \sigma_{11} & \sigma_{12} \\ \sigma_{21} & \sigma_{22} \end{pmatrix} = \begin{pmatrix} \sigma_1^2 & \rho\sigma_1\sigma_2 \\ \rho\sigma_1\sigma_2 & \sigma_2^2 \end{pmatrix} \tag{2.6}$$

Here, $\mu_i = \mathrm{E}(\mathrm{X}_i), \sigma_i^2 = \mathrm{Var}(X_i)$, and $\widetilde{\rho}$ is the correlation between X_1 and X_2. We will now describe a means of generating X_1 and X_2 from a pair of independent N(0,1) random variables N_1 and N_2. Note that we may express X_1 in terms of N_1, as follows:

$$X_1 = \mu_1 + \sigma_1 N_1$$

For X_2, we will try to write it in the following form:

$$X_2 = \mu_2 + aN_1 + bN_2$$

Recall that because N_1 is independent of N_2,

$$\mathit{Var}(X_2) = \mathrm{E}(X_2 - \mu_2)^2] = a^2 + b^2 = \sigma_2^2.$$

Also,

$$\mathit{Cov}(X_1, X_2) = \mathrm{E}[(X_1 - \mu_1)(X_2 - \mu_2)] = a\sigma_1 = \rho\sigma_1\sigma_2.$$

Solving these two equations, we get

$$a = \rho\sigma_2 \text{ and } b = \sqrt{(1-\rho^2)}\sigma_2.$$

In other words,

$$\begin{pmatrix} X_1 \\ X_2 \end{pmatrix} = \begin{pmatrix} \mu_1 \\ \mu_2 \end{pmatrix} + \begin{pmatrix} \sigma_1 & 0 \\ \rho\sigma_2 & \sqrt{1-\rho^2}\sigma_2 \end{pmatrix} \begin{pmatrix} N_1 \\ N_2 \end{pmatrix} \tag{2.7}$$

or in matrix notation, $\mathbf{X} = \boldsymbol{\mu} + \mathbf{LN}$ where $\mathbf{L}$ is lower triangular. Thus, by generating a pair of independent N(0,1) random variables, we can obtain $\mathbf{X}$ via the preceding affine transformation. The previous methodology works in general (for $\mathbf{X}$ having more than two components). In general, $\mathbf{X}$ can be written in the form

$$\mathbf{X} = \boldsymbol{\mu} + \mathbf{LN} \tag{2.8}$$

where $\mathbf{N}$ has the same number of components as does $\mathbf{X}$—i.e., same vector size—and consists of N(0,1) random variables. To connect the matrix $\mathbf{L}$ to $\boldsymbol{\Sigma}$ observe that

$$\boldsymbol{\Sigma} = \mathrm{E}[(\mathbf{X} - \boldsymbol{\mu})(\mathbf{X} - \boldsymbol{\mu})'] = \mathrm{E}[(\mathbf{LN})(\mathbf{LN})'] = \mathrm{E}[(\mathbf{LN})(\mathbf{N}'\mathbf{L}')].$$

Because $\mathbf{N}$ consists of $N(0,1)$ random variables, $\mathrm{E}[\mathbf{NN}'] = \mathbf{I}$, the identity matrix,[2] and we can write $\boldsymbol{\Sigma}$ as follows:

$$\boldsymbol{\Sigma} = \mathrm{E}[(\mathbf{LL}')] \tag{2.9}$$

Let $\mathbf{L} = \boldsymbol{\Sigma}^{1/2}$ so that $\mathbf{L}$ is a "square root" of $\boldsymbol{\Sigma}$. Furthermore, because $\boldsymbol{\Sigma}$ is symmetric and positive semi-definite, $\mathbf{L}$ can always be chosen to be a lower triangular matrix with real entries. Writing

$$\boldsymbol{\Sigma} = \mathbf{LL}' \tag{2.10}$$

is called the Cholesky factorization[3] of $\boldsymbol{\Sigma}$. Clearly, the key to generating $\mathbf{X}$ is the computation of the Cholesky factor $\mathbf{L}$. Thus, to produce correlated variables from uncorrelated (independent) ones, we need to find an $\mathbf{L}$ that solves the matrix equation (2.10).

We can use a Cholesky decomposition for $n = 3$ deviates. Suppose z_1, z_2, and z_3 are random samples from three independent normal distributions with the following correlation structure:

$$\widetilde{\boldsymbol{\rho}} = \begin{pmatrix} 1 & \rho_{12} & \rho_{13} \\ \rho_{21} & 1 & \rho_{23} \\ \rho_{31} & \rho_{32} & 1 \end{pmatrix}$$

where $\rho_y = \rho_{ji}$ because $\widetilde{\boldsymbol{\rho}}$ is symmetric. Random deviates with this correlation structure are

$$\begin{aligned} x_1 &= z_1 \\ x_1 &= \rho_{12} z_1 + \sqrt{1 - \rho_{12}^2}\, z_2 \\ x_3 &= \alpha_1 z_1 + \alpha_2 z_2 + \alpha_3 z_3 \end{aligned} \tag{2.11}$$

where

$$\begin{aligned} \alpha_1 &= \rho_{13} \\ \alpha_2 &= \frac{\rho_{23} - \rho_{12}\rho_{13}}{\sqrt{1 - \rho_{12}^2}} \\ \alpha_3 &= \sqrt{1 - \alpha_1^2 + \alpha_2^2} \end{aligned}$$

so that

$$\mathbf{L} = \begin{pmatrix} 1 & 0 & 0 \\ \rho_{12} & \sqrt{1 - \rho_{12}^2} & 0 \\ \rho_{13} & \frac{\rho_{23} - \rho_{12}\rho_{13}}{\sqrt{1 - \rho_{12}^2}} & \sqrt{1 - \rho_{13}^2 - \frac{(\rho_{23} - \rho_{12}\rho_{13})^2}{1 - \rho_{12}^2}} \end{pmatrix} \tag{2.12}$$

The following code generates a Cholesky decomposition in C++:

```
#include <vector>
#include "tnt.h"
#include "jama_cholesky.h"
#include "jama_eig.h"
using namespace TNT;
using namespace JAMA;
using namespace std;
```

MatrixUtil_genCholesky1

```
inline vector<double> genCholesky4(TNT::Array2D<double> R)
{
        int m = R.dim1();
        std::vector<double> normaldev;
        std::vector<double> defaultTime;
        TNT::Array2D<double> lb(m,m);  // lower-banded (lb) matrix
        TNT::Array1D<double> dw(m);
        TNT::Array1D<double> z(m);
        double deviate = 0.0;
        double sum = 0.0;
        double dt = 0.25;
        double normdeviate = 0.0;
        int i, j;

        JAMA::Cholesky<double> ch(R);
        lb = ch.getL();

        // generate uncorrelated deviates
        for (i = 0; i < m; i++)
        {
                normdeviate = mrng.genrand();
            deviate = normsinv(normdeviate);
                dw[i] = deviate;
        }

        // generate correlated deviates
        for (i = 0; i < m; i++)
        {
                sum = 0;
                for (j = 0; j < m; j++)
                {
                        sum = sum + lb[i][j]*dw[j];
                }
                z[i] = sum;
                normaldev.push_back(normalCalc(z[i]));
        }

        return normaldev;
}
```

An alternative approach to generating n correlated deviates, z_i, $i = 1, \ldots, n$, that are jointly normally distributed with mean zero and variance one with infinitesimal increments dz_i, is to use principal component analysis[4] to write their correlation/covariance matrix, Σ, as

$$\Sigma = \Gamma \Lambda \Gamma'$$

where Γ is the matrix of n eigenvectors $\mathbf{v}_i$'s, $i = 1, \ldots, n$, and Λ is the matrix of n associated eigenvectors λ_i's, Γ:

$$\mathbf{\Gamma} = \begin{bmatrix} v_{11} & v_{12} & \ldots & v_{1n} \\ v_{21} & v_{22} & \ldots & v_{2n} \\ \ldots & \ldots & \ldots & \ldots \\ v_{n1} & v_{n2} & \ldots & v_{nn} \end{bmatrix} \text{ and } \mathbf{\Lambda} = \begin{bmatrix} \lambda_1 & 0 & 0 & 0 \\ 0 & \lambda_2 & 0 & 0 \\ 0 & 0 & \ldots & 0 \\ 0 & 0 & 0 & \lambda_n \end{bmatrix}$$

This is the eigensystem representation of the covariance matrix of the correlated variables. Because the eigenvectors are linear combinations of the correlated variables that give independent variables, we can invert this relationship to obtain the linear combinations of independent variables, which reproduce the original covariance matrix. Because the transpose of Γ is equal to its inverse (the eigenvectors of Γ are orthogonal to each other), the rows of Γ represent the proportions of a set of n independent Brownian motions dw_i, $i = 1, \ldots, n$, which when linearly combined, reproduce the original correlated Brownian motions. The eigenvalues represent the variances of the independent Brownian motions. Thus, we can reproduce the correlated Brownian motions dz_i from the linear combination of the independent Brownian motions dw_i, as follows:

$$\begin{aligned} dz_1 &= v_{11}\sqrt{\lambda_1}dw_1 + v_{12}\sqrt{\lambda_2}dw_2 + \ldots + v_{1n}\sqrt{\lambda_n}dw_n \\ dz_2 &= v_{21}\sqrt{\lambda_2}dw_1 + v_{22}\sqrt{\lambda_2}dw_2 + \ldots + v_{2n}\sqrt{\lambda_n}dw_n \\ &\ldots \\ dz_n &= v_{n1}\sqrt{\lambda_2}dw_1 + v_{n2}\sqrt{\lambda_2}dw_2 + \ldots + v_{nn}\sqrt{\lambda_n}dw_n \end{aligned}$$

This method is used extensively when pricing multivariate diffusion processes, such as a (stochastic volatility) spread option (see §2.13) where correlated deviates must be generated.

The following is the code that generates the previous eigenvalue-eigenvector decomposition for MAX_SIZE-defined correlated deviates:

MatrixUtil_genEigenValue.h

```
inline TNT::Array1D<double> genEigenValue(TNT::Array2D<double>& R)
{
        int m = R.dim1();

        TNT::Array2D<double> lb(m,m);  // lower-banded (lb) matrix
        double deviate = 0.0;
        double normdeviate = 0.0;
        unsigned long seed = 0.0;
```

```
        double sum = 0.0;
        double dt = 0.25;
        TNT::Array1D<double> dw(m);
        TNT::Array1D<double> z(m);
        TNT::Array1D<double> D(m);
        TNT::Array2D<double> V(m,m);
        vector<double> dz;
        vector<double> eigenValue;
        vector<double> eigenVector[MAX_SIZE];
        vector<double>::iterator eigenVecIter;

        int i, j;
        JAMA::Eigenvalue<double> eig(R);
        // get eignvalues
        D = eig.getRealEigenvalues();
        // get eigenvectors
        V = eig.getV();
        // store eigenvalues
        for (i = 0; i < m; i++)
                eigenValue.push_back(D[i]);

        // stores rows of eigenvectors so that we can compute
        // dz[i] = v[i][1]*sqrt(eigenvalue[1])*dw1+ v[i][2]*
        //   sqrt(eigenvalue[2])*dw2 + ...
        for (i = 0; i < m; i++)
                for (j = 0; j < m; j++)
                        eigenVector[i].push_back(V[i][j]);

        // generate uncorrelated deviates
        for (i = 0; i < m; i++)
        {
                normdeviate = mrng.genrand();
            deviate = normsinv(normdeviate);
                dw[i] = deviate; //sqrt(dt);
        }
        // generate correlated deviates
        for (i = 0; i < m; i++)
        {
                sum = 0;
                eigenVecIter = eigenVector[i].begin();
                for (j = 0; j < m; j++)
                {
                        sum += (*eigenVecIter)*sqrt(eigenValue[j])*dw[j];
                           eigenVecIter++;
                }
                z[i] = sum;
        }
        return z;
}
```

The code makes use of the Template Numerical Toolkit (TNT) matrix library.[5] The matrix library contains many matrix manipulation and computational routines, such as for the computation of the eigenvectors and eigenvalues from a given (symmetric) matrix like the covariance/correlation matrix Σ. However, such a covariance/correlation matrix that is passed into the method `genEigenValue` needs to be known *a priori*. One can make

assumptions about what these will be or try to estimate them from historical data. For example, if one wants to estimate the correlation between the deviates of a stock and its volatility, one could use the estimated historical correlation. However, because correlation estimates are time-varying and unstable, we must use caution when inputting a specified correlation matrix at different times.

It is also important to note that we do not use the computer's built-in random number generator `rand()` for generating standard normal deviates because research has shown that a long series of (random) numbers generated from the computer's built-in clock contains serial correlation. In other words, the numbers are not truly random—they are pseudo-random. To resolve this problem, we use a Mersenne Twister random number generator,[6] which can generate a much longer series of random numbers with no serial correlation. This is particularly important in pricing credit baskets and CDO tranches because generating a long series of random default times is required over possibly millions of simulations (see Chapter 6, "Credit Derivatives"). The following code is a Mersenne Twister random number class in C++.

Random.h

```
#include "Random.h"

CRandomMT::CRandomMT() : left(-1)
{
        SeedMT(DEFAULT_SEED);
}

CRandomMT::CRandomMT(ULONG _seed) : left(-1), seedValue(_seed)
{
        SeedMT(seedValue);
}

void CRandomMT::SeedMT(ULONG seed)
 {
    //
    // We initialize state[0..(N-1)] via the generator
    //
    //   x_new = (69069 * x_old) mod 2^32
    //
    // from Line 15 of Table 1, p. 106, Sec. 3.3.4 of Knuth's
    // _The Art of Computer Programming_, Volume 2, 3rd ed.
    //
    // Notes (SJC): I do not know what the initial state requirements of
    // the Mersenne Twister are, but it seems this seeding generator could
    // be better.  It achieves the maximum period for its modulus (2^30)
    // if x_initial is odd (p. 20-21, Sec. 3.2.1.2, Knuth); if x_initial
    // can be even, you have sequences like 0, 0, 0, ...; 2^31, 2^31,
    // 2^31, ...; 2^30, 2^30, 2^30, ...; 2^29, 2^29 + 2^31, 2^29, 2^29 +
    // 2^31, ..., etc. so I force seed to be odd below.
    //
    // Even if x_initial is odd, if x_initial is 1 mod 4 then
    //
    //   the          lowest bit of x is always 1,
    //   the  next-to-lowest bit of x is always 0,
    //   the 2nd-from-lowest bit of x alternates
```

```
    //     ... 0 1 0 1 0 1 0 1 ... ,
    //   the 3rd-from-lowest bit of x 4-cycles
    //     ... 0 1 1 0 0 1 1 0 ... ,
    //   the 4th-from-lowest bit of x has the 8-cycle
    //     ... 0 0 0 1 1 1 1 0 ... ,
    //     ...
    //
    // and if x_initial is 3 mod 4 then
    //
    //   the          lowest bit of x is always 1,
    //   the  next-to-lowest bit of x is always 1,
    //   the 2nd-from-lowest bit of x alternates
    //     ... 0 1 0 1 0 1 0 1 ... ,
    //   the 3rd-from-lowest bit of x 4-cycles
    //     ... 0 0 1 1 0 0 1 1 ... ,
    //   the 4th-from-lowest bit of x has the 8-cycle
    //     ... 0 0 1 1 1 1 0 0 ... ,
    //    ...
    //
    // The generator's potency (min. s>=0 with (69069-1)^s = 0 mod 2^32) is
    // 16, which seems to be alright by p. 25, Sec. 3.2.1.3 of Knuth.  It
    // also does well in the dimension 2..5 spectral tests, but it could be
    // better in dimension 6 (Line 15, Table 1, p. 106, Sec. 3.3.4, Knuth).
    //
    // Note that the random number user does not see the values generated
    // here directly since reloadMT() will always munge them first, so
    // maybe none of all of this matters.  In fact, the seed values made
    // here could even be extra-special desirable if the Mersenne Twister
    // theory says so-- that's why the only change I made is to restrict to
    // odd seeds.
    //

    register ULONG x = (seed | 1U) & 0xFFFFFFFFU, *s = state;
    register int    j;

    for(left=0, *s++=x, j=N; --j;
        *s++ = (x*=69069U) & 0xFFFFFFFFU);
        seedValue = seed;       // Save the seed value used - DHL
 }


ULONG CRandomMT::ReloadMT(void)
 {
    register ULONG *p0=state, *p2=state+2, *pM=state+M, s0, s1;
    register int    j;

    if(left < -1)
        SeedMT(seedValue);      // Changed to make use of our
                                // seed value - DHL

    left=N-1, next=state+1;

    for(s0=state[0], s1=state[1], j=N-M+1; --j; s0=s1, s1=*p2++)
        *p0++ = *pM++ ^ (mixBits(s0, s1) >> 1) ^ (loBit(s1) ? K : 0U);

    for(pM=state, j=M; --j; s0=s1, s1=*p2++)
        *p0++ = *pM++ ^ (mixBits(s0, s1) >> 1) ^ (loBit(s1) ? K : 0U);
```

```
    s1=state[0], *p0 = *pM ^ (mixBits(s0, s1) >> 1) ^ (loBit(s1) ? K : 0U);
    s1 ^= (s1 >> 11);
    s1 ^= (s1 <<  7) & 0x9D2C5680U;
    s1 ^= (s1 << 15) & 0xEFC60000U;
    return(s1 ^ (s1 >> 18));
}
```

It is also important to note that before the random number generator routine is called `genrand()`, `sgenrand(seed)` must be called once (for instance, in the class constructor) where seed is any 32-bit integer except for 0.

2.4 IMPORTANCE SAMPLING

Although variance reduction based on importance sampling has not been as widely used as other variance reduction techniques, it is a very efficient technique. The idea behind importance sampling is to simulate more sample paths on the area that matters.[7] For instance, for a deep out-of-the-money call option, most of the time the payoff from the simulation is 0; simulating more sample paths with positive payoffs should reduce the variance in the estimation.[8] Mathematically speaking, "the fundamental idea behind importance sampling is that under certain regularity conditions, expectation under one probability measure can be expressed an expectation under another probability measure through the Radon-Nikodym theorem."[9] The appropriate choice of the probability measure will effectively reduce the variance associated with estimation. A change in probability measure will change the drift of the simulated process.

To illustrate the importance sampling framework, consider a European call option $C(T;\omega)$ that expires at time T on an underlying asset $S(T;\omega)$ valued on a sample path ω with a payoff

$$C(T;\omega) = \mathit{Max}(S(T;\omega) - X, 0)$$

Under the risk-neutral (martingale) measure Q, an estimate of the present value of the option is

$$\widehat{C}(T;\omega) = e^{-\int_0^T r(t;\omega)dt} C(T;\omega)$$

where the price at time 0 is

$$C_0 = E^Q \left[e^{-\int_0^T r(t;\omega)dt} C(T;\omega) \right]$$

A direct estimate for C_0 is obtained by simulating the risk-neutral distribution of the underlying asset(s) and taking the sample mean over replications of $\widehat{C}(T;\omega)$. However, by the Radon-Nikodym theorem, if measure Q is absolutely continuous with respect to some other measure P, then

$$C_0 = E^P \left[\widehat{C}(T;\omega) \frac{dQ}{dP} \right]$$

which gives an alternative estimator for simulation under P

$$\widehat{C}(T;\omega)\frac{dQ}{dP}$$

where $\frac{dQ}{dP}$ is the Jacobian of the measure change—i.e., the Radon-Nikodym derivative. It can be shown that the estimator is asymptotically an unbiased estimator of the option price. However, the new estimator may have different estimation variance, thus the potential for variance reduction.

The variance of the new estimator is

$$E^P\left[\left(\widehat{C}(T;\omega)\frac{dQ}{dP}\right)^2\right] - C_0^2 = E^P\left[\left(\widehat{C}(T;\omega)\right)^2\left(\frac{dQ}{dP}\right)^2\right] - C_0^2$$

To minimize the variance of the estimator, one needs to solve a stochastic minimization optimization

$$\min_{\theta\in\Theta} V(\theta)$$

where

$$V(\theta) = E^P\left[\left(\widehat{C}(T;\omega)\frac{dQ}{dP}\right)^2\right] - C_0^2$$

and the new measure is in the family of probability measures $\{P(\theta;\omega) : \theta \in \Theta\}$, where θ is the parameter and for any $\theta \in \Theta$, measure Q is absolutely continuous with respect to $P(\theta;\omega)$.

Define an infinitesimal perturbation analysis (IPA) estimator as

$$\widehat{C}^2(T,\omega)\frac{\partial f(\theta,\omega)}{\partial\theta} \tag{2.13}$$

where

$$f(\theta,\omega) = \frac{dQ}{dP(\theta)}$$

is the change of measure function.

To use the IPA, we need to change the drift in the Brownian motion to compute $\frac{\partial f(\theta,\omega)}{\partial\theta}$. Suppose the underlying asset price under the risk-neutral measure Q is an Ito process defined by the following stochastic differential equation:

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

where $\widetilde{z}_t$ is a standard Brownian motion under Q. Define the family of $P(\theta)$ as the equivalent probability measures with respect to Q introduced by changing the drift term of $\widetilde{z}_t$ by θ. Then, by Girsanov's theorem, we know under $P(\theta)$

$$dS_t = \left(\mu(S_t,t) + \theta\sigma(S_t,t)\right)dt + \sigma(S_t,t)dz_t$$

where z_t is a Brownian motion under P, and

$$z_t = \widetilde{z} - \theta t$$

The change of measure process is given by

$$\begin{aligned}\frac{dQ}{dP} &= \exp\left(-\theta z_t - \frac{1}{2}\theta^2 T\right) \\ &= \exp\left(-\theta \widetilde{z}_t + \frac{1}{2}\theta^2 T\right) \end{aligned} \tag{2.14}$$

so

$$\frac{\partial f(\theta,\omega)}{\partial \theta} = (-\widetilde{z}_T + \theta T)\, e^{\left(-\theta \widetilde{z}_T + \frac{1}{2}\theta^2 T\right)}. \tag{2.15}$$

If we perform a measure to change to obtain a new IPA estimator under P, we obtain:

$$\widehat{C}^2(T,\omega)(-z_T)e^{\left(-2\theta z_T - \theta^2 T\right)} \tag{2.16}$$

which we call the IPA-Q, because it was derived under the Q measure.

Example 1

Suppose the stock price follows a geometric Brownian motion; then

$$dS_t = \mu S_t dt + \sigma S_t d\widetilde{z}_t$$

where μ is the drift (mean rate of return) and σ is the volatility (standard deviation rate of return). If we define

$$\lambda = \mu + \theta\sigma$$

then λ is the mean rate of return of S_t under the probability measure P. Thus, we can also use the rate of return λ as the parameter, because it is equivalent to θ. The IPA estimator given in (2.16) in terms of λ is as follows:

$$C^2(T,\omega)\left(-\frac{\widetilde{z}_T}{\sigma} + \frac{\lambda-\mu}{\sigma^2}T\right) e^{\left(-\frac{\lambda-\mu}{\sigma}\widetilde{z}_T + \frac{1}{2}\frac{(\lambda-\mu)^2}{\sigma^2}T\right)} \tag{2.17}$$

Example 2

Consider Asian options as in Vazquez-Abad and Dufresne (1998), where the underlying stock follows geometric Brownian motion

$$dS_t = rdt + \sigma d\widetilde{z}t$$

where r is the risk-free rate interest rate and σ is the volatility. The payoff function of the option at maturity T is given by

$$C(T) = \mathit{Max}(A(T) - K, 0)$$

where the average price is defined over the equally spaced discrete time points $N_0 + 1, \ldots, N$, i.e.,

$$A(T) = \frac{1}{N - N_0} \sum_{i=N_0+1}^{N} S_{\frac{i}{N}T}.$$

The IPA estimator (IPA-Q) is given by

$$e^{-2rT} \left[Max(A(T) - K, 0)\right]^2 f^2(\lambda) \left(-\frac{z_T}{\sigma}\right) \tag{2.18}$$

where

$$f(\lambda) = \exp\left(-\frac{\lambda - r}{\sigma} z_T - \frac{(\lambda - r)^2}{2\sigma^2} T\right) \tag{2.19}$$

An alternative IPA estimator (IPA-VA) is given by Vazquez-Abad and Dufresne (1998):

$$2e^{-2rT} \left[Max(A(T) - K, 0)\right]^2 f^2(\lambda) x \left[(A(T) - K)\left(\frac{z_T}{\sigma} - \frac{(\lambda - r)^2}{\sigma^2}\right) + \frac{1}{N - N_0} \sum_{i=N_0+1}^{1} \frac{iT}{N} S_{\frac{iT}{N}}\right] \tag{2.20}$$

Su and Fu (2000) provide results that show that the IPA-Q estimator leads to smaller variances than the IPA-VA estimator because the IPA estimator needs only to compute the derivative of dQ/dP with respect to λ, while the IPA-VA estimator involves computation of the derivative of dQ/dP and the derivative of the payoff function with respect to λ.

2.5 Importance Sampling Example in Matlab

The following Matlab code prices a straddle using importance sampling.

ImportanceSampling.m

```
%% The following code calculates the price of a straddle with
%% payoff = 30 - S  if S <= 30
%%        = 0       if 30 <= S <= 70
%%        = S - 70  if S >= 70
%%
%% with interest rate r = 2% countinuously compounded
%% volatility = 0.2, S(0) = 50 and time to expiry T = 1

%% Importance sampling has been used here to make MC simulations
%% more effective

clear;
format long;

r = .02;
s0 = 50;
vol = 0.2;
```

```
T = 1;
dt = 1/120;
M = T/dt;         %% Number of time steps
N = 10000;        %% Number of Simulations
sum_strad = 0;
sum_sT = 0;
sum_call = 0;
sum_put = 0;

a_30 = 2.55;    %% a for put with strike 30
a_70 = -1.68;   %% a for call with strike 70

for i = 1:N
    e_70 = randn(1);
    e_30 = randn(1);
    sT(i) = s0*exp(.2*(e_70 - a_70));   %% MC stock prices with imp
                                        %% sampling at time T
    sT(i) = max(sT(i), 0);              %% Making spurious paths
                                        %% where S(T) < 0 equal to 0

    %% Price of a call option with strike 70 calculated through
    %% importance sampling
    call(i) = exp(-r)*(max(sT(i) - 70, 0)*exp(a_70*e_70 - (a_70^2)/2));


    sT(N + i) = 50*exp(.2*(e_30 - a_30));    %% MC stock prices with imp
                                             %% sampling for strike price
                                             %% of 30 at time T = 1
    sT(N + i) = max(sT(N+i), 0);

    %% Price of a put option with strike 30 calculated through
    %% importance sampling
    put(i) = exp(-r)*(max(30 - sT(N + i), 0)*exp(a_30*e_30 - (a_30^2)/2));

    %% price of a straddle by adding call and put price
    strad(i) = call(i) + put(i);

    sum_strad = sum_strad + strad(i);
    avg_strad(i) = sum_strad/i;     %% recording avg price at each
                                    %% iteration for convergence dig.
end

%% Convergence Diagram
figure;
N_vector = [1:1:N];
plot(N_vector, avg_strad);
title('Convergence Diagram for Straddle price');
xlabel('Number of Samples');
ylabel('Monte Carlo Estimates');

straddle = avg_strad(N);    %% straddle price with N iterations

%% Standard Error
sumSE = 0;
```

```
for k = 1:N
    sumSE = sumSE + (straddle - strad(k))^2;
end

SE = sqrt(sumSE/(N*(N-1)));
```

Figure 2.1 shows the convergence for the straddle price for the 10,000 simulation paths.

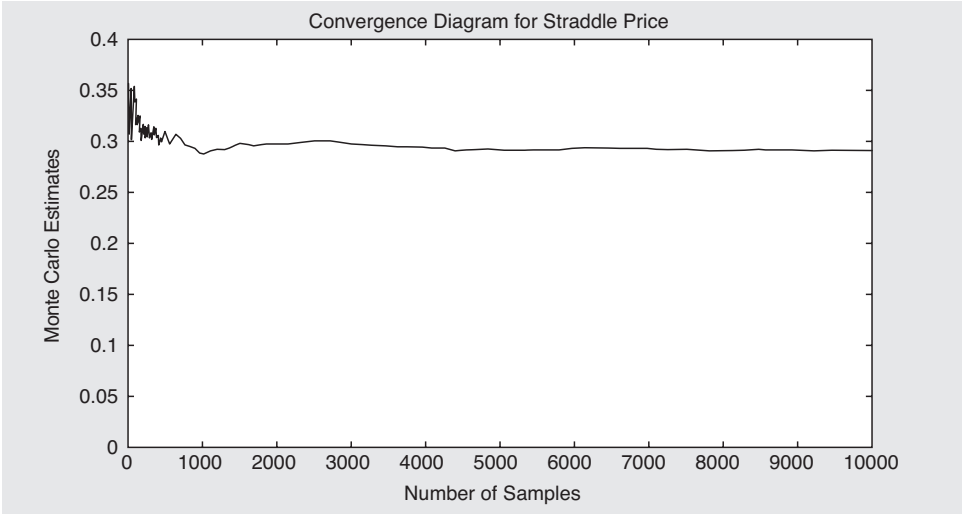


Figure 2.1

In this implementation, both importance sampling and moment matching have been used to price the straddle:

ImportanceMomentMatching.m

```
%% The following code calculates the price of a straddle with
%% payoff = 30 - S  if S <= 30
%%        = 0       if 30 <= S <= 70
%%        = S - 70  if S >= 70
%%
%% with interest rate r = 2% countinuously compounded
%% volatility = 0.2, S(0) = 50 and time to expiry T = 1

%% Techinques like Moment Matching and Importance sampling
%% have been used here to make MC simulations more effective


clear;
format long;

r = .02;
s0 = 50;
vol = 0.2;
T = 1;
```

```
dt = 1/120;
M = T/dt;         %% Number of time steps
N = 10000;        %% Number of Simulations
sum_strad = 0;
sum_sT = 0;
sum_call = 0;
sum_put = 0;

a_30 = 2.55;
a_70 = -1.68;

e = randn(2*N, 1);
avge = mean(e);
stdev_e = std(e);

%% 1f

for i = 1:N
    e_70 = (e(i,1) - avge)/stdev_e;      %% generating a random variate and
                                         %% moulding it for importance
                                         %% sampling for strike price of 70

    e_30 = (e(N+i, 1) - avge)/stdev_e;   %% generating a random variable and
                                         %% moulding it for importance for
                                         %% strike price of 30

    sT(i) = s0*exp(.2*(e_70 - a_70)*T);    %% MC stock prices with
                                           %% imp sampling at time T
    sT(i) = max(sT(i), 0);                 %% Making spurious paths where
                                           %% S(T) < 0 equal to 0

    %% Price of a call option with strike 70 calculated through
    %% importance sampling
    call(i) = exp(-r*T)*(max(sT(i) - 70, 0)*exp(a_70*e_70 - (a_70^2)/2));


    sT(N + i) = 50*exp(.2*(e_30 - a_30));    %% MC stock prices with imp
                                             %% sampling for strike price
                                             %% of 30 at time T = 1
    sT(N + i) = max(sT(N+i), 0);             %% spurious paths

    %% Price of a put option with strike 30 calculated through
    %% importance sampling
    put(i) = exp(-r)*(max(30 - sT(N + i), 0)*exp(a_30*e_30 - (a_30^2)/2));

    %% price of a straddle by adding call and put price
    strad(i) = call(i) + put(i);
    sum_strad = sum_strad + strad(i);
    avg_strad(i) = sum_strad/i;     %% recording avg price at each
                                    %% iteration for convergence dig.

end

%% Convergence Diagram
figure;
N_vector = [1:1:N];
plot(N_vector, avg_strad);
```

```
title('Convergence Diagram for Straddle price');
xlabel('Number of Samples');
ylabel('Monte Carlo Estimates');

%% Standard Error
sumSE = 0;

for k = 1:N
    sumSE = sumSE + (avg_strad(N) - strad(k))^2;
end

SE = sqrt(sumSE/(N*(N-1)));
```

Figure 2.2 shows the convergence for the straddle price for the 10,000 simulation paths, using importance sampling and moment matching.

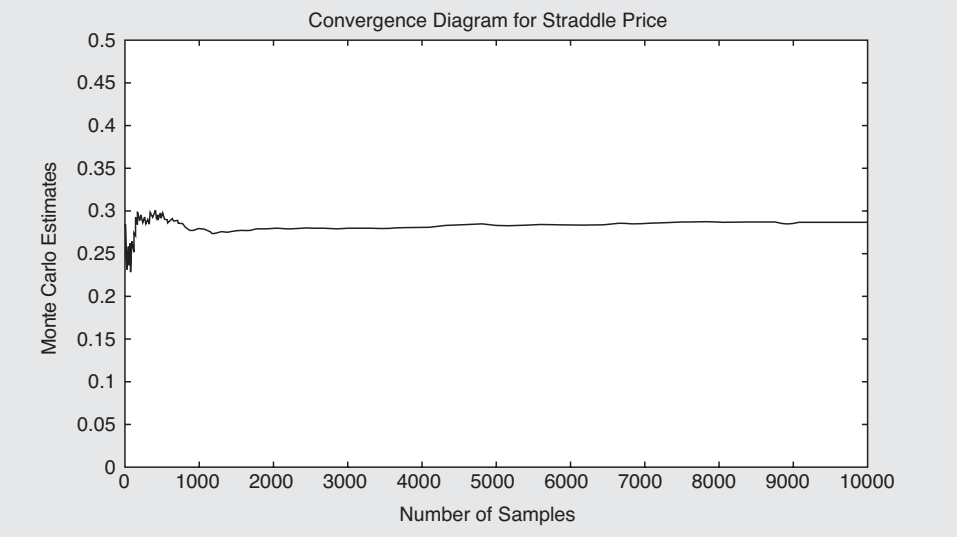


Figure 2.2

2.6 QUASI-RANDOM SEQUENCES

A quasi-random sequence, also called a low-discrepancy sequence, is a deterministic sequence of representative samples from a probability distribution. Quasi-random number generators (RNG) differ from pseudo RNGs in that pseudo RNGs try to generate "realistic" random numbers, while quasi generators create numbers that are evenly spaced in an interval—they have a more uniform discrepancy than pseudo RNGs.[10] For M simulations, quasi-random sequences have a standard error proportional to $1/M$, which is much smaller for large M than the standard error of pseudo RNGs, which is proportional to $1/\sqrt{M}$. Moreover, for a discrepancy of n points, D_n, low-discrepancy sequences have a discrepancy in the order of $O((\log n)^d/n)$, while a uniform random number sequence

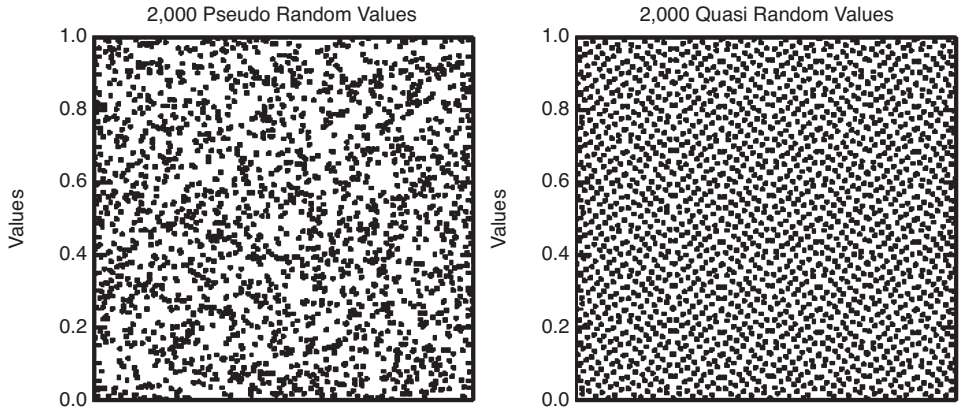


Figure 2.3

has a discrepancy in the order of $O(1/\sqrt{n})$. Thus, quasi RNGs are more efficient than pseudo-random numbers. Figure 2.3 shows how 2,000 quasi-random values are uniformly distributed, while the 2,000 pseudo-random values are not. As can be seen, the problem with pseudo-random numbers is that "clumpiness" occurs, which biases the results. A very large number of samples are needed to make the bias negligible. On the other hand, quasi-random numbers or low-discrepancy sequences are designed to appear random but not clumpy. In fact, a quasi-random sample is not independent from previous samples. It "remembers" the previous samples and tries to position itself away from all previous samples so that points are more uniformly distributed, and thus have a low discrepancy. This characteristic of low discrepancy sequences yields fast convergence in Monte Carlo simulation and is why it is preferred to pseudo-random numbers.

Two well-known low-discrepancy sequences are Sobol (1967) and Faure (1982).[11] The Sobol method uses a set of binary fractions called *direction numbers*. The jth number is generated by doing a bitwise exclusive or on all the direction numbers so that the ith bit of the number is nonzero. The effect is such that the bits toggle on and off at different rates. The kth bit switches once in 2^{k-1} steps so that the least-significant bit switches the fastest, and the most-significant bit switches the slowest.

The following is code to compute a Sobol sequence.[12] The methods are the inline part of the StatUtility class that contains methods for aiding in numerical and statistical computations.

```
#define GRAY(n) (n ^ ( n >> 1 ))  // for Sobol sequence
#define MAXDIM 5
#define VMAX 30

struct sobolp {
   double sequence[MAXDIM];
   int x[MAXDIM];
   int v[MAXDIM][VMAX];
   double RECIPD;
   int _dim;         // dimension of the sample space
```

```
   int _skip;
   unsigned long _nextn;
   unsigned long cur_seed;
};


class StatUtility {
public:
/*
   [in] numSamples    Number of samples desired
   [in] numDimensions The dimension of the sample spaces
   [out] realNum      Number actually created (<= numSamples)
*/
/*************************************************************************
  subolp_generateSamples
  Generates a Sobol sequence
  [in]: struct sobolp* config      : pointer to Sobol structure
                  double* samples  : pointer to sample values
 [out]: void

*************************************************************************/
inline void sobolp_generateSamples( struct sobolp* config, double*
samples)
{
   int i;
   nextSobol(config, config->cur_seed);
   config->cur_seed++;
   for(i = 0; i < config->_dim; i++ )
      samples[i] = config->sequence[i];
}

/*************************************************************************
  nextSobolNoSeed : Generates the next Sobol seed number to generate the
  next Sobol value
  [in]: struct sobolp* config      : pointer to Sobol structure
  [out]: void

*************************************************************************/
inline static void nextSobolNoSeed(struct sobolp* config) {
   int c = 1;
   int i;
   int save = config->_nextn;
   while((save %2) == 1)
    {
      c += 1;
           save  = save /2;
      }
      for(i=0;i<config->_dim;i++) {
             config->x[i] = config->x[i]^(config->v[i][c-1]<< (VMAX-c));
             config->sequence[i] = config->x[i]*config->RECIPD;
      }
      config->_nextn += 1;
}

/*************************************************************************
  sobolp_init : initializes the Sobol algorithm
  [in]: sobolp* config  : pointer to Sobol
```

```
        dim: dimension of the sample spaces
        seed: seed for Sobol number generator
 [out] : void
*************************************************************************/
void sobolp_init(struct sobolp* config, int dim, unsigned long seed)
{
  int d[MAXDIM], POLY[MAXDIM];
  int save;
  int m,i,j,k;

  config->_dim = dim;
  config->_nextn = 0;
  config->RECIPD = 1.0 / pow( 2.0, VMAX );
  config->cur_seed = seed;

  POLY[0] = 3;  d[0] = 1;  /* x + 1          */
  POLY[1] = 7;  d[1] = 2;  /* x^2 + x + 1    */
  POLY[2] = 11; d[2] = 3;  /* x^3 + x + 1    */
  POLY[3] = 19; d[3] = 4;  /* x^4 + x  + 1   */
  POLY[4] = 37; d[4] = 5;  /* x^5 + x^2 + 1 */

  for(i=0; i < config->_dim; i++ )
    for(j = 0; j < d[i]; j++ )
      config->v[i][j] = 1;

  for( i = 0; i < config->_dim; i++ )
     for( j = d[i]; j < VMAX; j++ )
     {
       config->v[i][j] = config->v[i][j-d[i]];
       save = POLY[i];
       m = pow( 2, d[i] );
       for( k = d[i]; k > 0; k-- )
      {
          config->v[i][j] = config->v[i][j] ^ m*(save%2)*config->v[i][j-k];
          save = save/2;
             m = m/2;
      }
    }

  for( i = 0; i < config->_dim; i++ )
    config->x[i]=0;

  config->_skip = pow( 2, 6 );

  for( i = 1; i <= config->_skip; i++ )
    nextSobolNoSeed(config);
       }
 };
```

A Sobol implementation in MATLAB is provided with the code that accompanies this book. Here is a Monte Carlo implementation using the Sobol sequence:

MCPricer_MonteCarloSobol.cpp

```
/*************************************************************************
*
  MonteCarloSobol : Values a European call option using Faure sequence for
    variance reduction
  [in]: double S: asset price
        double X: strike price
        double vol: volatility
        double rate: risk-free rate
        double div:  dividend yield
        double T: option maturity
             char type: type of option
        long N: number of time steps
        long M: number of simulations
 [out] :  callValue
*************************************************************************
*/
vector<double> MCPricer::MonteCarloSobol(double price, double strike,
  double vol, double rate, double div, double T, char type, long N, long M)
{
        int i, j;
        double sum1 = 0.0;
        double sum2 = 0.0;
        double S1 = price;              // stock price for +deviate
        double S2 = price;              // stock price for -deviate
        double lnS1 = log(price);   // log of the initial stock
                                    // price for +deviate
        double lnS2 = log(price);   // log of the initial stock
                                    // price for -deviate
        double SD;                 // standard deviation
        double SE;                 // standard error
        long dim =  N;             // dimension of Sobol sequence
        double dt = T/N;                    // time step
        double mu = rate - div - 0.5*vol*vol;  // drift
        double rands[100];
        double val = 0;
        int cnt = 0;
        vector<double> value;
        struct sobolp sp;

        srand(unsigned(time(0)));      // initialize RNG
        long seed = (long) rand() % 100;
        // initialize Sobol sequence
        util.sobolp_init(&sp,dim,seed);

        for (i = 0; i < M; i++)
        {
                // initalize stock price for the next simulation
                lnS1 = log(price);
                lnS2 = log(price);

                for (j = 0; j < N; j++)
```

```
                {

                        // generate Sobol samples
                        util.sobolp_generateSamples(&sp,rands);
                        // generate path and antithetic path
                        lnS1 = lnS1 + mu*dt + vol*sqrt(dt)*rands[cnt];
                        lnS2 = lnS2 = mu*dt + vol*sqrt(dt)*(-rands[cnt]);

                        // keep track of Sobol number to use
                        if ((cnt + 1) % N == 0)
                               cnt = 0;
                        else
                               cnt++;
                }

                // convert back to lognormal random variables
                S1 = exp(lnS1);
                S2 = exp(lnS2);

                if (type == 'C')
                        val = 0.5*(max(0, S1 - strike) +
                                max(0, S2 - strike));
                else
                        val = 0.5*(max(0, strike - S1) +
                                max(0,strike - S2));

                sum1 = sum1 + val;
                sum2 = sum2 + val*val;
        }

        val = exp(-rate*T)*sum1/M;
        value.push_back(val);
        // compute standard deviation
        SD = sqrt((exp(-2*rate*T)/(M-1))*(sum2 - (sum1*sum1)/M));
        value.push_back(SD);

    // compute standard error
        SE = SD/sqrt(M);
        value.push_back(SE);

        return value;
}
```

The number of time steps to be used along each path should equal the dimension of the Sobol sequence.

In a Monte Carlo simulation, the number of time steps N is the dimension of a low-discrepancy sequence—that is, the number of independent quasi-random numbers to be generated simultaneously. The quasi-random numbers are generated simultaneously so that the samples and increments along the path are independent and identically distributed. Let $x_k, k = 1, \ldots, N$, be N quasi-random numbers. Then, the Faure sequence of length M (the number of simulations) is defined by

$$x_k = \sum_{l=0}^{m} \frac{a_{k,l}}{p^{l+1}} \tag{2.21}$$

where m is the number of digits in the p representation of M—i.e.,

$$m = \int \left[\frac{\log(M)}{\log(p)}\right], \; a_{0,l} = \int \left[\frac{M\%p^{l+1}}{p}\right],$$
$$a_{k,l} = \sum_{q=l}^{m} \frac{q!}{l!(q-l)!} a_{k-1,q}\%p \tag{2.22}$$

and p is the smallest prime number greater than or equal to N.[13] The "int" operator denotes the integer part of the expression in brackets, and % denotes the modulo operator that is the remainder after division.

