Monte Carlo Simulation With Java and C++

Michael J. Meyer

PREFACE

Classes in object oriented programming languages define abstract concepts which can be arranged in logical hierarchies using class inheritance and composition. This allows us to implement conceptual frameworks taylored specifically for the intended domain of application.

Such frameworks have been constructed for application to the most diverse areas such as graphical user interface development, database manipulation, data visualization and analysis, networking and distributed computing.

The framework provides the terms in which the problems in its domain of application should be described and analyzed and the parts from which solutions can be assembled. The crucial point is the class design, that is, the definition of concepts of the right degree of generality arranged in the proper relation to each other. If a good choice is made it will be easy and pleasant to write efficient code. To create a good design requires a thorough understanding of the application domain in addition to technical skills in object oriented programming.

In this regard developers writing code dealing with problems in some field of mathematics can draw on the enormous effort, sometimes spanning centuries, made by mathematicians to discover the right concepts and to polish them to the highest degree of efficiency and elegance. It suggests itself that one should follow the theoretical development by systematically defining the mathematical concepts as they occur in the abstract theory. Object oriented languages make such an approach easy and natural.

In this book we will undertake this program in the field of probability theory. Probability theory has an appeal far beyond mathematics and across all levels of sophistication. It is too large to admit exhaustive treatment. Consequently this effort must of necessity be partial and fragmentary. The basic concepts of *random variable*, *conditional expectation*, *stochastic process*

and *stopping time* are defined and several concrete instances of these notions implemented. These are then applied to a random assortment of problems. The main effort is devoted to the pricing and hedging of options written on liquidly traded securities.

I have credited only sources that I have actually used and apologize to all authors who feel that their contributions should have been included in the references. I believe that some of the material is new but no comprehensive survey of the existing literature has been carried out to ascertain its novelty.

Michael J. Meyer
March 30, 2002

ACKNOWLDEGEMENTS

I am grateful to P. Jaeckel for useful discussion on low discrepancy sequences and Bermudan swaptions.

Contents

List of Figures

Chapter 1

Introduction

The theory of random phenomena has always had widespread appeal not least because of its application to games of chance and speculation. The fundamental notion is that of the expected value of a bet. The Monte Carlo method computes