The following is code to implement a Faure sequence. The function *generateFaure* is an inline function in the StatUtility class. Other helper inline utility functions are also provided.

```
   #include "TNT.h"

    class StatUtility
    {
     public:

/************************************************************************
**
    generateFaure: This function generates a Faure sequence of length M
 [in] N : number of time steps
          M: number of  simulations
 [out]: vector<double>  X: (the Faure sequence)

   ************************************************************************
   ***/
inline vector<double> generateFaure(long N, long M)
{
       int p = generatePrime(N);
       int l, q, k;
       long v1, v2, v3;
       long value = 0;
       Array2D<long> a(N,N);
       int m = (int)(log(M)/log(p));
       if (m == 0)
             m = 1;
       long x[] = {0};
       unsigned long fact = 0;


       for (k = 1; k <= N; k++) // number of time steps
       {
              for (l = 0; l <= m; l++)
              {
                    value = pow(p,l+1);
                    a[0][l] = (int)((M % value)/p);

                    for (q = l; q <= m; q++)
                    {
                        v1 = factorial(q);
```

```
                        v2 = factorial(q-l);
                        v3 = factorial(l);
                             fact = v1/(v2*v3);

                        value = fact*a[k-1][q] % p;
                        a[k][l] = a[k][l] +  value;
                    }
                    x[k] = x[k] + a[k][l]/pow(p,l+1);
            }
            X.push_back((double)x[k]);
      }


      return X;
}

/*************************************************************************
***/
 factorial   This function computes the factorial of a number
 [in]:   N : number to factorialize
 [out]: N!
*************************************************************************
***/
inline long factorial(long N)
{

if ((N == 1) || (N == 0))
return 1;
else
      return N*factorial(N-1);
}

/*************************************************************************
***/
  generatePrime This function computes the smallest prime greater than or
    equal to N
  [in]: long N :  find prime >= N
  [out]:  prime >= N
*************************************************************************
***/
inline long generatePrime(long N)
{
long i = N;
bool flag = false;

do
{
            // check if number is prime
            if ((i % 2 != 0) && (i % 3 != 0) && (i % 4 != 0)
              && (i % 5 != 0) && (i % 7 != 0) && (i % 8 != 0)
              && (i % 9 != 0))
                    flag = true;
            else
            i++;

}
while (flag != true);
```

```
return i;
}

/*************************************************************************
****
 Function: polarRejection
 This function computes two standard deviates using polar rejection
   (transformation) method
 Returns the first deviate and stores the second deviate in a vector Y so
 that it can be used for another call rather than throwing it away.
 [in]: double y : seed value
       i:   ith standard deviate
 [out]: Y[i]  // ith standard normal deviate in Y
*************************************************************************
***/
inline double polarRejection(double y, int i)

       double w = 0.0;
       double x1, x2, z1, z2, c;
       double temp = 0.0;
       double *idum = &y;

       do
       {
             x1 = gasdev((long*)idum);
             x2 = gasdev((long*)idum);

             w = x1*x1 + x2*x2;
       }
       while (w >= 1);

       c = sqrt(-2*(log(w)/w));
       z1 = c*x1;
       Y.push_back(z1);
       z2 = c*x2;
       Y.push_back(z2);

       return Y[i];
}
...
...
};
```

The following is a Monte Carlo implementation using the Faure sequence to value a European call option with maturity T.

```
/*********************************************************************
  Function: MonteCarloFaureQuasiRandom
  Values a European call option using Faure sequence for variance reduction
  [in]: double S: asset price
        double X: strike price
        double vol: volatility
        double rate: risk-free rate
        double div:  dividend yield
        double T: option maturity
```

```
        long N: number of time steps
        long M: number of simulations
 [out] :  callValue
**************************************************************************/
double MCPricer::FaureQuasiRandom(double S, double X, double vol, double
rate, double div, double T, long N, long M)
{
      int i, j, k;
      double dt = T/N;       // step step
      double mudt = (rate - div - 0.5*vol*vol)*dt;  // drift
      double voldt = vol*sqrt(dt);                  // diffusion term
      double sum = 0.0;
      double sum1 = 0.0;
      double lnSt, lnSt1, St, St1;
      double lnS = log(S);
      double deviate = 0.0;
      double callValue = 0.0;
      double SD = 0.0;       // standard deviation
      double SE = 0.0;       // standard error
      vector<double> x;      // stores Faure sequence

      k = 0;
      for (i = 1; i <= M; i++)
      {
           // generate Faure sequence
            x = util.generateFaure(N,M);

           // initialize log asset prices for next simulation path
            lnSt = lnS;
            lnSt1 = lnS;

            for (j = 0; j < N; j++)
            {
                  // get standard normal deviate using polar rejection
                  // method
                  deviate = util.polarRejection(x[j],k);

                  lnSt = lnSt + mudt + voldt*deviate;

                  // compute antithetic
                  lnSt1 = lnSt1 + mudt + voldt*(-deviate);
                  // increment index to retrieve deviate stored in
                  // vector Y in polar rejection method
                  k++;
            }
            St = exp(lnSt);
            St1 = exp(lnSt1);

            callValue = 0.5*(max(0, St - X) + max(0,St1-X));
            sum = sum + callValue;
            sum1 = sum1 + callValue*callValue;
      }

      callValue = exp(-rate*T)*(sum/M)

      SD = sqrt(exp(-2*rate*T)*(sum1/M) - callValue*callValue);
      cout << "stdev =   "  <<  SD << endl;
```

```
    SE = SD/sqrt(M-1);
    cout << "stderr =   "  <<  SE << endl;

    return callValue;
}
```

2.7 VARIANCE REDUCTION TECHNIQUES

Suppose we can simulate an independent and identically distributed (i.i.d) sequence $\{f_i^*, i = 1, \ldots, M\}$, where each f_i^* has expectation f and variance σ^2. An estimator of f based on M simulations is then the sample mean:

$$\frac{1}{M}\sum_{i=1}^{M} f_i^* \tag{2.23}$$

By the central limit theorem, for large M, this sample mean is approximately normally distributed with mean f and standard deviation $\frac{\sigma}{\sqrt{M}}$. The estimate of the option's price converges to the true option's price f with the standard error $\frac{\sigma}{\sqrt{M}}$. The 95% confidence interval is $f - 1.96\frac{\sigma}{\sqrt{M}} < f^* < f + 1.96\frac{\sigma}{\sqrt{M}}$.

Convergence of the crude Monte Carlo is slow to the order of $1/\sqrt{M}$. To increase the accuracy 10 times, we need to run 100 times more simulations of sample paths. On the other hand, decreasing the variance σ^2 by a factor of 10 does as much for error reduction as increasing the number of simulations by a factor of 100.

The simplest variance reduction procedure is *antithetic variates*. For each path simulated with ε_i^k (denoted as the ith deviate, $i = 1 \ldots N$ on the kth path, $k = 1 \ldots M$), an identical path is simulated with $-\varepsilon_i^k$. The payoff F_k for the path is calculated with ε_i^k, and also the payoff $\widehat{F}_k$ for the path with $-\varepsilon_i^k$. Then the average is taken: $\frac{1}{2}(F_k + \widehat{F}_k)$. Although ε_i^k are samples from a standard normal distribution with mean 0, in a sample, you will get some non-zero mean. The antithetic variates procedure corrects this bias by averaging out the deviations and centers the mean at 0.

Another variance reduction technique is to use *control variates*. Control variates are random variables, whose expected value (mean) that we know, which is correlated with the variable we are trying to estimate—i.e., the derivative security we want to value.[14] The principle underlying this technique is to "use what you know." Suppose you are trying to simulate an estimate for the price of a complex security. Suppose also that you know a closed-form analytical formula for the price of a similar, but simpler, security. The price of the complex security can be represented as follows:

$$f_{complex} = f_{simple} + (f_{complex} - f_{simple}) \tag{2.24}$$

Because you know the price f_{simple} from the closed-form formula, you need only estimate the difference $\varepsilon^* = (f_{complex} - f_{simple})$ via the following simulation:

$$f^*_{complex} = f_{simple} + \varepsilon^* \tag{2.25}$$

Because the securities are similar, the difference ε^* is small relative to the value f_{simple}, and we can find the "bulk" of the value of our complex security exactly without errors, and our errors are in the relatively smaller estimate of the difference in ε^*.

As an example for practical use of control variates, we use arithmetic and geometric Asian options. We know the analytical formula for the price of a geometric Asian option (see Chapter 5, "Collateralized Debt Obligations"), but in practice, we are most interested in the price of an arithmetic Asian option. There is no simple analytical formula for arithmetic Asian options. We note that the price of otherwise identical arithmetic and geometric Asian options are rather close. Thus, we can represent the price of an arithmetic Asian option as follows:

$$f^*_{arithmetic} = f^*_{geometric} + \varepsilon^*$$

We evaluate this technique next in Section 2.8.

2.8 Monte Carlo Antithetic Example in Matlab

The following Matlab code gives an example of option pricing using antithetics to reduce the standard error estimate.

Antithetic.m

```
%%  Author: Chetan Jain & Jim Carson
%%  The program calculates the estimate Monte Carlo value using antithetic
%%  variables of a call option on a stock with dynamics
%%  dY = 2(0.04 - Y)dt + 0.2YdW and
%%  dS/S = 0.01dt + sqrt(Y)*(-0.7dW + (sqrt(0.51))dZ)
%%  where S(0) = 99.7503 and Time = 3 months
%%  It also calculates the standard error and the Black Scholes implied
%%  volatility of call option

clear;
format long;

T = 0.25;
dt = 1/120;
M = T/dt;           %%  Number of time steps

y0 = 0.04;          %%  Value of Y at time zero
s0 = 99.7503;       %%  Value of S at time zero
N = 50000;          %%  Number of Monte Carlo simulations
N = N/2;            %%  Adjusting N for Antithetic variables

for i = 1:N         %%  Generate N Monte Carlo simulations
    y(1) = y0;
    y1(1) = y0;
    s(1) = s0;
    s1(1) = s0;

    for j = 2:M             %%  Generate sample path
        xi = randn(1);
        xj = randn(1);
        y(j) = y(j-1) + 2*(0.04 - y(j-1))*dt + 0.2*y(j-1)*sqrt(dt)*xi;
```

```
        y(j) = max(y(j),0);
        s(j) = s(j-1)*( 1 + 0.01*dt + sqrt(y(j-1)*dt)*(-0.7*xi +
                 (sqrt(0.51))*xj));
        s(j) = max(s(j),0);

        %%  Antithetic variates
        y1(j) = y1(j-1) + 2*(0.04 - y1(j-1))*dt + 0.2*y1(j-1)*sqrt(dt)*-xi;
        y1(j) = max(y1(j),0);
        s1(j) = s1(j-1)*( 1 + 0.01*dt + sqrt(y1(j-1)*dt)*(-0.7*-xi +
                  (sqrt(0.51))*-xj));
        s1(j) = max(s1(j),0);

    end

    yT(i) = y(M);           %% Save terminal values of y
    yT1(i) = y1(M);         %% Save Antithetic terminal values of y
    sT(i) = s(M);           %% Save terminal values of s
    sT1(i) = s1(M);         %% Save Antithetic terminal values of s

    c(i) = exp(- 0.01 * 0.25) * max( sT(i) - 100, 0);       %% Call values
    c(i + N) = exp(- 0.01 * 0.25) * max( sT1(i) - 100, 0);  %% Antithetic
                                                            %% call values

end

%%  MC estimates
sumc = 0;
ConvDiagram_c = zeros(N,1);

for k = 1:2*N
    sumc = sumc + c(k);
    ConvDiagram_c(k) = sumc/(k);

end

avgc = sumc/(2*N);

N_vector = [1:1:2*N];
plot(N_vector, ConvDiagram_c);
title('Convergence Diagram for European Call Option Price');
xlabel('Number of Samples');
ylabel('Monte Carlo Estimate (with antithetic variables)');

figure;
plot(N_vector, ConvDiagram_c);
title('Convergence Diagram for European Call Option Price [Detail]');
xlabel('Number of Samples');
ylabel('Monte Carlo Estimate (with antithetic variables)');
axis([0 500 0 6]);

%% Standard Error of Call prices

sumSEc = 0;
for k = 1:2*N
    sumSEc = sumSEc + (c(i) - avgc)^2 ;
end
```

```
SEc = sqrt(sumSEc/(N*2));


%% Implied Volatility
vol = blsimpv(s0, 100, 0.01, 0.25, avgc, 10);
```

Running the Matlab function returns the following result:

```
vol =

        0.1944540
```

Figure 2.4 shows the convergence of a European call price option using antithetics, and Figure 2.5 shows the convergence of the price in detail.

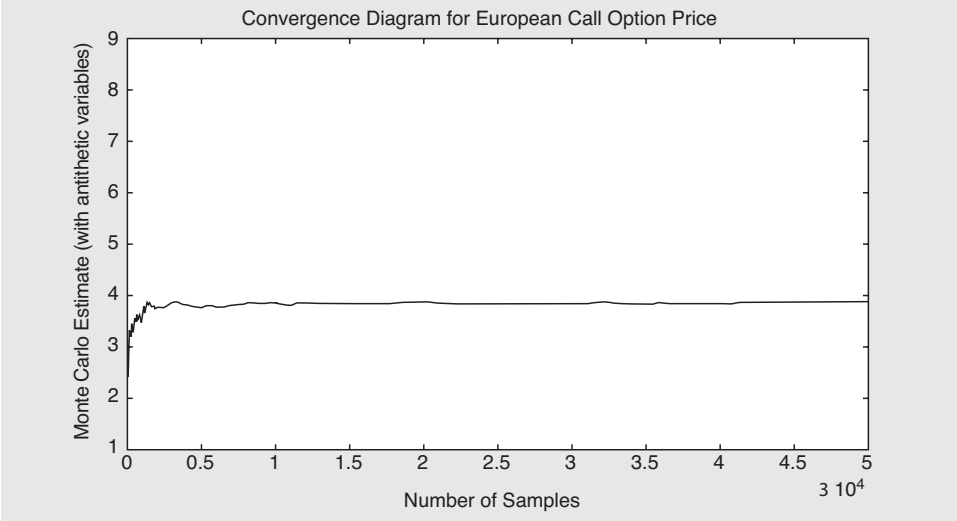


Figure 2.4

2.9 MONTE CARLO IMPLEMENTATION IN C++

The best way to simulate geometric Brownian motion (GBM) of the underlying asset (random variable) is to use the process for the natural logarithm of the variable, which follows arithmetic Brownian motion (ABM) and is normally distributed. Let $x(t) = \ln(S(t))$. Then we have

$$dx(t) = (r - q - \frac{1}{2}\sigma^2)dt + \sigma dz(t) \tag{2.26}$$

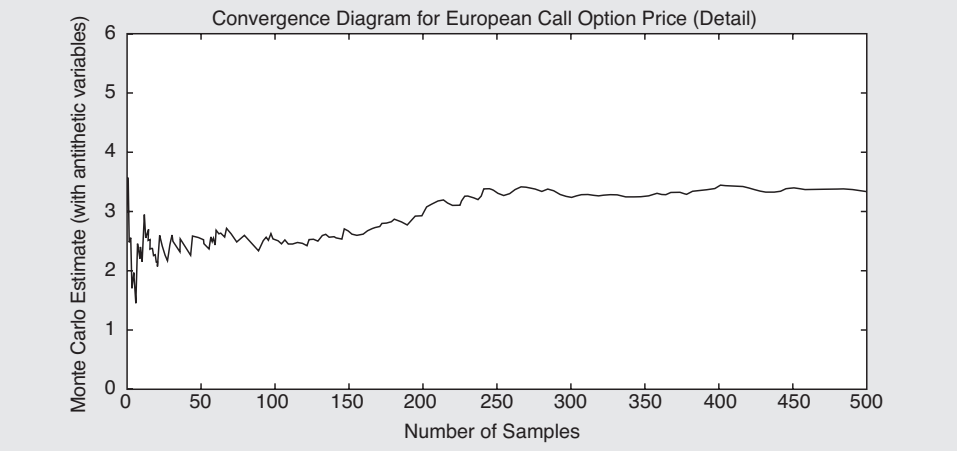


Figure 2.5

Equation (2.18) can be discretized by changing the infinitesimal changes dx, dt, and dz into small discrete changes Δx, Δt, and Δz:

$$\Delta x = (r - q - \frac{1}{2}\sigma^2)\Delta t + \sigma \Delta z \tag{2.27}$$

This representation involves no approximation because it is actually the solution of the SDE in (2.8), which can be written as follows:

$$x(t + \Delta t) = x(t) + (r - q - \frac{1}{2}\sigma^2)\Delta t + \sigma(z(t + \Delta t) - z(t)) \tag{2.28}$$

We can write equation (2.20) in terms of the underlying asset price S

$$S(t + \Delta t) = S(t) \exp\left(\left(r - q - \frac{1}{2}\sigma^2\right)\Delta t + \sigma\left(z(t + \Delta t) - z(t)\right)\right) \tag{2.29}$$

where $z(t)$ is standard Brownian motion. The random increment $z(t + \Delta t) - z(t)$ has mean zero and variance Δt. It can be simulated by random samples of $\varepsilon\sqrt{\Delta t}$, where ε is a standard deviate drawn from a standard normal distribution. Dividing up the time to maturity (the time period over which we want to simulate), T, into N time steps, each time step is of size $\Delta t = T/N$. Consequently, we can generate values of $S(t)$ at the end of these intervals, $t_i = i\Delta t, i = 1, \ldots, N$ using equation (2.20) by computing

$$x(t_i) = x(t_{i-1}) + (r - q - \frac{1}{2}\sigma^2)\Delta t + \sigma\varepsilon_i\sqrt{\Delta t},\ i = 1, \ldots, N \tag{2.30}$$

then computing

$$S(t_i) = \exp(x(t_i))i = 1, \ldots, N \tag{2.31}$$

for each time step of each of the M simulations, and then finally computing the payoff $Max(0, S(T) - X)$ at maturity—i.e., the final time step on a given path. To obtain an estimate $\widehat{C}$ of the call price, C, we take the discounted average of all the simulated payoffs:

$$\widehat{C} = e^{-rT} \frac{1}{M} \sum_{j=1}^{M} Max(0, S_j(T) - X) \tag{2.32}$$

Note that to compute a European call estimate under GBM, we can let $N = 1$. Moreover, because we have a closed-form solution to the underlying SDE, samples of $S(T)$ can be found directly without simulating the entire path. In general, however, $N > 1$, because there are many types of derivatives—i.e., path-dependent options—where only an approximate discretization of the SDE to the continuous SDE can be used by taking small time steps.

The following code implements the Monte Carlo simulation under the risk-neutral process in (2.23). It uses the antithetic variates to reduce the variance:

MCPricer_MonteCarloAntithetic.cpp

```
/**************************************************************************
 MonteCarloAntithetic : values a European call option using antithetic
                        variates
 [in]:   double price: asset price
         double strike: strike price
         double vol: volatility
         double rate: risk-free rate
         double div:  dividend yield
         double T: option maturity
         long N: number of time steps
         long M: number of simulations
  [out]     double value (call value)
**************************************************************************/
vector<double> MCPricer::MonteCarloAntithetic(double price, double strike,
  double vol, double rate, double div, double T, long M, long N, char type)
{

        int i, j;
        double deviate;             // random deviate
        double sum1 = 0.0;          // sum of payoffs
        double sum2 = 0.0;          // sum of squared payoffs
        double S1 = price;          // stock price for +deviate
        double S2 = price;          // stock price for -deviate
        double lnS1 = log(price);   // log of the initial stock price
                                    // for +deviate
        double lnS2 = log(price);   // log of the initial stock price
                                    // for -deviate
        double SD;                  // standard deviation
        double SE;                  // standard error
        double dt = (double) T/N;
        double val = 0;
        vector<double> value;
        double mu = rate - div - 0.5*vol*vol; // drift

        // initialize rng
```

```
        mrng.sgenrand(unsigned(time(0)));

        for (i = 0; i < M; i++)
        {
                // initalize stock price for the next simulation
                lnS1 = log(price);
                lnS2 = log(price);

                for (j = 0; j < N; j++)
                {
                       // generate random deviate
                        deviate = mrng.genrand();
                        // simulate paths
                        lnS1 = lnS1 + mu*dt + vol*sqrt(dt)*deviate;
                        lnS2 = lnS2 + mu*dt + vol*sqrt(dt)*(-deviate);
                }

                // convert back to lognormal random variables

                S1 = exp(lnS1);
                S2 = exp(lnS2);

                if (type == 'C')
                        val = 0.5*(max(0, S1 - strike) +
                                max(0,S2 - strike));
                else  // if put
                        val = 0.5*(max(0, strike - S1) +
                                max(0, strike - S2));

                sum1 = sum1 + val;
                sum2 = sum2 + val*val;
        }

        val = exp(-rate*T)*sum1/M;
        value.push_back(val);

        // compute standard deviation
        SD = sqrt((exp(-2*rate*T)/(M-1))*(sum2 - (sum1*sum1)/M));
        value.push_back(SD);

        // compute standard error
        SE = SD/sqrt(M);
        value.push_back(SE);

        return value;
}
```

Suppose we want to calculate the price of European call option in a Black-Scholes world with antithetic variance reduction where $S = 50$, $X = 50$, $r = 5.5\%$, $q = 2\%$, $T = 0.75$ (nine months), and $\sigma = 0.20$. We make the following method call to *MonteCarloAntithetic* with $M = 100$, 1000, 10000, and 100000 (changing M in the method call), simulations, and $N = 10$ time steps.

Table 2.1 summarizes the results. Notice that as the number of simulations increases, both the standard deviation and standard error decreases. Moreover, as the number of sim-

Table 2.1

Num. of Simulations	MC Call Price	Standard Deviation	Standard Error
100	3.935	3.458	0.360
1000	4.013	3.200	0.101
10000	4.057	3.153	0.032
100000	4.037	3.093	0.010

ulations increases by a factor of 10, the standard error decreases by approximately a factor of 3.

The Black-Scholes price is $4.03. Thus, increasing the number of simulations narrows the confidence interval of the estimate because the standard error decreases.

Monte Carlo is used extensively for simulating outcomes—i.e., profit and losses (P/L) of dynamic trading strategies. A single sample diffusion path is simulated, and the dynamic trading strategy is executed over this path. The P/L is then calculated. Then, the process is repeated M times for M sample paths. The mean is an estimate of the expected P/L from the trading strategy. A standard deviation of P/Ls around this mean tells you how stable the trading strategy is. Monte Carlo is extensively used in risk and portfolio management to compute value-at-risk (VAR) of a portfolio. A confidence level is chosen—for example, 95% or 99%—and the underlying factors of each security are simulated, and the P/L of each position in each security is calculated based on the realization of the simulated factor values. The process is repeated for each security M times, and then, based on the simulated P/L probability distribution generated by the aggregated P/L of all securities in the portfolio, the VAR can be computed by looking at the P/L value that lies to the left of confidence level of the P/L probability distribution.

As a practical application, suppose we want to implement a dynamic replication strategy $\{(\Delta_t, N_t), 0 \leq t \leq T\}$ of going long a stock and going short a call option on the stock. We know from (1.48) that the option price at time t is $f_t = \Delta_t S_t - N_t A_t$. The following code implements a dynamic replication strategy on a call option that matures in 20 weeks, with a strike of 50, a volatility of 20%, a risk-free rate of 5.0%, and a current stock price of $49. We are always rebalancing between our long equity position and short position in the money market account (to finance our stock purchases) so that the strategy is self-financing.

Monte Carlo is also well suited for valuation of spread options and basket options (options on a portfolio of assets). Consider two stocks, S_1, and S_2, that each follow risk-neutral price processes

$$dS_1 = (r - q_1)S_1 dt + \sigma_1 S_1 dz_1$$

and

$$dS_2 = (r - q_2)S_2 dt + \sigma_2 S_2 dz_2$$

where $dz_1 dz_2 = \rho dt$. Price paths follow a 2D discretized geometric Brownian motion that can be simulated according to

$$S_{1,i+1} = S_{1,i} \exp((r - q_1 - \frac{\sigma_1^2}{2})\Delta t + \sigma_1 \sqrt{\Delta t}\varepsilon_{1,i+1}$$

and

$$S_{2,i+1} = S_{2,i} \exp((r - q_2 - \frac{\sigma_2^2}{2})\Delta t + \sigma_2\sqrt{\Delta t}(\rho\varepsilon_{1,i+1} + \sqrt{1-\rho^2}\varepsilon_{2,i+1}),$$

for $i = 0, 1, \ldots, N$, where $\varepsilon_{1,i+1}$ and $\varepsilon_{2,i+1}$ are samples from the standard normal. The same seed is used for all simulations. The following is an implementation to value a European spread call (or put) option with payoff

$$e^{-r(T-t)} E^Q \left[Max(S_1(T) - S_2(T) - X, 0) | \Im_t\right]$$

for a call and

$$e^{-r(T-t)} E^Q \left[Max(X - S_1(T) + S_2(T), 0) | \Im_t\right]$$

for a put.

The C++ code to price a spread option is:

MCPricer_calcMCEuroSpreadOption.cpp

```
/*************************************************************************
      calcMCEuroSpreadOption :  computes the value of a European spread
        option
       [in]   double price1: price of asset 1
                double price2: price of asset 2
                double strike: strike price
                double rate : risk free rate
                double vol1: volatility of asset 1
                double vol2: volatility of asset 2
                double div1: dividend yield of asset 1
                double div2: dividend yield of asset 2
                double rho : correlation of dz1 and dz2
                double T:  maturity of option
                char type: option type (C)all or (P)ut
                long M: number of simulations
                long N: number of time steps
         [out] : price of spread option
*************************************************************************/
vector<double> MCPricer::calcMCEuroSpreadOption(double price1, double
   price2, double strike, double rate, double vol1, double vol2, double
   div1, double div2, double rho, double T, char type, long M, long N)
{
        N = 1;                      // No path dependency
        int i, j;                   // counter
        double dt = T/N;            // size of time step
        double mu1 = (rate - div1 - 0.5*vol1*vol1);   // drift for stock
                                                      // price 1
        double mu2 = (rate - div2 - 0.5*vol2*vol2);   // drift for stock
                                                      // price 2
        double srho = sqrt(1 - rho*rho);
        double sum1 = 0.0;          // sum of all the call values on stock 1
                                    // at time T
        double sum2 = 0.0;          // sum of all the call values on stock 2
                                    // at time T
        double S1 = 0.0;            // stock price 1
```

```
        double S2 = 0.0;            // stock price 2
        double deviate1 = 0.0;      // deviate for stock price 1
        double deviate2 = 0.0;      // deviate for stock price 2
        double z1 = 0.0;            // correlated deviate for stock price 1
        double z2 = 0.0;            // correlated deviate for stock price 2
        double CT = 0.0;            // option price at maturity
        double SD = 0.0;            // standard deviate of price
        double SE = 0.0;            // standard error of price
        double val = 0.0;           // spread option price today
        vector<double> value;

        mrng.sgenrand(unsigned(time(0))); // initialize Mersenne RNG

        for (i = 0; i < M; i++)
        {
                // initialize prices for each simulation
                S1 = price1;
                S2 = price2;

                for (j = 0; j < N; j++)
                {
                        // generate deviates
                        deviate1 = mrng.genrand();
                        deviate2 = mrng.genrand();

                        // calculate correlated deviates
                        z1 = deviate1;
                        z2 = rho*deviate1 + srho*deviate2;
                        S1 = S1*exp(mu1*dt + vol1*z1*sqrt(dt));
                        S2 = S2*exp(mu2*dt + vol2*z2*sqrt(dt));
                }

                if (type == 'C')
                        CT = max(S1 - S2 - strike, 0);
                else
                        CT = max(strike - S1 + S2,0);

                sum1 = sum1 + CT;
                sum2 = sum2 + CT*CT;
        }


        val = exp(-rate*T)*(sum1/M);
        value.push_back(val);

        SD = sqrt((sum2 - sum1*sum1/M)*exp(-2*rate*T)/(M-1));
        value.push_back(SD);

        SE = SD/sqrt(M);
        value.push_back(SE);

        return value;
}
```

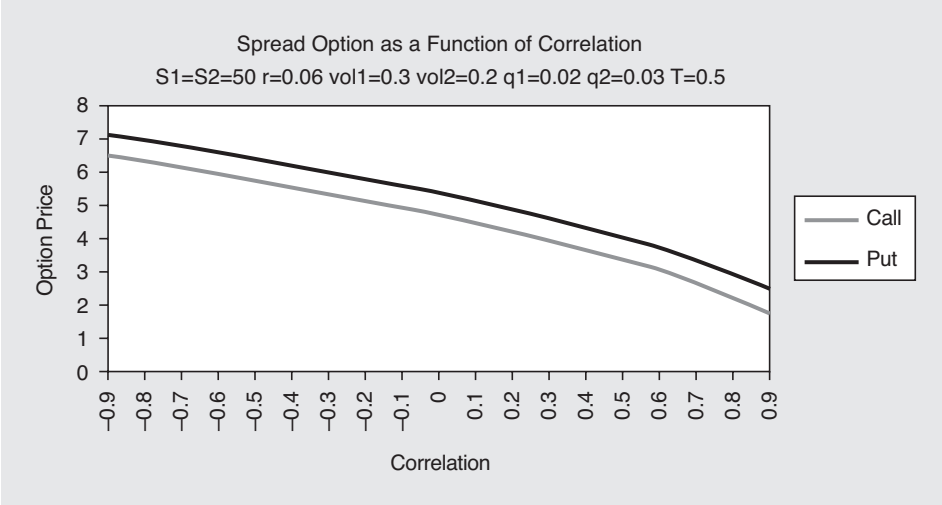


Figure 2.6

Suppose $S_1 = 50$, $X = 1$, $r = 0.06$, $q_1 = 2\%$, $q_2 = 3\%$, $\sigma_1 = 30\%$, $\sigma_2 = 20\%$, and $T = 0.5$. Figure 2.6 is a plot of the price of call and put spread option prices as a function of correlation $\rho = -1, -0.9, \ldots, 0.9, 1$, for $M = 100,000$ simulations. Note that because the option is European, we can speed the computation by using only $N = 1$ time step. Notice that as the correlation increases, the option price (monotonically) decreases for both call and put spread options. When $\rho = 1$, the second random factor $\varepsilon_{2,i}$ cancels out of the second geometric Brownian equation for S_2. Consequently, both S_1 and S_2 are being driven by only one (the same) source of randomness or uncertainty, $\varepsilon_{1,i}$. Thus, $z_{1,i}$ and $z_{2,i}$ move in the same direction so that random movements of both assets occur in the same direction and make the spread $S_1 - S_2$ decrease because the same direction movements are offset. On the other hand, when $\rho = -1$, the randomness of $+\varepsilon_{1,i}$ in the equation of the first asset is offset by $-\varepsilon_{1,i}$ in the equation of the second asset, and so movements in one direction of one asset are magnified by movements in the opposite direction of the other asset. Thus, a widening of the spread $S_1 - S_2$ occurs, making the option worth more. Other numerical techniques for valuing spread options include the Fast Fourier Transform (FFT) method, which is discussed in the next section.

2.10 Fast Fourier Transform

We discuss the Fast Fourier transform (FFT) method for pricing derivatives when higher dimensional processes cannot be efficiently priced with binomial trees. Because a two-dimensional binomial scheme is not efficient and does not converge to the true theoretical price, it may be necessary to use numerical methods to price a bivariate (and higher variate) option, such as a spread option. In particular, FFT can be used.[15] FFT is faster than finite-difference methods and integro-differential equations.

The FFT is an efficient algorithm for computing

$$w(k) = \sum_{i=1}^{N} e^{-\frac{2\pi i}{N}(j-1)(k-1)} x(j) \quad k = 1, \ldots, N \tag{2.33}$$

where N is usually a power of 2. The FFT reduces the number of multiplications in the required N summations from an order of N^2 to that of $N \log_2(N)$, a considerable reduction in computing time.

Suppose we want to first value a European call option that matures at time T written on a single underlying stock price S_T. The characteristic function of the log stock price, $s_T = \log(S_T)$, is given by $\phi_T(u) = E\left[e^{ius_T}\right]$. In many cases, the characteristic function is known analytically and can be computed numerically. It has been shown that using Fourier transforms, the risk-neutral probability of a call with strike X finishing in-the-money is given by

$$P(S_T > X) = \Pi_2 = \frac{1}{2} + \frac{1}{\pi} \int_0^\infty \text{Re}\left[\frac{e^{-iu\ln(X)}\phi_T(u)}{iu}\right] du.$$

The delta of the option, denoted Π_1, is numerically computed as

$$\Pi_1 = \frac{1}{2} + \frac{1}{\pi} \int_0^\infty \text{Re}\left[\frac{e^{-iu\ln(X)}\phi_T(u-i)}{iu\phi_T(-i)}\right] du$$

where Re is the real part of the complex expression. Assuming no dividends and constant interest rates, the option price is given by the following:

$$C = S\Pi_1 - Xe^{-rT}\Pi_2$$

However, the FFT cannot be used to evaluate the integrals because the integrand is singular at the required evaluation point $u = 0$.

Madan and Carr (1999) developed a method to overcome this problem so that the FFT can be used given its efficient computational speed. Consider the initial call price $C_T(k) = \int_{-\infty}^{k} e^{-rT}(e^s - e^k)q_T(s)ds$, where $q_T(s)$ is the risk-neutral probability density of s and k is the log strike price. Because $C_T(k)$ tends to the initial stock price, S_0, as k tends to $-\infty$, the call price function is not square integrable. To obtain a square integrable function, we use a modified call price $c_T(k) = e^{\alpha k}C_T(k)$ for $\alpha > 0$. Positive values of α assist in the integrability of $c_T(k)$ over the negative log strike axis but poses problems over the positive log strike axis. Madan and Carr (1999) suggest that an upper bound for α can be computed from the analytical expression for the characteristic function (given later in equation (2.36)) and the required condition $E\left[S_T^{\alpha+1}\right] < \infty$.[16] Moreover, they suggest that $\frac{1}{4}$ of this upper bound serves as a good choice.

The Fourier transform of $c_T(k)$ is defined by:

$$\psi_T(v) = \int_{-\infty}^{\infty} e^{ivk} c_T(k) dk \tag{2.34}$$

By taking the inverse Fourier transform, we can obtain an analytical expression of $\psi_T(v)$ in terms of the characteristic function ϕ_T and thus compute call prices numerically:

$$C_T(k) = \frac{e^{-\alpha k}}{2\pi} \int_{-\infty}^{\infty} e^{ivk}\psi_T(v)dv \frac{e^{-\alpha k}}{\pi} \int_{-\infty}^{\infty} e^{ivk}\psi_T(v)dv \tag{2.35}$$

where the second equality holds because $C_T(k)$ is real, which implies that the function $\psi_T(v)$ is odd in its imaginary part and real in its even part.[17]

The expression for $\psi_T(v)$ is determined as follows:

$$\begin{aligned}
\psi_T(v) &= \int_{-\infty}^{\infty} e^{ivk} \int_{k}^{\infty} e^{\alpha k} e^{-rT}(e^s - e^k) q_T(s) ds \\
&= \int_{-\infty}^{\infty} e^{-rT} q_T(s) \int_{-\infty}^{s} (e^{s+\alpha k} - e^{(1+\alpha)k}) e^{ivk} dk ds \\
&= \int_{-\infty}^{\infty} e^{-rT} q_T(s) \left[\frac{e^{(\alpha+1+iv)s}}{\alpha + iv} - \frac{e^{(\alpha+1+iv)s}}{\alpha + 1 + iv} \right] ds \\
&= \frac{e^{-rT}\phi_T(v - (\alpha+1)i)}{\alpha^2 + \alpha - v^2 + i(2\alpha+1)v}
\end{aligned} \tag{2.36}$$

Call prices can be computed by substituting in (2.36) into (2.35).

Using the Trapezoid Rule for the integrand in (2.35) and setting $v_j = \eta(j-1)$, Madan and Carr use the following approximation for the call price:

$$C_T(k) = \frac{e^{-\alpha k}}{\pi} \sum_{j=1}^{N} e^{-iv_j k}\psi_T(v_j)\eta \tag{2.37}$$

where the effective upper limit of integration is $N\eta$. To use the FFT, Madan and Carr utilize a strike price spacing of λ and are interested in at-the-money call values $C(k)$, which correspond to k near 0. The values for k are given by

$$k_u = -b + \lambda(u-1), \text{ for } k = 1, \ldots, N \tag{2.38}$$

which gives a log range from $-b$ to b where $b = \frac{N\lambda}{2}$. Substituting (2.38) into (2.37) yields

$$C_T(k_u) \approx \frac{e^{-\alpha k_u}}{\pi} \sum_{j=1}^{N} e^{-iv_j(-\frac{N\lambda}{2} + \lambda(u-1))}\psi_T(v_j)\eta \tag{2.39}$$

Because $v_j = (j-1)\eta$, we can write (2.39) as

$$C_T(k_u) \approx \frac{e^{-\alpha k_u}}{\pi} \sum_{j=1}^{N} e^{-i\lambda\eta(j-1)(u-1)} e^{i\frac{N\lambda}{2}v_j}\psi_T(v_j)\eta \tag{2.40}$$

Moreover, from (2.33), we know

$$\lambda\eta = \frac{2\pi}{N}. \tag{2.41}$$

Madan and Carr then incorporate Simpson's rule weightings into the summation in order to obtain an accurate integration with larger values of η because larger η values lead to call prices at strike spacings relatively small so that more strike prices will lie in the desired region near the stock price.[18] Consequently, Madan and Carr write the analytical expression for the call price as

$$C_T(k_u) = \frac{e^{-\alpha k_u}}{\pi} \sum_{j=1}^{N} e^{-i\frac{2\pi}{N}(j-1)(u-1)} e^{i\frac{N\lambda}{2}v_j} \psi(v_j) \frac{\eta}{3}(3 + (-1)^j + \delta_{j-1}) \tag{2.42}$$

where δ_j is the Kronecker delta function that is unity for $j = 0$ and 0 otherwise. Because (2.42) is an exact application of the FFT, the FFT can be used.

2.11 FFT Implementation in Matlab

The following is the FFT implementation, written by Madan, Carr, and Liu (2000), in Matlab to generate the results in their paper:

optfftm.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% function y=optfftM(alpha, p0,f,r,t,par,chatfun, measure)
%
% Option Pricing using FFT
% Outputs:
%       K, strike price (nxt)
%       Opt, Option price (nxt)
% Inputs:
%       alpha-- dampening parameter
%       p0---  initial spot price (1x1)
%       f---     forward price  (tx1)
%       r--              riskfree rate  (tx1)
%       t--              maturity (tx1)
%       par--            other parameters regarding the cf of log rice
%  chatfun --CF of the log price logST/S_t or logST (S_t is scaled to 1)
%  measure=0 (default: spot: returns equal distance moneyness=K/S);
%            1: forward: Returns equal distance moneyness=K/F
%
%
% Written by Carr and Madan (1999), adopted by Liuren Wu
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [K,Opt]=optfftM2(alpha, p0,f,r,t,par, chatfun, measure,nn,eta)

if nargin==7
   measure=0;
   nn=4096;
   eta=0.05;
end
```

```
if nargin==8
   nn=4096;
   eta=0.05;
end
p=1;
fp=f./p0;

%eta=.05;
lambda=2.*pi./(nn.*eta);
b=nn.*lambda./2;
nn1=nn./2-round(.95./lambda); nn2=nn./2+round(.95./lambda);
%nn1=1918; nn2=2180;

mat=length(t);
u=[1:nn]';
vjv=eta*(u-1);
zz1=optpfftM(vjv,alpha,p,fp,r,t,par,chatfun,measure);

fac=exp(i*b*vjv).*(eta/3).*(3+(-1).^u);
fac(1)=eta/3;
nf=size(zz1,2);
zz=zz1.*repmat(fac,1,nf);

gg=fft(zz);
ku=-b+lambda*(u-1);
g=real(gg);
cpf=repmat((exp(-alpha*ku)/pi),1,mat).*g;
ss=exp(ku);


cpfo=cpf(nn1:nn2,:);
sso=ss(nn1:nn2);

nnf=length(sso);
if measure==0
   K=repmat(sso*p0,1,mat);
   Opt=cpfo*p0;
else
   K=sso*f';
   Opt=cpfo.*(repmat(f',nnf,1));
end

return
```

optpfftm.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% function y = optpfftM(v,alpha,p,f,r,t,par,chatfun,measure)
%
% Transform the cf of logST./S_t into the dampended cf
% alpha--the dampen factor
% Written by Carr and Madan(1999), adopted by Liuren Wu, Feb,2000
% Modified by Liuren Wu 11/12/2000 for fixed log(K/F)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function y = optpfftM(v,alpha,p,f,r,t,par,chatfun,measure);
```

```
  n=length(v(:)); lt=length(t);
  en=ones(n,1); et=ones(1,length(t));

  u=v-(alpha+1).*i;  %nx1
  u1=exp(-r.*t); %tx1
  u2=eval([chatfun '(u,p,f,r,t,par)']); %nxt
  u3=alpha.^2+alpha-v.^2+i.*v.*(2.*alpha+1); %nx1

  y=repmat(u1',n,1).*u2./repmat(u3,1,lt);

  if measure>0
     y=y.*(repmat(p./f',n,1)).^(repmat(i*u,1,lt));
  end

return
```

lscf.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% function y =lscf(u,p,f,r,t,par)
%
% CF of log S_T under log-stable model
%(Carr and Wu, 2003)
% par=[lambda, alpha];
% Liuren Wu, Feb, 2000 and after
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function y =lscf(u,p,f,r,t,par);

sig=par(1);
alph=par(2);
lambda= - sig.^alph.*sec(pi*alph/2);
iu=i*u;

psij=lambda*(iu-(iu).^alph);
u1=iu*log(f'); %nxt
u2=psij*t';
y=exp(u1-u2);

return

% and the following is the main function for testing the functions:
% test;
% Price options under the Black-Scholes model and the Log Stable model
% using FFT

close all; clear all; format short; format compact;

S=100;
T=[1 6 12]'/12;
nt=length(T);
r=zeros(nt,1); q=0;
F=S*exp((r-q).*T);

alpha=6; measure=0; eta=1;  nn=2^10;
```

```
chatfun=['bscf']; sigma=0.21; par=sigma^2;
[Kn,Cn1]=optfftM2(alpha,S,F,r,T,par,chatfun,measure,nn,eta);

figure(1)
plot(Kn(:,1), Cn1)
axis tight
chatfun=['lscf']; par=[0.15; 1.5];
[Kn,Cn2]=optfftM2(alpha,S,F,r,T,par,chatfun,measure,nn,eta);

figure(2)
plot(Kn(:,1), Cn2)
axis tight

return
```

bscf.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% function y = bscf(u,p,f,r,t,par)
%
% CF of log S_T under jump-diffusion
% dS/S=(r-q-\alpha g)dt +(e^k-1) dJ(\alpha), k\sim N(\omega,\eta)
%
% par=(sigma, alpha, omega, eta );
% CGMY model with maturity dependent C(t).
%
% Liuren Wu, Feb, 2000 and after
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function y =bscf(u,p,f,r,t,par);

v=par(1);

psid=0.5*(i*u+u.^2); %nx1

u1=i*u*log(f'); %nxt
u2=psid*(v.*t)';
y=exp(u1-u2);

return
```

Figure 2.7 shows a plot of the FFT solution for various maturities.

2.12 PATH-DEPENDENT VALUATION

To price a path-dependent derivative by Monte Carlo, meaning the payoff is dependent on the entire path taken by the underlying security, we need to estimate the conditional expectation. Suppose we want to price a European-style claim of the underlying process (i.e., the payoff)

$$f_T = F(\{S_t, 0 \leq t \leq T\})$$

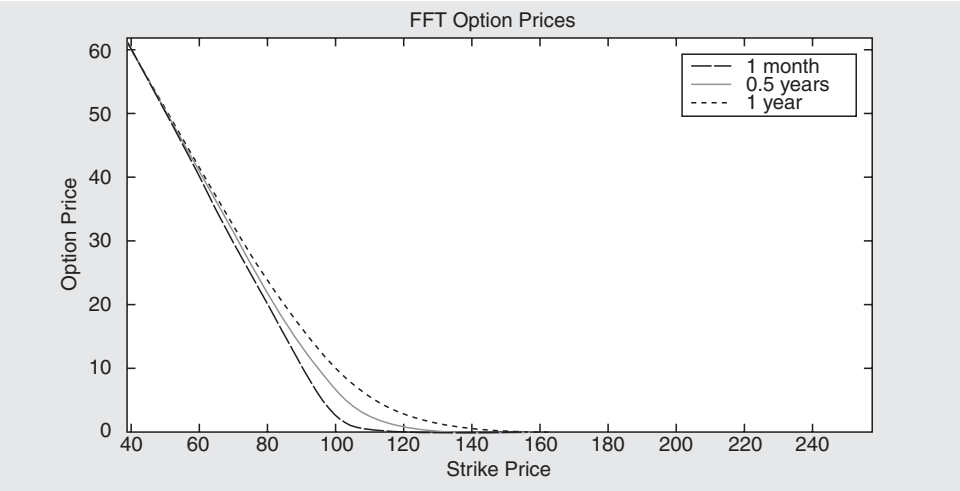


Figure 2.7

which depends on the entire path from 0 to T. The risk-neutral pricing formula gives the price of the security at time 0 as a discounted expectation:

$$f_o = e^{-rT} E^Q_{0,S} \left[F \left(\{ S_t, 0 \leq t \leq T \} \right) \right] \tag{2.43}$$

The expectation is calculated over all possible paths of the risk-neutral process from 0 to T started at $(S, 0)$. We can estimate this expectation as follows:

1. Divide the path into N time steps Δt and simulate M sample paths of the risk-neutral diffusion process.

2. Calculate the terminal payoff for each path. The payoff on the kth path, $\left\{ S^k_i, i = 0, 1, \ldots, N \right\}$, $k = 1, 2, \ldots, M$ ($i-$time counter on a give path; k - counts different paths):

$$F(S^k_0, S^k_1, \ldots, S^k_N)$$

3. Discount with the risk-free rate r.

4. Crude Monte Carlo estimate f^* of the security price is just an average of all the discounted payoffs over *M* sample paths generated:

$$f^*_0 = e^{-r\tau} \frac{1}{M} \sum_{k=1}^{M} F(S^k_0, S^k_1, \ldots, S^k_N) \tag{2.44}$$

f_0^* is an MC estimate of the N-dimensional integral

$$f_0^* = e^{-r\tau} \int_0^\infty \dots \int_0^\infty F(S_0, S_1, \dots, S_N) p^Q \cdot (S_N, t_N | S_{N-1}, t_{N-1}) \dots p^Q(S_1, t_1 | S, t) dS_1 \dots dS_N \tag{2.45}$$

where

$$p^Q(S_{i+1}, t_{i+1} | S_i, t_i) = \frac{1}{S_{i+1}\sqrt{2\pi\sigma^2\tau}} \exp\left\{ -\frac{\left(\ln\left(\frac{S_{i+1}}{S_i}\right) - \mu\tau\right)^2}{2\sigma^2\tau} \right\},$$

$$\mu = r - \frac{\sigma^2}{2}.$$

MC simulation is used to calculate these multidimensional integrals involving integration over multiple points on the path.

Suppose we want to price an Asian option, an option whose value depends on the average price of the underlying security over the life of the option, by simulation. We generate M sample paths $\left\{S_i^k, i = 0, 1, \dots, N\right\}$, $k = 1, 2, \dots, M$, index i counts time points on a given path, index k counts paths, $t_i = i\Delta t$, $\Delta t = \frac{\tau}{N}$, $S_{i+1} = S_i \exp\left\{\mu\Delta t + \varepsilon_{i+1}\sigma\sqrt{\Delta t}\right\}$. Compute the average price A_k for each path:

$$A_k = \frac{1}{N}\sum_{i=1}^{N} S_i^k$$

Estimate the option price:

$$f(S,t) = \frac{e^{-r\tau}}{M}\sum_{i=1}^{M} Max(A_k - X, 0) \tag{2.46}$$

To reduce variance, always use antithetic variates. Note that it is not necessary to save all the prices on the path and then compute the average price. The average price can be computed efficiently by the recurrent relation

$$A_{k,j+1} = \frac{1}{j+1}(jA_{k,j} + S_{j+1})$$

where A_j is the average price between time 0, t_0, and t_j. One updates the average at each time step. This saves computing time and memory.

The following is a Monte Carlo implementation to price an Asian option using geometric averaging on a stock with $S = 45$, $X = 42$, $\sigma = 20\%$, $r = 5.5\%$, $q = 1.5\%$, and $T = 1$ using $M = 1000$ simulations with $N = 10$ time steps (equally spaced fixing times) so that $t_i - t_{i-1} = T/N = \Delta t$ for all $i = 1, \dots, N$. We make the following call:

```
void main()
{

   MonteCarlo mc;

   double price  mc.calcMCGAsianPrice(45,42,0.20,0.055,0.015,1,'C');cout <<
"Geometric Asian price = " <<  price << endl;
}
```

The function implementation is:

MCPricer_calcMCGAsianPrice.cpp

```
/*************************************************************************
 calcMCGAsianPrice: computes the price of a geometric Asian option using
        Monte Carlo simulation
 [in]: double price: initial stock price
       double strike  : strike price
       double vol     : volatility
       double rate    : risk free rate
       double div     : dividend yield
       double T       : time to maturity
       char type      : (C)all or (P)ut
       long M         : number of simulations
       long N         : number of time steps
  [out] double : price of geometric Asian option
*************************************************************************/
vector<double> MCPricer::calcMCGAsianPrice(double price, double strike,
double vol, double rate, double div, double T, char type, long M, long N)
{
   // initialize variables
        double G = 0.0;            // price of geometric average
                                   // Asian option
        int i, j;
        double deviate;            // normal deviate
        double S = 0.0;            // stock price
        double sum = 0.0;          //  sum of payoffs
        double sum2  = 0.0;        //  sum of squared payoffs
        double product = 0.0;
        double payoff = 0.0;       // option payoff
        double deltat = 0.0;       // step size
        double val = 0.0;
        vector<double> value;
        double stddev, stderror = 0.0;
        double dt = (double) T/N;  // compute change in step size
        double mu = rate - div - 0.5*vol*vol;// compute drift
    mrng.sgenrand(unsigned(time(0)));    // initialize rng

        // for each simulation
        for (i = 0; i <= M; i++)
        {
                S = price;
                product = 1;

                for (j = 0; j < N; j++)
```

```
                {
                        deviate = mrng.genrand();  // generate random
                                                   // deviate
                        S  = S*exp(mu*dt + vol*sqrt(dt)*deviate);
                        product *= S;
                }

                // compute geometric average
                G = pow(product,(double)1/N);
                if (type == 'C')
                        payoff = max(G - strike,0);
                else
                        payoff = max(strike - G,0);

                sum += payoff;
                sum2 += payoff*payoff;
        }

        val = exp(-rate*T)*(sum/M);
        value.push_back(val);

        stddev = sqrt((sum2 - sum*sum/M)*exp(-2*rate*T)/(M-1));
        value.push_back(stddev);

        stderror = stddev/sqrt(M);
        value.push_back(stderror);

    return value;
}
```

The price of the geometric average Asian call is \$5.76, with a standard deviation of 5.00 and a standard error of 0.50. We now price an arithmetic Asian price option using the same values for the geometric average Asian call.

MCPricer_calcMCAAsianPrice.cpp

```
vector<double> MCPricer::calcMCAAsianPrice(double price, double strike,
   double vol, double rate, double div, double T, char type, long M,
   long N)
{
        // initialize variables
        double A = 0.0;                  // arithmetic average
        double mu = 0.0;                 // drift
        int i, j;
        double deviate;                  // normal deviate
        double stddev = 0.0;             // standard deviation
        double stderror = 0.0;           // standard error
        double S = 0.0;                  // stock price
        double sum = 0.0;
        double sum1 = 0.0;
        double sum2 = 0.0;
        double payoff = 0.0;
        double val = 0.0;
        vector<double> value;
        double dt = (double) T/N;           // compute step size
```

```
        mu = rate - div - 0.5*vol*vol;          // compute drift
    mrng.sgenrand(unsigned(time(0)));   // initializer RNG

        // for each simulation
        for (i = 0; i <= M; i++)
        {
                S = price;
                sum1 = 0;

                for (j = 0; j < N; j++)
                {
                    deviate = mrng.genrand();
                    S  = S*exp(mu*dt + vol*sqrt(dt)*deviate);
                    sum1 += S;
                }
                A = sum1/N;

                if (type == 'C')
                      payoff = max(A - strike, 0);
                else
                     payoff = max(strike - A,0);

                 sum += payoff;
                 sum2 += payoff*payoff;
        }
        val = exp(-rate*T)*(sum/M);
        value.push_back(val);

        stddev = sqrt((sum2 - sum*sum/M)*exp(-2*rate*T)/(M-1));
        value.push_back(stddev);

        stderror = stddev/sqrt(M);
        value.push_back(stderror);

    return value;
}
```

The price of the arithmetic average is approximately \$5.90, with a standard deviation of 5.134 and a standard error of 0.514. It is not surprising that the arithmetic average is higher than the geometric average price because the arithmetic average of a series of values is always greater than or equal to the geometric average of a series of values.

As discussed, the geometric average option makes a good control variate for the arithmetic average option. It lowers the standard deviation, and thus standard error, of the estimate. The following is an implementation for pricing an arithmetic average option using a geometric average control variate:

MCPricer_calcMCAAsianGCVPrice.cpp

```
vector<double> MCPricer::calcMCAAsianGCV(double price, double strike,
double vol, double rate, double div, double T, char type, long M, long N)
{
        int i, j;                                   // counters
        double geo = 0.0;       // geometric average
        double ave = 0.0;       // arithmetic average
```

```
double mu = 0.0;         // drift
double stddev = 0.0;     // standard deviation
double stderror = 0.0;   // standard error
double portfolio = 0.0;
double deviate;          // standard deviate
double S = 0.0;          // stock price
double sum = 0.0;        // sum of payoffs
double sum1 = 0;         // sum of squared payoffs
double product = 0.0;
double payoff = 0.0;     // option payoff
double dt = 0.0;         // time step
double val = 0.0;
vector<double> value;    // stores value, SD, SE

mrng.sgenrand(unsigned(time(0)));  // initialize RNG

dt = (double) T/N;                 // step size
mu = rate - div - 0.5*vol*vol;     // drift

// for each simulation
for (i = 0; i <= M; i++)
{
        S = price;
        product = 1;
        sum = 0;
        sum1 = 0;

        for (j = 0; j < N; j++)
        {
                deviate = mrng.genrand();
                S  = S*exp(mu*dt + vol*sqrt(dt)*deviate);
                sum = sum + S;
                product *= S;
        }
        // calculate arithmetic average
        ave = sum/N;
        // calculate geometric average
        geo = pow(product,(double)1/N);

        if (type == 'C')
                payoff = max(0, (ave - strike) - (geo - strike));
        else
                payoff = max(0, (strike - ave) - (strike - geo));

        sum += payoff;
        sum1 += payoff*payoff;
}

portfolio =  exp(-rate*T)*(sum/M);
value = calcMCGAsianPrice(price,strike,vol,rate,div,T,'C',M,N);
val = portfolio + value[0];
value.push_back(val);

stddev = sqrt((sum1 - sum*sum/M)*exp(-2*rate*T)/(M-1));
value.push_back(stddev);

stderror = stddev/sqrt(M);
```

```
        value.push_back(stderror);

        return value;
}
```

The arithmetic price is approximately \$5.90, with a standard deviation of 0.185 and a standard error of 0.019. Table 2.2 summarizes the Monte Carlo results of the Asian option using different methods and using $M = 1000$ simulations and $N = 10$ time steps per path. Note how small the standard error is using the control variate technique than the arithmetic average Monte Carlo without it.

Table 2.2

Monte Carlo	Estimate	Standard Deviation	Standard Error
Arithmetic Average	\$5.898	5.134	0.514
Geometric Average	\$5.760	5.000	0.500
Geometric Control Variate	\$5.898	0.185	0.019

2.13 Monte Carlo Pricing of Asian Currency Option in Matlab

Here is an implementation in Matlab to value an Asian currency option.

AsianMC.m

```
NRepl = 1000;
S0 = 1.1692;    % spot exchange rate
rf = 0.0124;    % foreign risk rate
rr = 0.0221;    % domestic risk free rate
r = rf - rr;
T = 1;
sigma = 0.100336;   % volatility
NSamples = 51;

function [P,CI] = AsianMC(S0,X,r,T,sigma,NSamples,NRepl)

randn('state',sum(100*clock))
tic

NSteps = 51
dd = NSteps/NSamples
X = 1.1700;
Payoff = zeros(NRepl,1);
for i=1:NRepl
   randn('state',sum(100*clock))
   Path=AssetPaths1(S0,r,sigma,T,NSteps,1);
   Payoff(i) = max(0, mean(Path(1:dd:(NSteps))) - X);
end

[P,aux,CI] = normfit( exp(-rf*T) * Payoff)
```

```
%hist(Payoff,100)
toc

function SPaths=AssetPaths1(S0,mu,sigma,T,NSteps,NRepl)

 dt = T/NSteps;
 nudt = (mu-0.5*sigma^2)*dt;
 sidt = sigma*sqrt(dt);
 Increments = nudt + sidt*randn(NRepl, NSteps);
 LogPaths = cumsum([log(S0)*ones(NRepl,1) , Increments] , 2);
 SPaths = exp(LogPaths);
```

The output is as follows:

```
P =

   0.028890

aux =

   0.044193

CI =

   0.026148
   0.031633

Elapsed time is 1.680099 seconds.
```

2.14 Finite Difference Methods

We discuss numerical methods known as finite difference methods for pricing derivatives by approximating the diffusion process that the derivative must follow. Finite difference methods are a means for generating numerical solutions to partial differential equations and linear complementary (free boundary) problems such as those used to price American options. Finite difference schemes are useful for valuation of derivatives when closed-form analytical solutions do not exist or for solutions to complicated multi-factor (multi-dimensional) models. By discretizing the continuous-time partial differential equation that the derivative security must follow, it is possible to approximate the evolution of the derivative and, therefore, the present value of the security.

2.15 Explicit Difference Methods

Binomial and trinomial trees work well for pricing European and American options. However, there are alternative numerical methods that can be used to value these standard options, as well as more complex derivatives with nonlinear payoffs such as exotic options. Finite difference methods are used to price derivatives by solving the differential equation in conjunction with the initial asset price condition and boundary value condition(s)—i.e.

payoffs—that the derivative must also satisfy. The differential equation is converted into a system of difference equations that are solved iteratively.

Consider the Black-Scholes (BS) PDE

$$\frac{\partial f}{\partial t} + (r-q)S\frac{\partial f}{\partial S} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 f}{\partial S^2} = rf \tag{2.47}$$

subject to the payoff condition $f(S_T, T) = (S_T - X)^+$. We can extend the trinomial tree approach by creating a rectangular grid or lattice by adding extra nodes above and below the tree so that we have $2N_j + 1, N_j \geq N$, nodes at every time step i rather than $2i + 1$. In a similar manner to trinomial trees, when implementing finite difference methods, we divide space and time into discrete intervals, Δt and Δx, which generates the lattice.

We add boundary conditions to the grid, which determines option prices as a function of the asset price for high and low values so that $\frac{\partial f}{\partial S} = 1$ for S large and $\frac{\partial f}{\partial S} = 0$ for S small. We can simplify the BS PDE by replacing the PDE with finite differences. Thus, we can discretize the PDE to develop a numerical finite-difference scheme. First, we simplify the PDE; let $x = \ln(S)$ so that

$$\frac{\partial f}{\partial t} + \mu\frac{\partial f}{\partial x} + \frac{1}{2}\sigma^2 \frac{\partial^2 f}{\partial x^2} = rf$$

where $\mu = r - q$. To get rid of the rf term on the LHS, let $u(x, t)$ be a new function: $u(x, t) = e^{r(T-t)} f(e^x, t)$. u is a forward price of the option f and satisfies the PDE:

$$\frac{1}{2}\sigma^2 \frac{\partial^2 u}{\partial x^2} + \mu\frac{\partial u}{\partial x} = -\frac{\partial \mu}{\partial t} \tag{2.48}$$

We will discretize this PDE by taking the central difference of the state variable, x, and the forward difference of the time variable t. Denote $u_{i,j} = u(x_j, t_i)$, $t_i = i\Delta t$, and $x_j = j\Delta x$. Substituting the finite differences into the PDE:

$$\begin{aligned}\frac{1}{2}\sigma^2\left(\frac{u_{i+1,j+1} - 2u_{i+1,j} + u_{i+1,j-1}}{\Delta x^2}\right) + \mu\left(\frac{u_{i+1,j+1} - u_{i+1,j-1}}{2\Delta x}\right) = \\ -\left(\frac{u_{i+1,j} - u_{i,j}}{\Delta t}\right)\end{aligned} \tag{2.49}$$

Rearranging terms, we have the recurrent relation for the forward option price

$$u_{i,j} = \widetilde{p}_u u_{i+1,j+1} + \widetilde{p}_m u_{i+1,j} + \widetilde{p}_d u_{i+1,j-1} \tag{2.50}$$

where

$$\begin{aligned}\widetilde{p}_u &= \frac{\sigma^2\Delta t}{2\Delta x^2} + \frac{\mu\Delta t}{2\Delta x} \\ \widetilde{p}_m &= 1 - \frac{\sigma^2\Delta t}{\Delta x^2} \\ \widetilde{p}_d &= \frac{\sigma^2\Delta t}{2\Delta x^2} - \frac{\mu\Delta t}{2\Delta x}\end{aligned} \tag{2.51}$$

Note that $\widetilde{p}_u + \widetilde{p}_m + \widetilde{p}_d = 1$. Denote $\alpha = \frac{\Delta t}{(\Delta x)^2}$ and $\beta = \frac{\mu \Delta t}{\Delta x}$. We can rewrite (2.51) as $\widetilde{p}_u = \frac{\sigma^2}{2}(\alpha + \beta)$, $\widetilde{p}_m = 1 - \sigma^2 \alpha$, and $\widetilde{p}_d = \frac{\sigma^2}{2}(\alpha - \beta)$.

The relationship of trinomial trees to finite-difference discretizations of the BS PDE can be seen as follows. Substitute the present value of the option $f_{i,j} = e^{-r(T-t_i)} u_{i,j}$ into (2.50). We arrive at the backward induction relationship:

$$f_{i,j} = e^{-r\Delta t} \left(\widetilde{p}_u f_{i+1,j+1} + \widetilde{p}_m f_{i+1,j} + \widetilde{p}_d f_{i+1,j-1} \right) \tag{2.52}$$

This is similar to the backward induction methodology in a trinomial tree. This is equivalent to the discounted expectation of the forward option price (under a risk-neutral measure). Thus, we have shown that the explicit finite difference scheme is equivalent to approximating the diffusion process by a discrete trinomial process.

Figure 2.8 is an explicit finite difference discretization.

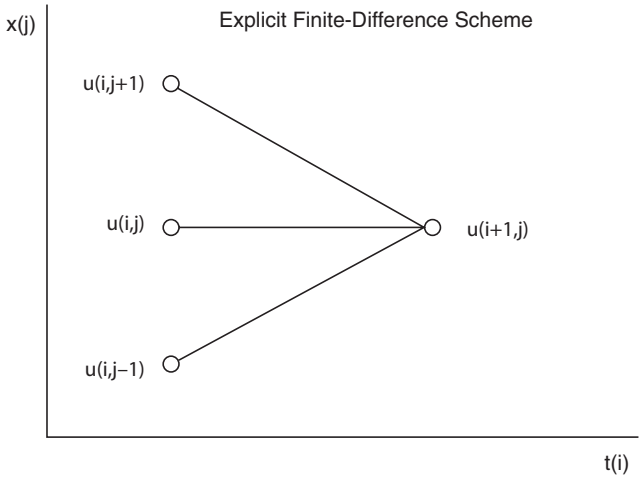


Figure 2.8

The following Matlab code is an implementation of the explicit finite difference scheme.

ExplicitFiniteDifference.m

```
%%  Explicit Finite Difference Scheme
%%  Author: Chetan Jain & Jim Carson
%%  This code implements the explicit finite difference scheme for the
%%  following PDE:
%%  ut = (1 + x^2)uxx,  -1<x<1, 0<t<1
%%  u(x, 0) = x^4, -1 <= x <= 1
%%  u(-1, t) = u(1, t) = 1, 0<= t <= 1

clear;
format long;

%%  Enter k
```

```
k = 4;

%%  Enter alpha
alpha = .69;

%% Initialize x interval
delta_x = 2^(-k);
LeftEndpoint = -1;
RightEndpoint = 1;
InitialNminus = LeftEndpoint/delta_x;
InitialNplus  = RightEndpoint/delta_x;

%%  Initialize t interval
delta_t = alpha*(delta_x^2);
T = 1;
M = T/delta_t;

%%  Ensure positive indexes for Matlab and same width as x interval
Nminus = 1;
Nplus = InitialNplus - InitialNminus + Nminus;

%%  Boundary condition w.r.t. x
for (i = 1:Nplus)
    n = InitialNminus + i - 1;
    oldu(i) = (n*delta_x)^4;
end

%%  m loop
for (m=1:M)

    %%  Boundary condition for w.r.t. t
    newu(Nminus) = 1;
    newu(Nplus) = 1;

    %%  Get u_m+1 from u_m
    for (i = 2:Nplus - 1)
        n = InitialNminus + i - 1;
        newu(i) = alpha*(1+(n*delta_x)^2)*oldu(i+1) +
                    (1-2*alpha*(1+(n*delta_x)^2))*oldu(i) +
                    alpha*(1+(n*delta_x)^2)*oldu(i-1);
    end

    %%  Transplant memory
    oldu(:) = newu(:);

end

%%  Make plot
plot(oldu);


%%  Use linear interpolation to find the requested values

    values(Nminus) = oldu(Nminus);    %%  Initialize values() with
                                      %%  boundary condition
    w = 1;
```

```
for (i = 1:Nplus)
    n = InitialNminus + i - 1;
    if (n*delta_x >= InitialNminus*delta_x + w*0.2)
        values(w+1) = oldu(i) + (oldu(i-1)-oldu(i))*(n*delta_x - w*0.2 -
                        InitialNminus*delta_x)/delta_x;
        w = w + 1;
    end

end
```

2.16 Explicit Finite Difference Implementation in C++

The following is an implementation of an explicit finite difference scheme.[19] N is the total number of time steps, where each time step is $\Delta t = T/N$ and $|M|$ is the total number of state movements (in either direction from the initial state at time 0) where the state variable is $S_j, j = -M \ldots -1, 0, 1, \ldots M$.

```
/*************************************************************************
 explicitDiffAmerican: values an American option using the
   explicit difference method
        [in]: double price : asset price
              double strike : strike price
              double vol : volatility
              double rate: risk-free rate
              double div: dividend yield
              double T: maturity
              int N: number of time steps
              int M: number of space steps
              char type: (C)all or (P)ut
              char bc: boundary conditions (D)irichlet or (N)eumann
        [out] : option price
*************************************************************************/
double ExplicitDiffMethod::explicitDiffAmerican(double price, double
strike, double vol, double rate, double div, double T, int N, int M,
char type, char bc)
{
      int i, j;
      double dt = T/N;
      double drift = rate - div - 0.5*(vol*vol);
      double dx = vol*sqrt(3*dt/2);
      double pu, pm, pd;
      Array2D<double> C(N,M);   // stores option prices
      Array2D<double> S(N,M);   // stores stock prices

      pu = (vol*vol*dt)/(2*dx*dx) + (drift*dt)/(2*dx);
      pm = 1.0 - (vol*vol*dt)/(dx*dx);
      pd = (vol*vol*dt)/(2*dx*dx) - (drift*dt)/(2*dx);

      // initialize asset prices at maturity
      for (j = -M; j <= M; j++)
          S[N][j] = price*exp(j*dx);
```

```
if (type == 'C')
{
      for (j = -M; j <= M; j++)
      {
            C[N][j] = max(S[N][j] - strike,0);
      }
      // boundary conditions for high and low asset prices
      for (i = 0; i < N; i++)
      {
            if (bc == 'D')
            {
                  C[i][-M] = 0.0;
                  C[i][M] = max(S[N][M] - strike,0);
            }
            else
            {
                  C[i][M] =  C[i][M-1] + (S[i][M] - S[i][M-1]);
                  C[i][-M] = C[i][-M+1];
            }
      }

      for (i = N-1; i >= 0; i--)
      {
            for (j = M-1; j >= -(M-1); j--)
            {
                  C[i][j] =  pu*C[i+1][j+1] + pm*C[i+1][j] +
      pd*C[i+1][j-1];
                  C[i][j] = max(S[N][j] - strike, C[i][j]);
            }
      }
}
else //if (type == 'P')
{
      // boundary conditions for high and low asset prices
      for (i = 0; i < N; i++)
      {
            C[i][0] = strike;
            C[i][M]  = 0;
      }
      for (j = -M; j <= M; j++)
      {
            C[N][j] = max(strike - S[N][j],0);
      }

      for (j = -M; j <= M; j++)
      {
            C[N][j] = max(strike - S[N][j],0);
      }
      // boundary conditions for high and low asset prices
      for (i = 0; i < N; i++)
      {
            if (bc == 'D')
            {
                  C[i][-M] = strike;
                  C[i][M] = max(0,strike - S[N][j]);
            }
```

```
                else  // Neumann bc
                {
                      C[i][M] = C[i][M-1];
                      C[i][-M] = C[i][-M+1] + (S[i][-M] - S[i][-M+1]);
                }
          }

          for (i = N-1; i >= 0; i--)
          {
                for (j = M-1; j >= -M; j--)
                {
                      C[i][j] =  pu*C[i+1][j+1] + pm*C[i+1][j] +
                        pd*C[i+1][j-1];
                      C[i][j] = max(strike - S[N][j], C[i][j]);

                 }
          }

     }
     return C[0][0];
}
```

Suppose we want to price an ATM American-style call option with $S = 50$, $X = 50$, $\sigma = 0.20$, $r = 0.06$, $q = 0.03$, $N = 4$, $M = 5$, and $T = 1$. Figure 2.9 is a lattice generated using the `explicitDiffAmerican` method.

The value of the call option marching backward from the maturity date $T = 1$ is \$4.76. It can be shown that as the number of time steps $N \to \infty$, so the explicit difference schemes will converge to the price using a trinomial diffusion process.

		t	1.0	0.75	0.50	0.25	0.0
S	j	i	**0**	1	2	3	4
72.20	**3**		22.20	22.20	22.20	22.20	22.20
63.88	**2**		15.47	15.09	14.65	14.36	13.88
56.51	**1**		9.39	8.70	7.97	6.94	6.51
50.00	**0**		**4.76**	4.03	3.13	2.24	0.00
44.24	**–1**		1.91	1.33	0.77	0.00	0.00
39.14	**–2**		0.55	0.26	0.00	0.00	0.00
34.63	**–3**		0.00	0.00	0.00	0.00	0.00

Figure 2.9

2.17 IMPLICIT DIFFERENCE METHOD

If the backward difference, $\frac{u_{i,j}-u_{i-1,j}}{\Delta t}$, is used instead of the forward difference for the time derivative, $\frac{\partial f}{\partial t}$ in (2.49), we will get an implicit difference scheme in which $u_{i+1,,j}$ is implicitly dependent on $u_{i,j+1}$, $u_{i,j}$, and $u_{i,j-1}$.

$$u_{i+1,j} = \widetilde{p}_u u_{i,j+1} + \widetilde{p}_m u_{i,j} + \widetilde{p}_d u_{i,j-1} \tag{2.53}$$

where the probabilities $\widetilde{p}_u$, $\widetilde{p}_m$, and $\widetilde{p}_d$ are defined in (2.51). If we substitute the present value of the option $f_{i,j} = e^{-r(T-t_i)}u_{i,j}$ into (2.53), we get the risk-neutral expected value:

$$f_{i+1,j} = e^{-r\Delta t}(\widetilde{p}_u f_{i,j+1} + \widetilde{p}_m f_{i,j} + \widetilde{p}_d f_{i,j-1}) \tag{2.54}$$

If $f_{i,j}$ is a put option, then when the stock price is zero, we get the boundary condition:

$$f_{i,-M} = X \quad i = 0, 1, \ldots, N \tag{2.55}$$

The value of the option tends to zero as the stock price tends to infinity. We may use the boundary condition:

$$f_{i,M} = 0 \quad i = 0, 1, \ldots, N \tag{2.56}$$

The value of the put at maturity (time T) is:

$$f_{N,j} = \max[X - S_j, 0] \quad j = -M, \ldots, -1, 0, 1, \ldots M \tag{2.57}$$

The following figure, Figure 2.10, is an implicit finite-difference discretization. Equations (2.55), (2.56), and (2.57) define the value of the put option along the boundaries of

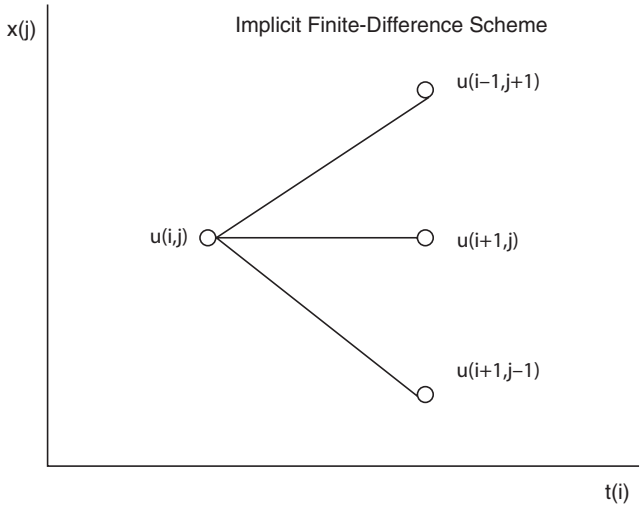


Figure 2.10

the grid. To solve for the value of f at all other points, we use equation (2.54). First, the points corresponding to $T - \Delta t$ are solved. Equation (2.54) with $i = N - 1$ yields $2M - 1$ linear simultaneous equations:

$$f_{N,j} = e^{-r\Delta t}(\widetilde{p}_u N-1,j+1 + \widetilde{p}_m f_{N-1,j} + \widetilde{p}_d f_{N-1,j-1}) \quad j = -M+1, \ldots, M+1 \tag{2.58}$$

Unlike the explicit finite difference method, each equation cannot be solved individually for the option values at time step i. These equations must be considered with the boundary conditions. The system of equations can be rewritten as a tridiagonal matrix form. We can rewrite (2.53) as

$$\begin{bmatrix} \widetilde{p}_m & \widetilde{p}_d & 0 & \ldots & \ldots & \ldots & 0 \\ \widetilde{p}_u & \widetilde{p}_m & \widetilde{p}_d & 0 & \ldots & \ldots & 0 \\ 0 & \widetilde{p}_u & \widetilde{p}_m & \widetilde{p}_d & 0 & \ldots & 0 \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & \ldots & 0 & \widetilde{p}_u & \widetilde{p}_m & \widetilde{p}_d & 0 \\ 0 & \ldots & \ldots & 0 & \widetilde{p}_u & \widetilde{p}_m & \widetilde{p}_d \\ 0 & \ldots & \ldots & \ldots & 0 & \widetilde{p}_u & \widetilde{p}_d \end{bmatrix} \begin{bmatrix} u_{i,-M} \\ u_{i,-M+1} \\ u_{i,-M+2} \\ \ldots \\ u_{i,M-2} \\ u_{i,M-1} \\ u_{i,M} \end{bmatrix} = \begin{bmatrix} u_{i+1,-M} \\ u_{i+1,-M+1} \\ u_{i+1,-M+2} \\ \ldots \\ u_{i+1,M-2} \\ u_{i+1,M-1} \\ u_{i+1,M} \end{bmatrix} \tag{2.59}$$

where the (probability) elements of the matrix in (2.59) are given in (2.51).

Let β_U be the upper boundary (for a put $\beta_U = 0$ if S is much larger than X) and β_L be a lower boundary (for a put $\beta_L = X$ if $S = 0$) when the asset price reaches the high and low points, respectively. Then $u_{i+1,-M} = \beta_L$ and $u_{i+1,M} = \beta_U$. However, we will be using the partial derivatives (Neumann boundary conditions) of the option price at the boundaries when we use finite difference schemes. Note that $u_{N,j} = f(S) = Max(S - X, 0)$ for a call and $Max(X - S, 0)$ for a put option, $j = -M, \ldots, M$. We can rewrite (2.59) as

$$\mathbf{M}\mathbf{u}^i = \mathbf{b}^{i+1} \quad i = 0, 1, \ldots, N-1 \tag{2.60}$$

which can be solved for $\mathbf{u}^i$ because $\mathbf{M}$ is nonsingular—i.e., it can be inverted:

$$\mathbf{u}^i = \mathbf{M}^{-1}\mathbf{b}^{i+1} \tag{2.61}$$

where $\mathbf{M}^{-1}$ is the inverse of M. Making use of the boundary conditions, we can solve (2.61) iteratively starting at time $i = N - 1$ and solving for $\mathbf{u}^{N-1}$. We know $\mathbf{b}^N$ because they are given by the boundary conditions in (2.55), (2.56), and (2.57). After we solve for

$\mathbf{u}^{N-1}$ at time $i = N-1$, we can use it to solve for $\mathbf{u}^{N-2}$ at time $i = N-2$ because $\mathbf{b}^{N-1} = \mathbf{u}^{N-1}$. Thus, we can solve for each $\mathbf{u}^i$, $i = N-1, \ldots, 0$, sequentially working backward starting from time $i = N-1$ until we solve for $\mathbf{u}^0$, which gives us a vector solution of option prices.

Because $\mathbf{M}$ is tridiagonal—i.e., only the diagonal, super-diagonal, and sub-diagonal entries are nonzero—we do not have to store all the zeros, but just the non-zero elements. The inverse of $\mathbf{M}$, $\mathbf{M}^{-1}$ is not tridiagonal and requires a lot more storage than M.

We can rewrite the system of equations in (2.59) as

$$\begin{bmatrix} 1 & p^*_{-M,d} & 0 & \ldots & \ldots & \ldots & 0 \\ 0 & 1 & p^*_{-M+1,d} & 0 & \ldots & \ldots & 0 \\ 0 & 0 & 1 & p^*_{-M+2,d} & 0 & \ldots & 0 \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & \ldots & 0 & 0 & 1 & p^*_{M-2,d} & 0 \\ 0 & \ldots & \ldots & 0 & 0 & 1 & p^*_{M-1,d} \\ 0 & \ldots & \ldots & \ldots & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} u_{i,-M} \\ u_{i,-M+1} \\ u_{i,-M+2} \\ \ldots \\ u_{i,M-2} \\ u_{i,M-1} \\ u_{i,M} \end{bmatrix} = \begin{bmatrix} u^*_{i+1,-M} \\ u^*_{i+1,-M+1} \\ u^*_{i+1,-M+2} \\ \ldots \\ u^*_{i+1,M-2} \\ u^*_{i+1,M-1} \\ u^*_{i+1,M} \end{bmatrix} \tag{2.62}$$

where

$$p^*_{-M,d} = \frac{\widetilde{p}_d}{\widetilde{p}_m}, \quad p^*_{j+1,d} = \frac{\widetilde{p}_d}{\widetilde{p}_m - \widetilde{p}_u p^*_{j,d}} \quad j = -M+1, \ldots, M-2 \tag{2.63}$$

and

$$u^*_{i+1,-M} = \frac{u_{i+1,-M}}{\widetilde{p}_m}, \quad u^*_{i+1,j+1} = \frac{u_{i+1,j+1} - \widetilde{p}_u u^*_{i+1,j}}{\widetilde{p}_m - \widetilde{p}_u p^*_{j,d}} \qquad j = -M+1, \ldots, M-1 \tag{2.64}$$

Solving (2.63) and (2.64) from bottom to top, we get:

$$u_{i,M} = u^*_{i+1,M}, \quad u_{i,j} = u^*_{i+1,j} - p^*_{j,d} u_{i,j+1} \qquad i = 0 \ldots N-1,\ j = -M+1, \ldots, M-1 \tag{2.65}$$

We could also use an LU decomposition to solve for $\mathbf{u}^i$ in (2.60) without having to invert $\mathbf{M}$.

2.18 LU DECOMPOSITION METHOD

In an LU decomposition, we decompose the matrix $\mathbf{M}$ into a product of a lower triangular matrix $\mathbf{L}$ and an upper triangular matrix $\mathbf{U}$, namely $\mathbf{M} = \mathbf{LU}$, of the form:

$$\begin{bmatrix} p_m & p_d & 0 & \dots & \dots & \dots & 0 \\ p_u & p_m & p_d & 0 & \dots & \dots & 0 \\ 0 & p_u & p_m & p_d & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & 0 & p_u & p_m & p_d & 0 \\ 0 & \dots & \dots & 0 & p_u & p_m & p_d \\ 0 & \dots & \dots & \dots & 0 & p_u & p_d \end{bmatrix} =$$

$$\begin{bmatrix} 1 & 0 & \dots & \dots & & 0 & \\ \ell_M & 1 & & & & & \\ 0 & & \ell_{M-1} & 1 & & & \\ \dots & & & \ell_{M-2} & 1 & & \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ & & & & & & \\ 0 & & & & & \ell_{-M+1} & 1 \end{bmatrix} \cdot \tag{2.66}$$

$$\begin{bmatrix} y_M & z_M & 0 & \dots & \dots & 0 & \\ 0 & y_{M-1} & z_{M-1} & & & & \\ 0 & & y_{M-2} & z_{M-2} & & & \\ 0 & & \dots & \dots & \dots & 0 & \\ \dots & \dots & \dots & \dots & \dots & \dots & \\ & & & & & z_{-M+1} & \\ 0 & & & 0 & 0 & y_{-M} & \end{bmatrix}$$

In order to determine the quantities ℓ_j, y_j, and z_j, $j = -M+1, \dots, M-1$, we multiply the two matrices on the right-hand side of (2.66) and equation the result to the left-hand side. After some simple calculations:

$$\begin{aligned} y_M &= p_m \\ y_j &= p_m - \frac{p_u p_d}{y_{j-1}} && j = -M+1, \dots, M-1 \\ z_j &= -p_d,\ \ell_j = -\frac{p_u}{y_j} && j = -M+1, \dots, M-1 \end{aligned} \tag{2.67}$$

The only quantities we need to calculate and store are the y_j. $j = -M+2, \dots, M-1$. We can rewrite the original problem in $\mathbf{M}\mathbf{u}^i = \mathbf{b}^{i+1}$ as $\mathbf{L}(\mathbf{U}\mathbf{u}^i) = \mathbf{b}^{i+1}, i = 0, \dots, N-1$, which may be broken down into two simpler subproblems:

$$\mathbf{L}\mathbf{q}^i = \mathbf{b}^i,\ \mathbf{U}\mathbf{u}^{i+1} = \mathbf{q}^i \tag{2.68}$$

where $\mathbf{q}^i$ is an intermediate vector. We eliminate the ℓ_j from $\mathbf{L}$ and the z_j from $\mathbf{U}$ using (2.67); the solution procedure is to solve two subproblems:

$$\begin{bmatrix} 1 & 0 & \dots & \dots & 0 & & \\ -\frac{p_u}{y_M} & 1 & & & & & \\ 0 & & -\frac{p_u}{y_{M-1}} & 1 & & & \\ \dots & & & & & & \\ \dots & \dots & \dots & \dots & \dots & \dots & \\ 0 & \dots & & & & -\frac{p_u}{y_{-M+1}} & 1 \end{bmatrix} \begin{bmatrix} q_{i,M} \\ q_{i,M-1} \\ \dots \\ \dots \\ q_{i,-M+1} \\ q_{i,-M} \end{bmatrix} = \begin{bmatrix} b_{i,M-1} \\ b_{i,M-2} \\ \dots \\ \dots \\ b_{i,-M+2} \\ b_{i,-M+1} \end{bmatrix} \tag{2.69}$$

and

$$\begin{bmatrix} y_M & p_d & \dots & \dots & 0 \\ & y_{M-1} & p_d & & 0 \\ 0 & & & & \\ \dots & \dots & & & \\ \dots & & & & \\ & & & & \\ \dots & \dots & \dots & \dots & p_d \\ 0 & \dots & & & y_{-M} \end{bmatrix} \begin{bmatrix} u_{i,M} \\ u_{i,M-1} \\ \dots \\ \dots \\ u_{i,-M+1} \\ u_{i,-M} \end{bmatrix} = \begin{bmatrix} q_{i,M} \\ q_{i,M-1} \\ \dots \\ \dots \\ q_{i,-M+1} \\ q_{i,-M} \end{bmatrix} \tag{2.70}$$

The intermediate quantities $q_{i,j}$ are found by forward substitution. We can read off the value of $q_{i,M}$ directly, while any other equation in the system relates only $q_{i,j}$ and $q_{i,j-1}$. If we solve the system in decreasing i-incidental order, we have $q_{i,j}$ available at the time we have to solve for $q_{i,j-1}$. Consequently, we can find $q_{i,j}$ as follows:

$$q_{i,M} = b_{i,M}, \quad q_{i,j} = b_{i,j} + \frac{p_d q_{i,j-1}}{y_{j-1}}, \quad j = -M+1, \dots, M-1 \tag{2.71}$$

Solving (2.70) for the $u_{i,j}$ (after we find the intermediate $q_{i,j}$) is achieved through backward substitution. We can read $u_{i+1,-M}$ directly (it is the value of the boundary), and if we solve in increasing i-incidental order, we can find all of the $u_{i,j}$ in the same manner:

$$u_{i+1,-M} = \frac{q_{i,-M}}{y_{-M}} \; u_{i+1,j} = \frac{q_{i,j} + p_d u_{i+1,j+1}}{y_j} \; j = -M+1, \dots, M-1 \tag{2.72}$$

At the boundary, we get the following conditions:

$$u_{i,M} - u_{i,M-1} = \beta_U \tag{2.73}$$

$$u_{i,-M+1} - u_{i,-M} = \beta_L \tag{2.74}$$

β_U and β_L are the derivatives at the boundary so that

$$\beta_U = S_{i,M} - S_{i,M-1}, \quad \beta_L = 0$$

for a call and

$$\beta_U = 0, \beta_L = -1 \cdot (S_{i,-M+1} - S_{i,-M})$$

for a put.

2.19 LU DECOMPOSITION EXAMPLE IN MATLAB

The following Matlab code gives an LU decomposition used in a forward difference scheme.

ForwardDifference.m

```
%%  Forward Difference Scheme
%%  Author: Chetan Jain and Jim Carson
%%  Boundary value problem for the function u = u(x) defined for
%%  0 <= x <= 1
%%  uxx + x*ux + u = 0,  u(0) = 1, u(1) = exp(-1/2)
%%  error at grid point n, e = u - u(n*delta_x)
%%  LU decomposition has been used here
%%  Values of e(N/2) have been computed


clear;

for k = 2:12

%%  Initialize
delta_x = 2^(-k);
DELTA_x(k-1) = delta_x;   %%  Storing delta_x for plots
N = 1/delta_x;
alpha = 1/(delta_x^2);      %%  Parameter in finite difference scheme

Nminus = 1;
Nplus = N + 1;

%%  Boundary conditions
u_0 = 1;
u_N = exp(-.5);

%%  Create vector b
b(Nminus) = alpha*u_0;
for i = Nminus+1:Nplus-3
    b(i) = 0;
end
b(Nplus-2) = (alpha + (Nplus-2))*u_N;


%%  LU Decomposition
%%%%%%%%%%%%%%%%%%%%
```

```
%%  Find y
y(Nminus) = 2*alpha;
for i = Nminus+1:Nplus-2
    y(i) = (2*alpha + i) - (alpha^2 + alpha*i)/y(i-1);
end

%%  Find z
for i = Nminus:Nplus-3
    z(i) = -(alpha + i);
end

%%  Find l
for i = Nminus:Nplus-3;
    l(i) = -alpha/y(i);
end

%%  Find q
q(Nminus) = b(Nminus);
for i = Nminus+1:Nplus-2
    q(i) = b(i) + (alpha*q(i-1))/y(i-1);
end

%%  Find u
u(Nplus-1) = u_N;
u(Nplus-2) = q(Nplus-2)/y(Nplus-2);

for  i=Nplus-3:-1:1
    u(i) = (q(i) + (alpha + i)*u(i+1))/y(i);
end


%%  Determine error at grid point:

j = (Nplus/2) + .5;
error(k-1) = u(j) - exp(-.5*((j*delta_x)^2));
k = k+1;

end % end k "for" loop

%%  Plots

plot(-log(DELTA_x), -log(abs(error)))
xlabel('-log(delta(x))')
ylabel('-log |error_N_/_2|')
title('Forward Difference Error Plot:  -log |error_N_/_2| as a function
  of -log(delta(x))')
```

Figure 2.11 shows a forward difference error plot as function of the space step. In particular, the error is linearized by taking the negative value of the error logarithm and the space step.

The figure shows that the logarithm of the inverse error increases as the logarithm of the inverse of the step size, $(\Delta x)^{-1}$, increases.

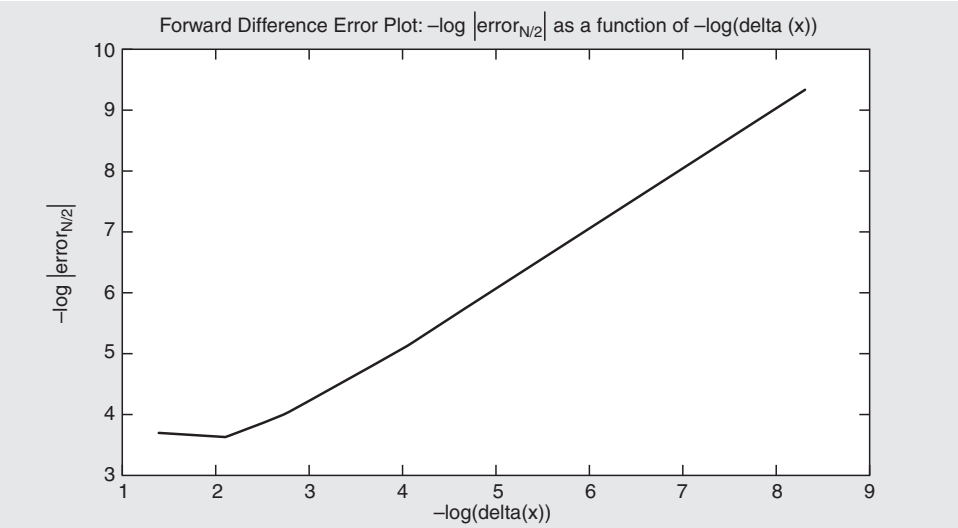


Figure 2.11

2.20 IMPLICIT DIFFERENCE EXAMPLE IN MATLAB

The following Matlab function prices a vanilla European call/put using the implicit finite difference method.

Implicit_Difference.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Author: Olivier Rochet (June 2005)
% This function prices a Vanilla European Call/Put using the Implicit
% Scheme of the Finite Difference Method.
% Parameters are as follows:
% OptionType = 1 for a Call or 0 for a Put
% SO = initial asset price ; K = strike price ; r = risk free rate ; q =
% dividend % rate ; T = time to maturity ; sig = volatility ;
% Smin = minimum stock price ;
% Smax = maximum stock price ; Ds = stock price step size ; Dt = time step
% size.
% This function keeps track of the time required to price the option.
% Note: When pricing using Finite Difference Methods, you increase accuracy
% by making the mesh finer and finer. In our case adequate pricing results
% are obtained with Ds = 0.5 and Dt = 1/1200 with Smin = 20 and Smax = 300.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [P] = ImplicitEuro(OptionType,SO,K,r,q,T,sig,Smin,Smax,Ds,Dt)

tic % Keep track of time

% Calculate number of stock price steps and take care of rounding.
N = round((Smax - Smin) / Ds);
```

```
Ds = (Smax - Smin) / N;
% Calculate number of time steps and take care of rounding.
M = round(T/Dt);
Dt = (T/M);

A=zeros(N,N); % A matrix
S=zeros(N,1); % stock price vector
V=zeros(N,1); % option value vector
matsol=zeros(N,M+1); % solution matrix

for i=1:1:N % Generate S and V vectors
    S(i)=Smin + i*Ds;
    if OptionType == 1
        V(i)=max(S(i)-K,0); % Call: Payoff that is initial condition
    else
        V(i)=max(K-S(i),0); % Put: Payoff that is initial condition
    end
end

for i=1:1:N % Build A matrix
    % Set up coefficients
    Alpha = 0.5*(sig^2)*(S(i)^2)*(Dt/(Ds^2));
    Betha = (r-q)*S(i)*(Dt/(2*Ds));
    Lph=-Alpha+Betha;
    Dph=1+r*Dt+2*Alpha;
    Uph=-Alpha-Betha;
    % Fill A matrix
    if i==1
        A(i,i) = Dph + 2*Lph;
        A(i,i+1) = Uph - Lph;
    elseif i==N
        A(i,i-1) = Lph - Uph;
        A(i,i) = Dph + 2*Uph;
    else
        A(i,i-1) = Lph;
        A(i,i) = Dph;
        A(i,i+1) = Uph;
    end
end

matsol(:,1)=V; % Initiate first column of matrix solution with payoff
               % that is initial condition

invA = A^-1; % Invert matrix A before performing calculations

for k=1:M % Generate solution matrix
    matsol(:,k+1)=invA*matsol(:,k);
end

% find closest point on the grid and return price
% with a linear interpolation if necessary

DS = SO-Smin;

indexdown = floor(DS/Ds);
indexup = ceil(DS/Ds);
```

```
if indexdown == indexup
    P = matsol(indexdown,M+1);
else
    P = matsol(indexdown,M+1)+ (DS - matsol(indexdown,M+1))...
        *(matsol(indexup,M+1) - matsol(indexdown,M+1))/Ds;
end

toc % Keep track of time

SO = 50;      % initial stock price
K = 52;       % strike price
r = 0.025;    % risk free rate
q = 0.01;     % dividend yield
sig = 0.3;    % volatility
Smin = 20;    % min stock price on grid
Smax = 300;   % max stock price on grid
Ds = 0.5;     % stock step size
Dt = 1/1200; % time step size

P = ImplicitEuro(OptionType,SO,K,r,q,T,sig,Smin,Smax,Ds,Dt)
```

```
Output:

P =
    5.38588
```

The following Matlab code implements the implicit difference scheme method to solve a PDE using SOR.

ImplicitDifference.m

```
%% SOR
%% Author: Chetan Jain & Jim Carson
%% This code implements the fully implicit finite difference scheme
%% for the following PDE using SOR:
%% ut = 3y^2uyy - 3yuy,  1 < y < e,  0<t<1
%% u(y, 0) = 0,      1 <= y <= e
%% u(1, t) = t,      0<= t <= 1
%% u(e, t) = t^2     0 <= t <= 1

clear;
format long;

%%  Enter alpha
alpha = 0.1;

%%  Choose omega for SOR algorithm
omega = 1;

k=3;

%% Initialize x interval
delta_x = 2^(-k);
LeftEndpoint = 0;
RightEndpoint = 1;
```

```
InitialNminus = LeftEndpoint/delta_x;
InitialNplus  = RightEndpoint/delta_x;

%%  Ensure positive indexes for Matlab and same width as x interval
Nminus = 1;
Nplus = InitialNplus - InitialNminus + Nminus;

%%  Initialize t interval
delta_t = alpha*(delta_x^2);
beta = delta_t/delta_x;
T = 1;
M = T/delta_t;

%%  Boundary condition w.r.t. x
for (i = Nminus:Nplus)
   % n = InitialNminus + i - 1;
    oldu(i,1) = 0;
end

%%  Boundary condition w.r.t. t
for (i = 1:M+1)
    oldu(1,i) = (i-1)*delta_t;
    oldu(Nplus,i) = ((i-1)*delta_t)^2;
end

%%  SOR algorithm
%%%%%%%%%%%%%%%%%%

for (i = 2:M+1)
    newu = ones(Nplus,1)    %% Create vector newu
    count = 0;
    while (sum((newu(2:Nplus-1,1) - oldu(2:Nplus-1,i)).^2) > 0.000001)
        count = count + 1;
        oldu(2:Nplus-1,i) = newu(2:Nplus-1);

        for (j = 2:Nplus-1)
              y(j) = (1/(1+6*alpha))*(oldu(j,i-1) + 3*(alpha + beta)*
                       oldu(j-1,i) - 3*(beta - alpha)*oldu(j+1,i));
              newu(j,1) = oldu(j,i) + omega*(y(j) - oldu(j,i));

        end
    end  % end while loop

end

oldu(:,M+1)
plot(oldu(:,M+1))


%%  Use linear interpolation to find the requested values

    values(Nminus) = oldu(Nminus,M+1);    %%  Initialize values() with
                                          %%  boundary condition
    w = 1;

for (i = 1:Nplus)
    n = Nminus + i - 1;
```

```
    if (n*delta_x >= Nminus*delta_x + w*0.2)
        values(w+1) = oldu(i,M+1) + (oldu(i-1,M+1)-oldu(i,M+1))*
                        (n*delta_x - w*0.2 - Nminus*delta_x)/delta_x;
        w = w + 1;
    end

end
```

2.21 CRANK-NICOLSON SCHEME

The Crank-Nicolson is a type of finite difference scheme that is used to overcome the stability limitations imposed by the stability and convergence restrictions of the explicit finite difference scheme. The Crank-Nicolson converges faster than the implicit and explicit finite difference schemes. The rate of convergence of the Crank-Nicolson scheme is $O((\Delta t)^2)$, whereas it is $O((\Delta t))$ for the implicit and explicit finite difference methods.

Essentially, the Crank-Nicolson method is an average of the implicit and explicit methods. Consider a simple diffusion equation. If we use a forward difference approximation for the time partial derivative, we obtain the explicit scheme

$$\frac{u_{i+1,j} - u_{i,j}}{\Delta t} + O(\Delta t) = \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{(\Delta x)^2} + O((\Delta t)^2)$$

and if we take the backward difference, we get the implicit scheme

$$\frac{u_{i+1,j} - u_{i,j}}{\Delta t} + O(\Delta t) = \frac{u_{i+1,j+1} - 2u_{i+1,j} + u_{i+1,j-1}}{(\Delta x)^2} + O((\Delta t)^2)$$

Taking the average of these two equations, we get:[20]

$$\begin{aligned} &\frac{u_{i+1,j} - u_{i,j}}{\Delta t} + O(\Delta t) = \\ &\frac{1}{2}\left(\frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{(\Delta x)^2} + \frac{u_{i+1,j+1} - 2u_{i+1,j} + u_{i+1,j-1}}{(\Delta x)^2}\right) + \\ &O((\Delta x)^2) \end{aligned} \tag{2.75}$$

Ignoring the error terms, we get the Crank-Nicolson scheme

$$\begin{aligned} u_{i+1,j} - \frac{1}{2}\alpha\left(u_{i+1,j+1} - 2u_{i+1,j} + u_{i+1,j-1}\right) = \\ u_{i,j} + \frac{1}{2}\alpha\left(u_{i,j+1} - 2u_{i,j} + u_{i,j-1}\right) \end{aligned} \tag{2.76}$$

where $\alpha = \frac{1}{2}\frac{\Delta t}{(\Delta x)^2}$. Notice that $u_{i+1,j-1}$, $u_{i+1,j}$, and $u_{i+1,j+1}$ are now determined implicitly in terms of $u_{i,j}$, $u_{i,j+1}$, and $u_{i,j-1}$. Equation (2.76) can be solved in the same manner as the implicit scheme in (2.53) because everything on the right-hand side can be evaluated explicitly if the $u_{i,j}$ are known. Denote the left-hand side of (2.76) by $Z_{i,j}$. The problem of solving (2.74) reduces to first computing

$$Z_{i,j} = (1 - \alpha)u_{i,j} + \frac{1}{2}\alpha\left(u_{i,j-1} + u_{i,j+1}\right) \tag{2.77}$$

which is an explicit formula for $Z_{i,j}$, and then solving

$$(1+\alpha)u_{i+1,j} - \frac{1}{2}\alpha\left(u_{i+1,j-1} + u_{i+1,j+1}\right) = Z_{i,j}. \tag{2.78}$$

We can write (2.78) as a linear system

$$\mathbf{A}\mathbf{u}^{\mathbf{i+1}} = \mathbf{b}^{\mathbf{i}} \tag{2.79}$$

where the matrix $\mathbf{A}$ is given by

$$\mathbf{A} = \begin{bmatrix} 1+\alpha & -\frac{1}{2}\alpha & 0 & \dots & 0 \\ -\frac{1}{2}\alpha & 1+\alpha & -\frac{1}{2}\alpha & & \dots \\ 0 & -\frac{1}{2}\alpha & & \dots & 0 \\ \dots & & \dots & & -\frac{1}{2}\alpha \\ 0 & 0 & \dots & -\frac{1}{2}\alpha & 1+\alpha \end{bmatrix} \tag{2.80}$$

and the vectors $\mathbf{u}^{\mathbf{i+1}}$ and $\mathbf{b}^{\mathbf{i}}$ are given by

$$\mathbf{u}^{\mathbf{i+1}} = \begin{bmatrix} u_{i+1,N^-+1} \\ \dots \\ u_{i+1,0} \\ \dots \\ u_{i+1,N^+-1} \end{bmatrix}, \ \mathbf{b}^{\mathbf{i}} = \begin{bmatrix} Z_{i,N^-+1} \\ \dots \\ Z_{i,0} \\ \dots \\ Z_{i,N^+-1} \end{bmatrix} + \frac{1}{2}\alpha \begin{bmatrix} u_{i+1,N^-} \\ 0 \\ \dots \\ 0 \\ u_{i+1,N^+} \end{bmatrix} \tag{2.81}$$

The vector on the far right-hand side of (2.81), in $\mathbf{b}^{\mathbf{i}}$ comes from the boundary conditions applied at the end points of a finite mesh where $x = N^-\Delta x$ and $x = N^+\Delta x$. N^- and N^+ are integers, chosen to be sufficiently large so that no significant errors are introduced.

To implement the Crank-Nicolson scheme, we first generate the vector $\mathbf{b}^{\mathbf{i}}$ using known quantities. Then, we use either an LU decomposition solver or an SOR solver to solve the system (2.79). The scheme is both stable and convergent for all values of $\alpha > 0$.

We can apply the Crank-Nicolson scheme to the Black-Scholes diffusion equation by replacing time and space derivatives with finite differences centered at the time step $i+1/2$.

$$\begin{aligned} &\frac{1}{2}\sigma^2\left(\frac{(u_{i+1,j+1} - 2u_{i+1,j} + u_{i+1,j-1}) + (u_{i,j+1} - 2u_{i,j} + u_{i,j-1})}{2\Delta x^2}\right) + \\ &\quad \mu\left(\frac{(u_{i+1,j+1} - u_{i+1,j-1}) + (u_{i,j+1} - u_{i,j-1})}{4\Delta x}\right) + \frac{u_{i+1,j} - u_{i,j}}{\Delta t} - \\ &\quad r\left(\frac{u_{i+1,j} + u_{i,j}}{2}\right) = 0. \end{aligned}$$

which can be written as

$$\begin{aligned} p_u u_{i,j+1} + p_m u_{i,j} + p_d u_{i,j-1} = &-p_u u_{i+1,j+1} - \\ &(p_m - 2)\, u_{i+1,j} - p_d u_{i+1,j-1} \end{aligned} \tag{2.82}$$

where

$$p_u = -\frac{1}{4}\Delta t\left(\frac{\sigma^2}{\Delta x^2}+\frac{\mu}{\Delta x}\right)$$
$$p_m = 1+\Delta t\frac{\sigma^2}{2\Delta x^2}+\frac{r\Delta t}{2} \tag{2.83}$$
$$p_d = -\frac{1}{4}\Delta t\left(\frac{\sigma^2}{\Delta x^2}-\frac{\mu}{\Delta x}\right)$$

We can write (2.82) in the form in (2.77). First, set

$$Z_{i,j} = p_u u_{i,j+1} + p_m u_{i,j} + p_d u_{i,j-1}$$

and then solve

$$-p_u u_{i+1,j+1} - (p_m - 2)u_{i+1,j} - p_d u_{i+1,j-1} = Z_{i,j}$$

using an LU decomposition.

2.22 Asian Option Pricing Using Crank-Nicolson in Matlab

The following Matlab code prices an Asian option using the Crank-Nicolson method.

Asian_Option.m

```
% Author: Jan Vecer
% Crank - Nicolson finite difference
% Thomas algorithm

tic
clear

%parameters

r=0.05;  % risk free
sig=.5;  % volatility
K1=0;    % parametro per floating strike
K2=2;    % strike per fixed asian
S0=2;
gamma=0;
T =1;    %

%space and time constraints

z=[-1 1];
t=[0 1];

%time grid

n=200;
dt=(t(2)-t(1))/n;
```

```
tvec = t(1):dt:t(2);

%space grid

m=200;
dz=(z(2)-z(1))/m;
zvec = z(1):dz:z(2);

%strategy

%q = zeros(length(tvec),1);
%q = (1/(r*t(2)))*(1-exp(-r*(t(2)-tvec(:))));

thalf=0.5*(tvec(1:n)+tvec(2:n+1));

q = (1/(r*t(2)))*(1-exp(-r*(t(2)-thalf(:))));

%constants

mu=dz^2/dt;

u=zeros(n+1,m+1);

%terminal condition

%u(n+1,:)=ones;
%u(n+1,:)=zvec;

u(n+1,:)=max(zvec-K1,0);

%computation

for i=n:-1:1,

   %if mod(i,20)==0
   %   fprintf('%3.0f',i)
   %end

%tridiagonal terms

a=(sig^2)*(zvec(3:m)-q(i)).^2;
b=(sig^2)*(zvec(2:m)-q(i)).^2 + 2*mu;
c=(sig^2)*(zvec(2:m-1)-q(i)).^2;
d=(sig^2)*(zvec(2:m)-q(i)).^2 - 2*mu;

%explicit part of the scheme (right hand side)

aux      = d.*u(i+1,2:m)-[(1/2)*c.*u(i+1,3:m),0]-[0,(1/2)*a.*u(i+1,2:m-1)];
aux(1)   = aux(1) - (1/2)*((sig^2)*(q(i)-zvec(2)).^2*u(i+1,1));
aux(m-1) = aux(m-1)-(1/2)*((sig^2)*(q(i)-zvec(m)).^2*u(i+1,m+1));

%left hand side of the equation

Dleft=-diag(b)+(1/2)*diag(c,1)+(1/2)*diag(a,-1);

%corner elements of the matrix
```

```
Dleft(m-1,m-1)=Dleft(m-1,m-1)+ (sig^2)*(q(i)-zvec(m)).^2;
Dleft(m-1,m-2)=Dleft(m-1,m-2)- (1/2)*(sig^2)*(q(i)-zvec(m)).^2;
Dleft(1,1) = Dleft(1,1) + (sig^2)*(q(i)-zvec(2)).^2;
Dleft(1,2) = Dleft(1,2) - (1/2)*(sig^2)*(q(i)-zvec(2)).^2;

%solving implicit part of Crank Nicolson

u(i,2:m)=(Dleft\aux')';

%corner elements

u(i,m+1)=2*u(i,m)-u(i,m-1);
u(i,1)=2*u(i,2)-u(i,3);

end

%prices
  point(1)=(1-exp(-r*t(2)))/(r*t(2))-exp(-r*t(2))*(K2/S0);
  S0*spline(zvec,u(1,:),point(1)), m, n

toc
```

```
ans =

    0.2464

m =

   200

n =

   200

Elapsed time is 6.995622 seconds.
```

ENDNOTES

1. Note that to compute each simulated path, the initial asset price, asset volatility, time to maturity, number of time steps, and drift term need to be specified.
2. $\mathrm{E}[\mathbf{N}\mathbf{N}'] = E\begin{bmatrix} \chi^2(1) & 0 \\ 0 & \chi^2(1) \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \mathbf{I}$. Note that the expectation (mean) of a chi-squared random variable is its number of degrees of freedom.
3. See *Numerical Recipes in C* for the code implementing the algorithm, pgs. 96–98. Also see the Template Numerical Toolkit.
4. Clewlow, L. and Strickland, C. (1998), pg. 128.
5. See *http://math.nist.gov/tnt/.*
6. See *http://www.math.keio.ac.jp/matumoto/emt.html.*
7. Su, Y. and Fu, M. (2000), pg. 587.
8. Ibid., pg. 587.

9. Ibid., pg. 587.
10. The mathematical definition of discrepancy of n sample points is $D_n = \sup_J \left| \frac{A(J;n)}{n} - V(J) \right|$ where $J = \prod_{i=1}^{d} [0, u_i) = [0, u_1)^d$, $u_i \leq 1$, d is the dimension, $A(J;n)$ are the number of points landed in region J, and $V(J)$ is the volume of J.
11. There are other low-discrepancy sequences, including Halton (1960), Niederreiter (1992), and Niederreiter and Shiue (1995).
12. The code was adapted from Austen McDonald *(http://www.prism.gatech.edu/ ∼gte363v/montecarlo/)*, who massaged the original Sobol version into a parallel version.
13. Clewlow, L. and Strickland, C. (1998), pg. 131.
14. Clewlow, L. and Strickland, C. (1998), pg. 95.
15. The material in the section follows the work of Madan and Carr (1999). See Madan and Carr (1999), "Option Valuation Using the Fast Fourier Transform"; see also Dempster and Hong, *Spread Option Valuation and the Fast Fourier Transform*, Judge Institute of Management Studies at the University of Cambridge (July 2000).
16. See Madan and Carr (1999), "Option Valuation Using the Fast Fourier Transform," pg. 6.
17. Ibid, pg. 5.
18. Ibid., pg. 11.
19. In order to keep consistent with the notion given—e.g., $S_j, j = -M \ldots -1, 0, 1, \ldots M$—we allow array indices to be negative. Although this is not considered good programming practice, the array indices cannot go out of bounds because proper memory is allocated in initialization to handle the total memory needed during the implementation. The reader can easily modify the code so that the indices remain positive—e.g., go from $j = 0, 1, \ldots 2^*M$—because there are |M| nodes above and below the nodes in the middle.
20. It can be shown that the error terms in (2.73) are accurate to rather than

CHAPTER 3

COPULA FUNCTIONS

SECTIONS

Copula functions are essential instrumentalities in the pricing of structured credit products given the ability of these functions to incorporate correlation dependencies of the underlying credits. They can be characterized in terms of density functions (for elliptical copulae) and generator functions (for Archimedean copulas). Copulae are instrumental in pricing credit products like credit default swaps, credit swap indices (CDXs), and collateralized debt obligations (CDOs) that require modeling dependency and correlation structures of the underlying reference entities. In §3.1, we provide the definition and basic properties of copula functions. In §3.2, we discuss classes of copula functions, including the multivariate Gaussian copula and multivariate Student's *t* copula. In §3.3, we review Archimedean copulae, a large and flexible class of copulae for modeling dependency structures. In §3.4, we discuss calibrating copulae. We review using the exact maximum likelihood method for the Gaussian and Student's *t* copula. We discuss the inference functions for margins methods (IFM), as well as the canonical maximum likelihood method for calibrating the copula parameters. We also discuss the Bouyè, Mashal, and Zeevi methods. In addition, we provide Matlab implementations for both of these calibration methods. In §3.5, we discuss and provide numerical results from calibrating copulae to real-market data. The work follows directly from Galiani (2003). In §3.6, some Excel copulae examples in Excel are given.

3.1 DEFINITION AND BASIC PROPERTIES OF COPULA FUNCTIONS

An n-dimensional copula is a function $C : [0,1]^n \rightarrow [0,1]$ that has the following properties:

1. $C(\mathbf{u})$ is increasing in each component $\mathbf{u}_k$ with $k \in \{1, 2, ..., n\}$.

2. For every vector $\mathbf{u} \in [0,1]^n$, $C(\mathbf{u}) = 0$ if at least one coordinate of the vector $\mathbf{u}$ is 0, and $C(\mathbf{u}) = \mathbf{u}_k$ if all the coordinates of $\mathbf{u}$ are equal to 1 except the k-th one.

3. For every $\mathbf{a}, \mathbf{b} \in [0,1]^n$ with $\mathbf{a} \leq \mathbf{b}$ given a hypercube $\mathbf{B} = [\mathbf{a}, \mathbf{b}] = [a_1, b_1] \times [a_2, b_2] \times \cdots \times [a_n, b_n]$ whose vertices lie in the domain of C, its volume[1] $V_C(\mathbf{B}) \geq 0$.

The definition shows that C is a multivariate distribution function with uniformly distributed marginals. The statistical interpretation of the preceding properties will become further meaningful once we adapt the definition of copula functions to a vector or random variables. But first, we will need an auxiliary theory, which constitutes one of the most relevant results in the copula methodology.

Theorem 1 (Sklar): Let G be an n-dimensional distribution function with margins $F_1, F_2, \ldots, F_n$. Then there exists an n-dimensional copula C such that, for $\boldsymbol{x} \in \Re^n$, we have

$$G(x_1, x_2, \ldots, x_n) = C(F_1(x_1), F_2(x_2), \ldots, F_n(x_n)). \tag{3.1}$$

Moreover, *if* $F_1, F_2, \ldots, F_n$ are continuous, then C is unique. Sklar's theorem expresses the basic idea of dependence modeling via copula functions, by stating that for any multivariate distribution function, the univariate marginals (the distribution functions in case of random variables) and the dependence structure can be separated, with the latter completely described by a copula function. As Scaillet (2000), Sklar's theorem has an important corollary:

Corollary 1: Let G and C be, respectively, an n-dimensional distribution function (with continuous univariate marginals $(F_1, F_2, \ldots, F_n)$ and an n-dimensional copula function. Then for any $\boldsymbol{u} \in [0,1]^n$, we have

$$C(u_1, u_2, \ldots, u_n) = G(F_1^{-1}(u_1), F_2^{-1}(u_2), \ldots, F_n^{-1}(u_n)), \tag{3.2}$$

where $F_i^{-1}(u_i)$ denotes the inverse of the cumulative distribution function, namely, for $u_i \in [0,1]$, $F_i^{-1}(u_i) = \inf\{x : F_i(x) \geq u_i\}$. The importance of equation (3.2) will be clear in the following sections after we present a general framework for simulation of random numbers generated by a specific copula. Let $(X_1, X_2, \ldots, X_n)'$ be an n-dimensional vector of random variables with distribution functions $(F_1, F_2, \ldots, F_n)$ and joint distribution function G. Then, by Sklar's theorem, if $(F_1, F_2, \ldots, F_n)$ are continuous functions, $(X_1, X_2, ..., X_n)'$ has a unique copula, as described by the following representation:

$$\begin{aligned} G(x_1, x_2, \ldots, x_n) &= P(X_1 \leq x_1, \ldots, X_n \leq x_n) \\ &= C(F_1(x_1), F_2(x_2), \ldots, F_n(x_n)) \end{aligned}$$

This representation of copula functions allows to further re-establish the last two properties of definition 1. In fact, property (2) follows from the fact that, by the so-called "probability-integral transform" (see Casella and Berger [2000]), if the random variables X and Y have continuous distribution function F_X and F_Y, then the random variables

$U = F_X(X)$ and $V = F_Y(Y)$ are uniformly distributed on $[0, 1]$ and, therefore, in the bivariate case,

$$C(u, 1) = P(U \leq u, V \leq 1) = P(U \leq u) = u$$

and

$$C(u, 0) = P(U \leq u, V \leq 0) = 0$$

Property (3) ensures that the copula function C respects the defining characteristic of a proper multivariate distribution function, assigning non-negative weights to all rectangular subsets in $[0, 1]^n$.

By applying Sklar's theorem and by exploiting the relation between the distribution and the density function,[2] we can easily derive the multivariate copula density $c(F_1(x_1), \ldots, F_n(x_n))$ associated with a copula $C(F_1(x_1), \ldots, F_n(x_n))$:

$$\begin{aligned} f(x_1, ..., x_n) &= \frac{\partial^n [C(F_1(x_1), ..., F_n(x_n))]}{\partial F_1(x_1) ... \partial F_n(x_n)} \cdot \prod_{i=1}^{n} f_i(x_i) \\ &= c(F_1(x_1), ..., F_n(x_n)) \cdot \prod_{i=1}^{n} f_i(x_i), \end{aligned}$$

where we define

$$c(F_1(x_1), \ldots, F_n(x_n)) = \frac{f(x_1, \ldots, x_n)}{\prod_{i=1}^{n} f_i(x_i)}. \tag{3.3}$$

As we will see in §3.2, knowledge of the associated copulate density will be particularly useful in order to calibrate its parameters to real-market data.

3.2 Classes of Copula Functions

There are different families of copula functions that can be classified. The most common are the Gaussian and t copulas belonging to the elliptical family.[3]

Multivariate Gaussian Copula

Definition 1: Let R be a symmetric, positive definite matrix with $diag(R) = \mathbf{1}$ and let $\mathbf{\Phi}_R$ be the standardized multivariate normal distribution correlation matrix R. Then the multivariate Gaussian copula is defined as

$$C(u_1, u_2, \ldots, u_n; R) = \mathbf{\Phi}_R(\Phi^{-1}(u_1), \Phi^{-1}(u_2), \ldots, \Phi^{-1}(u_n)), \tag{3.4}$$

where $\Phi^{-1}(u)$ denotes the inverse of the normal cumulative distribution function. The associated multinormal copula density is obtained by applying equation (3.3):

$$c(\Phi(x_1),\ldots,\Phi(x_n)) = \frac{f^{gaussian}(x_1,\ldots,x_n)}{\prod_{i=1}^{n} f_i^{gaussian}(x_i)} = \frac{\frac{1}{(2\pi)^{n/2}|R|^{1/2}}\exp\left(-\frac{1}{2}\boldsymbol{x}'R^{-1}\boldsymbol{x}\right)}{\prod_{i=1}^{n}\frac{1}{\sqrt{2\pi}}\exp\left(-\frac{1}{2}x_i^2\right)}$$

Thus, fixing $u_i = \Phi(x_i)$, and denoting $\varsigma = (\Phi^{-1}(u_1),\ldots,\Phi^{-1}(u_n))'$ the vector of the Gaussian univariate inverse distribution functions, we have the following:

$$c(u_1,u_2,\ldots,u_n;R) = \frac{1}{|R|^{1/2}}\exp\left[-\frac{1}{2}\varsigma'(R^{-1}-I)\varsigma\right] \tag{3.5}$$

Figure 3.1 shows the surface of the Gaussian copula density as depicted in (3.5) for the bivariate case with correlation r.

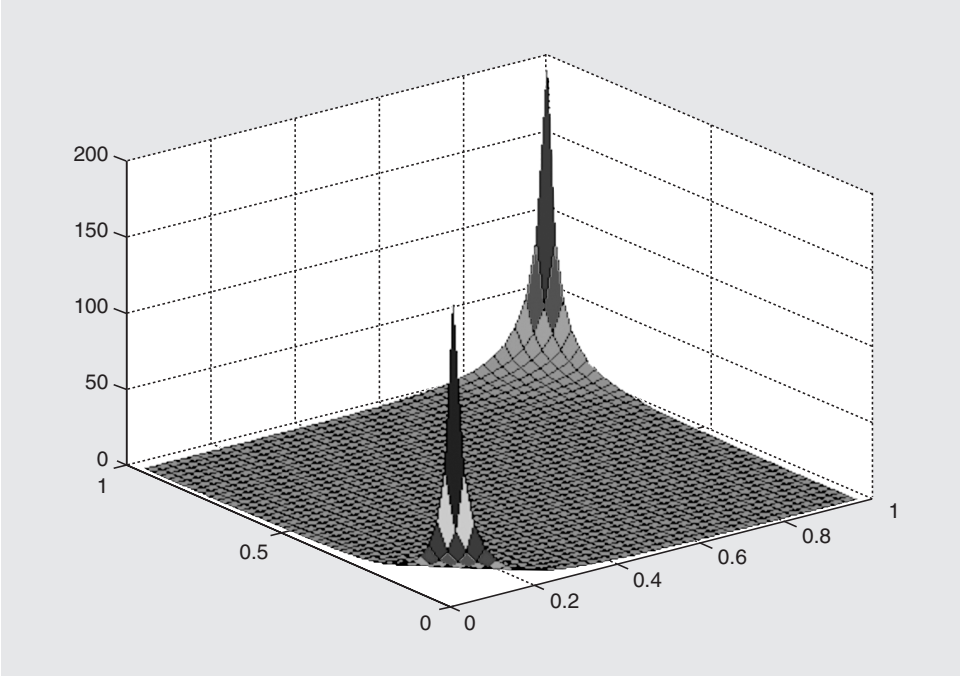


Figure 3.1 Gaussian copula density for bivariate case

The following Matlab code computes the Gaussian copula density surface.

gaussian_copula_density.m

```
% This script plot the density of a bivariate gaussian copula function
% Pairwise correlation is set at 50%

R=ones(2,2);
r=.5;

R(1,2)=r;
R(2,1)=r;
X=zeros(2,1);
U=zeros(2,1);
gc=zeros(39,39);
h=0;

for i=0.025:.025:.975
    h=h+1;
    k=0;
    for j=0.025:.025:.975
        X=[i;j];
        k=k+1;
        U=norminv(X);
        block1=1/(det(R)^0.5);
        block2=-0.5*U'*(inv(R)-ones(2,2))*U;
        gauss_grid(h,k)=block1*exp(block2);
    end
end
surf(gauss_grid)
```

Multivariate Student's *T* Copula

Definition 2: Let R be a symmetric, positive definite matrix with $diag(R) = \mathbf{1}$ and let $T_{R,v}$ be the standardized multivariate Student's t distribution with correlation matrix R and v degrees of freedom.[4] Then the multivariate Student's t copula function is defined as

$$C(u_1, u_2, \ldots, u_n; R, v) = T_{R,v}(t_v^{-1}(u_1), t_v^{-1}(u_2), \ldots, t_v^{-1}(u_n)) \tag{3.6}$$

where $t_v^{-1}(u)$ denotes the inverse of the Student's t cumulative distribution function. The associated Student's t copula density is obtained by applying equation (3.3):

$$\begin{aligned} c(u_1, u_2, ..., u_n; R, v) &= \frac{f^{Student}(x_1, \ldots, x_n)}{\prod_{i=1}^{n} f_i^{Student}(x_i)} \\ &= |R|^{-1/2} \frac{\Gamma\left(\frac{v+n}{2}\right)}{\Gamma\left(\frac{v+1}{2}\right)} \left[\frac{\Gamma\left(\frac{v}{2}\right)}{\Gamma\left(\frac{v+1}{2}\right)}\right]^n \frac{\left(1 + \frac{\varsigma' R^{-1} \varsigma}{v}\right)^{-\frac{v+n}{2}}}{\prod_{i=1}^{n} \left(1 + \frac{\varsigma_i^2}{v}\right)^{-\frac{v+1}{2}}} \end{aligned} \tag{3.7}$$

where $\varsigma = (t_v^{-1}(u_1), t_v^{-1}(u_2), \ldots, t_v^{-1}(u_n))'$.

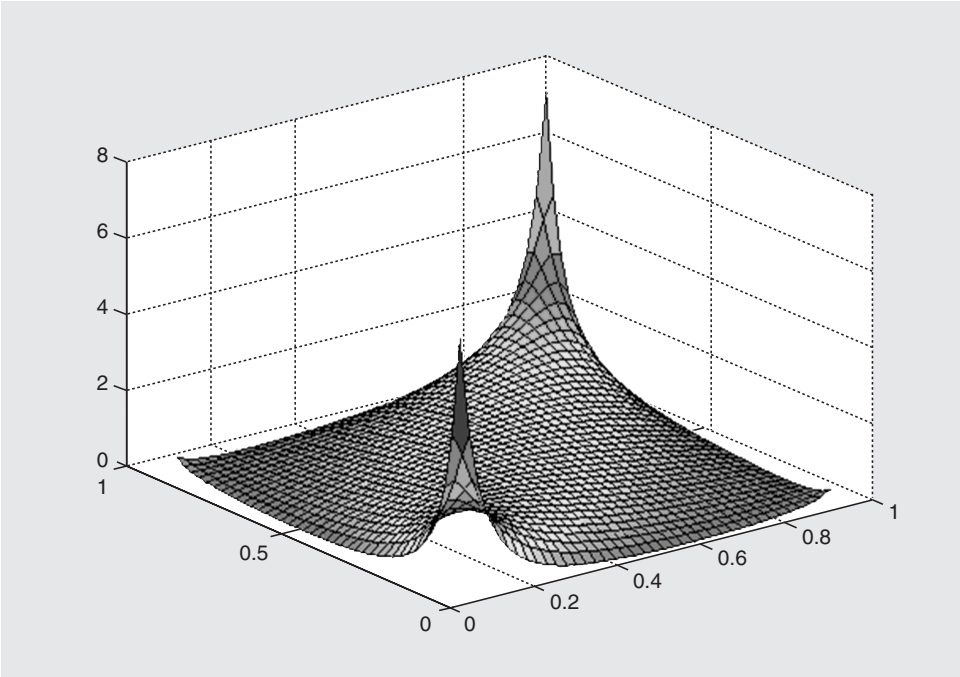


Figure 3.2 Student's t copula density for bivariate case

Figure 3.2 shows the surface of the Student's t copula density as given in (3.7) for a bivariate case with correlation r.

The following Matlab code computes a t copula density surface:

t_copula_density1.m

```
% This script plots che bivariate Student's t copula with 50% correlation
% and 3 degrees of freedom

R=ones(2,2);
r= .5;

R(1,2)=r;
R(2,1)=r;
X=zeros(2,1);
U=zeros(2,1);
gc2=zeros(39,39);
h=0;
DoF=3; % degrees of freedom
d=2; %dimension
Block_1=0;
Block_2=0;
for i=0.025:.025:.975
    h=h+1;
```

```
    k=0;
    for j=0.025:.025:.975
        X=[i;j];
        k=k+1;
        y=(tinv(X,DoF));
        A=gamma((DoF+d)/2)*((gamma(DoF/2))^(d-1));
        B=((gamma((DoF+1)/2))^d)*((det(R))^0.5);
        Block_1=A/B;
        A=0;
        B=0;

        Block_2=1;
        for l=1:d
            C=(1+((y(l)^2)/DoF))^(-(DoF+1)/2);
            Block_2=Block_2*C;
            C=0;
        end
        D=y'*inv(R)*y/DoF;
        Block_3=(1+D)^(-(DoF+d)/2);
        D=0;
        t_grid(h,k)=(Block_1/Block_2)*Block_3;
    end
end

surf(t_grid)
```

3.3 ARCHIMEDEAN COPULAE

Archimedean copulae constitute an important class of copula functions not only because of their analytical tractability (many of the most common Archimedean copulae have closed form expression), but also because they provided a large spectrum of different dependence structures.

Following the analysis of Nelson (1999), consider a function $\varphi : [0,1] \to [0,\infty)$ such that:

- φ is continuous.
- $\varphi'(u) < 0$ for all $u \in [0,1]$.
- $\varphi(1) = 0$.

Define the pseudo-inverse of φ as the function $\varphi^{[-1]} : [0,\infty) \to [0,1]$ such that:

$$\varphi^{[-1]}(t) = \begin{cases} \varphi^{-1}(t) & \text{for } 0 \le t \le \varphi(0) \\ 0 & \text{for } \varphi(0) \le t \le \infty \end{cases}$$

Now, if φ is convex, then the function $C : [0,1]^2 \to [0,1]$, defined as

$$C(u,v) = \varphi^{[-1]}\left[\varphi(u) + \varphi(v)\right] \tag{3.8}$$

is an Archimedean copula and φ is called the generator of the copula. Furthermore, if $\varphi(0) = \infty$, the pseudo-inverse describes an ordinary inverse function (that is, $\varphi^{[-1]} = \varphi^{-1}$) and we call φ and C, respectively, a strict generator and a strict Archimedean copula.

Gumbel Copula: Let $\varphi(t) = (-\ln t)^\theta$ with $\theta \geq 1$. Then, using equation (3.8), we have

$$C_\theta^{Gumbel}(u,v) = \varphi^{-1}[\varphi(u) + \varphi(v)] = \exp\left\{-\left[(-\ln u)^\theta + (-\ln v)^\theta\right]^{1/\theta}\right\}.$$

Clayton Copula: Let $\varphi(t) = (t^{-\theta} - 1)/\theta$ with $\theta \in [-1, \infty) \setminus \{0\}$. Then, using (3.8), we have

$$C_\theta^{Clayton}(u,v) = \max[(u^{-\theta} + v^{-\theta} - 1)^{1/\theta}, 0].$$

Note that if $\theta > 0$, then $\varphi(0) = \infty$, and we can simplify the preceding equation as

$$C_\theta^{Clayton}(u,v) = (u^{-\theta} + v^{-\theta} - 1)^{1/\theta}. \tag{3.9}$$

Frank Copula: Let $\varphi(t) = -\ln \frac{e^{-\theta t} - 1}{e^{-\theta} - 1}$ with $\theta \in \Re \setminus \{0\}$. Then, using equation (3.8), we have

$$C_\theta^{Frank}(u,v) = -\frac{1}{\theta} \ln\left[1 + \frac{(e^{-\theta u} - 1)(e^{-\theta v} - 1)}{e^{-\theta} - 1}\right]. \tag{3.10}$$

We can generalize the Archimedean copulae framework to the multivariate case. Following the analysis of Embrechts, Lindskog, and McNeil (2001), we have the following theorem:

Theorem 2(Kimberling): Let $\varphi : [0,1] \to [0,\infty)$ be a continuous, strictly decreasing function such that $\varphi(0) = \infty$ and $\varphi(1) = 0$, and let φ^{-1} be the inverse of φ. Then, for all $n \geq 2$, the function $C : [0,1]^n \to [0,1]$ defined as

$$C(u_1, u_2, \ldots, u_n) = \varphi^{-1}[\varphi(u_1) + \varphi(u_2) + \ldots + \varphi(u_n)]$$

is an n-dimensional Archimedean copula if and only if φ^{-1} is completely monotone[5] on $[0,\infty)$.

3.4 CALIBRATING COPULAE

Calibrating copula parameters to real-market data is an important step in pricing structured credit products to ensure accuracy and robustness in pricing. In the following analysis, we consider a random sample represented by the time series $\boldsymbol{X} = (X_{1t}, X_{2t}, ..., X_{Nt})_{t=1}^T$ where N stands for the number of underlying assets—e.g., loans—and T represents the number of observations (on a daily monthly, quarterly, or yearly basis) available.

Exact Maximum Likelihood Method (EML)

Let Θ be the parameter space and θ be the k-dimensional vector of parameters to be estimated. Let $L_t(\theta)$ and $l_t(\theta)$ be, respectively, the likelihood and the log-likelihood function for the observation at time t. Define the log-likelihood function $l(\theta)$ as the following:

$$l(\theta) = \sum_{t=1}^{T} l_t(\theta) \tag{3.11}$$

Consider the canonical expression for density function as expressed by equation (3.3). We can expand (3.11) as follows:

$$l(\theta) = \sum_{t=1}^{T} \ln c(F_1(x_1^t), ..., F_N(x_N^t)) + \sum_{t=1}^{T}\sum_{n=1}^{N} \ln f_n(x_n^t). \tag{3.12}$$

Define the maximum likelihood estimator, as the vector $\widehat{\theta}$ such that

$$\widehat{\theta} = (\widehat{\theta}_1, \widehat{\theta}_2, \ldots, \widehat{\theta}_k) \in \arg\max\{l(\theta) : \theta \in \Theta\}.$$

Gaussian Copula: Let $\Theta = \{R : R \in \Re^{NxN}\}$ denote the parameter space with R being a symmetric and positive definite matrix. Applying equation (3.12) to the case of the gaussian copula density given in equation (3.5) yields

$$l^{gaussian}(\theta) = -\frac{T}{2}\ln|R| - \frac{1}{2}\sum_{t=1}^{T} \varsigma_t'(R^{-1} - I)\varsigma_t. \tag{3.13}$$

Assuming that the log-likelihood function in (3.13) is differentiable in θ and that the solution of the equation $\frac{\partial}{\partial\theta}\ell(\theta) = 0$ defines a global maximum, we can easily recover the maximum likelihood estimator $\widehat{\theta} = \widehat{R}$ for the gaussian copula whose log-likelihood is given in equation (3.13):

$$\frac{\partial}{\partial R^{-1}}\ell^{gaussian}(\theta) = \frac{T}{2}R - \frac{1}{2}\sum_{t=1}^{T}\varsigma_t'\varsigma_t$$

and therefore

$$\widehat{R} = \frac{1}{T}\sum_{t=1}^{T}\varsigma_t'\varsigma_t. \tag{3.14}$$

Student's t Copula: Let $\theta = \left\{(v, R) : v \in [2, \infty), R \in \Re^{NxN}\right\}$, with R being a symmetric and positive definite matrix, denote the parameter space. We can apply (3.14) to the case of the Student's t copula density given in (3.7). In this case, the calculation is more involved, leading to the following:

$$\begin{aligned}\ell^{Student}(\theta) = T\ln\frac{\Gamma\left(\frac{v+N}{2}\right)}{\Gamma\left(\frac{v}{2}\right)} - NT\ln\frac{\Gamma\left(\frac{v+1}{2}\right)}{\Gamma\left(\frac{v}{2}\right)} - \frac{T}{2}\ln|R| - \\ \frac{v+N}{2}\sum_{t=1}^{T}\ln\left(1 + \frac{\varsigma_t'R^{-1}\varsigma_t}{v}\right) + \\ \frac{v+1}{2}\sum_{t=1}^{T}\sum_{n=1}^{N}\ln\left(1 + \frac{\varsigma_{nt}^2}{v}\right).\end{aligned} \tag{3.15}$$

Unlike the case of Gaussian copula, the calibration of the Student's t copula via the EML method is more complicated because it requires a simultaneous estimation (see, for

example, Johnson and Kotz (1972)) of the parameters of the marginals and the parameters related to the dependence structure. But this procedure, as indicated by Mashal and Naldi (2002), requires a large amount of data and is computationally intensive. For that reason, the alternative methodologies of the inference functions for margins method (IFM) and canonical maximum likelihood method (CML) are employed.

The Inference Functions for Margins Method (IFM)

The IFM, based on the work of Joe and Xu (1996), exploiting the fundamental idea of copula theory (that is, the separation between the univariate margins and the dependence structure), expresses equation (3.12) as

$$\ell(\theta) = \sum_{t=1}^{T} \ln c(F_1(x_1^t;\theta_1),\ldots,F_N(x_N^t;\theta_N);\boldsymbol{\alpha}) + \sum_{t=1}^{T}\sum_{n=1}^{N} \ln f_n(x_n^t;\theta_n). \quad (3.16)$$

The peculiarity of (3.16) relies in the separation between the vector of the parameters for the univariate marginals $\boldsymbol{\theta} = (\theta_1,\ldots,\theta_N)$ and the vector of the copula parameters $\boldsymbol{\alpha}$. In other words, the calibration of the copula parameters to market data is performed via a two-stage procedure:

1. Estimation of the vector of the parameters for the marginal univariates $\boldsymbol{\theta} = (\theta_1,...,\theta_N)$ via the EML method. For instance, considering the time series of the ith underlying asset, we have[6]

$$\widehat{\theta}_i = \arg\max_{\theta_t} \sum_{t=1}^{T} \ln f_i(x_i^t;\theta_i).$$

2. Estimation of the vector of copula parameters $\boldsymbol{\alpha}$, using the previous estimators $\widehat{\boldsymbol{\theta}} = (\widehat{\theta}_1 \ldots,\widehat{\theta}_N)$:

$$\widehat{\alpha}_{IFM} = \arg\max_{\alpha} \sum_{t=1}^{T} \ln c\left(F_1(x_1^t;\widehat{\theta}_1),\ldots,F_N(x_n^t;\widehat{\theta}_N);\boldsymbol{\alpha}\right).$$

The IFM estimator is then defined as the vector $\boldsymbol{\theta}^{IFM} = (\widehat{\boldsymbol{\theta}},\widehat{\boldsymbol{\alpha}}_{IFM})$.

The Canonical Maximum Likelihood Method (CML)

Both the EML and IFM methods are based on an exogenous specification, and thus imposition, of the parametric form of the univariate marginals.[7] An alternative method, which does not imply any *a priori* assumption on the distributional form of the marginals, is the CML method and relies on the concept of the *empirical marginal transformation*. The transformation tends to approximate the unknown parametric marginals $\widehat{F}_n(\cdot)$, for $n = 1,\ldots,N$, with the empirical distribution functions $\widehat{F}_n(\cdot)$, defined as follows

$$\widehat{F}_n(\cdot) = \frac{1}{T}\sum_{t=1}^{T} 1_{\{X_{nt}\leq\cdot\}} \text{ for } n = 1,\ldots,N, \quad (3.17)$$

where $1_{\{X_{nt} \leq \cdot\}}$ represents the indicator function. The CML is then implemented via a two-stage procedure:

1. Transformation of the initial data set $X = (X_{1t}, X_{2t}, \ldots, X_{Nt})_{t=1}^{T}$ into uniform variates, using the empiricial marginal distribution—that is, for $t = 1, \ldots, T$, let $\widehat{u}_t = (\widehat{u}_1^t, \widehat{u}_2^t, \ldots, \widehat{u}_N^t) = \left\lfloor \widehat{F}_1(X_{1t}), \widehat{F}_2(X_{2t}), \ldots, \widehat{F}_N(X_{Nt}) \right\rfloor$.

2. Estimation of the vector of the copula parameters $\boldsymbol{\alpha}$, via the following relation:

$$\widehat{\boldsymbol{\alpha}}_{CML} = \arg\max_{\boldsymbol{\alpha}} \sum_{t=1}^{T} \ln c\left(\widehat{u}_1^t, \widehat{u}_2^t, \ldots, \widehat{u}_N^t; \boldsymbol{\alpha}\right).$$

The CML estimator is then defined as the vector $\boldsymbol{\theta}^{CML} = \widehat{\boldsymbol{\alpha}}_{CML}$.

3.5 Numerical Results for Calibrating Real-Market Data

In this section, we will present an application, as shown by Galiani (2003), of the CML method for calibrating the parameter of the Student's t copula to real-market data. We consider a portfolio of four stocks (Fiat, Merrill Lynch, Ericsson, and British Airways) with 990 daily observations spanning from August 14, 1999 to July 15, 2003. There are two approaches provided: the first developed by Bouyè *et al*., based on a recursive optimization procedure for the correlation matrix; the second, proposed by Mashal and Zeevi, based on the rank correlation estimator given by the Kendall's tau.

Bouyé, Durrelman, Nikeghbali, Riboulet, and Roncalli Method

This procedure is composed of a series of subsequent steps and is summarized and implemented as follows:

1. Starting from the random sample X of stock returns, transform the initial data set into the set of uniform variates $\widehat{U}$ using the empirical marginal transformation and using the canonical maximum likelihood method (CML) given in §3.4.3.

2. For each value of the degrees of freedom v on a specified range, assess the correlation matrix R_v^{CML} using the following routine:

   i. For each fixing date t, let $\xi_t = (t_v^{-1}(\widehat{u}_{1t}), t_v^{-1}(u_{2t}), \ldots, t_v^{-1}(\widehat{u}_{2t}))'$ for $t = 1, \ldots, T$.

   ii. Estimate the exact maximum likelihood (EML) estimator $\widehat{R}$ of the correlation matrix for the Gaussian copula, using equation (3.14). Then set $R_0 = \widehat{R}$.

   iii. Obtain $R_{v,k+1}$ via the following recursive scheme:[8]

$$R_{v,k+1} = \frac{v+N}{Tv} \sum_{t=1}^{T} \frac{\xi_t' \xi_t}{\left(1 + \frac{\xi_t' R_{v,k}^{-1} \xi_t}{v}\right)}$$

iv. Rescale matrix entries in order to have unit diagonal elements:

$$(R_{v,k+1})_{i,j} = \frac{(R_{v,k+1})_{i,j}}{\sqrt{(R_{v,k+1})_{i,i}\,(R_{v,k+1})_{j,j}}}$$

v. Repeat procedure iii–iv until $R_{v,k+1} = R_{v,k}$ and set $R_v^{CML} = R_{v,k}$.

3. Find the CML estimator v^{CML} of the degrees of freedom by maximizing the log-likelihood function of the Student's t copula density:

$$v^{CML} = \arg\max_{v\in\Theta} \sum_{t=1}^{T} \log c^{Student}(\widehat{u}_1^t, \widehat{u}_2^t, \ldots, \widehat{u}_N^t; R_v^{CML}, v)$$

Figure 3.3 plots the log-likelihood function of the t copula density as a function of the degrees of freedom, by which we can see that the estimated number of degrees of freedom is 10.

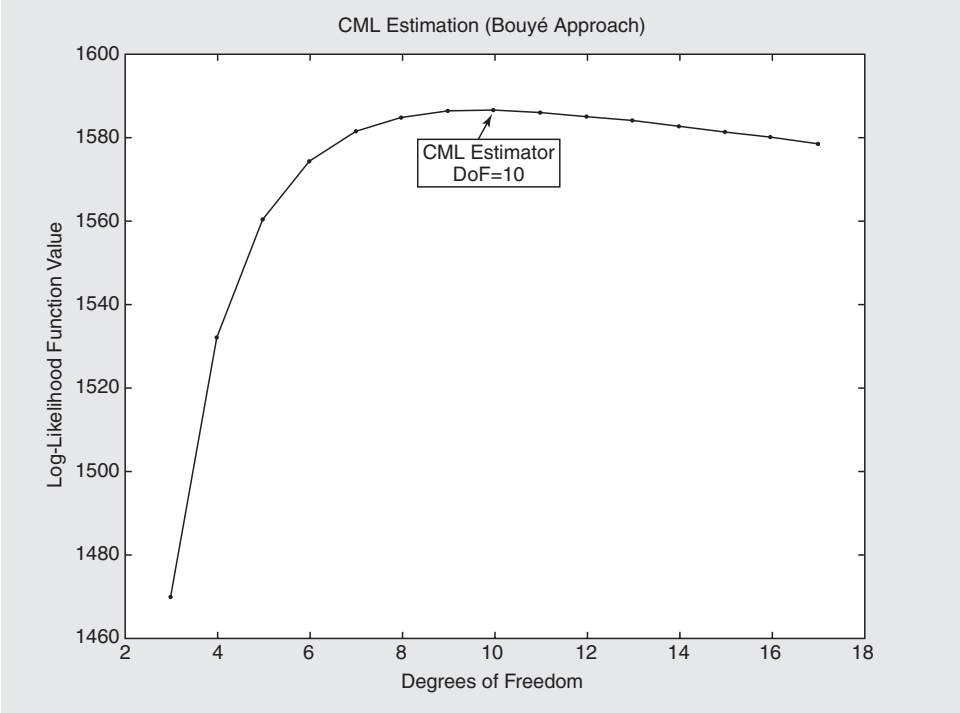


Figure 3.3 Log-likelihood function of the t copula density using Bouyè method

IMF_t_CORR.m

```
function [U_sample,CORR]=IFM_t_CORR(R,DoF,M)

% This function, starting from a random sample M, estimate the correlation
% matrix for the Student's t copula via the Bouye(2000), p.42 algorithm.
% Moreover it returns the uniform variate sample via the
% probability-integral transformation with Student's t margins.
% R:    correlation matrix for the gaussian copula
% DoF:  degrees of freedom for the multivariate t copula density
% M:    random sample of equity returns

N=size(R,2);
T=size(M,1);
U_emp=zeros(size(M));
U_st=zeros(size(M));
t_CORR=zeros(N);
CORR=zeros(N);

for n=1:N
    U_emp(:,n)=emp_dis(M(:,n));
end

for i=1:T
    U_st(i,:)=tinv(U_emp(i,:),DoF);
end

for k=1:100
    dummy_CORR=zeros(N);

        if k==1
            for i=1:T
                term=(U_st(i,:)'*U_st(i,:))/(1+(U_st(i,:)*
                       inv(R)*U_st(i,:)'/DoF));
                dummy_CORR=dummy_CORR+term;
            end
            t_CORR=dummy_CORR*(DoF+N)/(T*DoF);
            for xx=1:N
                for yy=1:xx
                   CORR(xx,yy)=t_CORR(xx,yy)/(sqrt(t_CORR(xx,xx))*
                                sqrt(t_CORR(yy,yy)));
                   CORR(yy,xx)=CORR(xx,yy);
                end
            end
        else
            for i=1:T
                term=(U_st(i,:)'*U_st(i,:))/(1+(U_st(i,:)*inv(CORR)*
                       U_st(i,:)'/DoF));
                dummy_CORR=dummy_CORR+term;
            end
            t_CORR=dummy_CORR*(DoF+N)/(T*DoF);
            for xx=1:N
                for yy=1:xx
                   CORR(xx,yy)=t_CORR(xx,yy)/(sqrt(t_CORR(xx,xx))*
                                sqrt(t_CORR(yy,yy)));
                   CORR(yy,xx)=CORR(xx,yy);
                end
            end
```

```
        end
end

U_sample=U_st;
```

emp_dis1.m

```
function X=emp_dis(Y)
% Compute the empirical marginal transformation, for the asset return
% samples, as described in Mashal(2002), "Beyond Correlation", pag.15
% As input we provide the i-th row of our d-columns sample
% i: #observation       d: #asset in the basket

X=zeros(length(Y),1);                   % create a vector of d dimension
count=0;
for k=1:length(Y)
    for s=1:length(Y)
        if Y(s)<=Y(k)
            count=count+1;
        end
    end
    X(k)=count/length(Y);               % compute the empirical
                                        % distribution function
    if X(k)==1
        X(k)=.999999;                   % otherwise, if 1, then we have
                                        % problem in using the t inverse
    end
    count=0;
end
```

t_copula_density2.m

```
function c=t_copula_density(U,DoF,Corr)

% gives the t-copula density for fixed input of "U" ( t-th row of the
% uniform sample with N elements), degrees of freedom "DoF", and
% correlation matrix "Corr"
%
N=length(U);
A=0;
B=0;
C=0;
D=0;
Block_1=0;
Block_2=1;
Block_3=0;

% Now, we split formula 4, pag. 12 of Mashal(2002) "Beyond Correlation" in
% 3 blocks

A=gamma((DoF+N)/2)*((gamma(DoF/2))^(N-1));
B=((gamma((DoF+1)/2))^N)*((det(Corr))^0.5);
Block_1=A/B;
```

```
for n=1:N
    C=(1+((U(n)^2)/DoF))^(-(DoF+1)/2);
    Block_2=Block_2*C;
end

D=U'*inv(Corr)*U/DoF;
Block_3=(1+D)^(-(DoF+N)/2);

c=(Block_1/Block_2)*Block_3;
```

pseudo_sample_IFM.m

```
% Starting from a random sample M, this script, generates a matrix (T*N) of
% uniform variates through the probability-integral transformation with
% gaussian margins.

% T:    # of observation in the time series for each asset
% N:    # of assets

M = xlsread('C:\copula\DataSet_new.xls'); % equity return database
N=size(M,2);
T=size(M,1);
ML_CORR=zeros(N);
Average=zeros(N,1);
Deviation=zeros(N,1);
U_nor1=zeros(size(M));
U_nor2=zeros(size(M));

for i=1:N
    Average(i)=mean(M(:,i));
    Deviation(i)=std(M(:,i),1);
    U_nor1(:,i)=normcdf((M(:,i)-Average(i))/Deviation(i));

end

for i=1:T
    U_nor2(i,:)=norminv(U_nor1(i,:));
    ML_CORR=ML_CORR+U_nor2(i,:)'*U_nor2(i,:);
end

ML_CORR=ML_CORR/T;
```

t_copula_CML

```
function MLE=t_copula_CML(DoF)
% calculate the target function to be maximized as described in Step 3
% pag. 43 of Mashal's (2002) paper
pseudo_sample
T=size(U,1);                    % number of observation for each stock
Log_L=zeros(T,1);
Sum_Log_L=0;
% For each observation compute the t-copula density and sum over
for t=1:T
    Log_L(t)=log(t_copula_density(U(t,:),DoF,CORR_EST));
```

```
    Sum_Log_L=Sum_Log_L+Log_L(t);
end

MLE=Sum_Log_L;
```

Mashal and Zeevi Method

The Bouyè procedure can be computationally heavy when dealing with several underlying assets and large data sets, and, as pointed out by Mashal and Zeevi (2002), affected by numerical instability due to the inversion of close to singular matrices. Consequently, Mashal and Zeevi propose to estimate the correlation matrix for the Student's t copula via a rank correlation estimator, namely the Kendall's tau, exploiting the result included in the following theorem:

Theorem 3: Let $X \sim E_N(\mu, \Sigma, \varphi)$, where for $i, j \in \{1, 2, \ldots, N\}$, X_i and X_J are continuous. Then,

$$\tau(X_i, X_j) = \frac{2}{\pi} \arcsin R_{i,j} \tag{3.18}$$

where $E_N(\mu, \Sigma, \varphi)$, denotes the N-dimensional elliptical distribution with parameters (μ, Σ, φ), and $\tau(X_i, X_j)$ and $R_{i,j}$ indicate, respectively, the Kendall's tau[9] and the Pearson's linear correlation coefficient for the random variables (X_i, X_j).

Proof: See Lindkog, McNeil, and Schmock (2001).

1. Starting from the random sample X of stock prices, transform the initial data set into the set of uniform variate $\widehat{U}$ using the empirical marginal transformations described in §3.4.3.

2. From equation (3.18), estimate the correlation matrix R^{CML}.

3. Find the CML estimator v^{CML} of the degrees of freedom by maximizing the log-likelihood function of the Student's t copula density:

$$v^{CML} = \arg\max_{v \in \Theta} \sum_{t=1}^{T} \log c^{Student}(\widehat{u}_1^t, \widehat{u}_2^t, \ldots, \widehat{u}_N^t; R^{CML}, v)$$

Figure 3.4 plots the log-likelihood function of the t copula density as a function of the degrees of freedom, by which we can see that the estimated number of degrees of freedom is 9.

The corresponding calibrated correlation matrix R_9^{CML} is shown in Table 3.1.

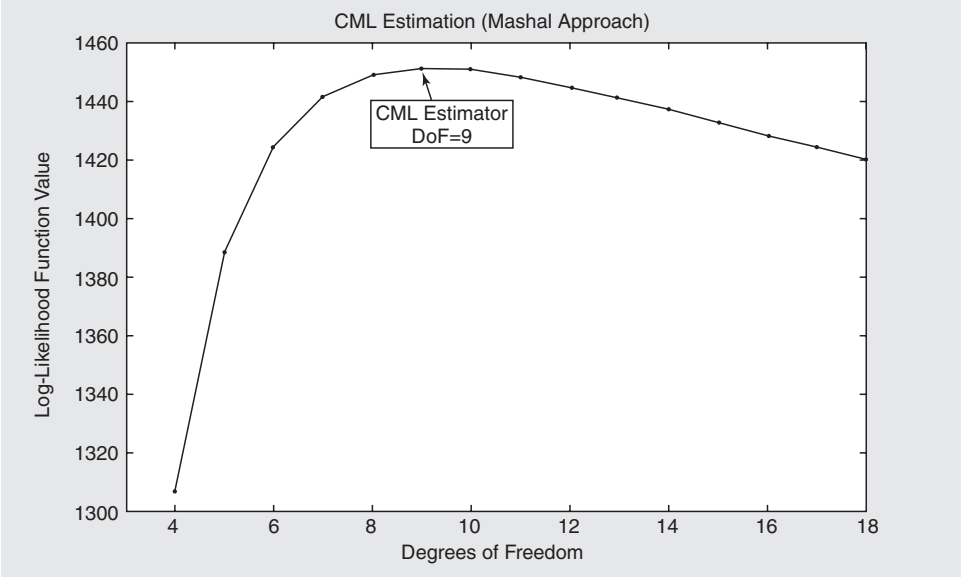


Figure 3.4 Log-likelihood function of the t copula density using Mashal and Zeevi method

Table 3.1 Correlation Matrix

1	0.34771	0.81475	0.77631
0.34771	1	0.62666	0.65706
0.81475	0.62666	1	0.77288
0.77631	0.65706	0.77288	1

The following is the Matlab implementation of the Mashal and Zeevi method:

t_copula_density3.m

```
function c=t_copula_density(U,DoF,Corr)

% gives the t-copula density for fixed input of "U" ( i-th row of the
% pseudo sample with d elements), degrees of freedom "DoF", and
% correlation matrix "Corr"
%
d=length(U);
y=zeros(d,1);          % create the vector of univariate t r.v.s.
z=zeros(d,1);
z=DoF;
y=(tinv(U,z))';        % vector the inverse of the t cdf

% Now, we split formula 4, pag. 12 of Mashal(2002) "Beyond Correlation"
% in 3 blocks

A=gamma((DoF+d)/2)*((gamma(DoF/2))^(d-1));
```

```
B=((gamma((DoF+1)/2))^d)*((det(Corr))^0.5);
Block_1=A/B;
A=0;
B=0;

Block_2=1;
for k=1:d
    C=(1+((y(k)^2)/DoF))^(-(DoF+1)/2);
    Block_2=Block_2*C;
    C=0;
end

D=y'*inv(Corr)*y/DoF;
Block_3=(1+D)^(-(DoF+d)/2);
D=0;

c=(Block_1/Block_2)*Block_3;
```

emp_dis2.m

```
function X=emp_dis(Y)
% Compute the empirical marginal transformation, for the asset return
% samples, as described in Mashal(2002), "Beyond Correlation", pag.15
% As input we provide the i-th row of our d-columns sample
% i: #observation       d: #asset in the basket

X=zeros(length(Y),1);                   % create a vector of d dimension
count=0;
for k=1:length(Y)
    for s=1:length(Y)
        if Y(s)<=Y(k)
            count=count+1;
        end
    end
    X(k)=count/length(Y);               % compute the empirical
                                        % distribution function
    if X(k)==1
        X(k)=.999999;                   % otherwise, if 1, then we have
                                        % problem in using the t inverse
    end
    count=0;
end
```

kendall.m

```
function TAU=KENDALL(A,B);

% This function computes the Kendall's tau for a bivariate vector of
% observations

A=A(:);
B=B(:);
N=length(A);
N1=0;N2=0;S=0;
```

```
for J=1:N
    A1=A(J+1:N)-A(J);
    A2=B(J+1:N)-B(J);
    AA=A1.*A2;
    i=find(AA>=0);
    lni=length(i);
    k=find(~AA(i));                    % find the zero products.
    dis=length(AA)-lni;                % discordant pairs.
    con=lni-length(k);                 % concordant pairs
    l=length(find(~(~A1(i(k)))));      % add up the extra zeros
    m=length(find(~(~A2(i(k)))));
    N1=N1+con+dis+l;
    N2=N2+con+dis+m;
    S=S+con-dis;
end
TAU=S/sqrt(N1*N2);
```

pseudo_sample.m

```
% Starting from a random sample M, this script creates a pseudo sample and
% compute the estimated correlation matrix using Kendall's Tau as described
% in Mashal(2001), "Beyond Correlation", p.41.

% T:    # of observation in the time series for each asset
% N:    # of assets

M = xlsread('C:\copula\DataSet_new.xls'); % equity price database
N=size(M,2);
T=size(M,1);
U=zeros(size(M));
corr_matrix=zeros(N);

% for each underlying asset we calculate the empirical distribution
for n=1:N
    U(:,n)=emp_dis(M(:,n));
end

% create the correlation matrix (via Kendall's tau), using the procedure
% described in page 42-42 of Mashal(2001) "Beyond Correlation"

for i=1:N
    for j=i:N
        if i==j
            corr_matrix(i,j)=1;
        else
        corr_matrix(i,j)=sin(pi*0.5*KENDALL(U(:,i),U(:,j)));
        corr_matrix(j,i)=corr_matrix(i,j);
        end
    end
end


CORR_EST=corr_matrix;
CORR_LIN=corrcoef(U);
```

The corresponding calibrated correlation matrix R^{CML} is shown in Table 3.2.

Table 3.2 Calibrated Correlation Matrix

1	0.44818	0.90208	0.83975
0.44818	1	0.67615	0.68552
0.90208	0.67615	1	0.84178
0.83975	0.68552	0.84178	1

3.6 USING COPULAS IN EXCEL

Figure 3.5 shows random Monte Carlo simulations to value a portfolio using bivariate Frank copula. Figure 3.6 shows a simulation for the Clayton copula.

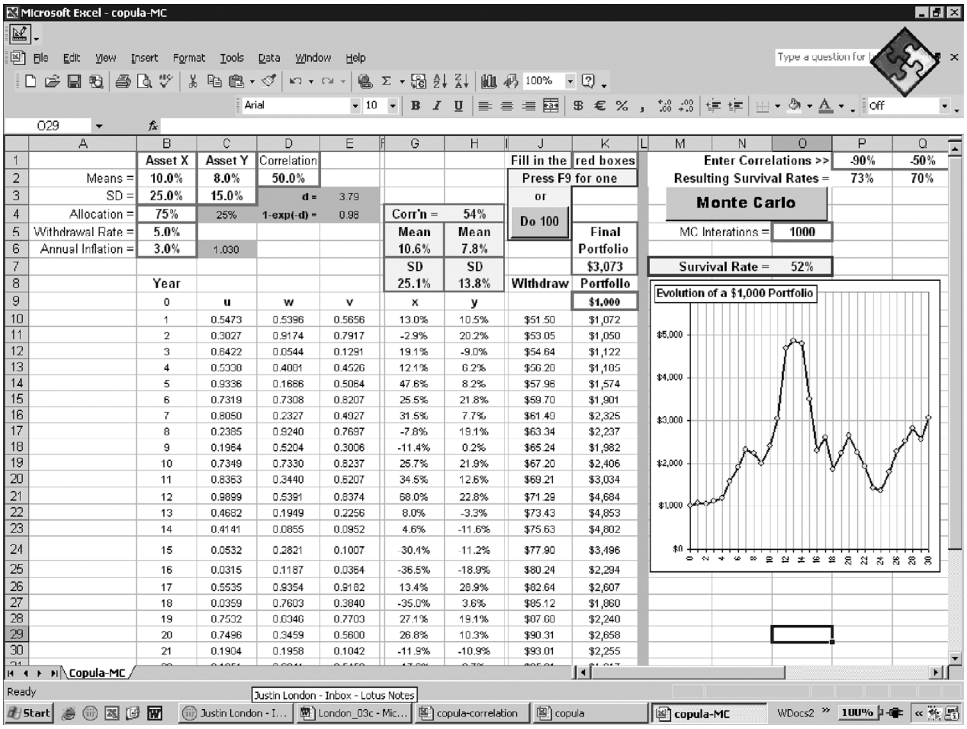


Figure 3.5 Random Monte Carlo simulations

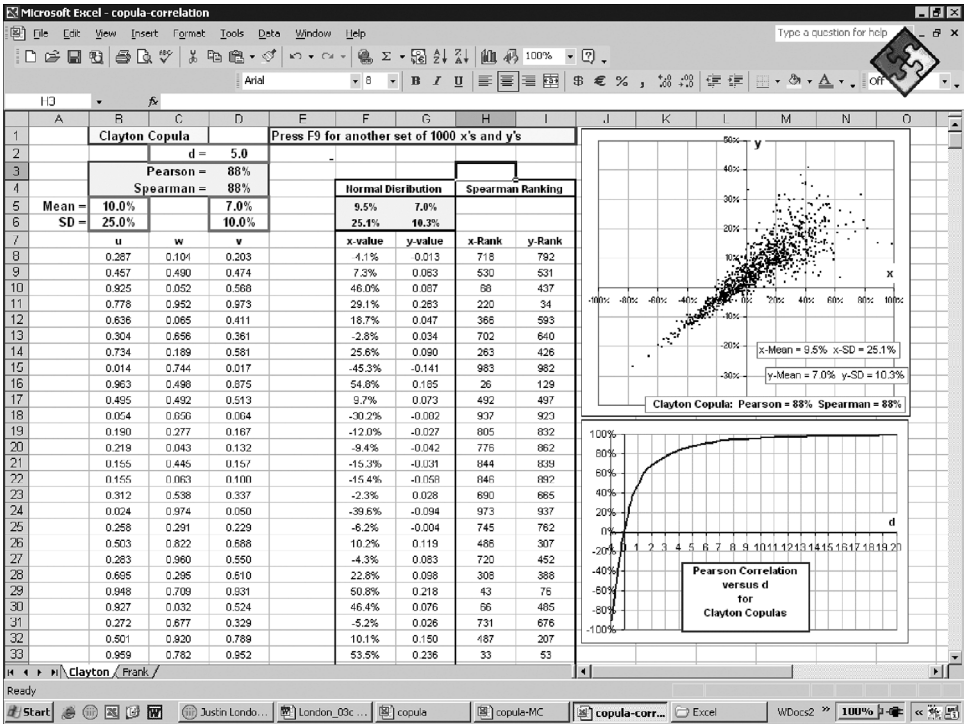


Figure 3.6 Clayton copula simulation

ENDNOTES

1. The volume $V_C(\mathbf{B})$ of a n-box $\mathbf{B} = [\mathbf{a}, \mathbf{b}]$ is defined as follows:

$$V_C(\mathbf{B}) = \sum_d \operatorname{sgn}(\mathbf{d}) C(\mathbf{d})$$
$$= \sum_{i_1=1}^{2} \sum_{i_2=1}^{2} \cdots \sum_{i_n=1}^{2} (-1)^{i_1+i_2+\ldots+i_n} C(d_{1i_1}, d_{2i_2}, \ldots, d_{ni_n}) \geq 0$$

   where $d_{j1} = a_j$ and $d_{j2} = b_j$ for all $j \in \{1, 2, \ldots, n\}$.
2. In the univariate case, the density function $f(x)$ of a random variable X can be obtained by the cumulative distribution function via the following relation:

$$f(x) = \frac{\partial F(x)}{\partial x}.$$

3. Based on the definition of Fang, Kotz, and Ng (1987), if $\mathbf{X}$ is an n-dimensional vector of random variables and for some $\boldsymbol{\mu} \in \Re^n$, and some nxn nonnegative definite, symmetric matrix $\boldsymbol{\Sigma}$, the characteristic function $\varphi_{X-\mu}(\mathbf{t})$ of $\mathbf{X} - \boldsymbol{\mu}$ is a function of the quadratic form $\boldsymbol{t'\Sigma t}$, then $\mathbf{X}$ has an elliptical distribution with parameters $(\boldsymbol{\mu}, \boldsymbol{\Sigma}, \boldsymbol{\varphi})$ and we write $\mathbf{X} \sim \mathbf{E}^n(\boldsymbol{\mu}, \boldsymbol{\Sigma}, \boldsymbol{\varphi})$.

4. Following Johnson and Kotz (1972), p. 134, given a random vector $\mathbf{X} = (X_1, \ldots, X_n)'$ with a joint standardized multinormal distribution with correlation matrix R and a χ^2_v-distributed random variable S, independent from $\mathbf{X}$, we define the standardized multivariate Student's t joint density function with correlation matrix R and v degrees of freedom, as the joint distribution function of the random vector $\mathbf{Y} = (\frac{X_1}{S/\sqrt{v}}, \ldots, \frac{X_n}{S/\sqrt{v}})'$:

$$f(\mathbf{y}) = \frac{\Gamma\left(\frac{v+n}{2}\right)}{\Gamma\left(\frac{v}{2}\right)} \frac{1}{(\pi v)^{n/2} |R|^{1/2}} \left(1 + \frac{\boldsymbol{y}' R^{-1} \boldsymbol{y}}{v}\right)^{-\frac{(v+n)}{2}}$$

5. A function $f(t)$ is said to be completely monotone on the interval D if it has derivatives of all orders which alternate in sign, that is

$$(-1)^k \frac{d^k}{dt^k} f(t) \geq 0$$

for all t in the interior of D and $k = 0, 1, 2, 3....$

6. Mashal and Naldi (2002) suggest procedures for estimating the parameters of the marginals based on a numerical optimization routine for the likelihood of the univariate returns. Namely, starting from a univariate return sample, $\{X_t\}_{t=1}^T$, they construct a new sample $\{\widehat{X}_t\}_{t=1}^T = \left\{\frac{X_t - m}{H}\right\}_{t=1}^T$ where m is the location parameter and H is the scale factor, to be distributed as a Student's t random variable with v degrees of freedom. Therefore, given a parameter space $\Theta = \{\theta : m \in \Re, H > 0, v > 2\}$ with $\theta = (m, H, v)$, they define the maximum likelihood estimator $\widehat{\theta} = \left(\widehat{m}, \widehat{H}, \widehat{v}\right)$ as $\widehat{\theta} = \arg\max_{\theta \in \Theta} \prod_{t=1}^T f_v^{student}\left(\widehat{X}_t\right)$.
7. Note that for equation (3.15), we have fixed $\varsigma = (t_v^{-1}(u_1), t_v^{-1}(u_2), \ldots, t_v^{-1}(u_N))'$. Similarly, with regard to equation (3.13), we have $\varsigma = (\Phi^{-1}(u_1), \ldots, \Phi^{-1}(u_N))'$.
8. This equation is derived by the maximum output of the log-likelihood function for the t copula density given in equation (3.15):

$$\frac{\partial \ell^{Student}(\theta)}{\partial R^{-1}} = \frac{T}{2} R - \frac{v+N}{2} \sum_{t=1}^T \frac{\frac{\xi_t' \xi_t}{v}}{\left(1 + \frac{\xi_t' R^{-1} \xi_t}{v}\right)}$$

from where we note that the ML estimator of R must satisfy the following equation:

$$R_{ML} = \frac{v+N}{Tv} \sum_{t=1}^T \frac{\xi_t' \xi_t}{\left(1 + \frac{\xi_t' R_{ML}^{-1} \xi_t}{v}\right)}$$

9. Using the definition given by Embrechts, Lindskog, and McNeil (2001), let (x, y) and $(\widetilde{x}, \widetilde{y})$ be two observations from a vector (X, Y) of continuous random variables. Then (x, y) and $(\widetilde{x}, \widetilde{y})$ are said to be concordant if

$$(x - \bar{x}) - (y - \bar{y}) > 0$$

and discordant otherwise. The Kendall's tau is then defined as the probability of concordance minus the probability of discordance; namely, given two pairs (X, Y) and $\left(\widetilde{X}, \widetilde{Y}\right)$

of independent random variables with the same distribution $F(.,.)$, we have:

$$\tau = P\left[(X - \bar{X})(Y - \bar{Y}) > 0\right] - P\left[(X - \bar{X})(Y - \bar{Y}) < 0\right]$$

As suggested by Meneguzzo and Vecchiato (2002), given two series X_t and Y_t with $t = 1, \ldots, T$, the consistent estimator of the Kendall's tau is then computed as

$$\tau = \frac{2}{T(T-1)} \sum_{i<j} \operatorname{sgn}\left[(X_i - X_j)(Y_i Y_j)\right],$$

where

$$\operatorname{sgn}(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ -1 & \text{if } x < 0 \end{cases}$$

CHAPTER 4

MORTGAGE-BACKED SECURITIES

SECTIONS

Mortgage-backed securities (MBSs) and mortgage pass-throughs (PT) are claims on a portfolio of mortgages. MBSs are created when a federal agency, mortgage banker, bank, or investment company buys up mortgages of a certain type—i.e., FHA (Federal Home Administration) or VA (Veterans' Administration) insured—and then sells claims on the cash flows from the portfolio as MBSs, with the proceeds of the MBS sale being used to finance the purchase of the mortgages. There are two types of MBS: agency and conventional (private-label).[1]

Agency MBS, such as a GNMA pass-through, are securities with claims on a portfolio of mortgages insured against default risk by FHM, VA, or FmHA (Farmers Mortgage Home Administration). A mortgage banker, bank, or investment company presents a pool of FHA, VA, or FmHA mortgages of a certain type (30-year fixed, 15-year variable rate, etc.) to GNMA (Ginnie Mae). If the mortgage pool is in order, GNMA will issue a separate guarantee that allows the MBSs on the mortgage pool to be issued as a GNMA PT. Other agency MBSs include the Federal Home Loan Mortgage Corporation (FHLMC) MBSs, which are claims on a portfolio of conventional mortgages. The FHLMC issues agency MBSs, whereby the FHLMC buys mortgages from the mortgage originator, and then creates an MBS referred to as a *participation certificate*, which it issues through a network

of dealers. FHLMC has a swap program whereby FHLMC swaps MBSs for a savings and loan's or commercial bank's portfolio of mortgages of a certain type. Other government agencies such as FNMA (Fannie Mae) issue several types of MBS: participation certificates, swaps, and PTs. With these certificates, homeowners' mortgage payments pass from the originating bank through the issuing agency to the holds of the certificates.

Conventional types, also known as private-label types, are issued by commercial banks (via their holding companies), S&Ls, mortgage bankers, and investment companies. Conventional issued MBSs include those issued by Prudential Home, Chase Mortgage, Citi-Corp Housing, Ryland/Saxon, GE Capital, and Countrywide. Conventional PTs must be registered with the SEC. These PTs are often insured with external insurance in the form of a letter of credit (LOC) of the private-label issuer, as well as internal insurance through the creation of senior and junior classes of the PT structured by the private-label issuer.

There is both a primary and a secondary market for MBS. In the primary market, investors buy MBSs issued by agencies or private-label investment companies either directly or through dealers. Many of the investors are institutional investors. Thus, the creation of MBS has provided a tool for having real estate financed more by institutions. In the primary market, MBS issue denominations are typically between $25,000 to $250,000 (with some as high as $1M) and some have callable features. In the secondary market, MBSs are traded over-the-counter (OTC). OTC dealers are members of the Mortgage-Backed Securities Dealer Association (MSDA).

MBSs are some of the most complex securities to model and value due to their sensitivity to prepayment and interest rates, which affects the timing, frequency, and size of cash flows to investors. Cash flows (CFs) from MBSs are the monthly CFs from the portfolio of mortgages (referred to as the *collateral*). Cash flows include interest on principal, scheduled principal, and prepaid principal. Cash flow analysis is essential in the valuation of any MBS given their impact by the underlying features of the MBS, including *weighted average maturity* (WAM), *weighted average coupon rate* (WAC), *pass-through rate* (PT rate), and *prepayment rate* or *speed*. The WAM is effectively the duration, or weighted length of time, of all the payment of MBS cash flows to be paid out to investors. The WAC is the rate on a portfolio of mortgages (collateral) that is applied to determine scheduled principal. The PT rate is the interest on principal and is lower than the WAC, with the difference going to the MBS issuer. The prepayment rate or speed is the assumed prepayment rate made by homeowners of mortgages in the pool.

In this chapter, we discuss MBS pricing and modeling in detail. In §4.1, we discuss prepayment and PSA models for MBS pricing. In §4.2, we give numerical examples using Excel of how the prepayment models work. In §4.3, we discuss MBS pricing, quoting, and the value and return to investors based on different prepayment and interest rate assumptions. In §4.4, we discuss prepayment risk and the average life of MBS. In §4.5, we review in detail a numerical implementation in C++ and Excel for valuation and cash flow analysis of MBS using Monte Carlo simulation. In §4.6, we give numerical examples using the Fixed-Income Toolbox in Matlab. In §4.7, we discuss MBS derivatives, including collateralized mortgage obligations (CMOs) and sequential-pay tranche structures. We give examples using Excel. In §4.8, we give an implementation of a CMO in C++. In §4.9, we discuss planned amortization classes (PAC) and their structures. In §4.10, we review stripped MBS, including interest-only (IO) and principal-only (PO) securities. In §4.11,

we discuss interest rate risk of MBSs. Finally, in §4.12, we discuss hedging MBS and using MBS for balance sheet asset-liability management.

4.1 PREPAYMENT MODELS

MBS valuation models typically assume a prepayment rate or speed. Investors and issuers apply different prepayment models in analyzing MBS. Most models, though, are compared to a benchmark model or rate. The benchmark model is the one provided by the Public Securities Association (PSA). PSA measures speed by the Conditional Prepayment Rate (CPR). CPR is the proportion of the remaining mortgage balance that is prepaid each month and is quoted on an annual basis. The monthly rate is referred to as the Single-Monthly Mortality rate (SMM) and is given by:

$$SMM = 1 - (1 - CPR)^{1/12} \tag{4.1}$$

The estimated monthly prepayment is:

$$\textit{Monthly prepayment} = SMM \cdot [\textit{Beginning of month balance} - \textit{Sched. prin. for month}]$$

For example, if CPR $= 6\%$, beginning-of-the-month balance $= \$100$M, and scheduled principal for month $= \$3$M, then the estimated prepaid principal for the month would be $\$0.499$M:

$$SMM = 1 - [1 - .06]^{1/12} = .005143$$
$$\textit{Monthly prepaid principal} = .005143[\$100M - \$3M] = \$0.499M$$

In the PSA model, CPR depends on the maturity of the mortgages. PSA's standard model assumes that for a 30-year mortgage (360 months), the CPR is equal to .2% the first month, grows at that rate for 30 months to equal 6%, and stays at 6% for the rest of the mortgage's life. This model is referred to as the 100% PSA model. Figure 4.1 shows the prepayment rate as a function of time in months.

The estimation of CPR for month t is:

$$CPR = \begin{cases} 0.06\left(\frac{t}{30}\right), & \text{if } t \le 30 \\ 0.06, & \text{if } t > 30 \end{cases} \tag{4.2}$$

As an example, the CPR for month five is:

$$CPR = .06\left(\frac{5}{30}\right) = .01$$
$$SMM = 1 - [1 - .01]^{1/12} = .000837$$

PSA's model can be defined in terms of different speeds by expressing the standard model (100% PSA) in terms of a higher or lower percentage, such as 150% or 50%. In a period of lower rates, the PSA model could be 150%, and in a period of higher rates, it

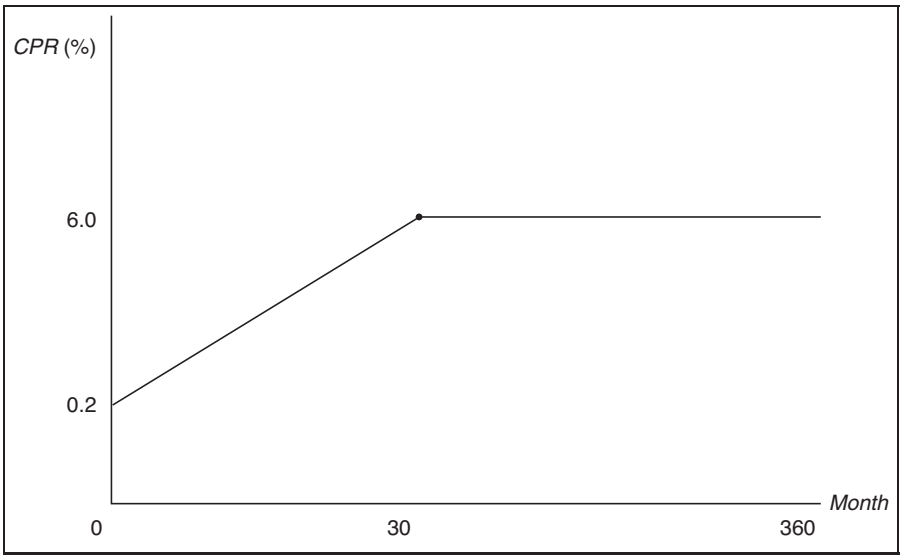


Figure 4.1 100% PSA Model

could be 50%. For the 100% PSA model, the average time a 30-year mortgage is held is 17 years; for a 225% PSA model, it is 8 years. Figure 4.2 shows the different prepayment rates as a function of time in months.

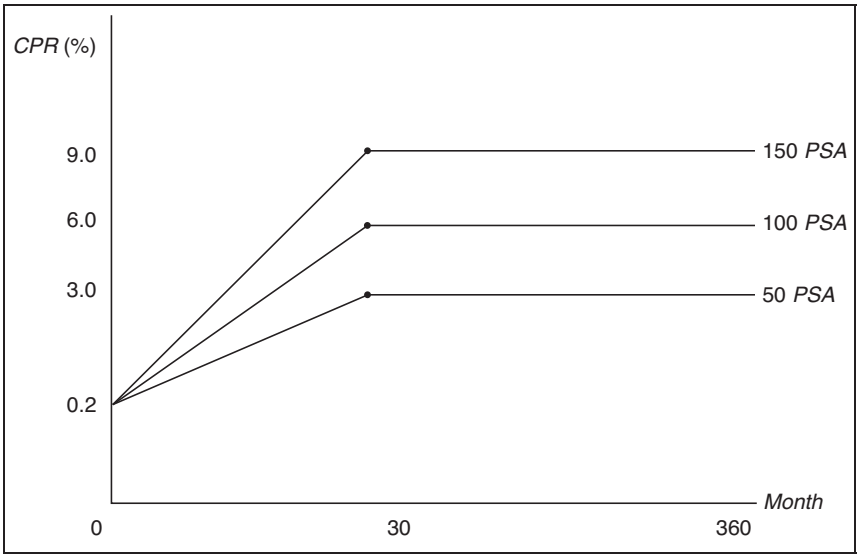


Figure 4.2 PSA Models

Suppose we want to compute the CPR and SMM for month five with 165 PSA speed. Then we compute the following based on (4.1) and (4.2):

$$\begin{aligned} \textit{CPR} &= .06\left(\frac{5}{30}\right) = .01 \\ 165\textit{CPR} &= 1.65(.01) = .0165 \\ \textit{SMM} &= 1 - [1 - .0165]^{1/12} = .0001386 \end{aligned}$$

4.2 Numerical Example of Prepayment Model

Let $p =$ monthly scheduled mortgage payment, $F_0 =$ the face value of the underlying mortgage pool of the MBS, $M = \textit{WAM} =$ weighted average of the number of months remaining until maturity, $I =$ interest rate payment, $\textit{SP} =$ scheduled principal payment, $\textit{PP} =$ prepaid principal, $R_A =$ annual interest rate (WAC), $B_i, i = 1, \ldots, 360$, the remaining mortgage balance in month i, and $\textit{CF}_i, i = 1, \ldots, 360$, the cash flow in the ith month. Note that the balance in month 1 is the initial face value of the MBS pool, $B_1 = F_0$. The following formula gives p:

$$p = \frac{F_0}{\left(\frac{1-1/(1+(R_A/12))^M}{R_A/12}\right)} \tag{4.3}$$

Consider a mortgage portfolio with an underlying face value of \$100M, a WAC $= 9\%$, a WAM $= 360$ months, and a prepayment speed $= 100\%$ PSA. We need to compute the various cash flows for the first month. The first monthly principal payment is as follows:

$$p = \frac{\$100M}{\left(\frac{1-1/(1+(.09/12))^{360}}{.09/12}\right)} = \$804,600$$

The interest payment is:

$$I = \left(\frac{.09}{12}\right)\$100M = \$750,000$$

The scheduled principal payment is:

$$\textit{SP} = \$804,600 - \$750,000 = \$54,600$$

The estimated prepaid principal in the first month is:

$$\begin{aligned} \textit{CPR} &= \left(\frac{1}{30}\right).06 = .002 \\ \textit{SMM} &= 1 - [1 - .002]^{1/12} = .0001668 \\ \textit{PP} &= .0001668[\$100M - \$54,620] = \$16,667 \end{aligned}$$

The first-year cash flow is computed as:

$$\begin{aligned} CF_1 &= p + PP + I \\ &= \$804,600 + \$750,000 + \$16,667 = \$821,295 \end{aligned}$$

The beginning balance for the second month is:

$$\begin{aligned} B_2 &= B_1 - SP - PP \\ &= \$100M - \$0.0546 - \$0.01667M = \$99.9287M \end{aligned}$$

The second-month cash flows are computed as follows. The second monthly payment is:

$$p = \frac{\$99.9287M}{\left[\frac{1-1/(1+(.09/12))^{359}}{.09/12}\right]} = \$804,488$$

The interest payment is:

$$I = \left(\frac{.09}{12}\right)\$99.9287M = \$749,465$$

and the scheduled principal payment is:

$$SP = \$804,488 - \$749,465 = \$55,023$$

The estimated prepaid principal is:

$$\begin{aligned} CPR &= \left(\frac{2}{30}\right).06 = .004 \\ SMM &= 1 - [1 - .004]^{1/12} = .0003339 \\ PP &= .0003330[\$99.9287M - \$55,023] = \$33,352 \end{aligned}$$

Thus, the second-month cash flows are computed as:

$$CF_2 = \$749,400 + \$55,023 + \$33,352 = \$837,840$$

The remaining month cash flows are computed similarly.

Consider now a mortgage portfolio with a face value of \$100M, a WAC $= 8.125\%$, a WAM $= 357$ months, a PT rate $= 7.5\%$, and a prepayment $= 165\%$ PSA. Note that because the WAM is not 360 months, but rather 357 months, the pool age is "seasoned" so that the first month of payments actually starts in month four, and not month one. Moreover, interest payments are calculated using the PT rate. However, scheduled principal and mortgage payments are computed using the WAC rate.

In this example, the schedule monthly mortgage payment is:

$$p = \frac{\$100M}{\left[\frac{1-1/(1+(.08125/12))^{357}}{.08125/12}\right]} = \$743,970$$

The interest payment (which uses the PT rate) is

$$I = \left(\frac{.075}{12}\right)\$100M = \$625,000$$

and the scheduled principal payment (which uses the WAC rate) is

$$SP = \$743,970 - (.08125/12)(\$100M) = \$66,880.$$

The estimated prepaid principal using the 165% PSA model is:

$$CPR = 1.65\left(\frac{4}{30}\right).06 = .0132$$
$$SMM = 1 - [1 - .0132]^{1/12} = .0011067$$
$$PP = .0011067[\$100M - \$.06688] = \$110,600$$

The first-month cash flow (starting in month four) is:

$$CF_1 = \$625,000 + \$66,880 + \$110,600 = \$802,480$$

The beginning mortgage balance for month two is:

$$\$100M - \$66,880 - \$110,600 = \$99.822M$$

The scheduled monthly mortgage payment in the second month is:

$$p = \frac{\$99.822M}{\left[\frac{1-1/(1+(.08125/12))^{356}}{.08125/12}\right]} = \$743,140$$

The second-month interest payment is:

$$I = \left(\frac{.075}{12}\right)\$99.822M = \$623,890$$

The scheduled principal payment in month two is:

$$SP = \$743,140 - (.08125/12)(\$99.822M) = \$67,260$$

The estimated prepaid principal is:

$$CPR = 1.65\left(\frac{5}{30}\right).06 = .0165$$
$$SMM = 1 - [1 - .0165]^{1/12} = .00139$$
$$PP = .00139[\$99.822M - \$.06726] = \$138,210$$

Thus, the second-month cash flow is computed as:

$$CF_2 = \$623,890 + \$67,260 + \$138,210 = \$829,360$$

Table 4.1 shows the cash flows for the first few months.

The Excel spreadsheet MBS1.xls shows the complete computations for every month. Parameters can be changed (for different assumptions) to generate different cash flows.

Table 4.1

Month	Balance	Sch. Payment	Interest	Sch.Pr.	CPR	SMM	Prepayment	CF	Ending Bal.
1	100,000,000.00	743,967.06	625,000.00	66,883.73	0.0132	0.0011067	110,597.15	802,480.87	99,822,519.13
2	99,822,519.13	742,153.62	623,890.74	66,271.98	0.0165	0.0013855	138,213.22	828,375.94	99,618,033.93
3	99,618,033.93	740,145.25	622,612.71	65,648.15	0.0198	0.0016652	165,771.24	854,032.10	99,386,614.54
4	99,386,614.54	737,942.81	621,166.34	65,012.61	0.0231	0.0019457	193,248.74	879,427.69	99,128,353.20
5	99,128,353.20	735,547.31	619,552.21	64,365.76	0.0264	0.0022271	220,623.21	904,541.17	98,843,364.23
6	98,843,364.23	732,959.93	617,771.03	63,707.98	0.0297	0.0025093	247,872.18	929,351.19	98,531,784.07
7	98,531,784.07	730,181.99	615,823.65	63,039.70	0.0330	0.0027925	274,973.22	953,836.56	98,193,771.16
8	98,193,771.16	727,214.96	613,711.07	62,361.30	0.0363	0.0030765	301,903.97	977,976.35	97,829,505.88

Source: Johnson, S. (2004)

4.3 MBS Pricing and Quoting

The prices of an MBS are quoted as a percentage of the underlying mortgage balance. The mortgage balance at time t, F_t, is quoted as a proportion of the original balance. This is called the pool factor pf_t:

$$pf_t = \frac{F_t}{F_0} \tag{4.4}$$

Suppose, for example, an MBS backed by a collateral mortgage pool originally worth \$100M, a current pf of .92, and quoted at $95 - 16$ (note: 16 is 16/32) would have a market value of \$87.86M, as calculated:

$$\begin{aligned} F_t &= (pf_t)F_0 \\ &= (.92)(\$100M) = \$92M \end{aligned}$$

so that

$$\text{Market Value} = (.9550)(\$92M) = \$87.86M$$

The market value is the clean price; it does not take into account accrued interest, denoted AI. For an MBS, accrued interest is based on the time period from the settlement date (two days after the trade) to the first day of the next month. For example, if the time period is 20 days, the month is 30 days, and the WAC = 9%, then AI is \$.46M:

$$AI = \left(\frac{20}{30}\right)\left(\frac{.09}{12}\right)\$92M = \$460,000$$

The full market value would be \$88.32M:

$$\text{FullMktValue} = \$87.86M + \$460,000 = \$88.32M$$

The market price per share is the full market value divided by the number of shares. If the number of shares is 400, then the price of the MBS based on a $95 - 16$ quote would be \$220,080:

$$\textit{MBS price} = \frac{\$88.32M}{400} = \$220,800$$

The value of an MBS is equal to the present value (PV) of security's cash flows (CFs); thus, the value is a function of the MBS's expected CFs and the interest rate. In addition, for MBS, the CFs are also dependent on rates R: A change in rates will change the prepayment of principal and either increase or decrease early CFs:

$$V_{MBS} = f(CFs, R)$$

where

$$CF = f(R).$$

Since cash flows, CFs, are a function of rates, the value of MBS is more sensitive to interest rate changes than a similar corporate bond. This sensitivity is known as *extension risk*. Note the following relationships:

$$\textit{if } R \downarrow \Rightarrow \text{lowerdiscountrate} \Rightarrow V_M \uparrow \text{ (just like any other bond)}$$

and

$$\begin{aligned}&\textit{if } R \downarrow \Rightarrow \text{Increases prepayment} :\Rightarrow V_M \uparrow\\ &\quad \text{Earlier CFs} \uparrow\end{aligned}$$

On the other hand,

$$\textit{if } R \uparrow \Rightarrow \text{higherdiscountrate} \Rightarrow V_M \downarrow$$

and

$$\begin{aligned}&\textit{if } R \uparrow \Rightarrow \text{Decreases prepayment} :\Rightarrow V_M \downarrow\\ &\qquad \Rightarrow \text{Earlier CFs} \downarrow\end{aligned}$$

so that an increase in rates will reduce the market value of the MBS, leading to extension risk.

There are various exogenous and endogenous factors that influence prepayment other than refinancing rates. One is housing turnover—the long-term rate at which borrowers in a pool prepay their mortgages because they sell their homes. Another is the seasoning period, the number of months over which base voluntary prepayments (housing turnover, cash-out refinancing, and credit upgrades, but not rate refinancing or defaults) are assumed to increase to long-term levels. Other factors include credit curing—the long-term rate at which borrowers prepay their mortgages because improved credit and/or increased home pool prices enable them to get better rates and/or larger loans. As the pool burns out, the rate of curing declines.[2] Default, expressed as a percentage of the PSA Standard Default Assumption (SDA), affects prepayment, as well as the maximum rate-related CPR for burnout—CPR is lower for a pool that has experienced no prior rate-related refinancing. The lower the ratio, the faster the pool burns out.[3]

Many Wall Street firms use proprietary reduced-form prepayment models that use past prepayment rates and endogenous variables to explain prepayment. These models are calibrated to fit observed payment data, unrestricted by theoretical considerations.[4]

4.4 Prepayment Risk and Average Life of MBS

Average life is the weighted average of the MBS's or MBS collateral's time periods, with the weights being the periodic cash flow payments divided by the total principal. For example, the original average life of the 30-year, \$100M, 9%, 100 PSA mortgage (the first example in §4.2) portfolio is 12.077 years, computed as follows:

$$\textit{Ave. life} = \frac{1}{12}\frac{(1(\$71,295) + 2(\$88,376) + \ldots + 360(\$135,281))}{\$100,000,000} = 12.077$$

In general, the average life of the MBS can be computed by the following formula:

$$\textit{Ave. life} = \frac{1}{12}\frac{\sum_{i=1}^{360} i * CF_i}{F_0} \tag{4.5}$$

Prepayment risk can be measured in terms of how responsive (sensitive) an MBS's or MBS collateral's average life is to changes in prepayment speed (change in PSA) or equivalently to changes in rates (because rate changes are the major factor affecting speed):

$$\text{prepayment risk} = \frac{\Delta \textit{Ave. life}}{\Delta \textit{PSA}} \cong \frac{\Delta \textit{Ave. life}}{\Delta R} \tag{4.6}$$

An MBS or its collateral would have zero prepayment risk if

$$\text{prepayment risk} = \frac{\Delta \textit{Ave. life}}{\Delta \textit{PSA}} = 0.$$

One of the more significant innovations in finance occurred in the 1980s with the development of derivative MBS, such as Planned Amortization Classes (PACs), which had different prepayment risk features, including some derivatives with zero prepayment risk.

Assumptions of prepayment rates can be made based on the probability of refinancing rates changing. For example, if there is a high probability that the Federal Open Markets Committee (the Fed) will lower rates (based, for example, on media reports that they intend to do so in the near future), refinancing rates can be expected to fall as well so that more homeowners will refinance their mortgages at lower rates. This in turn will increase the speed of prepayment and thus of the cash flows to investors. PSA rates should then be adjusted upward. Conversely, if the Fed is expected to raise rates as a response to, say, inflation, refinancing rates can be expected to rise, decreasing prepayment risk and lengthening the average life of the MBS. PSA rates should then be adjusted downward.

The best way to model refinancing rate scenarios is through Monte Carlo simulation. One first constructs an interest rate tree—i.e., a binomial tree[5]—with both the spot rates and refinancing rates at each node. One runs many simulation paths sampling from possible interest rate paths that rates could possibly take in the tree. For each simulation path, one estimates the cash flows based on the refinancing rates at each step along the path. (Each time step along the path corresponds to a time step made in the short-rate tree.) Specifically, Monte Carlo simulation can be used to determine the MBS's theoretical value or rate of return through the following steps:

1. Simulate interest rates. Use a binomial interest-rate tree to generate different paths for spot rates and refinancing rates.

2. Specify a prepayment model based on the spot rates.

3. Generate CF paths for a mortgage portfolio, MBS, or tranche.

4. Determine the PV of each path, the distribution of the path, the average (theoretical value), and standard deviation. Alternatively, given the market value, determine each path's rate of return, distribution, average, and standard deviation.

Step 1

In step 1, to simulate interest rates, we generate interest rate paths from a binomial interest-rate tree.[6] For example, assume a three-period binomial tree of one-year spot rates, R_t^S, and refinancing rates, R_t^{ref}, where $R_0^S = 6\%$, $R_0^{ref} = 8\%$, $u = 1.1$, and $d = .9091 = 1/1.1$. With three periods, there are four possible rates after three periods (years), and there are eight possible paths in the binomial tree shown in Figure 4.3. Table 4.2 shows eight short-rate paths simulated from the preceding binomial tree (the eight possible paths rates can take in the tree).

Suppose we have a mortgage portfolio with a par value of \$1M, a WAM $= 10$ years, a WAC $= 8\%$, PT rate $= 8\%$, annual cash flow payments, the mortgages are insured against default risk, and has a balloon payment at the end of year 4 equal to the balance at the

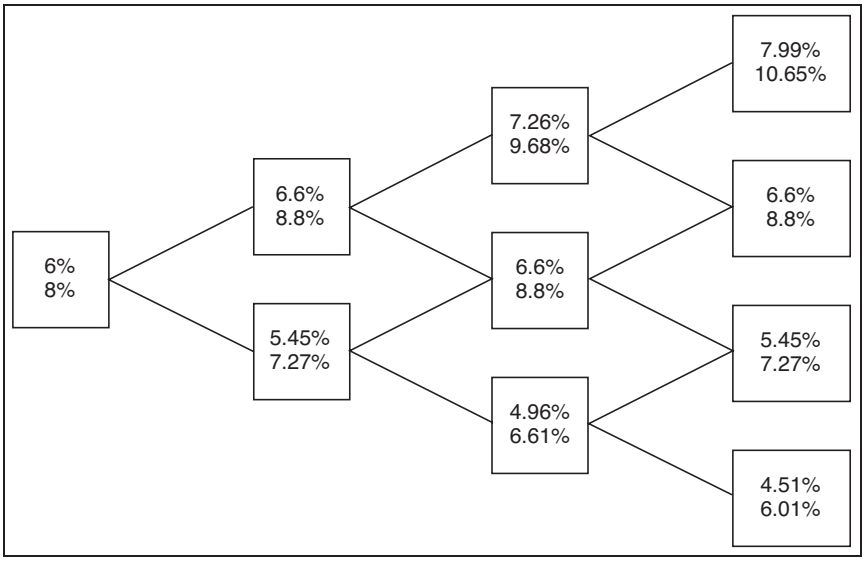


Figure 4.3 Binomial tree for spot and refinancing rates

Table 4.2

	Year 1	Year 2	Year 3	Year 4
Path 1	8.0000%	7.2728%	6.6117%	6.0107%
Path 2	8.0000%	7.2728%	6.6117%	7.2728%
Path 3	8.0000%	7.2728%	8.0000%	7.2728%
Path 4	8.0000%	8.8000%	8.0000%	7.2728%
Path 5	8.0000%	7.2728%	8.0000%	8.0000%
Path 6	8.0000%	8.8000%	8.0000%	8.0000%
Path 7	8.0000%	8.8000%	9.6800%	8.8000%
Path 8	8.0000%	8.8000%	9.6800%	10.6480%

beginning of year 4 (e.g. the scheduled principal in year 4). We compute the scheduled monthly mortgage payment:

$$p = \frac{\$1,000,000}{\frac{1-(1/1.08)^{10}}{.08}} = \$149,029$$

If we initially assume no prepayment risk, then we obtain the cash flows shown in Table 4.3. The balloon payment at the end of year 4 is:

$$\begin{aligned} \textit{Balloon} &= \textit{Balance}(\textit{yr4}) - \textit{Sch.prin}(\textit{yr4}) \\ &= \$775,149 - \$86,957 = \$688,946 \end{aligned}$$

The cash flow in year 4 can be computed as

$$\begin{aligned} CF_4 &= \textit{Balloon} + p \\ &= \$688,946 + \$149,029 = \$837,973 \end{aligned}$$

or equivalently, as

$$\begin{aligned} CF_4 &= \textit{Balance}(\textit{yr4}) + \textit{Interest} \\ &= \$775,903 + \$62,513 = \$837,973 \end{aligned}$$

Table 4.3

Year	Balance	P	Interest	Scheduled Principal	CF
1	$1,000,000	$149,029	$80,000	$69,029	$149,029
2	$930,970	$149,029	$74,478	$74,551	$149,029
3	$856,419	$149,029	$68,513	$80,516	$149,029
4	$775,903	$149,029	$62,072	$86,957	$837,975

Step 2

The second step of the Monte Carlo process is to specify a prepayment model. Suppose we specify the prepayment schedule shown in Table 4.4. The CPR is determined by the value of the spread $X = \textit{WAC} - R^{\textit{ref}}$.

Table 4.4

Range					CPR
		X	$\leq$	0	5%
0	$<$	X	$\leq$	0.5%	10%
0.5%	$<$	X	$\leq$	1.00%	20%
1.00%	$<$	X	$\leq$	1.25%	30%
1.25%	$<$	X	$\leq$	2.0%	40%
2.0%	$<$	X	$\leq$	2.5%	50%
2.5%	$<$	X	$\leq$	3.0%	60%
		X	$>$	3.0%	70%

Step 3

The third step of the valuation process is the estimation of cash flows for each path based on the simulated path of refinancing rates, the spread X, and thus the CPR (see Table 4.5).

The calculation of the cash flows for the first path are shown as follows. In year 1, the scheduled mortgage payment is:

$$p = \frac{\$1,000,000}{\frac{1-(1/1.08)^{10}}{.08}} = \$149,029$$

The interest payment is:

$$I = 0.08(\$1,000,000) = \$80,000$$

The scheduled principal is:

$$SP = \$149,029 - \$80,000 = \$69,029$$

Table 4.5 (continued next page)

Path 1 Year	1 Ref	2 Balance	WAC	3 Interest	4 Sch. Prin.	5 CPR	6 Prepaid	7 CF
1	0.072728	1000000	0.08	80000	69029.49	0.2	186194.1	335223.6
2	0.066117	744776.4	0.08	59582.11	59641.48	0.4	274054	393277.6
3	0.060107	411081	0.08	32886.48	38647.68	0.4	148973.3	220507.5
4		223460	0.08	17876.8				241336.8
Path 2 Year	1 Ref	2 Balance	WAC	3 Interest	4 Sch. Prin.	5 CPR	6 Prepaid	7 CF
1	0.072728	1000000	0.08	80000	69029.49	0.2	186194.1	335223.6
2	0.066117	744776.4	0.08	59582.11	59641.48	0.4	274054	393277.6
3	0.072728	411081	0.08	32886.48	38647.68	0.2	74486.66	146020.8
4		297946.6	0.08	23835.73				321782.4

Source: Johnson, S. (2004)

Table 4.5 (continued)

Path 3	1	2		3	4	5	6	7
Year	Ref	Balance	WAC	Interest	Sch. Prin.	CPR	Prepaid	CF
1	0.072728	1000000	0.08	80000	69029.49	0.2	186194.1	335223.6
2	0.08	744776.4	0.08	59582.11	59641.48	0.05	34256.75	153480.3
3	0.072728	650878.2	0.08	52070.25	61192.16	0.2	117937.2	231199.6
4		471748.8	0.08	37739.91				509488.7

Path 4	1	2		3	4	5	6	7
Year	Ref	Balance	WAC	Interest	Sch. Prin.	CPR	Prepaid	CF
1	0.088	1000000	0.08	80000	69029.49	0.05	46548.53	195578
2	0.08	884422	0.08	70753.76	70824.26	0.05	40679.89	182257.9
3	0.072728	772917.8	0.08	61833.43	72665.69	0.2	140050.4	274549.5
4		560201.7	0.08	44816.14				605017.9

Path 5	1	2		3	4	5	6	7
Year	Ref	Balance	WAC	Interest	Sch. Prin.	CPR	Prepaid	CF
1	0.072728	1000000	0.08	80000	69029.49	0.2	186194.1	335223.6
2	0.08	744776.4	0.08	59582.11	59641.48	0.05	34256.75	153480.3
3	0.088	650878.2	0.08	52070.25	61192.16	0.05	29484.3	142746.7
4		560201.7	0.08	44816.14				605017.9

Path 6	1	2		3	4	5	6	7
Year	Ref	Balance	WAC	Interest	Sch. Prin.	CPR	Prepaid	CF
1	0.088	1000000	0.08	80000	69029.49	0.05	46548.53	195578
2	0.08	884422	0.08	70753.76	70824.26	0.05	40679.89	182257.9
3	0.088	772917.8	0.08	61833.43	72665.69	0.05	35012.61	169511.7
4		665239.5	0.08	53219.16				718458.7

Path 7	1	2		3	4	5	6	7
Year	Ref	Balance	WAC	Interest	Sch. Prin.	CPR	Prepaid	CF
1	0.088	1000000	0.08	80000	69029.49	0.05	46548.53	195578
2	0.096	884422	0.08	70753.76	70824.26	0.05	40679.89	182257.9
3	0.088	772917.8	0.08	61833.43	72665.69	0.05	35012.61	169511.7
4		665239.5	0.08	53219.16				718458.7

Path 8	1	2		3	4	5	6	7
Year	Ref	Balance	WAC	Interest	Sch. Prin.	CPR	Prepaid	CF
1	0.088	1000000	0.08	80000	69029.49	0.05	46548.53	195578
2	0.096	884422	0.08	70753.76	70824.26	0.05	40679.89	182257.9
3	0.10648	772917.8	0.08	61833.43	72665.69	0.05	35012.61	169511.7
4		665239.5	0.08	53219.16				718458.7

Source: Johnson, S. (2004)

The prepaid principal is:

$$PP = 0.20(\$1,000,000 - \$69,029) = \$186,194$$

The cash flow in year 1 is:

$$CF_1 = \$80,000 + \$69,029 + \$186,194 = \$335,223$$

For year 2, along path 1, we have a balance of:

$$B_2 = \$1,000,000 - \$69,029 - \$186,194 = \$744,776$$

The scheduled monthly mortgage payment is:

$$p = \frac{\$744,446}{\frac{1-(1/1.08)^9}{.08}} = \$119,223$$

The interest payment is:

$$I = 0.08(\$744,776) = \$59,582$$

The scheduled principal payment is:

$$SP = \$119,223 - \$59,582 = \$59,641$$

The prepaid principal in the second year is:

$$PP = 0.4(\$755,776 - \$59,641) = \$274,054$$

The cash flow is:

$$CF_2 = \$59,582 + \$59,641 + \$274,052 = \$393,277$$

In year 3, on path 1, the balance is:

$$B_3 = \$744,776 - \$59,641 - \$274,054 = \$411,081$$

The scheduled monthly mortgage payment is:

$$p = \frac{\$411,081}{\frac{1-(1/1.08)^8}{.08}} = \$71,543$$

The interest payment is:

$$I = 0.08(\$411,081) = \$32,886$$

The scheduled principal payment is:

$$SP = \$71,543 - \$32,886 = \$38,648$$

The prepaid principal in the third year is:

$$PP = 0.4(\$411,081 - \$38,648) = \$148,973$$

The cash flow is:

$$CF_3 = \$32,886 + \$38,648 + \$148,973 = \$220,507$$

Finally, in year 4, the balance is:

$$B_4 = \$411,081 - \$38,648 - \$148,943 = \$223,460$$

The interest payment is:

$$I = 0.08(\$223,460) = \$17,877$$

The cash flow is:

$$\begin{aligned} CF_4 &= B_4 + I \\ &= \$223,460 + \$17,877 = \$241,337 \end{aligned}$$

The cash flows for all the other paths are computed similarly.

Step 4

The fourth step of the valuation process is the valuation of the cash flows along each of the paths. The PV of each path's cash flows are determined by specifying the appropriate discount rates. Because the mortgages are insured against default risk, the only risk investors are exposed to is prepayment risk. The risk premium for such risk is known as the *option adjusted spread* (OAS). The OAS is a measure of the spread over the government Treasury bonds rates provided by the MBS when all embedded options have been into account.[7] One can view the OAS as the market price for unmodeled risks (risks that the model cannot capture), such as the forecast error associated with prepayments. The OAS is the spread, such that when added to all the spot rates on all interest rate paths, make the average present value of the paths equal to the observed market price (plus accrued interest). Thus, it equates the observed market *price* of a security to its theoretical *value*. Mathematically, it is equivalent to the solution of K in

$$\begin{aligned} P^{Market} &= \frac{1}{N}\left[PV(path\,1) + PV(path\,2) + \ldots + PV(path\,N)\right] \\ &= \frac{1}{N}\left[\sum_{i=1}^{T}\frac{CF_i^{path\,1}}{(1+Z_i^1+K)^i} + \sum_{i=1}^{T}\frac{CF_i^{path\,2}}{(1+Z_i^2+K)^i} + \ldots + \right. \\ &\qquad \left. \sum_{i=1}^{T}\frac{CF_i^{path\,N}}{(1+Z_i^N+K)^i}\right] \end{aligned} \tag{4.7}$$

where Z_i^j is the zero rate at time i—i.e, month $i = 1, \ldots, T$ on path $j = 1, ..., N$. Typically, $T = 360$ and $N = 1,024$.

The cash flow "yield" that is a standard measure in evaluating any MBS is the *static spread*. This is the yield spread in a static scenario—i.e., no volatility interest rates—of a bond over the entire theoretical Treasury spot rate curve (not a single point on the Treasury yield curve). The magnitude of this spread depends on the steepness of the yields curve: the steeper the curve, the greater the difference between the bond and Treasury yields.[8] There are two ways to compute the static spread. The first approach is to use today's yield curve to discount future cash flows and keep the mortgage refinancing rate fixed at today's mortgage rate.[9]

Because the mortgage refinancing rate is fixed, the investor can usually specify a reasonable prepayment rate, which can be used to estimate the bond's future cash flows until the maturity of the bond. The second approach, known as the *zero volatility OAS*, computes the static spread by allowing the mortgage rates to go up the curve as implied by forward interest rates.[10] In this case, a prepayment model is needed to determine the vector of future prepayment rates (a prepayment schedule) implied by the vector of future refinancing rates. After a static spread and OAS is computed, the implied cost of the prepayment option embedded in any MBS can be computed by calculating the different between the OAS (at the assumed volatility of interest rates) and the static spread. That is

$$\textit{Option cost} = \textit{Static spread} - \textit{OAS} \tag{4.8}$$

Consequently, because, in general, a tranche's option cost is more stable than its OAS in the face of uncertainty of interest rate movements, then, for small market moves, the OAS of a tranche may be approximated by recalculating the static spread and subtracting its option cost. This is quite useful because the OAS is computationally expensive to evaluate while the static spread is cheap and easy to compute.[11]

It is important to point out that investors in MBS hold the equivalent of long positions in noncallable bonds and short positions in call (prepayment) options.[12] The noncallable bond is a collection of zero-coupon bonds—i.e., Treasury strips—and the call option gives the borrower the right to prepay the mortgage at any time prior to maturity of the loan.[13] Thus, the value of MBS is the difference between the value of the noncallable bond and the value of the call (prepayment) option. The OAS is the spread differential between the bond component and the option value component of the MBS. The two main inputs into the computation of an OAS are the cash flows generated as a function of the principal (scheduled and unscheduled) and coupon payments, as well as the interest rate paths generated under an assumed term structure of the zero-coupon curve for discounting the cash flows.[14] At each cash flow date, the spot rate (observed from the interest rate path taken at the corresponding time step of the term structure) determines the discount factor for each cash flow.[15]

Denote z_t to be the appropriate zero discount rate for maturity t (i.e., t years or months), seen today (time 0) (and similarly, f_{tj}, the forward discount rate of maturity t seen at time j), and K, the option adjusted spread. In our simple four-step binomial example, the one-year forward (zero) rate at time 0 is $f_{10} = 8.0\%$; the one-year forward rates at time step 1 are $f_{11} = \{8.6\%, 7.45\%\}$; the one-year forward rates at time step 2 are $f_{12} = \{9.26\%, 8\%, 6.96\%\}$; and the one-year forward rates at time step 3 are $f_{13} = \{9.986\%, 8.6\%, 7.45\%, 6.51\%\}$.

The value of each path is obtained by discounting each cash flow by its risk-adjusted zero-spot rate, z. In our example of four time steps, the value of the MBS on path i is

$$V_i = \frac{CF_1}{1+z_1} + \frac{CF_2}{(1+z_2)^2} + \frac{CF_3}{(1+z_3)^3} + \frac{CF_4}{(1+z_4)^4}$$

where, because we can express zero rates in terms of forward rates, we have

$$\begin{aligned}
z_1 &= f_{10}, \\
z_2 &= ((1+f_{10})(1+f_{11}))^{1/2} - 1 \\
z_3 &= ((1+f_{10})(1+f_{11})(1+f_{12}))^{1/3} - 1 \\
z_4 &= ((1+f_{10})(1+f_{11})(1+f_{12})(1+f_{13}))^{1/4} - 1
\end{aligned}$$

In general, on path $i = 1, \ldots, N$,

$$z_T = \{(1+f_{10})(1+f_{11})(1+f_{12})...(1+f_{12T})\}^{1/T} - 1$$

Note that in our example, we have assumed one-year forward rates, but in a more complex and realistic implementation, we would be simulating future one-month rates over a period of 360 months. Thus, for each path, we would be simulating 360 one-month future interest rates, mortgage refinancing rates, and cash flows instead of just four, and we would be simulating many more paths—i.e., 1024, instead of eight.

The zero-rate calculations for path one are

$$\begin{aligned}
z_1 &= 0.08 \\
z_2 &= ((1.08)(1.074546))^{1/2} - 1 = 0.077269 \\
z_3 &= ((1.08)(1.074546)(1.069588))^{1/3} - 1 = 0.074703 \\
z_4 &= ((1.08)(1.074546)(1.069588)(1.06508))^{1/4} - 1 = 0.072289
\end{aligned}$$

so that the MBS value for path one is

$$V_1 = \frac{\$335,224}{1.08} + \frac{\$393,278}{(1.077269)^2} + \frac{\$220,507}{(1.04703)^3} + \frac{\$241,337}{(1.072289)^4} = \$1,009,470.$$

Table 4.6 shows the MBS computed for each of the eight paths.

The final step is to compute the theoretical value of the MBS by averaging over all values taken on each path. In this example, the theoretical value of the mortgage portfolio is the average of the MBS values computed on each of the eight paths:

$$\bar{V} = \frac{1}{N}\sum_{i=1}^{N} V_i \tag{4.9}$$

Evaluating (4.8), the theoretical value is \$997,235 or 99.7235% of par. Note, in addition to the theoretical value, we also can determine the variance of the distribution:

$$Var(V) = \frac{1}{N}\sum_{i=1}^{N}[V_i - \bar{V}]^2 \tag{4.10}$$

Equivalently, we can also compute the theoretical value by taking the weighted average of each MBS value computed on each path, where the weight is the probability of obtaining that value on the path (each up and down move is assumed to be 0.5), shown in Table 4.7.

Table 4.6 (continued next page)

Path 1 Year	7 CF	8 Z1,t-1	9 Zt0	10 Value	11 Prob.
1	335223.6	0.08	0.08	310392.2	0.5
2	393277.6	0.074546	0.07727	338883.5	0.5
3	220507.5	0.069588	0.074703	177647.1	0.5
4	241336.8	0.06508	0.072289	182547.5	
			Value =	1009470	0.125

Path 2 Year	7 CF	8 Z1,t-1	9 Zt0	10 Value	11 Prob.
1	335223.6	0.08	0.08	310392.2	0.5
2	393277.6	0.074546	0.07727	338883.5	0.5
3	146020.8	0.069588	0.074703	117638.5	0.5
4	321782.4	0.074546	0.074664	241252.6	
			Value =	1008167	0.125

Source: Johnson, S. (2004)

Table 4.6 (continued)

Path 3 Year	7 CF	8 Z1,t-1	9 Zt0	10 Value	11 Prob.
1	335223.6	0.08	0.08	310392.2	0.5
2	153480.3	0.074546	0.07727	132252.5	0.5
3	231199.6	0.08	0.078179	184465.3	0.5
4	509488.7	0.074546	0.07727	378300.6	
			Value =	1005411	0.125

Path 4 Year	7 CF	8 Z1,t-1	9 Zt0	10 Value	11 Prob.
1	195578	0.08	0.08	181090.8	0.5
2	182257.9	0.086	0.082996	155393.5	0.5
3	274549.5	0.08	0.081996	216742.2	0.5
4	605017.9	0.074546	0.080129	444493.9	
			Value =	997720.3	0.125

Path 5 Year	7 CF	8 Z1,t-1	9 Zt0	10 Value	11 Prob.
1	335223.6	0.08	0.08	310392.2	0.5
2	153480.3	0.074546	0.07727	132252.5	0.5
3	142746.7	0.08	0.078179	113892.1	0.5
4	605017.9	0.086	0.080129	444493.9	
			Value =	1001031	0.125

Path 6 Year	7 CF	8 Z1,t-1	9 Zt0	10 Value	11 Prob.
1	195578	0.08	0.08	181090.8	0.5
2	182257.9	0.086	0.082996	155393.5	0.5
3	169511.7	0.08	0.081996	133820.4	0.5
4	718458.7	0.086	0.082996	522269.5	
			Value =	992574 1	0.125

Path 7 Year	7 CF	8 Z1,t-1	9 Zt0	10 Value	11 Prob.
1	195578	0.08	0.08	181090.8	0.5
2	182257.9	0.086	0.082996	155393.5	0.5
3	169511.7	0.0926	0.086188	132277.2	0.5
4	718458.7	0.086	0.086141	516246.6	
			Value =	985008	0.125

Path 8 Year	7 CF	8 Z1,t-1	9 Zt0	10 Value	11 Prob.
1	195578	0.08	0.08	181090.8	0.5
2	182257.9	0.086	0.082996	155393.5	0.5
3	169511.7	0.0926	0.086188	132277.2	0.5
4	718458.7	0.09986	0.08959	509741.1	
			Value =	978502.5	0.125

Source: Johnson, S. (2004)

Table 4.7

Value	Prob.
1009470	0.125
1008167	0.125
1005411	0.125
997720.3	0.125
1001031	0.125
992574.1	0.125
985008	0.125
978502.5	0.125
Wt. Value	$997,235

4.5 MBS Pricing Using Monte Carlo in C++

To price MBS in C++, we create and define an MBS class that contains methods for MBS pricing via Monte Carlo simulations of spot rate paths in a binomial tree.

```
#ifndef _MBS_H__
#define _MBS_H__

#include <vector>
#include "math.h"
#include "time.h"
#include "Utility.h"
#include "TNT.h"
#define SIZE_X 100
#define SIZE_Y 100

using namespace std;
```

We define two global double array variables that will be used to store the spot rates and discount rates in the binomial tree.

```
static TNT::Array2D<double> spotRate(SIZE_X,SIZE_Y);
static TNT::Array2D<double> discountRate(SIZE_X,SIZE_Y);
```

The *MBS* class contains an overloaded constructor that accepts the notional principal, coupon, weighted average WAC, weighted average maturity (WAM), and option adjusted spread (OAS). The class contains a method `calcPrice` that first builds a binomial tree and then simulates the interest rate paths on the tree using Monte Carlo. The `calcPrice` method accepts the initial spot rate, mortgage refinance rate, the number of steps in the binomial tree, and the number of simulations. The *MBS* class also contains a method to compute the conditional prepayment rate (CPR) `calcCPR`, which accepts the current refinance rate and a method `computeZeroRates` that computes the current discount factor by ac-

cepting as input the current time step in the binomial tree and the stored history of discount rates on the current path. The MBS class contains a `calcPayment` function that computes the current mortgage payment by receiving the remaining principal and time to maturity as input. Finally, the MBS class contains a `getPrice` that returns the calculated MBS price, a `getStdDev` method that returns the standard deviation of the computed MBS price, and a `getStdError` method that returns the standard error of the computed MBS price.

MBS.h

```
// MBS.h: interface for the MBS class.
//
//////////////////////////////////////////////////////////////////////

#if !defined(AFX_MBS_H__76187F6C_FE6C_425F_97B6_7639548A3878__INCLUDED_)
#define AFX_MBS_H__76187F6C_FE6C_425F_97B6_7639548A3878__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
#include <vector>
#include "math.h"
#include "time.h"
#include "Utility.h"
#include "TNT\TNT.h"
#define SIZE_X 100
#define SIZE_Y 100

using namespace std;
static TNT::Array2D<double> spotRate(SIZE_X,SIZE_Y);
static TNT::Array2D<double> discountRate(SIZE_X,SIZE_Y);

class MBS
{

public:
        MBS();
        MBS(double principal, double coupon, double WAC, double WAM,
          double OAS) :
                faceValue(principal), coupon(coupon), WAC(WAC), WAM(WAM),
                OAS(OAS), T(WAM) { }
        virtual ~MBS()  { }
        double calcPayment(double principal, double T);   // compute
                                                          // payment
                                                          // amount
        void calcPrice(double initRate, double financeRate, int N,
          long int M);
        double calcCPR(double rate);
        void buildTree(double initRate, double financeRate, int N);
        double computeZeroRates(int cnt, vector<double> rate);
        double calcSMM(double x);
        double getPrice();
        double getStdDev();
        double getStdErr();
        double getMaturity();
        double getWAM();
```

```
        double getWAC();
        double getOAS();
  private:
        double OAS;                // option adjusted spread
        double faceValue;          // principal amount
        double coupon;             // coupon rate
        double WAM;                // weighted average maturity
        double WAC;                // weighted average coupon
        vector<double> zeroRates; // store discount zero coupon rates
        double T;                  // maturity of MBS
        double mbsPrice;           // price
        double stdDev;             // standard deviation
        double stdErr;             // standard error
};

#endif _MBS_H__

/*
class MBS
{
public:
        MBS();
        MBS(double principal, double coupon, double WAC, double WAM,
          double OAS) :
        faceValue(principal), coupon(coupon), WAC(WAC), WAM(WAM), OAS(OAS),
          T(WAM) { }
        virtual ~MBS();
class MBS
{
     public:
        MBS() { }
        MBS(double principal, double coupon, double WAC, double WAM,
          double OAS) :
                faceValue(principal), coupon(coupon), WAC(WAC), WAM(WAM),
                  OAS(OAS),
T(WAM) { }
        virtual ~MBS()  { }
        double calcPayment(double principal, double T); // compute
                                                        // payment
                                                        // amount
        void calcPrice(double initRate, double financeRate, int N,
          long int M);
        double calcCPR(double rate);
        double computeZeroRates(int cnt, vector<double> rate);
        double getPrice();
        double getStdDev();
        double getStdErr();
  private:
        double OAS;                    // option adjusted spread
        double faceValue;              // principal amount
        double coupon;                 // coupon rate
double WAM;                            // weighted average maturity
        double WAC;                    // weighted average coupon
        vector<double> zeroRates;      // store discount zero coupon rates
        double T;                      // maturity of MBS
        double mbsPrice;               // price
        double stdDev;                 // standard deviation
```

```
        double stdErr;                // standard error
};

#endif _MBS_H__

        double calcCPR(double rate);
private:
        double OAS;             // option adjusted spread
        double faceValue;       // principal amount
        double coupon;          // coupon rate
        double WAM;             // weighted average maturity
        double WAC;             // weighted average coupon
        vector<double> zeroRates;
        double valuationMBS();

        double computeZeroRates(int N, vector<double> rate);
        double T;
};
*/

//#endif // !defined(AFX_MBS_H__76187F6C_FE6C_425F_97B6_7639548A3878_
  _INCLUDED_)
```

The method definitions are

MBS.cpp

```
// MBS.cpp: implementation of the MBS class.
//
//////////////////////////////////////////////////////////////////////

#include "MBS.h"

//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////

void MBS::buildTree(double initRate, double financeRate, int N)
{

        Utility util;
        double u = 1.1;
        double d = 1/u;
        double p = (exp(initRate*T) - d)/(u - d);
        double deviate = 0.0;
        long seed = 0;
        double refRate = financeRate;
        long* idum = 0;
        double pay = faceValue;
        double faceAmount = 0.0;
        double interest = 0.0;
        double schedulePrincipal = 0.0;
        double prepaidPrincipal = 0.0;
        double CPR = 0.0;
        double balance = faceValue;
        double sum = 0.0;
```

```
double totalsum = 0.0;
double SMM = 0.0;
TNT::Array1D<double> CF(SIZE_X);   // cash_flow
vector<double> disc(0.0);

srand(unsigned(time(0)));
seed = (long) rand() % 100;
idum = &seed;
// build binomial tree for rates
for (int i = 0; i <= N; i++)
{
        for (int j = 0; j <= i; j++)
        {
                spotRate[i][j] = initRate*pow(u,j)*pow(d,i-j);
                discountRate[i][j] = spotRate[i][j] + OAS;
        }
}


faceAmount = faceValue;
int k = 0;
long int M = 10000;
int cnt = 0;
double r = 0.0;
int j = 0;

for (k = 0; k < M; k++)
{
        sum = 0.0;
        balance = faceValue;
        refRate = financeRate;
        j = 0;
        disc.clear();
        disc.empty();
        disc.push_back(discountRate[0][0]);

        for (i = 0; i < N; i++)
        {
                        balance = balance - (schedulePrincipal +
                                    prepaidPrincipal);
                        deviate = util.gasdev(idum);

                        if (deviate > 0)
                        {
                                j++;
                                refRate = refRate*u;
                        }
                        else
                        {
                                j--;
                                if (j < 0)
                                        j = 0;
                                refRate = refRate*d;
                        }
                        disc.push_back(discountRate[i+1][j]);
                        interest = coupon*balance;
                        pay = calcPayment(balance,WAM-i);
```

```
                                schedulePrincipal = pay - interest;

                                if (balance >= schedulePrincipal)
                                {
                                        CPR = calcCPR(refRate);
                                        SMM = calcSMM(CPR);
                                        prepaidPrincipal =
                                          SMM*(balance -
                                              schedulePrincipal);

                                        if (i != N-1)
                                                CF[i] = interest +
                                                        schedulePrincipal +
                                                        prepaidPrincipal;
                                        else
                                                CF[i] = interest + balance;

                                        r = computeZeroRates(i,disc);
                                        sum = sum + CF[i]/(pow(1+r,i+1));

                                }
                                else
                                        goto x;

                }
                x:
                totalsum = totalsum + sum;
        }
        double ave = (totalsum/M);
        std::cout << "MBS price = " << ave << endl;
}

double MBS::calcCPR(double rate)
{
        double CPR = 0.0;
        double value = WAC - rate;

        /*
        if (value <= 0)
                CPR = 0.05;
        else if ((value <= 0.005) && (value > 0))
                CPR = 0.10;
        else if ((value <= 0.01) && (value > 0.005))
                CPR = 0.20;
        else if ((value <= 0.0125) && (value > 0.01))
                CPR = 0.30;
        else if ((value <= 0.02) && (value > 0.0125))
                CPR = 0.40;
        else if ((value <= 0.025) && (value > 0.02))
                CPR = 0.50;
        else if ((value <= 0.03) && (value > 0.025))
                CPR = 0.60;
        else
                CPR = 0.70;
        */

        CPR = 100*(1-pow((1-(value/100)),12));
```

```
        return CPR;

}

double MBS::calcPayment(double fv, double T) {
        return (fv*coupon)/(1-pow(1/(1+coupon),T));
}

void MBS::calcPrice(double initRate, double financeRate, int N,
  long int M){

        Utility util;       // utility class for generating
                            // random deviates
        double u = 1.1;     // up move in binomial tree
        double d = 1/u;     // down move in binomial tree
        double p = (exp(initRate*T) - d)/(u - d);  // up probablity
        double deviate = 0.0;                      // random deviate
        long seed = 0;                             // seed
        double refRate = financeRate;    // refinance rate
        long* idum = NULL;               // pointer to seed value for RNG
        double pay = faceValue;          // face value of MBS
        double faceAmount = 0.0;         // face amount
        double interest = 0.0;           // interest payment
        double schedulePrincipal = 0.0;  // scheduled principal payments
        double prepaidPrincipal = 0.0;   // prepaid principal payments
        double CPR = 0.0;                // conditional prepayments
        double SMM = 0.0;                // monthly mortality
        double balance = faceValue;      // balance remaining
        double sum = 0.0;                // sum of discounted cash flows
                                         // along a path
        double totalsum = 0.0;  // total sum of all discounted cash flows
        double totalsum2 = 0.0;
        TNT::Array1D<double> CF(SIZE_X); // cash_flow
        vector<double> disc(0.0);        // stores discount rates


        // build binomial tree for rates
        for (int i = 0; i <= N; i++)
        {
                for (int j = 0; j <= i; j++)
                {
                        spotRate[i][j] = initRate*pow(u,j)*pow(d,i-j);
                        discountRate[i][j] = spotRate[i][j] + OAS;
                }
        }

        srand(unsigned(time(0)));
        seed = (long) rand() % 100;
        idum = &seed;
        faceAmount = faceValue;
        int k = 0;
        int cnt = 0;
        double r = 0.0;
        int j = 0;

        for (k = 0; k < M; k++)
```

```
        {
                sum = 0.0;
                balance = faceValue;
                refRate = financeRate;
                j = 0;
                disc.clear();
                disc.push_back(discountRate[0][0]);

                for (i = 0; i < N; i++)
                {
                        balance = balance - (schedulePrincipal +
                                    prepaidPrincipal);
                        deviate = util.gasdev(idum);

                        if (deviate > 0)
                        {
                               j++;
                               refRate = refRate*u;
                        }
                        else
                        {
                               j--;
                               if (j < 0)
                                j = 0;
                               refRate = refRate*d;
                        }
                        disc.push_back(discountRate[i+1][j]);

                        interest = coupon*balance;
                        pay = calcPayment(balance,WAM-i);
                        schedulePrincipal = pay - interest;

                        if (balance >= schedulePrincipal)
                        {
                           CPR = calcCPR(refRate);
                           SMM = calcSMM(CPR);
                           prepaidPrincipal = SMM*(balance -
                                                schedulePrincipal);

                           if (i != N-1)
                              CF[i] = interest + schedulePrincipal +
                                          prepaidPrincipal;
                           else
                              CF[i] = interest + balance;

                            r = computeZeroRates(i,disc);
                            sum = sum + CF[i]/(pow(1+r,i+1));

                        }
                        else   // break out of loop
                           goto x;

                }
                x:
                totalsum = totalsum + sum;
                totalsum2 = totalsum2 + sum*sum;
        }
```

```
        double ave = (totalsum/M);

        mbsPrice = ave;
        stdDev = sqrt(totalsum2 - (double)(totalsum*totalsum)/M)*(exp(-
        2*initRate*T)/(M-1));
        stdErr = (double) stdDev/sqrt(M);
}


double MBS::calcSMM(double CPR) {

         return (1 - pow((1 - CPR),(double)1/12));
}

double MBS::computeZeroRates(int cnt, vector<double> rate)
{

        double value = WAC+1;
        for (int j = 1; j <= cnt; j++)
            value = value*(1 + rate[j]);

        if (cnt == 0)
            value = WAC;
        else
           value = pow(value,(double)1/(cnt+1)) - 1;

        return value;
}

double MBS::getPrice() {
     return mbsPrice;
}

double MBS::getStdDev() {
     return stdDev;
}

double MBS::getStdErr() {
     return stdErr;
}

double MBS::getMaturity() {
        return T;
}

double MBS::getWAM() {
        return WAM;
}

double MBS::getWAC() {
        return WAC;
}

double MBS::getOAS() {
        return OAS;
}
```

Consider pricing an MBS with the parameters used previously:

Main-ch04.cpp

```
#include <fstream.h>
#include <stdlib.h>
#include <iostream.h>
#include <string.h>
#include <math.h>
#include <map>

#define SIZE_X 100
#include "CMO.h"

void main()
{

        std::cout.precision(7);
        double principal = 1000000;      //  underlying principal
                                         //  (notional) of MBS
        double coupon = 0.08;            //  coupon rate
        double WAC = 0.08;               //  weighted average
                                         //  coupon rate
        double WAM = 10;                 //  weighted average maturity
        double OAS = 0.02;               //  option adjusted spread
        double initSpotRate = 0.06;      //  spot rate
        double initRefinanceRate = 0.08; //  refinance rate
        int N = 10;                      //  number of time steps in tree
        long int M = 100000;             //  number of simulation paths
        MBS mbs(principal,coupon,WAC,WAM,OAS);

        std::cout << "Running Monte Carlo to price MBS..." << endl << endl;
        mbs.calcPrice(initSpotRate,initRefinanceRate,N,M);
        std::cout << "MBS Price = " << mbs.getPrice() << endl;
        std::cout << "Std Deviation = " << mbs.getStdDev() << endl;
        std::cout << "Std Error = " << mbs.getStdErr() << endl << endl;

        std::cout << "Pricing MBS with Simulations of Binomial
          Tree Paths..." << endl;
        MBS mbs1(principal,coupon,WAC,N,OAS);
        mbs1.buildTree(initSpotRate,coupon,N);


        vector<Tranche> tranche;
        Tranche trA('A',500000,0.06);
        tranche.push_back(trA);
        Tranche trB('B',300000,0.065);
        tranche.push_back(trB);
        Tranche trC('C',200000,0.07);
        tranche.push_back(trC);
        Tranche trZ('Z',100000, 0.075);
        tranche.push_back(trZ);

        std::cout << endl;
        std::cout << "Pricing CMO Tranches..." << endl << endl;
```

```
        CMO cmo(mbs,tranche);
        cmo.calcCashFlows(initSpotRate,initRefinanceRate,N,M);

}
```

The results are as follows:

```
MBS Price = 964386.69
Std Deviation = 110.07
Std Error = 1.10
```

We can improve the accuracy by increasing the number of simulations. For instance, if M $= 100,000$, then:

```
MBS Price = 964469.78
Std Deviation = 34.86
Std Error = 0.11
```

Thus, the price of the MBS is priced at roughly 96.5% of par. The more time steps, however, improves the accuracy of the computed price.

Continuous Time Model

The binomial model is a simple discrete model and does not capture the movement of interest rates in practice because at each step, rates can only go up or down—they cannot stay the same or move in between time steps. To capture a realistic evolution of interest rate movements, an arbitrage-free model of the term structure of interest rates is typically used. The short rate is assumed to follow a diffusion (a continuous time stochastic) process. The general form of these models is described in terms of changes in the short rate, as follows:

$$dr_t = \kappa(\theta - r)dt + \sigma r^{\alpha} dz_t, \quad r(0) = r_0$$

where dr_t represents an infinitesimal change in r_t over an infinitesimal time period, dt, and dz_t is a standard Wiener process. κ is the speed of mean-reversion, θ is the long-run mean of the interest rate process, α is the proportion conditional volatility exponent, and σ is the instantaneous standard deviation of changes in r_t. The various short-rate models differ by the parameter α. The Vasicek model assumes it is 0, the Cox-Ingersoll-Ross (CIR) model assumes it to be 0.5, and the Courtadon model assumes it to be 1.

In order to simulate the process, we discretize it as follows (assume Courtadon):

$$\Delta r_t = \kappa(\theta - r_t)\Delta t + \sigma r_t \sqrt{\Delta t} z_t, \quad r(0) = r_0$$

Many interest rate models have some form of mean reversion, reverting the generated interest rate paths to some "long-run" level. Without reversion, interest rates could obtain unreasonably high and low levels. Volatility, over time, would theoretically approach infinity. Similarly, a large percentage volatility assumption would result in greater fluctuations in yield, which in turn results in a greater probability of the opportunity to refinance. The increased probability in refinancing is a greater value attributed to the implied call option, and a higher resulting option cost.[16]

In addition to a more realistic term structure, we need to expand our prepayment model to reflect the effects of multiple factors that impact prepayment. We can utilize the Richard and Roll (1989) prepayment model, which is based on empirical estimation of the mortgagor's financing condition. The model tries to explain prepayments by observing actual prepayments and relating them to the measurable factors suggested by their economic theory of prepayments. The prepayment model makes a few assumptions. The maximum CPR is 50% and the minimum CPR is 0%. The midpoint CPR at 25% occurs at a WAC-refinance rate differential at 200 basis points. At midpoint, the maximum slope is 6% CPR for a 10 basis point rate shift.

The Richard and Roll (1989) model identifies four factors that should be included in any prepayment model:[17]

1. Refinancing incentive: borrower's incentive to refinance

$$RI(t) = a + b(\arctan(c + d(WAC - r_t)))$$

where[18]

$$a = (\max CPR + \min CPR)/2$$
$$b = 100(\max CPR - a)/(\pi/2)$$
$$d = \max slope/b$$
$$c = -d \; x \; midpoint \text{ diff.}$$

2. Seasoning (age of the mortgage):

$$Age(t) = \min(\frac{t}{30}, 1)$$

3. Seasonality (monthly multiplier): yearly trends in housing turnover[19]

$$MM(t) = (0.94, 0.76, 0.74, 0.95, 0.98, 0.92, 0.98, 1.10, 1.18, 1.22, 1.23, 0.98)$$

where t is the tth month, $t = 1...12$.

4. Burnout multiplier: A spike in refinancing due to incentives is followed by a burnout

$$BM(t) = 0.3 + 0.7\frac{B(t)}{B(0)}$$

where $B(t)$ is the mortgage balance at time t and $B(0)$ is the initial mortgage pool balance.

The annualized prepayment rate, $CPR(t)$, is equal to

$$CPR(t) = RI(t)xAge(t)xMM(t)xBM(t).$$

The cash flows for the MBS under this expanded prepayment model are as follows:

- $MP(t)$ is the scheduled mortgage payment for period (month) t:

$$MP(t) = B(t)\left(\frac{WAC/12}{1-(1+WAC/12)^{-WAM+t}}\right)$$

- $IP(t)$ is the interest payment for period t:

$$IP(t) = B(t)\left(\frac{WAC}{12}\right)$$

- $PP(t)$ is the principal prepayment for period t

$$PP(t) = SMM(t)(B(t) - SP(t))$$

where

$$SMM(t) = 1 - \sqrt[12]{1 - CPR(t)} \text{ and } SP(t) = MP(t) - IP(t).$$

- $SP(t)$ is the scheduled principal payment for period t, and $SMM(t)$ is the single monthly mortality-rate at time t.

The reduction in the mortgage balance for each month is given by

$$B(t+1) = B(t) - TPP(t)$$

where $TPP(t)$ is the total principal payment for period t. As before, we computed the expected cash flows at time t, $CF(t)$, of the MBS, and thus the MBS price P, using these formulas and Monte Carlo:

$$P = E^Q\left[\sum_{t=0}^{M} PV(t)\right] = E^Q\left[\sum_{t=0}^{M} df(t)CF(t)\right]$$

where

$$\begin{aligned}
PV &= \text{Present value for cash flow at time } t.\\
df(t) &= \prod_{k=1}^{t}\frac{1}{(1+r_k)} = \text{Discounting factor for time } t.\\
CF(t) &= MP(t) + PP(t) = TTP(t) + IP(t)\\
MP(t) &= SP(t) + IP(t)\\
TPP(t) &= SP(t) + PP(t).
\end{aligned}$$

Monte Carlo simulation can be used to help people like portfolio managers identify whether current MBS market prices are rich or cheap compared to their theoretical values and variances, and make potentially profitable trades to capture "mispricings" in the market

compared to their "true" theoretical values. One can use the information from the simulation to estimate the average life of each path and the mean and variance from all the paths. From this, one can estimate prepayment risk.

There are two types of cash flow analysis approaches. The first, static cash flow analysis, assumes a constant PSA, while the second, vector (or dynamic) cash flow analysis assumes that the PSA changes over time. The static cash flow methodology estimates CFs based on different PSA speeds, and then calculates the yields on the CFs for prices and for different PSA speeds, assuming a constant interest rate volatility assumption. Static CF analysis is useful in determining what is a good price given the estimated yields based on PSA speeds, duration, average life, and other features of the mortgage or MBS. Table 4.8 shows static analysis for different par value, price, and PSA speeds assumptions.

Based on CF analysis, an investor would be willing to pay 90.75% of par or less for the MBS, if they required a yield of 9.76% for an MBS investment with a PSA of 165 (or equivalently for an investment with an average life of 2.93, and duration of 2.57).

Table 4.8

Par Value	PSA Yield			Mean	Std Dev.
(Price as % of Par)	50%	100%	165%		
$44.127M (90.75)	8.37%	9.01%	9.76%	9.047%	.5681
$45.100M (92.75)	7.82%	8.31%	8.88%	8.3367%	.4330
$46.072M (94.75)	7.29%	7.63%	8.03%	7.6500%	.5234
$47.045M (96.75)	6.78%	6.97%	7.20%	6.9800%	.1717
$48.017M (98.75)	6.28%	6.34%	6.40%	6.3400%	.0490
Average Life (Years)	5.10	3.80	2.93		
Maturity	9.40	7.15	5.40		
Duration	4.12	3.22	2.57		
Vector Analysis:					
Month:	PSA				
1–36	50	100	165		
37–138	200	200	400		
139–357	300	300	400		
At $48.127M:					
Yield	6.02%	6.01%	6.00%	6.0100%	.00816
Average Life	3.51	2.71	2.63		
Duration	2.97	2.40	2.34		

Source: Johnson, S. (2004)

Vector analysis is a more dynamic approach. Vector analysis can be used like static CF analysis to determine prices given required yields. The example at the bottom of Table 4.8 shows vector analysis in which different PSA speeds are assumed for three subperiods.

In general, a decrease in PSA will benefit longer-maturity tranches more than shorter maturity tranches. Slowing down prepayment increases the OAS for all tranches, more for those tranches trading above par, as well as increases their price. However, changes in price are not as great for shorter duration tranches, as their prices do not move as much from a change in OAS as a longer duration tranche. Conversely, an increase in PSA will reduce the OAS and price of all tranches, especially if they are trading above par. Interest-only (IO) tranches and IO types of tranches will be adversely affected by an increase as well. A reduction in interest rate volatility increases the OAS and price of all tranches, though most of the increase is realized by the longer maturity tranches. The OAS gain for each of the tranches follows more or less the OAS durations of those tranches.[20] An increase in interest rate volatility will distribute the collateral's loss such that the longer the tranche duration, the greater the loss.

As part of the valuation model, option-adjusted duration and option-adjusted convexity are important measures. In general, duration measures the price sensitivity of a bond to a small change in interest rates. Duration can be interpreted as the approximate percentage change in price for 100-basis point parallel shift in the yield curve.[21]

For example, if a bond's duration is 3.4, this suggests that a 100-basis point increase in rates will result in a price decrease of approximately 3.4%. A 50-basis point increase in yields will decrease the price by roughly 1.7%. The smaller the basis point change, the better the approximated change will be.

The effective duration of an MBS (or any fixed-income security) can be approximated as follows:

$$\textit{Effective Duration} = \frac{V_- - V_+}{2V_0\Delta r} \tag{4.11}$$

where

V_- = Price if yield is decreased (per \$100 of par value) by Δr.

V_+ = Price if yield is increased (per \$100 of par value) by Δr.

V_0 = Initial price (per \$100 of par value).

Δr = Number of basis points change in rates used in calculate V_- and V_+.

Effective duration—in contrast to modified duration, which is the standard measure of duration—assumes that prices in the formula (4.11) are computed assuming cash flow changes when interest rates change. Modified duration, on the other hand, assumes that if interest rates change, the cash flow does not change so that modified duration is an appropriate measure for option-free securities like Treasury bonds, but not for securities with embedded options like MBS, where cash flows are affected by rate changes. Consequently, MBS use effective duration, also known as OAS duration, which can be computed using an OAS model as follows. First, the bond's OAS is found using the current term structure of interest rates. Next, the bond is reprised holding OAS constant, but shifting the term structure twice—one shift increases yields and one shift decreases yields generating two prices, V_- and V_+, respectively.[22]

Subsequently, effective duration can be used with a binomial tree or with CF analysis to measure the duration of a bond with option risk or an MBS. The following steps are utilized for using a binomial tree to value a bond with an embedded option. First, take a yield curve estimated with bootstrapping and value the bond, V_0, using the calibration approach. Then, let the estimated yield curve with bootstrapping decrease by a small amount and then estimate the price of the bond using the calibration approach: V_-. Let the estimated yield curve with bootstrapping decrease by a small amount, and then estimate the price of the bond using the calibration approach: V_+. Finally, calculate the effective duration in (4.11). Similarly, using static cash flow analysis, you can calculate effective duration as follows. For a given PSA, determine the prices associated with small yield changes (you can also use a model in which you assume PSA changes as rates change), and then use the formula (4.11). It is important to note that effective duration assumes only parallel shifts in the term structure and will not correctly predict the bond price change if shifts are not parallel.

Convexity is a measure of a security that is the approximate change in price that is not explained by duration. It can be viewed as the second-order term of the Taylor expansion of the bond price as a function of yield. Bonds with positive convexity will have a greater percentage increase in price than the percentage price decrease if the yield changes by a given number of basis points. Conversely, bonds with negative convexity will have a greater percentage price decrease than percentage price increase if yields change a given number of basis points. Although positive convexity is a desirable feature of a bond, a pass-through security can exhibit either positive or negative convexity, depending on the current mortgage refinancing rate relative to the rate on the underlying mortgage loans. Convexity can be computed as:

$$\frac{V_+ + V_- - 2(V_0)}{2V_0(\Delta r)^2} \tag{4.12}$$

If cash flows do not change when yields change, then the resulting convexity from (4.12) is a good approximation to the standard convexity of an option-free bond. However, if prices in (4.12) are derived by changing the cash flows change (by changing prepayment rates) when yields change, the resulting convexity is called *effective convexity*.[23] If prices are obtained by simulating the OAS via Monte Carlo simulation or by an OAS model, the resulting value is known as OAS convexity.

As an example of computing duration and convexity, consider a PSA 165 MBS with the following prices and yields shown in Table 4.9.

Table 4.9

Price	Yield
102.1875	6.75%
100.2813	7.00%
98.4063	7.25%

From (4.12), we find the duration is

$$\frac{102.1875 - 98.4063}{2(100.2813)(.0025)} = 7.54$$

and the convexity is

$$\frac{102.1875 + 98.4063 - 2(100.2813)}{2(100.2813)(.0025)^2} = 24$$

Thus, for a 25% change in the yield, the bond price will change by 7.54%, with a positive convexity of 24%—meaning 24% of the price change is not captured by the duration.

Figure 4.4 shows the simulated cash flows for a 30-year MBS with a 8.5% coupon on a $1,000,000 pool.

For MBS-pricing models where the underlying factor follows a diffusion process, see Kariya and Kobayashi (2000) for a one-factor (interest rate) valuation model and Kariya, Ushiyama, and Pliska (2002) for a three-factor (interest rate, mortgage rate, and housing price) valuation model.[24]

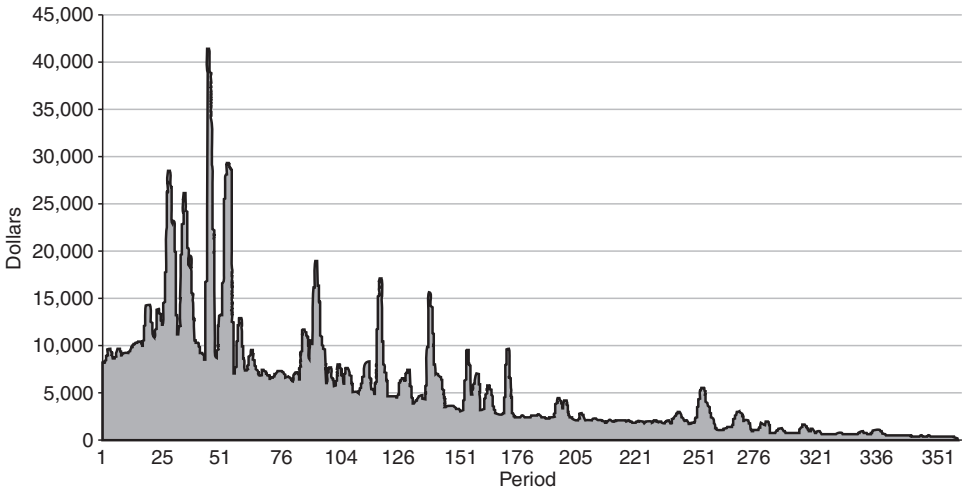


Figure 4.4 Simulated cash flows for a 30-year MBS. *Source: Bandic, I. (2002), pg. 23.*

4.6 Matlab Fixed-Income Toolkit for MBS Valuation

The Matlab Fixed-Income Toolkit can be used for MBS valuation and for computing many MBS measures, such as effective duration, convexity, and OAS. The following variables are inputs in MBS valuation in Matlab:

- **Price:** Clean price for every $100 of face value.
- **Yield:** Mortgage yield, compounded monthly (in decimal).
- **Settle:** Settlement date. A serial date or date string. Settle must be earlier than or equal to Maturity.
- **OriginalBalance:** Original balance value in dollars (balance at the beginning of each TermRemaining).
- **TermRemaining:** (Optional) Number of full months between settlement and maturity.
- **Maturity:** Maturity date. A serial date number of date string.
- **IssueDate:** Issue date. A serial date number or date string.
- **GrossRate:** Gross coupon rate (including fees), in decimal. Equal to WAC.
- **CouponRate:** Net coupon rate, in decimal. Default = GrossRate. Equal to PT rate.
- **Delay:** (Optional) Delay (in days) between payment from homeowner and receipt by bondholder. Default = 0 (no delay between payment and receipt).
- **NMBS:** Number of mortgage-backed securities.
- **PrepaySpeed:** (Optional) Relation of the conditional prepayment rate (CPR) to the benchmark model. Default = 0. Set PrepaySpeed to [] if you input a customized prepayment matrix.
- **PrepayMatrix:** (Optional) Used only when PrepayModel and PrepaySpeed are unspecified. Customized prepayment vector: A NaN-padded matrix of size max(TermRemaining)-by-NMBS. Each column corresponds to each MBS, each row corresponds to each month after settlement.
- **ZeroMatrix:** A matrix of three columns. Column 1: serial date numbers. Column 2: spot rates with maturities corresponding to the dates in Column 1. Column 3: Compounding of rates in Column 1. Values are 1 (annual), 2 (semiannual), 3 (three times per year), 4 (quarterly), 6 (bimonthly), 12 (monthly), and –1 (continuous).
- **Interpolation:** Interpolation method. Computes the corresponding spot rates for the bond's cash flow. Available methods are (0) nearest, (1) linear, and (2) cubic spline. Default = 1.

All inputs (except PrepayMatrix and ZeroMatrix) are NMBS x 1 vectors. The following variables are inputs for pricing bonds, which can in turn be used to find the implied yield curve for pricing mortgage-backed securities:

- **Face:** (Optional) Face value of each bond in the portfolio. Default = 100.
- **Yield:** Scalar or vector containing yield to maturity of instruments.

- **Settle:** Settlement date. A scalar or vector of serial date numbers. Settle must be earlier than or equal to Maturity.
- **Maturity:** Maturity date. A scalar or vector of serial date numbers of date strings.
- **ConvDates:** Conversion dates for the bonds. A matrix of serial date numbers.
- **CouponRates**: Matrix containing coupon rates for each bond in the portfolio in decimal form. The first column of this matrix contains rates applicable between Settle and dates in the first column of ConvDates.
- **Period:** (Optional) Number of coupons per year of the bond. A vector of integers. Allowed values are 0, 1, 2, 3, 4, 6, and 12. Default = 2 (semiannual).
- **Basis:** (Optional) Day count basis of the instrument. A vector of integers. 0 = actual/actual (default), 1 = 30/360, 2 = actual/360, 3 = actual/365, 4 = 30/360 (PSA compliant), 5 = 30/360 (ISDA compliant), 6 = 30/360 (European), and 7 = actual/365 (Japanese).
- **EndMonthRule:** (Optional) End-of-month rule. A vector. This rule applies only when Maturity is an end-of-month date for a month having 30 or fewer days. 0 = ignore rule, meaning that a bond's coupon payment date is always the same numerical day of the month. 1 = set rule on (default), meaning that a bond's coupon payment date is always the last actual day of the month.

Suppose we want to compute the cash flows and balances of an FHLMC mortgage pool with an initial balance of $10,000,000, PSA of 150, WAC = 8.125%, term of 360 months, and a remaining term of 357 months (the pool has been "seasoned" for three months). We use the following Matlab code:

```
OriginalBalance = 1000000;
GrossRate = 0.08125;
OriginalTerm = 360;
TermRemaining = 357;
PrepaySpeed = 125;
[Balance, Payment, Principal, Interest, Prepayment] =
  mbspassthrough(OriginalBalance,...
 GrossRate, OriginalTerm, TermRemaining, PrepaySpeed)
```

This code produces the output shown in Table 4.10 (Balance, Payment, Principal, Interest, and Prepayment).

These values are computed in the same way as in §4.2. Given a portfolio of mortgage-backed securities, we could compute the clean prices and accrued interest using the Matlab `mbsprice` function. Suppose the yield on the portfolio is 7.25%, the WAC (gross coupon) is 8.5%, the maturity is January 10, 2034, the issue date is January 10, 2004, and we want the price on five settlement dates: March 10, 2004; May 17, 2004; May 17, 2005; January 10, 2006; and June 10, 2006 with PT (coupon) rates of 7.5%, 7.875%, 7.75%, 7.95%, and 8.125%, on each of the MBS securities, respectively. Assume the delay in the start of payments is 20 days:

Table 4.10

Month	Balance	Payment	Principal	Interest	Prepayment
1	998,490.00	7439.7	668.8373	6770.8	836.6
2	996,780.00	7433.4	672.8021	6760.6	1045.4
3	994,850.00	7425.7	676.6479	6749	1253.8
4	992,700.00	7416.3	680.3719	6735.9	1461.6
5	990,350.00	7405.4	683.9716	6721.4	1668.7
6	987,790.00	7392.9	687.4443	6705.5	1875
7	985,020.00	7378.9	690.7876	6688.2	2080.4
8	982,040.00	7363.4	693.9991	6669.4	2284.7
9	978,850.00	7346.3	697.0763	6649.2	2487.7
10	975,460.00	7327.7	700.017	6627.7	2689.5
350	5,640.00	832.5	788.7487	43.8	36.7
351	4,820.00	827.1	788.9469	38.2	31.4
352	4,000.00	821.8	789.1451	32.6	26.1
353	3,190.00	816.4	789.3434	27.1	20.8
355	1,590.00	805.9	789.7401	16.2	10.3
356	790.00	800.7	789.9385	10.7	5.2
357	0.00	795.5	790.137	5.3	0

Source: Johnson, S. (2004)

```
% MBS.m  : compute MBS prices and accrued interest
Yield = 0.0725;
Settle = datenum(['10-Mar-2004';'17-May-2004';'17-May-2005';'10-Jan-
2006';'10-Jun-2006']);
Maturity = datenum('10-Jan-2034');
IssueDate = datenum('10-Jan-2004');
GrossCoupon = 0.085;
CouponRate = [0.075; 0.07875; 0.0775; 0.0795; 0.08125];
Delay = 20;
Speed = 150;
[Price, Accrt] = mbsprice(Yield, Settle, Maturity, IssueDate, ...
        GrossRate, CouponRate, Delay, Speed)
```

Table 4.11 shows the prices and accrued interest at each of the settlement dates.

Table 4.11

Settlement Date	Price	Accrued Interest
March 10, 2004	101.0937	0.0000
May 17, 2004	103.2801	0.1531
May 17, 2005	102.3677	0.1507
Jan 10, 2006	103.3897	0.0000
June 10, 2006	104.3008	0.0000

Suppose that we want to compute the OAS of the mortgage pool at the March 10, 2004 settlement date with a roughly a 28-year WAM remaining, given assumptions of a 100, 150, and 200 PSA speeds using the computed price and coupon on March 10, 2004 for the preceding mortgage pool. In Matlab, we first need to create a zero matrix constructed (implied) by bond prices (assume all bonds pay semiannual coupons) and yields:

```
Bonds = [datenum('11/21/2004')  0.045    100   2   3   1;
         datenum('02/20/2005')  0.0475  100   2   3   1;
         datenum('07/31/2007')  0.0500  100   2   3   1;
         datenum('08/15/2010')  0.0550  100   2   3   1;
         datenum('03/15/2012')  0.0575  100   2   3   1;
         datenum('02/15/2015')  0.0600  100   2   3   1;
         datenum('03/31/2020')  0.0650  100   2   3   1;
         datenum('08/15/2025')  0.0720  100   2   3   1;
         datenum('07/20/2034')  0.0850  100   2   3   1];

Yields = [0.0421; 0.0452;  0.0482; 0.0510;  0.0532;  0.0559;
0.0620; 0.0682; 0.0785];
% Since the above is Treasury data and not "selected" agency data, an
% ad-hoc method of altering the yield has been chosen for demonstration
% purposes
Yields = Yields + 0.025*(1./[1:9]');

% Get parameters from Bonds matrix
Settle = datenum('10-Mar-2004');
Maturity     = Bonds(:,1);
CouponRate   = Bonds(:,2);
Face         = Bonds(:,3);
Period       = Bonds(:,4);
Basis        = Bonds(:,5);
EndMonthRule = Bonds(:,6);

% compute bond prices
[Prices, AccruedInterest] = bndprice(Yields, CouponRate, ...
  Settle, Maturity, Period, Basis, EndMonthRule, [], [], [], [], ...
  Face);

% uses the bootstrap method to return a zero curve given a portfolio of
% coupon bonds and their prices
[ZeroRatesP, CurveDatesP] = zbtprice(Bonds, Prices, Settle);
SpotCompounding = 2*ones(size(ZeroRatesP));
ZeroMatrix = [CurveDatesP, ZeroRatesP, SpotCompounding];
Maturity = datenum('10-Jan-2034');
IssueDate = datenum('10-Jan-2004');
GrossRate = 0.085;
Delay = 20;
Interpolation = 1;
PrepaySpeed = [100 150 200];
Price = 101.0937;
CouponRate = 0.075;
Settle = datenum('10-Mar-2004');

OAS = mbsprice2oas(ZeroMatrix, Price, Settle, Maturity, ...
       IssueDate, GrossRate, CouponRate, Delay, Interpolation, ...
       PrepaySpeed)
```

Table 4.12

PSA	OAS
100	82.6670
150	101.8518
200	114.6088

The OAS results are shown in Table 4.12

We can compute the effective duration and convexity of the mortgage pool using the functions `mbsdurp` (duration given price), `mbsdury` (duration given yield), `mbsconvp` (convexity given price), and `mbsconvy` (convexity given yield). For instance, continuing with the example, we can compute the yearly duration, modified duration, and convexity of the pool on March 10, 2004 for the 100, 150, and 200 PSA speed assumptions by making the function calls following the code:

```
% compute regular duration and modified duration
[YearDuration, ModDuration] = mbsdurp(Price, Settle, Maturity,
  IssueDate, GrossRate, É CouponRate, Delay, PrepaySpeed)

% compute convexity
Convexity = mbsconvp(Price, Settle, Maturity, IssueDate, GrossRate,
  CouponRate, Delay,PrepaySpeed)
```

Table 4.13 shows duration, modified duration, and convexity for 100, 150, and 200 PSAs.

Table 4.13

PSA	Year Duration	Mod. Duration	Convexity
100	7.0669	6.8148	82.3230
150	6.1048	5.8881	62.2476
200	5.3712	5.1814	48.3368

4.7 Collateralized Mortgage Obligations (CMOs)

CMOs are securities backed by a pool of mortgages, MBSs, stripped MBSs, or CMOs. They are structured so that there are several classes of bonds; these classes are called tranches. Each tranche has a different priority claim on the principal. There are two general types of CMOs: sequential-pay tranches and planned and amortization class (PAC). In a sequential-pay tranche, each bond class is prioritized in terms of the order of the principal payment. Principal for each tranche is paid sequentially by priority: The first priority tranche's principal is paid entirely (retired) before the next class, which has its principal paid before the next class, and so on. This process continues until all the tranches in the structure are paid off. In general, the YTM of the first tranche is the lowest because it has

the shortest average life and the least prepayment risk. Each successive tranche has a longer average life and a higher YTM.

The last tranche in many plain vanilla structures does not receive interest until all the tranches with shorter maturities are paid off. These classes are known as accrual bonds or "Z bonds" due to their similarity to zero-coupon bonds. The interest that would be paid to the Z bond is used to pay the principal in the shorter maturity tranches, which shortens their average lives.

Figure 4.5 shows a hypothetical distribution of principal and interest cash flows between the sequential pay bonds and the Z bond.

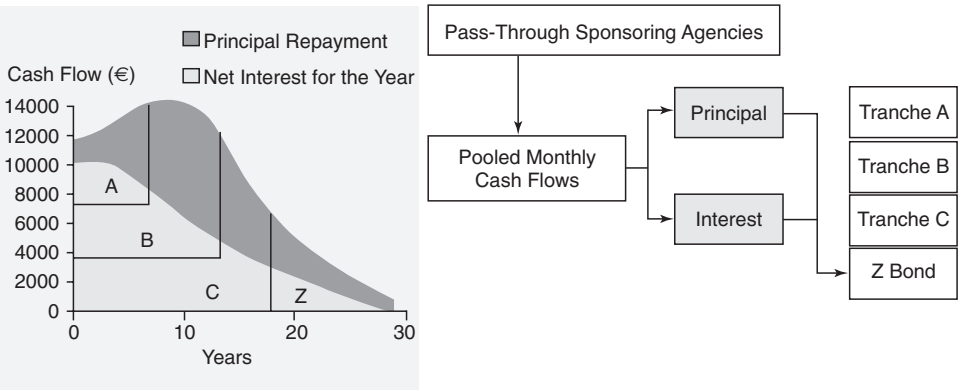


Figure 4.5 Sequential Pay Bonds and Z Bonds

Suppose we form sequential-pay tranches from the mortgage portfolio described in Table 4.14: \$100M mortgage portfolio, WAM = 357 months, WAC = 8.125%, PT rate = 7.5%, and prepayment speed = 165.

The distribution of cash flows is made as follows. Principal payments are first made to A, then to B, then to C, and so on down to the residual tranches. Principal payment includes both scheduled principal payment and prepaid principal. The coupon payment is based on the remaining balance in the tranche. Table 4.15 shows the structured tranche payments.

To attract certain types of investors, floating-rate and inverse floating-rate tranches are created. These two tranches can be created from an existing one such as the C tranche. For example, a floating rate class (FR) and inverse floating rate class (IFR) in C can be constructed such that the floating rate is, for example, LIBOR + 50 basis points, and the inverse floating rate is $28.5 - 3$ (LIBOR). The FR class is 75% of tranche C, and the IFR class is 25%. Thus, the tranches can be constructed as shown in Table 4.16.

Table 4.14

Tranche	Par	PT Rate
A	\$48.625M	7.5%
B	\$9M	7.5%
C	\$42.375M	7.5%

Table 4.15

Month	Balance 100,000,000.00	Interest	Principal	Tranch A Begin. Bal. 48,625,000.00	48,625,000.00 Interest	Principal
1	100,000,000.00	625,000.00	177,480.87	48,625,000.00	303,906.25	177,480.87
2	99,822,519.13	623,890.74	205,473.91	48,447,519.13	302,796.99	205,473.91
3	99,617,045.22	622,606.53	233,389.97	48,242,045.22	301,512.78	233,389.97
4	99,383,655.25	621,147.85	261,205.39	48,008,655.25	300,054.10	261,205.39
5	99,122,449.86	619,515.31	288,896.53	47,747,449.86	298,421.56	288,896.53
6	98,833,553.33	617,709.71	316,439.78	47,458,553.33	296,615.96	316,439.78
80	51,965,586.84	324,784.92	512,605.38	590,586.84	3,691.17	512,605.38
81	51,452,981.45	321,581.13	508,049.13	77,981.45	487.38	77,981.45
82	50,944,932.32	318,405.83	503,532.52	0.00	0.00	0.00
83	50,441,399.80	315,258.75	499,055.22	0.00	0.00	0.00
84	49,942,344.58	312,139.65	494,616.89	0.00	0.00	0.00
85	49,447,727.69	309,048.30	490,217.17	0.00	0.00	0.00
99	42,968,082.27	268,550.51	432,494.82	0.00	0.00	0.00
100	42,535,587.45	265,847.42	428,635.94	0.00	0.00	0.00
101	42,106,951.51	263,168.45	424,810.66	0.00	0.00	0.00
102	41,682,140.85	260,513.38	421,018.70	0.00	0.00	0.00
103	41,261,122.15	257,882.01	417,259.77	0.00	0.00	0.00
104	40,843,862.38	255,274.14	413,533.58	0.00	0.00	0.00
105	40,430,328.80	252,689.56	409,839.84	0.00	0.00	0.00
106	40,020,488.96	250,128.06	406,178.29	0.00	0.00	0.00
356	74,797.79	467.49	37,597.30	0.00	0.00	0.00
357	37,200.49	232.50	37,200.49	0.00	0.00	0.00

Tranche B Beginning Bal. 9000000	9000000 Principal	Interest	Tranche C Month	C: Beg. Bal Cum Int 42375000	Principal 0	Interest
9000000	0	56250	1	42375000	0	0
9000000	0	56250	2	42639843.8	0	0
9000000	0	56250	3	42904687.5	0	0
9000000	0	56250	4	43169531.3	0	0
9000000	0	56250	5	43434375	0	0
9000000	0	56250	54	56411718.8	0	0
9000000	196997.83	56250	55	56676562.5	0	0
8803002.17	899641.259	55018.764	56	56941406.3	0	0
7903360.91	894022.233	49396.006	57	57206250	0	0
904465.071	850793.615	5652.9067	65	59325000	0	0
53671.45651	53671.4565	335.4466	66	59589843.8	527084	372436.52
0	0	0	67	59062759.6	575606	369142.25
0	0	0	68	58487153.2	570502	365544.71
0	0	0	69	57916651	565442	361979.07
0	0	0	356	74797.7939	37597.3	467.48621
0	0	0	357	37200.4934	37200.5	232.50308

Source: Johnson, S. (2004)

Table 4.16

Tranche	Par	PT Rate
A	$48.625M	7.5%
B	$9M	7.5%
FR	$31.782M	LIBOR + 50 bps
IFR	$10.549M	28.5 − 3 (LIBOR)

Note that in the inverse floating rate equations, 28.5 is referred to as the cap rate (K) and 3 is called the leverage rate (L). Because the floating and inverse floating classes were created from class C, which paid a rate of 7.5%, the K and L were found such that:

$$.75[\mathit{LIBOR} + .5\%] + .25[K - L(\mathit{PLIBOR}) = 7.5\%$$

If LIBOR is 6%, then:

$$\begin{aligned} \mathit{FR} &= 6\% + .5\% = 6.5\% \\ \mathit{IFR} &= 28.5 - 3(6\%) = 10.5 \\ .75(6.5\%) &+ .25(10.5\%) = 7.5\% \end{aligned}$$

CMOs often have tranches with different rates. Such CMOs often include a special type of tranche known as a notional interest-only class, which receives only the residual interest. Notional IO classes are often described as paying a certain base interest on a notional principal. Consider the following CMO with an NIO shown in Table 4.17.

The notional principal of the NIO class is $13.75M. The interest-only (IO) class receives the excess interest of 7.5% over the rate paid on each class. For example, from class A, the IO class would receive 1.5% (7.5% − 6%) on $48.625M, which is $0.729375M. Capitalizing $.729375M at 7.5% yields a notional principal of $9.725M for the IO class on class A. The sum of the notional principals for each class yields the IO's notional principal of $13.75M, as shown in Table 4.18.

Suppose we have a $30,000,000 FHLMC mortgage pool with three tranches, A, B, and C, each with a size of $10,000,000. Assume the first tranche pool "balloons out" in 60 months, the second pool "balloons out" in 90 months, and the third is regularly amortized to maturity. The prepayment speeds are assumed to be 100, 165, and 200 for each tranche, respectively. Suppose that the delay before the first pass-through payment made after issue is 30 days, the WAC (*GrossRate* in Matlab) is 8.125%, the PT rate (*CouponRate* in Matlab)

Table 4.17

Tranche	Par	PT Rate
A	$48.625M	6.0%
B	$9M	6.5%
Z	$42.375M	7.0%
NIO	$13.750M	7.5%

Table 4.18

Tranche	Par	PT Rate	(.075−PT Rate) Par	Notional Principal
A	$48.625M	6.0%	$729,375	$9.725M
B	$9M	6.5%	$90,000	$1.2M
Z	$42.375M	7.0%	$211,875	$2.825M
			Total	$13.75M

is 7.5%, the issue date is March 1, 2004, the settlement date is March 1, 2004, and the maturity is March 1, 2034. The following Matlab code computes cash flows between settle and maturity dates, the corresponding time factors in months from settle, and the mortgage pool factor (the fraction of loan principal outstanding) for each tranche:

```
% mbsfamounts
% [output] CFlowAmounts: vector of cash flows starting from Settle
% through end of the last month (Maturity)
%                CflowDates: indicates when cash flows occur, including
% at Settle.  A negative number at Settle indicates
% accrued interest is due.
%                TFactors: vector of times in months from Settle,
% corresponding to each cash flow.
%                Factors: vector of mortgage factors (the fraction of
% the balance still outstanding at the end of each month).
Settle = [datenum('1-Mar-2004');
              datenum('1-Mar-2004');
              datenum('1-Mar-2004')];
Maturity = [datenum('1-Mar-2034')];
IssueDate = datenum('1-Mar-2004');
GrossRate = 0.08125;
CouponRate = 0.075;
Delay = 30;
PSASpeed = [100; 165; 200];
[CPR, SMM] = psaspeed2rate(PSASpeed);
PrepayMatrix = ones(360,3);
PrepayMatrix(1:60,1) = SMM(1:60,1);
PrepayMatrix(1:90,2) = SMM(1:90,2);
PrepayMatrix(:,3)=SMM(:,3)
 [CFlowAmounts, TFactors, Factors] = mbscfamounts(Settle, Maturity,
  IssueDate, ...
   GrossRate, CouponRate, Delay, [], PrepayMatrix)
```

The cash flows for the difference sequential tranches are shown in Table 4.19.

We can compute the price and accrued interest of each of the mortgage pools by using the following code:

```
[Price, AccrInt] = mbsprice(Yield, Settle, Maturity, IssueDate,
GrossRate, CouponRate, Delay, PrepaySpeed, PrepayMatrix)
```

The price and accrued interest is shown in Table 4.20. No tranche pool has any accrued interest because the settlement date is the same as the issue date.

Table 4.19

Month	Tranche A CF	Tranche B CF	Tranche C CF
1	0	0	0
2	70,708.00	71,794.00	72,379.00
3	72,368.00	74,534.00	75,702.00
4	74,015.00	77,256.00	79,004.00
5	75,649.00	79,957.00	82,283.00
59	95,117.00	105,150.00	108,680.00
60	94,592.00	104,190.00	107,470.00
61	94,070.00	103,240.00	106,270.00
62	7,580,500.00	102,290.00	105,080.00
63	0.00	101,350.00	103,910.00
64	0.00	100,420.00	102,750.00
88	0.00	80,383.00	78,340.00
89	0.00	79,637.00	77,455.00
90	0.00	78,897.00	76,579.00
91	0.00	78,163.00	75,713.00
92	0.00	4,811,500.00	74,857.00
93	0.00	0.00	74,009.00
358	0.00	0.00	2,017.40
359	0.00	0.00	1,976.90
360	0.00	0.00	1,936.80
361	0.00	0.00	1,897.30

Table 4.20

	Price	Accrued Interest
Tranche A	101.1477	0
Tranche B	100.8520	0
Tranche C	100.7311	0

We can compute the effective duration and convexity of the mortgage pool using the `mbsdurp` (duration given price), `mbsdury` (duration given yield), `mbsconvp` (convexity given price), and `mbsconvy` (convexity given yield). For instance, continuing with the example, we can compute the yearly duration, modified duration, and convexity of the pool on March 10, 2004 for the 100, 150, and 200 PSA speed assumptions by making the function calls following the code:

```
% compute regular duration and modified duration
[YearDuration, ModDuration] = mbsdurp(Price, Settle, Maturity,
IssueDate, GrossRate, É
   CouponRate, Delay, PrepaySpeed)

% compute convexity
Convexity = mbsconvp(Price, Settle, Maturity, IssueDate, GrossRate,
CouponRate, Delay, É
    PrepaySpeed)
```

The duration, modified duration, and convexity results returned from the Matlab functions above for 100, 150, and 200 PSAs are shown in Table 4.21. Figure 4.6 shows simulated cash flows for a sequential-pay CMO.

Table 4.21

PSA	Year Duration	Mod. Duration	Convexity
100	7.0669	6.8148	82.3230
150	6.1048	5.8881	62.2476
200	5.3712	5.1814	48.3368

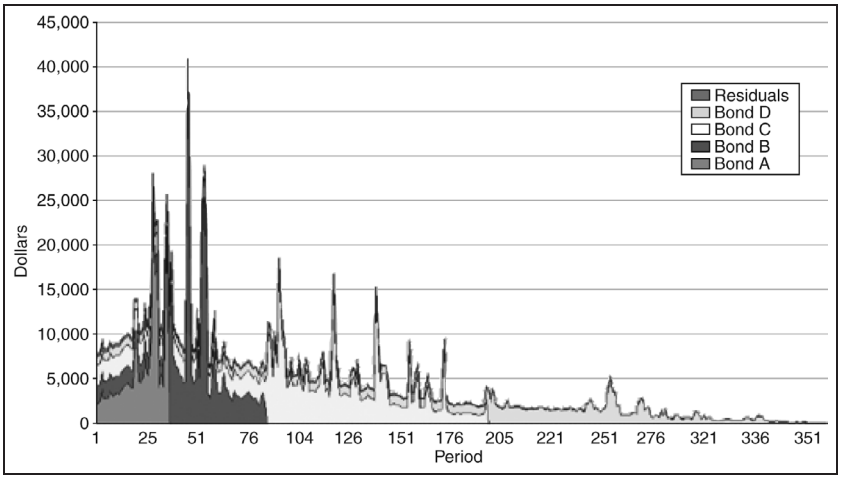


Figure 4.6 Simulated CMO Cash Flows. *Source: Bandic (2002), 28.*

4.8 CMO IMPLEMENTATION IN C++

CMO.h

```
#ifndef _CMO__
#define _CMO__

#include <vector>
#include <algorithm>
#include <numeric>
#include "Constants.h"
#include "MBS.h"



using namespace std;

class Tranche
{
        public:
                Tranche() {}
                Tranche(char clas, double balance, double coupon)
                        : initBalance_(balance), balance_(balance),
                            coupon_(coupon), clas_(clas) {}
                virtual ~Tranche() {}
                double initBalance_;
                double balance_;
                double coupon_;
                vector<double> cashFlows_;
                vector<double> sumCF_;
                vector<double> inter_;
                vector<double> principal_;
                vector<double> discount_;
                vector<double> T_;
                double price_;
                double interest_;
                double princip_;
                double averageLife_;
                char clas_;
};

class CMO
{
        public:
                CMO(MBS m, vector<Tranche> tr) : mbs(m), tranche(tr)
                {
                        for (int i = 0; i < tranche.size(); i++)
                                collateral_.push_back(tranche[i].balance_);
                }
                virtual ~CMO() { }
                void calcTrancheCF();
                inline double calcCPR(double SMM)
                  { return 100*(1-pow((1-(SMM/100)),12)); }
                inline double calcSMM(double scheduleBal,
                  double actualBal) {
```

```
                return 100*((double)(scheduleBal -
                  actualBal)/scheduleBal);
        }
                inline double calcPSA(double age, double CPR) {
                return 100*((double)(CPR/(0.2*min(age,30))));
        }
                inline double calcRefinance(double r) {

                double WAC = mbs.getWAC();
                double a = (double) 0.5/2;
                double b = (double) 100*((0.5 - a)/(PI/2));
                double d = (double) 0.06/b;
                double c = (double) -d*0.02;

                return (double) (a + b*(atan(c + d*(WAC - r))));
                }
                inline double calcBurnout(int t, Tranche tr,
                  double balance) {

                return (double) (0.3 +
                  0.7*((double)balance/1000000));
                }
                inline double calcMP(int t, Tranche tr, double balance)  {

                double WAC = mbs.getWAC();
                double WAM = mbs.getWAM();

                return balance*(((double)WAC/12)/(1-pow((1+
                  (double)WAC/12),-WAM+t)));
                }
                inline double calcIP(int t, Tranche tr, double r,
                  double balance) {

                double WAC = mbs.getWAC();

                return (balance)*((double)(tr.coupon_/12));
                }
                inline double calcPP(int t, Tranche tr, double r,
                  double balance) {

                double SMM = calcSMM1(t,tr,r,balance);
                double SP = calcSP(t,tr,r,balance);

                return SMM*(balance - SP);

                }
                inline double calcMM(int t) {

                double MM[12] = { 0.94, 0.76, 0.74, 0.95, 0.98,
                  0.92, 0.98, 1.10, 1.18, 1.22, 1.23, 0.98};
                int rem =  t % 12;

                if (t == 1)
                        rem = 1;

                return MM[rem-1];
```

```
                }
                inline double calcCPR1(int t, Tranche tr, double r,
                  double balance) {

                        double RI = calcRefinance(r);
                        double age = calcAge(t);
                        double MM = calcMM(t);
                        double BM = calcBurnout(t,tr,balance);

                        return RI*age*MM*BM;

                }
                inline double calcAge(int t) {

                        return min((double)t/30,1);
                }
                inline double calcSMM1(int t,Tranche tr, double r,
                  double balance) {

                        double CPR = calcCPR1(t,tr,r,balance);

                        return (1 - pow((1 - CPR),(double)1/12));
                }
                inline double calcSP(int t, Tranche tr, double r,
                  double balance) {

                        double MP = calcMP(t,tr,balance);
                        double IP = calcIP(t,tr,r,balance);

                        return MP - IP;
                }
                void calcCashFlows(double initRate, double financeRate,
                  int N, int M);
        private:
                MBS mbs;
                vector<Tranche> tranche;
                vector<double> collateral_;
};

#endif
```

Here are the method definitions:

CMO.cpp

```
#include "CMO.h"
#include "Utility.h"

void CMO::calcCashFlows(double initRate, double financeRate, int N, int M)
{
        Utility util;
        int i, t = 0;
        double r = 0.0715;
        const double kappa = 0.29368;
        const double vol = 0.11;
        const double theta = 0.08;
```

```
double deviate = 0;
long seed = 0;
long* idum = 0;
double balance = 0;
double sum = 0;
double S[4] = {0};
double sum1 = 0;
double sum2 = 0;
double sum3 = 0;
double sumA = 0;
double sumB = 0;
double sumC = 0;
double sumD = 0;
double CPR = 0;
double interest = 0.0;
double mbsPrice = 0;
double stdErr = 0;
double stdDev = 0;
double totalsum = 0;
double totalsumA = 0;
double totalsumB = 0;
double totalsumC = 0;
double totalsumD = 0;
double totalsum2 = 0;
double schedulePrincipal = 0;
double prepaidPrincipal = 0;
double discount = 0;
double principal = 0;
double pay = 0.0;
double r1 = 0.0;
double rr = 0.0;
int cnt = 0;
double trancheBal = 0.0;
double T = mbs.getMaturity();
double WAM = mbs.getWAM();
double OAS = mbs.getOAS();
double dt = (double) T/N;
double interest1 = 0;
vector<double> disc(0.0);
vector<double> time1;
TNT::Array1D<double> CF(SIZE_X);   // cash_flow
vector<double> p;

srand(unsigned(time(0)));
seed = (long) rand() % 100;
idum = &seed;

for (t = 1; t <= N; t++)
        time1.push_back((double)(t-1)/12);

for (i = 0; i < M; i++)
{
        r = initRate;
        sum = 0;
        sumA = 0;
        sumB = 0;
        sumC = 0;
```

```
sumD = 0;
schedulePrincipal = 0;
prepaidPrincipal = 0;
balance = 1000000;
cnt = 1;
disc.clear();
disc.empty();
disc.push_back(r);
p.clear();
p.push_back(0);

for (int j = 0; j < tranche.size(); j++)
{
        tranche[j].balance_ = collateral_[j];
        tranche[j].inter_.clear();
        tranche[j].principal_.clear();
        trancheBal = calcPP(0,tranche[j],r,
                        tranche[j].balance_) +
                        calcMP(0,tranche[j],
                        tranche[j].balance_);
        tranche[j].principal_.push_back(trancheBal);
        tranche[j].interest_ = calcIP(0,tranche[j],r,
                                tranche[j].balance_);
        tranche[j].inter_.push_back(tranche[j].interest_);
        S[j] = 0;
}

for (t = 1; t <= N; t++)
{
        balance = balance - (schedulePrincipal +
                    prepaidPrincipal);
        deviate = util.gasdev(idum);
        r = r + kappa*(theta - r)*dt +
              vol*r*sqrt(dt)*deviate;
        disc.push_back(r);
        interest = calcIP(t,tranche[cnt-1],r,balance);
        schedulePrincipal = calcMP(t,tranche[cnt-1],
                                balance);
        prepaidPrincipal = calcPP(t,tranche[cnt-1],r,
                               balance);
        tranche[cnt-1].balance_ = tranche[cnt-1].balance_ -
                                    schedulePrincipal -
                                    prepaidPrincipal;
        principal = schedulePrincipal + prepaidPrincipal;
        tranche[cnt-1].principal_.push_back(principal);
        tranche[cnt-1].princip_ = principal;
        p.push_back(principal);

        if (tranche[cnt-1].balance_ > 0)
                interest1 = calcIP(t,tranche[cnt-1],r,
                                tranche[cnt-1].balance_);
        else
                interest1 = 0;
        tranche[cnt-1].inter_.push_back(interest1);
        tranche[cnt-1].interest_ = interest1;

        for (int k = 1; k <= tranche.size(); k++)
```

```
{
        if (k != cnt)
        {
                interest1 = calcIP(t,tranche[k-1],r,
                            tranche[k-1].balance_);

                if (tranche[k-1].balance_ != 0 )
                {
                        tranche[k-1].inter_.
                          push_back(interest1);
                        tranche[k-1].interest_ =
                          interest1;
                }
                else
                {
                        tranche[k-1].inter_.
                          push_back(0.0);
                        tranche[k-1].interest_ =
                          0.0;
                }

                tranche[k-1].principal_.
                  push_back(0.0);
                tranche[k-1].princip_ = 0.0;
        }

        rr = mbs.computeZeroRates(t-1,disc);
        S[k-1] = (tranche[k-1].interest_ +
                    tranche[k-1].princip_)/
                    (pow(1+rr+OAS,
                    (double)(t-1)/12));

        if (k == 1)
                sumA = sumA + S[k-1];
        else if (k == 2)
                sumB = sumB + S[k-1];
        else if (k == 3)
                sumC = sumC + S[k-1];
        else
                sumD = sumD + S[k-1];
}

if (tranche[cnt-1].balance_ > 0)
{
        if (balance >= schedulePrincipal)
        {
                if (t != N)
                        CF[t-1]
                          = schedulePrincipal +
                                  interest +
                                  prepaidPrincipal;
                else
                        CF[t-1] = interest +
                                    balance;

                rr = mbs.computeZeroRates(t-1,disc);
                sum = sum + CF[t-1]/(pow(1+rr+OAS,
```

```
                                        (double)(t-1)/12));

                        }
                        else
                                goto x;
                }
                else
                {
                        tranche[cnt-1].balance_ = 0;
                        cnt++;
                }
        }

        x:
        totalsum = totalsum + sum;
        totalsumA = totalsumA + sumA;
        totalsumB = totalsumB + sumB;
        totalsumC = totalsumC + sumC;
        totalsumD = totalsumD + sumD;
        totalsum2 = totalsum2 + sum*sum;
}

calcTrancheCF();

for (int j = 0; j < tranche.size(); j++)
{
        sum1 = 0;
        sum2 = 0;
        for (i = 0; i < tranche[j].principal_.size(); i++)
        {
                sum1 = sum1 + (time1[i])*(tranche[j].principal_[i]);
                sum2 = sum2 + tranche[j].principal_[i];
        }
        tranche[j].averageLife_ = sum1/sum2;
}

sum1 = 0;
sum = accumulate(p.begin(),p.end(),0);
for (j = 0; j < p.size(); j++)
        sum1 = sum1 + time1[j]*p[j];

std::cout << endl;
std::cout << "collateral price = " << totalsum/M <<  " " <<
               "Ave.Life = " << sum1/sum << endl;
std::cout << "Tranche A price =  " << totalsumA/M << " " <<
               "Ave.Life = " << tranche[0].averageLife_ <<
endl;
std::cout << "Tranche B price =  " << totalsumB/M << " " <<
               "Ave.Life = " << tranche[1].averageLife_ <<
endl;
std::cout << "Tranche C price =  " << totalsumC/M << " " <<
               "Ave.Life = " << tranche[2].averageLife_ <<
endl;
std::cout << "Tranche Z price =  " << totalsumD/M << " " <<
               "Ave.Life = " << tranche[3].averageLife_ <<
endl;
```

```
        T = mbs.getMaturity();
        stdDev = sqrt(totalsum2 - (double)(totalsum*totalsum)/M)*
                    (exp(-2*initRate*T)/(M-1));
        stdErr = (double) stdDev/sqrt(M);
}


void CMO::calcTrancheCF()
{

        vector<Tranche>::iterator iter;
        vector<double>::iterator iter1;
        vector<double>::iterator iter2;
        int cnt = 1;

        for (iter = tranche.begin(); iter != tranche.end(); iter++)
        {
                iter2 = iter->inter_.begin();
                cnt = 1;
                for (iter1 = iter->principal_.begin(); iter1 !=
                                iter->principal_.end(); iter1++)
                {
                        std::cout << "Mo." << cnt << "  Class: " <<
                                        iter->clas_ << " " <<
                                        "Principal= " << *iter1
                        << " " << "Coupon= " << *iter2 << endl;
                        iter2++;
                        cnt++;
                }
        }
}
```

The main method is as follows:

```
void main()
{

      std::cout.precision(7);
      double principal = 1000000;      // underlying principal notional)
                                       // of MBS
      double coupon = 0.08;            // coupon rate
      double WAC = 0.08;               // weighted average coupon rate
      double WAM = 10;                 // weighted average maturity
      double OAS = 0.02;               // option adjusted spread
      double initSpotRate = 0.06;      // spot rate
      double initRefinanceRate = 0.08; // refinance rate
      int N = 10;                      // number of time steps in tree
      long int M = 100000;             // number of simulation paths

      MBS mbs(principal,coupon,WAC,WAM,OAS);

      vector<Tranche> tranche;
      Tranche trA('A',500000,0.06);
      tranche.push_back(trA);
      Tranche trB('B',300000,0.065);
      tranche.push_back(trB);
```

```
    Tranche trC('C',200000,0.07);
    tranche.push_back(trC);
    Tranche trZ('Z',100000, 0.075);
    tranche.push_back(trZ);

    std::cout << endl;
    std::cout << "Pricing CMO Tranches..." << endl << endl;
    CMO cmo(mbs,tranche);
    cmo.calcCashFlows(initSpotRate,initRefinanceRate,N,M);

}
```

The output is as follows:

```
Pricing CMO Tranches...

Mo.1   Class: A Principal= 51851.6  Coupon= 2500
Mo.2   Class: A Principal= 115430   Coupon= 1922.85
Mo.3   Class: A Principal= 114668.5 Coupon= 1349.507
Mo.4   Class: A Principal= 113798.2 Coupon= 780.5165
Mo.5   Class: A Principal= 113024   Coupon= 215.3967
Mo.6   Class: A Principal= 111815.5 Coupon= 0
Mo.7   Class: A Principal= 0        Coupon= 0
Mo.8   Class: A Principal= 0        Coupon= 0
Mo.9   Class: A Principal= 0        Coupon= 0
Mo.10  Class: A Principal= 0        Coupon= 0
Mo.1   Class: B Principal= 31110.96 Coupon= 1625
Mo.2   Class: B Principal= 0        Coupon= 1625
Mo.3   Class: B Principal= 0        Coupon= 1625
Mo.4   Class: B Principal= 0        Coupon= 1625
Mo.5   Class: B Principal= 0        Coupon= 1625
Mo.6   Class: B Principal= 0        Coupon= 1625
Mo.7   Class: B Principal= 110373.5 Coupon= 1027.143
Mo.8   Class: B Principal= 108932.9 Coupon= 437.0903
Mo.9   Class: B Principal= 107334.2 Coupon= 0
Mo.10  Class: B Principal= 0        Coupon= 0
Mo.1   Class: C Principal= 20740.64 Coupon= 1166.667
Mo.2   Class: C Principal= 0        Coupon= 1166.667
Mo.3   Class: C Principal= 0        Coupon= 1166.667
Mo.4   Class: C Principal= 0        Coupon= 1166.667
Mo.5   Class: C Principal= 0        Coupon= 1166.667
Mo.6   Class: C Principal= 0        Coupon= 1166.667
Mo.7   Class: C Principal= 0        Coupon= 1166.667
Mo.8   Class: C Principal= 0        Coupon= 1166.667
Mo.9   Class: C Principal= 0        Coupon= 1166.667
Mo.10  Class: C Principal= 105320.6 Coupon= 552.2968
Mo.1   Class: Z Principal= 10370.32 Coupon= 625
Mo.2   Class: Z Principal= 0        Coupon= 625
Mo.3   Class: Z Principal= 0        Coupon= 625
Mo.4   Class: Z Principal= 0        Coupon= 625
Mo.5   Class: Z Principal= 0        Coupon= 625
Mo.6   Class: Z Principal= 0        Coupon= 625
Mo.7   Class: Z Principal= 0        Coupon= 625
Mo.8   Class: Z Principal= 0        Coupon= 625
Mo.9   Class: Z Principal= 0        Coupon= 625
Mo.10  Class: Z Principal= 0        Coupon= 625
```

```
collateral price = 682754.6 Ave.Life = 0.4104376
Tranche A price =  564751.1 Ave.Life = 0.2279203
Tranche B price =  321741.6 Ave.Life = 0.5318972
Tranche C price =  108757.8 Ave.Life = 0.6266037
Tranche Z price =  5460.491 Ave.Life = 0
```

4.9 PLANNED AMORTIZATION CLASSES (PACS)

Planned amortization classes (PACs) (also called planned redemption obligations) are tranches set up such that they have zero (or at least minimum) prepayment risk. PACs are set up by applying low and high PSA speeds to the collateral. The PAC bond then receives a promise of the minimum CF each month, with a support bond created that receives the rest. These support classes, sometimes referred to as companions, absorb principal payments and pay off sooner if the PSA exceeds the PAC range. If the PSA is below the range, the companion classes have a longer life and amortization schedule. In either case, the PAC classes experience less volatility than they would in a sequential-pay structure because of the stability provided by the companions.

PACs are much less sensitive to prepayment risk than standard pass-throughs as long as the PSA speed falls between the low and high PSA thresholds used. Suppose one applied 90 and 300 PSA models to the collateral of a \$100M mortgage pool with a WAC $= 8.125\%$, WAM $= 357$ months, and PT rate $= 7.5\%$. This would yield two different monthly principals over the 357-month period. We also need to factor in a seasoning factor, a factor that accounts for the prepayment based on the season, which we assume to be 3. Table 4.22 shows the cash flows for a PAC with the above features.

To show how these calculations were made, we know that in the first month (period 1), the PAC balance is \$100,000,000. The computed interest for this first month is:

$$I_1 = (0.075/12) * (100,000,000) = \$625,000$$

The computed PAC principal payment is:

$$p = \frac{(0.08125/12)(100,000,000)}{1-(1/(1+0.08125/12))^{357}} = \$743,967.06$$

The PAC scheduled principal payment is:

$$743,967.06 - (0.085/12)(100,000,000) = 66,883.73$$

The low PAC PSA speed is assumed to be 90. We compute the PAC adjuster seasoning factor, which we take to be:

$$Min\left(\frac{month + seasoning\ factor}{number\ of\ months\ until\ fixed\ CPR}, 1\right)$$

Thus, in the first month, this value is:

$$Min\left(\frac{1+3}{30}, 1\right) = 0.13333$$

Table 4.22

Pac Period	Pac Low PSA Pr	Pac high PSA Pr	Pac Min. Principal	Pac Int 0.08	collateral Balance 100,000,000.00	collateral Interest	collateral Prncipal
1	127,042.38	268,982.80	127,042.38	381,088.87	100,000,000.00	625,000.00	177,480.87
2	142,460.86	319,853.03	142,460.86	380,294.85	99,822,519.13	623,890.74	205,473.91
3	157,844.28	370,548.61	157,844.28	379,404.47	99,617,045.22	622,606.53	233,389.97
4	173,185.47	420,991.54	173,185.47	378,417.95	99,383,655.25	621,147.85	261,205.39
5	188,477.28	471,103.66	188,477.28	377,335.54	99,122,449.86	619,515.31	288,896.53
6	203,712.56	520,806.83	203,712.56	376,157.55	98,833,553.33	617,709.71	316,439.78
7	218,884.17	570,023.17	218,884.17	374,884.35	98,517,113.54	615,731.96	343,811.60
102	363,181.39	371,031.65	363,181.39	129,910.54	41,682,140.85	260,513.38	421,018.70
103	361,690.25	364,654.51	361,690.25	127,640.65	41,261,122.15	257,882.01	417,259.77
104	360,206.39	358,384.81	358,384.81	125,380.09	40,843,862.38	255,274.14	413,533.58
105	358,729.77	352,220.75	352,220.75	123,140.18	40,430,328.80	252,689.56	409,839.84
106	357,260.36	346,160.58	346,160.58	120,938.80	40,020,488.96	250,128.06	406,178.29
107	355,798.12	340,202.57	340,202.57	118,775.30	39,614,310.67	247,589.44	402,548.63
356	150,750.84	2,573.92	2,573.92	31.76	74,797.79	467.49	37,597.30
357	150,372.35	2,507.45	2,507.45	15.67	37,200.49	232.50	37,200.49

Source: Johnson, S. (2004)

The PAC adjuster seasoning factor is then applied to the PAC CPR:

$$\left(\frac{PSA}{100}\right)(\textit{Fixed CPR})(\textit{adjuster seasoning factor}) = \left(\frac{90}{100}\right)(0.06)(0.1333) = 0.0072$$

The SMM is:

$$\textit{SMM} = (1-(1-\textit{PAC CPR})^{1/357} = (1-(1-0.0072)^{1/357} = 0.00060199$$

The PAC prepaid principal is:

$$\textit{SMM}(\textit{balance} - \textit{scheduled principal}) \cdot$$
$$0.00060199(100,000,000 - 66,883.73) = \$60,158.65$$

The PAC cash flow is:

$$\textit{PAC Interest} + \textit{PAC Scheduled Principal} + \textit{PAC Prepaid Principal} = 625,000 + 66,883.73 + 60,148.65 = \$752,042.38$$

We can now compute the PAC low PSA total prepayment amount:

$$\textit{PAC Scheduled Principal} + \textit{PAC Prepaid Principal} = 66,883.72 + 60,158.65 = \$127,042.38$$

The same computations are then made for the high PSA level of 300. The main difference is that the PAC CPR in this case is:

$$\left(\frac{PSA}{100}\right)(\textit{Fixed CPR})(\textit{adjuster seasoning factor}) = \left(\frac{300}{100}\right)(0.06)(0.1333) = 0.024$$

The PAC bond has an average life of 7.26 years. Moreover, between PSA speeds of 90 and 300, the PAC bond's average life is 7.26 years, implying no prepayment risk. Table 4.23 shows average life for the PAC and support bond for various PSA assumptions.

The PAC bond can be broken into other PAC tranches. The most common is a sequential-pay PAC. For example, one can form six sequential-pay PACs using the previous collateral. The average life for the PAC classes will be stable within the 90–300 PSA range; 90–300 is referred to as the *collar*. Some PACs will move outside that range. This is referred to as the *effective collar*. The more classes you have, the more narrow you make the windows, making the PAC resemble a bullet bond. Such PACs could be sold to liability-management funds to meet liabilities with certain liabilities or durations—cash flow matching. In the 1980s, one could find CMOs (especially PACs) with as many as 70 tranches; in the early 1990s, the average number of tranches was 24. Like the PACs, the support bond also can be divided into different classes: sequential-pay, floaters, accrual bonds, and so on.

Table 4.23

PSA Speed	PAC	Support	Collateral
0	10.36	23.84	20.36
50	8.04	21.69	15.36
90	7.26	20.06	12.26
100	7.26	18.56	11.67
150	7.26	12.56	9.33
200	7.26	8.36	7.69
250	7.26	5.35	6.52
300	7.26	3.11	5.64
350	6.61	2.91	4.98
400	6.06	2.74	4.45

Targeted amortization classes (TACs) also offer prepayment protection within a defined PSA range, but not below the PSA used to price the CMO. This could result in a lengthening of average life if prepayments slow down, and for this reason, TACs offer higher yields in relation to comparable PACs.

Figure 4.7 shows the cash flows paid to a hypothetical GNMA PAC 100/300.

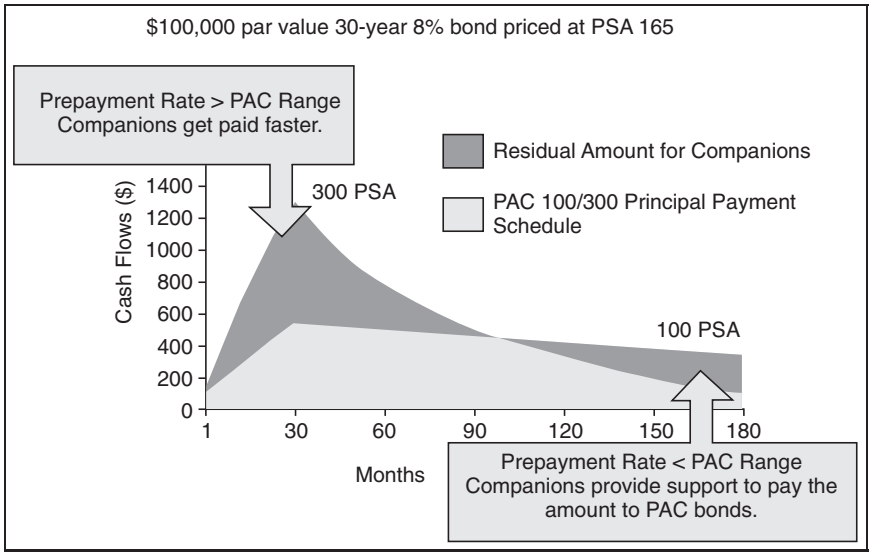


Figure 4.7 GNMA PAC 100/300 Cash Flows

4.10 Principal- and Interest-Only Strips

Stripped MBSs were introduced by FNMA in 1986. Any MBS can be "stripped" and sold separately by directing a collateral's cash flows into principal-only (PO) or interest-

only (IO) securities. IO classes receive just the interest on the mortgages. PO classes receive just the principal payments. The yield on PO bonds depends on the speed of prepayment. The faster the prepayment, the greater the yield. For instance, PO investors who paid \$75 million for a mortgage portfolio with a principal of \$100 million would receive a higher yield if the \$100 million were paid early (e.g., first years) than if it were spread out. POs have an inverse price-interest rate sensitivity relationship: If interest rates decrease, then prepayments increase so that the PO (yield) return increases and its price (value) increases. Analogously, if interest rates increase, then prepayments decrease so that the PO (yield) return decreases and its price (value) decreases.

Because IO investors receive interest on the outstanding principal, they want prepayments to be slow. For example, IO investors holding an IO claim on a \$400 million 7.5% pool would receive \$30 million ($= \$400 * 0.075$) if the principal were paid immediately. By contrast, if the principal were paid off by equal increments over four years, the return would be \$75 million (see Table 4.24).

Because a rate decrease augments speed, it lowers the return on an IO bond, causing its value to decrease. Whether IO bonds decrease in response to a rate decrease depends on whether this effect dominates the effect of lower discount rates on increasing value. In other words, when interest rates decrease, the prepayments increase, which decreases the return, and the value of the IO must be balanced against the increase in value from the effect of lower discount rates. The two effects may offset one another so that it is possible that there is a direct relationship between value and return for an IO bond.

Like CMOS, stripped MBSs are a derivative product. Both strips are extremely volatile and, as stated, dependent on prepayment rates. POs perform well in high prepayment environments when the principal purchased at a discount is returned at par, faster than expected, making them a bullish investment with a large, positive duration. IOs perform better if prepayments are slow because the principal remains outstanding for a longer period and interest payments continue, making them an investment with a negative duration—i.e., their price increases as interest rates increase and vice versa. IOs are often used to hedge interest rate risks in MBS or CMO portfolios. Portfolio losses that are caused by an increase in rates are partially or fully offset by a corresponding appreciation in the IO position, depending on the structure of the hedge.

Consider a stripped MBS with a collateral pool of \$100 million, WAM $= 357$ months, WAC $= 0.08125$, PT rate $= 0.075$, and a PSA $= 165$. The cash flows are given as shown in Table 4.25.

Table 4.24

Year 1	$(\$400M)(0.75) = \$30.0M$
Year 2	$(\$300M)(0.75) = \$22.5M$
Year 3	$(\$200M)(0.75) = \$15.0M$
Year 4	$(\$100M)(0.75) = \$7.5M$
Total	$\$75M$

Table 4.25

Period	Balance 100000000	Interest	Sch. Prin.	Prepaid Prin.	Principal		Stripped MBS PO	IO
1	100000000	625000	65216.47156	110598.9911	175815.4627		175815.4627	625000
2	99824184.54	623901.153	65592.16276	138216.4665	203808.6293		203808.6293	623901.2
3	99620375.91	622627.349	65951.60553	165774.6371	231726.2426		231726.2426	622627.3
4	99388649.67	621179.06	66294.44744	193250.204	259544.6515		259544.6515	621179.1
5	99129105.01	619556.906	66620.34673	220619.862	287240.2088		287240.2088	619556.9
6	98841864.81	617761.655	66928.97288	247860.3319	314789.3047		314789.3047	617761.7
7	98527075.5	615794.222	67220.00709	274948.3929	342168.4		342168.4	615794.2
104	40930020.29	255812.627	59884.94969	353521.5406	413406.4903		413406.4903	255812.6
105	40516613.8	253228.836	59775.10019	349946.5766	409721.6768		409721.6768	253228.8
106	40106892.12	250668.076	59665.45219	346403.4839	406068.9361		406068.9361	250668.1
107	39700823.19	248130.145	59556.00532	342891.9853	402447.9906		402447.9906	248130.1
108	39298375.2	245614.845	59446.75922	339411.8057	398858.5649		398858.5649	245614.8
109	38899516.63	243121.979	59337.71351	335962.6724	395300.3859		395300.3859	243122
110	38504216.25	240651.352	59228.86783	332544.3152	391773.183		391773.183	240651.4
111	38112443.06	238202.769	59120.22181	329156.4661	388276.6879		388276.6879	238202.8
112	37724166.38	235776.04	59011.77508	325798.8595	384810.6346		384810.6346	235776
113	37339355.74	233370.973	58903.52728	322471.2322	381374.7594		381374.7594	233371
356	75665.72172	472.910761	37703.25592	328.3705641	38031.62648		38031.62648	472.9108
357	37634.09524	235.213095	37634.09524	-3.02093E-12	37634.09524		37634.09524	235.2131

4.11 Interest Rate Risk

The MBS interest rate risk is similar to that of other fixed-income securities: When interest rates fall, price goes up and vice versa. However, the prepayment optionality "embedded" in MBS impacts the degree of price movement based on the relationship between the security's coupon rate and current mortgage rates. When a pass-through coupon is either at or above current mortgage rates, homeowners are more likely to exercise the prepayment option. As the likelihood of prepayment increases, the price of the MBS pass-through does not go up as much as that of an otherwise identical security with no optionality due to the increased prepayment risk. This is known as *negative convexity*.

When a pass-through coupon is below current mortgage rates, or "out of the money," homeowners are unlikely to exercise the prepayment option. Although the prepayment option is less likely to be exercised, the price of the pass-through still exhibits negative convexity because investors maintain their principal investment at lower levels than the current market rate for longer periods of time. This is known as *extension risk*. Figure 4.8 shows the inverse relationship between the price of a regular fixed-income security and interest rates. Figure 4.9 shows the negative convexity of an MBS.

4.12 Dynamic Hedging of MBS

Institutions hold significant positions in mortgage-backed securities (MBSs) for a variety of reasons. Hedging interest rate risk of MBSs is an important concern whether the positions reflect trades on relative value or inventory holdings due to main businesses. MBSs hedging is complicated by the fact that the timing of the cash flows is dependent on the prepayment behavior of the pool. In particular, mortgagors are more likely to prepay given the incentive to refinance when interest rates fall. Thus, fixed-rate investors are implicitly writing a call option on the corresponding fixed-rate bond.[25] Though other factors influence

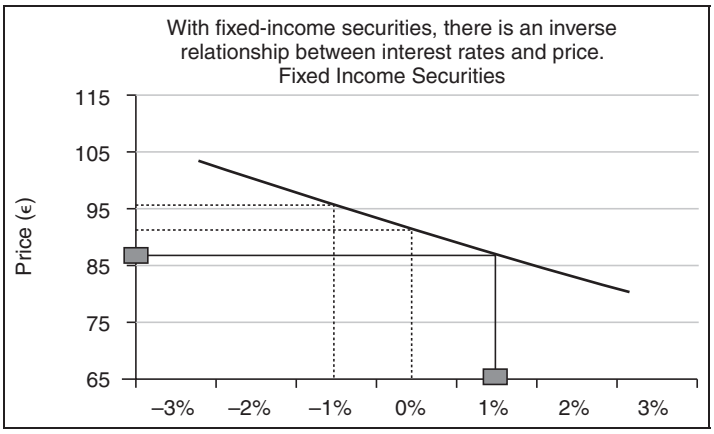


Figure 4.8 Inverse relationship between the price of a regular fixed income security and interest rates

prepayments—e.g., seasonality and burnout—interest rates are the predominant factor in valuing MBSs. Because of this predominance, U.S. Treasury securities, or more specifically, Treasury note (T-note) futures, are often used to hedge MBSs. There are two reasons: (1) T-note futures are very liquid instruments; and (2) the prices of those instruments are determined by the underlying term structure of interest rates and thus relate directly to the value of MBSs.[26] We follow the work of Boudoukh, Richardson, Stanton, and Whitelaw (1995) in the following discussion.

There are two common approaches to hedging MBSs using T-note futures. The first is purely empirical and involves the regression of past returns on MBSs against past returns on T-note futures. The estimated regression coefficient from the resulting relation can then be used to hedge the interest rate risk of MBSs using the risk in T-notes. The advantage of

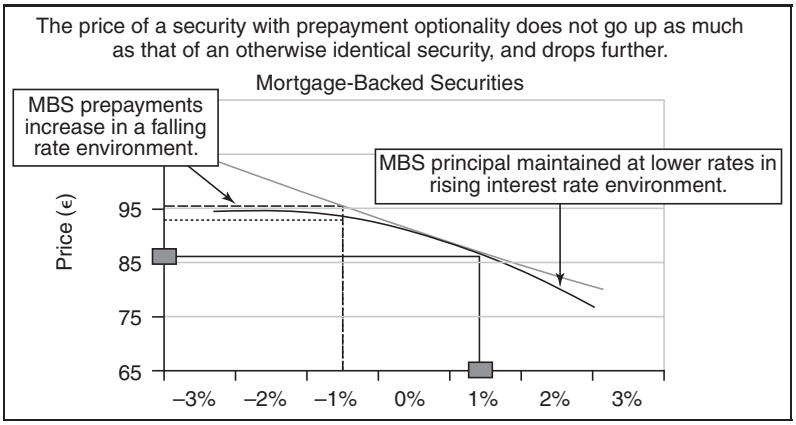


Figure 4.9 Negative convexity of a MBS

this method is that it does not involve strong assumptions regarding the underlying model for the evolution of interest rates or prepayments.[27] The disadvantage is that the method is static in nature. It does not explicitly adjust the hedge ratio for changes in interest rates and mortgage prepayments, which can potentially be detrimental from mishedging a large portfolio exposure.[28] Consequently, the observations used in the regression represent an average of the relation between MBSs and T-note futures only over the sample period, which may or may not be representative of the current period.

As an alternative, a second approach is model-based. It involves specification of the interest rate process and a prepayment model. The assumptions then help map an MBS pricing functional to interest rates and possibly other factors.[29] The approach represents a dynamic method for determining co-movements between MBS prices and T-note futures prices. These co-movements are completely specified by conditioning on current values of the relevant economic variables and on particular parameter values. The basic idea is to estimate a conditional hedge ratio between returns on an MBS and returns on a T-note futures. This is important for MBSs because, as interest rates change, expected future prepayments change, and thus the timing of the future prepayments change, and thus timing of the future cash flows also changes.

To estimate the conditional hedge ratio, a structural model is usually required (as with model-based MBS valuation approaches). There are two drawbacks: First, there is no consensus regarding what is a reasonable specification of how the term structure moves through time, and how these movements relate to prepayment behavior. The model price is going to be closely related to these possibly ad-hoc assumptions, which may be reasonable or unreasonable.[30] Second, and more subtle, is the recognition that the parameter values themselves may often be "chosen" or estimated from a static viewpoint.[31] For instance, empirical prepayment models often reflect ad hoc prepayment rates on data sets housing and interest rate factors. But any of the well-documented MBS-hedging fiascoes would imply that the resulting regression coefficients, which present an average of the relation of the past, do not have the same link to the variable factors describing the current period. In other words, static in-sample regression estimated coefficients are not accurate estimators of future out-of-sample coefficients.

One method that has worked well in reducing the error between in-sample and out-of-sample estimators is the probability density estimation method.

The Multivariable Density Estimation Method

Multivariate density estimation (MDE) is a method for estimating the joint density of a set of variables. Given the joint and marginal densities of these variables, the corresponding distributions and conditional moments, such as the mean, can be calculated. The estimation relates the expected return on an MBS to the return on a T-note futures, conditional on relevant information available at any point in time. We have T observations, $\mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_T$, where each $\mathbf{z}_t$ is an m-dimensional vector that might include the MBS and T-note futures returns, as well as several variables describing the state of the economy. One popular consistent measure of the joint density is the Parzen (fixed window width) density estimator:

$$\widehat{f}(\mathbf{z}^*) = \frac{1}{Th^m}\sum_{t=1}^{T} K\left(\frac{z^* - z_t}{h}\right)$$

where $K(\cdot)$ is called the kernel function (with the property that it integrates to unity) and is often chosen to be a density function, h is window or smoothing parameter (which helps determine how tight the kernel function is), and $\widehat{f}(z^*)$ is the estimate of the probability density at $\mathbf{z}^*$. The density at any point $\mathbf{z}^*$ is estimated as the average of densities centered at the actual data points $\mathbf{z}_t$. The further a data point is away from the estimation point, the less it contributes to the estimated density. Consequently, the estimate is highest near high concentrations of data points and lowest when observations are sparse.[32] A commonly used kernel is the multivariate normal density:

$$K(\mathbf{z}) = \frac{1}{(2\pi)^{m/2}} e^{-\frac{1}{2}\boldsymbol{z'z}}$$

Let $\mathbf{z}_t = (R_{t+1}^{mbs}, R_{t+1}^{TN}, \mathbf{x}_t)$, where R_{t+1}^{mbs} and R_{t+1}^{TN} are the one-period returns on the MBS and T-note futures from t and $t+1$, respectively, and $\mathbf{x}_t$ is an $(m-2)$-dimensional vector of factors known at time t. We can then obtain the conditional mean, $E\left[R_{t+1}^{mbs}|R_{t+1}^{TN}, \mathbf{x}_t\right]$—i.e., the expected MBS return given movements in the T-note return—conditional on the current economic state as described by $\mathbf{x}_t$. Specifically,

$$\begin{aligned} E\left[R_{t+1}^{mbs}|R_{t+1}^{TN}, \mathbf{x}_t\right] &= \int R_{t+1}^{mbs} \frac{f(R_{t+1}^{mbs}, R_{t+1}^{TN}, \mathbf{x}_t)}{f_1(R_{t+1}^{TN}, \mathbf{x}_t)} dR^{mbs} \\ &= \frac{\sum_{i=1}^{t} R_{t+1-i}^{mbs} K_1^{t-i}(\cdot,\cdot)}{\sum_{i=1}^{t} K_i^{t-i}(\cdot,\cdot)} \end{aligned} \tag{4.13}$$

where $K_1^{t-i}(\cdot,\cdot) = K_1((R_{t+1-i}^{TN} - R_{t+1}^{TN})/h^{TN}), (\mathbf{x}_{t-i} - \mathbf{x}_t)/h))$.

$K_1(\cdot,\cdot)$ is the marginal density, $\int K(z) dR^{mbs}$, which is also a multivariate normal density. The expected return in equation (4.13) is simply a weighted average of past returns where the weights depend on the levels of the conditioning variables relative to their levels in the past.

Given $E\left[R_{t+1}^{mbs}|R_{t+1}^{TN}, \mathbf{x}_t\right]$, a hedge ratio can be formed by estimating how much the return on the MBS changes as a function of changes in the T-note futures return, conditional on currently available information $\mathbf{x}_t$. That is

$$\frac{\partial E\left[R_{t+1}^{mbs}|R_{t+1}^{TN},\mathbf{x}_t\right]}{\partial R_{t+1}^{TN}} = \frac{\sum_{i=1}^{t} R_{t+1-i}^{mbs}\frac{\partial K_1^{t-i}(\cdot,\cdot)}{\partial R_{t+1}^{TN}}}{\sum_{i=1}^{t} K_1^{t-1}(\cdot,\cdot)} - \frac{\sum_{i=1}^{t} R_{t+1-i}^{mbs}K_1^{t-i}(\cdot,\cdot)\sum_{i=1}^{t}\frac{\partial K_1^{t-1}(\cdot,\cdot)}{\partial R_{t+1}^{TN}}}{\left[\sum_{i=1}^{t} K_1^{t-i}(\cdot,\cdot)\right]^2} \tag{4.14}$$

where

$$\frac{\partial K_1^{t-i}(\cdot,\cdot)}{\partial R_{t+1}^{TN}} = -\left[\frac{(R_{t+1-i}^{TN} - R_{t+1}^{TN})}{(h^{TN})^2}\right] K_1^{t-i}(\cdot,\cdot).$$

A couple of points can be made. First, equation (4.14) provides a formula for the hedge ratio between an investor's MBS position and T-note futures. For example, if $\frac{\partial E\left[R_{t+1}^{mbs}|R_{t+1}^{TN},\mathbf{x}_t\right]}{\partial R_{t+1}^{TN}}$ equals 0.5, then for every \$1 of an MBS held, the investor should short \$0.50 worth of T-note futures. Second, the hedge ratio will change dynamically, depending on the current economic state described by $\mathbf{x}_t$. For example, suppose $\mathbf{x}_t$ is an $m-2$ vector of term structure variables. As these variables change, whether they are the level, slope, or curvature of the term structure, the hedge ratio may change in response. Thus, the appropriate position in T-note futures will vary over time. Third, the hedge ratio is a function of the unknown return on the T-note futures. If the conditional relation between MBS returns and T-note futures returns is always linear, then the same hedge ratio will be appropriate, regardless of how T-note futures move. If the relation is not linear, then the investor must decide what type of T-note moves to hedge. For example, the investor might want to form the MBS hedge in the neighborhood of the conditional mean of the T-note futures return because many of the potential T-note futures will lie in that region. On the other hand, it may be the case that the investor is concerned about the tails of the distribution T-note futures returns, and thus adjusts the hedge ratio to take account of potential extreme moves in interest rates and T-note futures. Fourth, the hedge ratio is horizon specific. In contrast to the instantaneous hedge ratio, the method's implied hedge ratio directly reflects the distribution of MBS returns over the relevant horizon. Thus, different hedge ratios may be appropriate for daily, weekly, or monthly horizons.

The static OLS regression coefficient, or hedge ratio, is given by

$$\beta = \frac{\sum_{i=1}^{t} R_{t+1-i}^{mbs}R_{t+1-i}^{TN} - T\mu_{mbs}\mu_{TN}}{\sum_{i=1}^{t}(R_{t+1-i}^{TN} - \mu_{TN})^2} \tag{4.15}$$

where $\mu_{mbs} = \frac{1}{T}\sum_{i=1}^{t} R_{t+1-i}^{mbs}$ and $\mu_{TN} = \frac{1}{T}\sum_{i=1}^{t} R_{t+1-i}^{TN}$.

In contrast, the dynamic hedging method explicitly takes into account the current economic state. Equation (4.14) can be rewritten as

$$\frac{\partial E\left[R_{t+1}^{mbs}|R_{t+1}^{TN},\mathbf{x}_t\right]}{\partial R_{t+1}^{TN}} = \sum_{i=1}^{t} R_{t+1-i}^{mbs}\left[\frac{(R_{t+1-i}^{TN}-R_{t+1}^{TN})}{(h^{TN})^2}\right]w_i(t) - \left[\sum_{i=1}^{t} R_{t+1-i}^{mbs}w_i(t)\right]\cdot \left[\sum_{i=1}^{t}\left(\frac{(R_{t+1-i}^{TN}-R_{t+1}^{TN})}{(h^{TN})^2}\right)w_i(t)\right] \tag{4.16}$$

where $w_i(t) = \frac{K_1^{t-i}(\cdot,\cdot)}{\sum_{i=1}^{t} K_1^{t-i}(\cdot,\cdot)}$.

The hedge ratio in (4.16) is constructed by taking past pairs of MBS and T-note futures returns, and then differentially weighting these pairs' co-movements by determining how "close" $(R_{t+1-i}^{TN}, \mathbf{x}_{t-i})$ pairs are to a chosen value of R_{t+1}^{TN} and current information $\mathbf{x}_t$. The dynamic hedge ratio is similar in spirit to a regression hedge, except that the weights are no longer constant, but instead depend on current information. $w_i(t)$ puts little weight on the observation pair $(R_{t+1-i}^{mbs}, R_{t+1-i}^{TN})$ if the current information $\mathbf{x}_t$ is not close to $\mathbf{x}_{t-i}$ in a distributional sense. The hedge ratio adjusts to current economic conditions. For example, if interest rates are currently high, but the term structure is inverted, then more weight will be given to past co-movements between MBS and T-note futures in that type of interest rate environment.

Boudoukh, Richardson, Stanton, and Whitelaw (1995) apply the method to weekly 30-year fixed-rate GNMA MBS (with 8%, 9%, and 10% coupons) and T-note futures data over the period January 1987 to May 1994. The GNMA prices represent dealer-quoted prices on X% coupon-bearing GNMAs traded for forward delivery on a *to be announced* (TBA) basis.[33] Performing an out-of-sample analysis, their research shows that the dynamic hedging method performs considerably better than the static regression method. For instance, in hedging weekly returns on 10% GNMA, the dynamic method reduces the volatility of the GNMA return from 41 to 24 basis points, whereas a static method manages only 29 basis points of residual volatility. Furthermore, only 1 basis point of the volatility of the dynamically hedged return can be attributed to risk associated with U.S. Treasuries in contrast to 14 basis points of interest rate risk in the statically hedged return.

The results of Boudoukh, Richardson, Stanton, and Whitelaw (1995) shown in Table 4.26 compares the mean, volatility, and autocorrelation of unhedged returns on GNMA TBAs and hedged returns using two different approaches. The approaches involve hedging GNMAs with T-note futures, resulting in the hedged return, $R_{t+1}^{mbs} - \beta_{t+}R_{t+1}^{TN}$, where R_{t+1}^{mbs} and R_{t+1}^{TN} are the out-of-sample returns on GNMAs and T-note futures respectively, and the hedge ratio, is estimated using the prior 150 weeks of data in one of two ways: (1) a linear hedge based on a regression of past R_{t+1}^{mbs} on R_{t+1}^{TN}, and (2) a MDE hedge using the distribution of R_{t+1}^{mbs} and R_{t+1}^{TN}, conditional on the 10-year yield at time t. The estimation is performed on a rolling basis and covers the out-of-sample period, December 1989 to May 1994. Results are reported for both weekly and overlapping monthly returns.

Table 4.26

	GNMA 8		GNMA 9		GNMA 10	
	1 wk.	4 wks.	1 wk.	4 wks.	1 wk.	4 wks.
Unhedged						
Mean (%)	.078	.326	.077	.316	.069	.286
Vol. (%)	.685	1.364	.531	.999	.414	.746
Autocorr.	.011	.138	-.019	.109	-.045	.082
Linear Hedge						
Mean (%)	.007	.041	.017	.086	.024	.126
Vol. (%)	.318	.599	.309	.573	.286	.524
Autocorr.	.015	-.107	.012	.021	.060	.039
MDE Hedge						
Mean (%)	.006	.123	.020	.178	.027	.189
Vol. (%)	.285	.583	.245	.472	.242	.440
Autocorr.	.011	-.096	.016	-.083	.036	-.085

Source: Boudoukh, Richardson, Stanton, and Whitelaw (1995)

Figure 4.10 shows hedge ratios for hedging weekly 10% (top) and 8% (bottom) GNMA returns using the 10-year T-note futures. Hedge ratios are estimated on a 150-week rolling basis using both a linear regression and MDE. The MDE hedge ratios condition on the level of the 10-year T-note yield.

Figure 4.11 shows the expected weekly return on a 10% (top) and an 8% (bottom) GNMA as a function of the contemporaneous 10-year T-note futures return, conditional on three different levels of the 10-year T-note yield. The relation is estimated using MDE over the period January 1987 to May 1994. Returns are in percent per week.

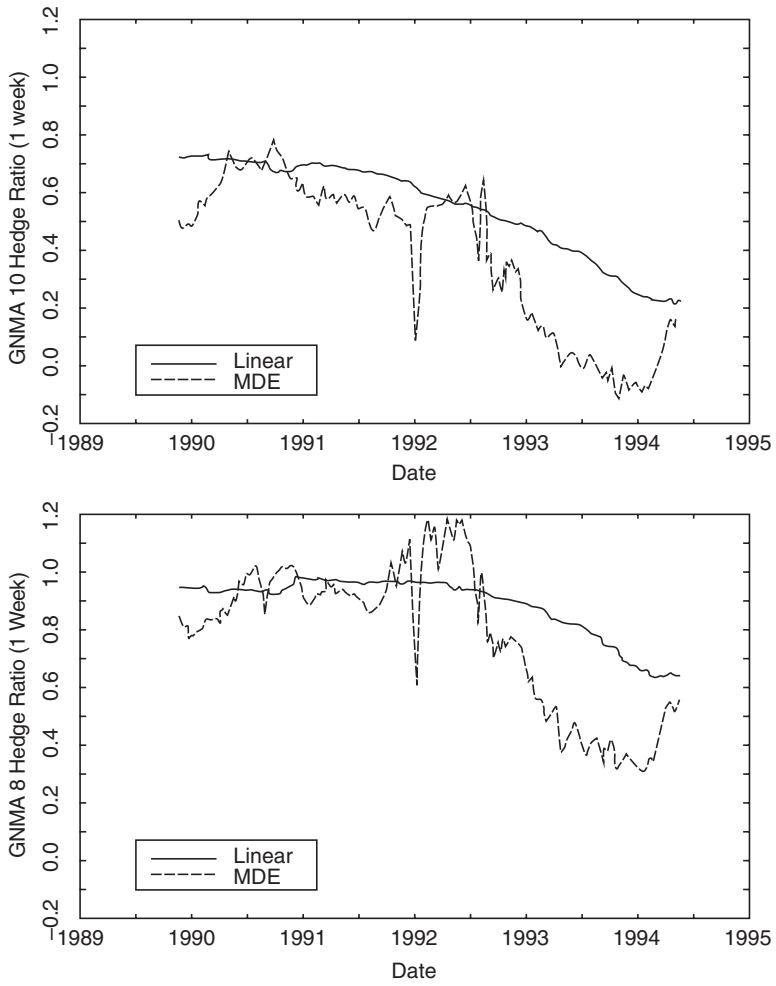


Figure 4.10 Hedge ratios for hedging weekly 10% (top) and 8% (bottom) GNMA returns using the 10-year T-note futures

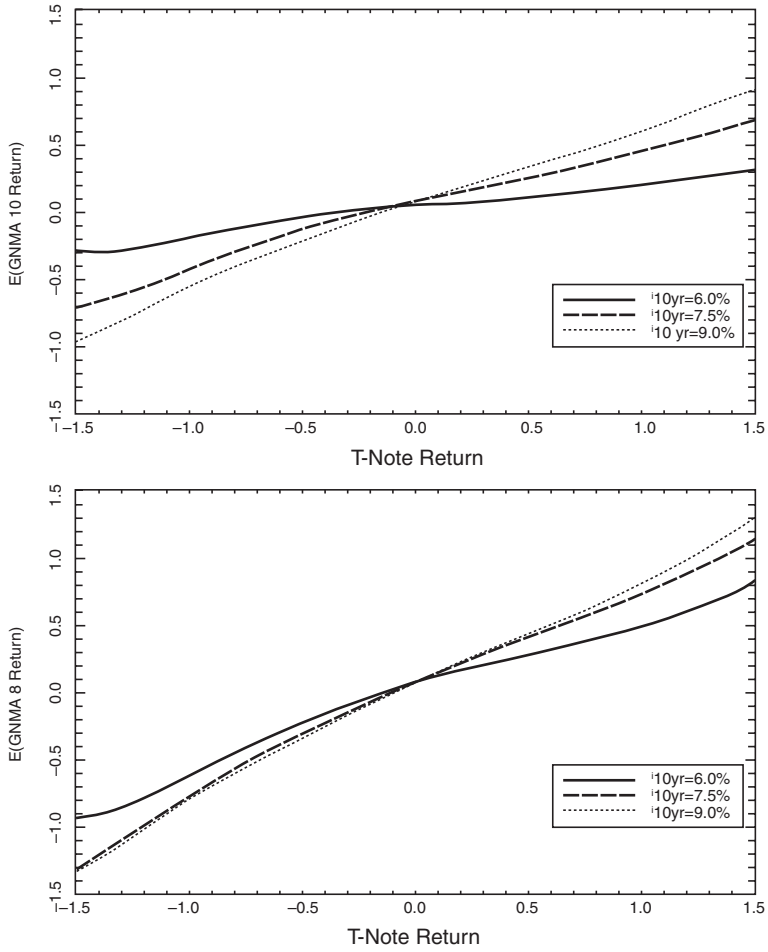


Figure 4.11 Expected weekly return on a 10% (top) and an 8% (bottom) GNMA as a function of the contemporaneous 10-year T-note futures return, conditional on three different levels of the 10-year T-note yield

ENDNOTES

1. Throughout this chapter, we follow the direct work of Dr. Stafford Johnson, Professor of Finance at Xavier University, at *http://www.academ.xu.edu/johnson/*.
2. Obazee, P. (2002), pp. 338–339. See "Understanding the Building Blocks for OAS Models" in *Interest Rate, Term Structure, and Valuation Modeling*. Edited by Frank J. Fabozzi, Wiley & Sons.
3. Id., pp. 338–339.
4. Id., pp. 338–339.
5. In practice, one would use a more sophisticated term structure model than the binomial model such as the Hull-White, Black-Derman-Toy, Black-Karasinski, or Cox-Ingersoll-Ross interest rate models.
6. We give the C++ code later in the chapter.
7. Hull, J. (1996), pg. 391.
8. Fabozzi, F., Richard, S., and Horwitz, D. (2002), pp. 445–446. "Monte Carlo Simulation/OAS Approach to Valuing Residential Real Estate-Backed Securities" in *Interest Rate, Term Structure, and Valuation Modeling*, Wiley & Sons.
9. Id., pp. 445–446.
10. Id., pg. 446.
11. Id., pg. 453.
12. Obazee, P. (2002), pp. 315–344. See "Understanding the Building Blocks for OAS Models" in *Interest Rate, Term Structure, and Valuation Modeling*. Edited by Frank J. Fabozzi, Wiley & Sons.
13. Id., pg. 317.
14. Id., pg. 318.
15. Id., pg. 319.
16. Bandic, I., pg. 11.
17. The refinancing incentive and seasoning factor were previously discussed, but are
18. Formulas from Davidson/Herskovitz (1996).
19. Monthly parameters were taken from Figure 3 in Richard Roll (1989).
20. Fabozzi, F., Richard, S., and Horwitz, D. (2002), pg. 459. "Monte Carlo Simulation/OAS Approach to Valuing Residential Real Estate-Backed Securities" in *Interest Rate, Term Structure, and Valuation Modeling*, Wiley & Sons.
21. Id., pg. 454.
22. Fabozzi, F., Richard, S., and Horwitz, D. (2002), pg. 454.
23. Id., pg. 455.
24. The one-factor model has the capacity to describe the burnout effect of prepayment by embedding heterogeneity of prepayment behavior into the MBS valuation as a function of mortgage rates. In contrast, the three-factor model is based on discrete-time, no-arbitrage pricing theory, making an association between prepayment behavior and cash flow patterns where prepayment behavior is due to refinancing (caused by changes in interest rates) and rising housing prices by incentive response functions.
25. Boudoukh, Richardson, Stanton, and Whitelaw (1995), pg. 1.
26. Id., pg. 1.
27. Id., pg. 1.

28. For a discussion of some of the problems associated with static hedges, see, for example, Breeden (1991) and Breeden and Giarla (1992). With respect to linear regression hedges in particular, Batlin (1987) discusses the effect of the prepayment option on the hedge ratio between MBSs and T-note futures.
29. Davidson and Herkowitz (1992) provide an analysis of the various theoretical methodologies for valuing MBSs in practice. The advantages and disadvantages of each approach are discussed in detail. With respect to the particular issue of hedging MBSs, Roberts (1987) gives an analysis, focusing primarily on model-based approaches to MBS valuation.
30. Id.
31. Id., pg. 2.
32. Id., pg. 3.
33. The TBA market is most commonly employed by mortgage originators who have a given set of mortgages that have not yet been pooled. However, trades can also involve existing pools, on an unspecified basis. This means that, at the time of the agreed-upon transaction, the characteristics of the mortgage pool to be delivered (e.g., the age of the pool, its prepayment history, and so on) are at the discretion of the dealer. Nonetheless, as long as new mortgages with the required coupon are being originated, these pools are likely to be delivered because seasoned pools are more valuable in the interest rate environment that characterizes a sample period. Thus, GNMA TBAs are best viewed as forward contracts on generic, newly issued, securities.

This is the first book for professionals with prebuilt, fully tested code you can use to start modeling and pricing complex derivatives. All the code in the book may be downloaded by the book's purchasers from a secure Web site, and is designed for both ease of use and ease of adaptation.

Unique Features:

- Provides ready-to-use derivatives pricing tools that cannot be found in any other book
- Includes models for the fastest-growing areas, including weather, energy, and power derivatives, CDOs, and credit derivatives
- Monte Carlo simulation, copula methodologies, and finite differences are covered in detail

The derivatives industry is growing at breakneck speed: hundreds of financial institutions now market complex derivatives; thousands of financial and technical professionals need to model them accurately and effectively. Now, for the first time, one book brings together proven, tested real-time models created for each of today's leading modeling platforms: C++, MATLAB, and Microsoft Excel. Using this book's models, professionals can save months of development time, while improving the accuracy and reliability of the models they create. The book shows how to implement pricing algorithms for a wide variety of complex derivatives, including rapidly emerging instruments covered in no other book. Utilizing actual Bloomberg data, the book covers credit derivatives, CDOs, mortgage-backed securities, asset-backed securities, fixed-income securities, and today's increasingly important weather, power, and energy derivatives. Along the way, the book presents underlying theory and math in the context of practical implementation, covering everything from Monte Carlo simulation to copula methods and finite differences.

Justin London has developed fixed-income and equity models for trading companies and his own quantitative consulting firm. He has analyzed and managed bank corporate loan portfolios using credit derivatives in the Asset Portfolio Group of a large bank in Chicago, Illinois, as well as advised several banks in their implementation of derivative trading systems. London is the founder of a global online trading and financial technology company. A graduate of the University of Michigan, London holds a BA in economics and mathematics, an MA in applied economics, and an MS in financial engineering, computer science, and mathematics, respectively.

Pub Date: November 2006 • $189.99 • Cloth • ISBN 0-13-196259-0
National Marketing Campaign
For further information contact: Amy Fandrei at
amy.fandrei@pearsoned.com • 317.428.3082

FT Press
FINANCIAL TIMES
www.ftpress.com | An imprint of Pearson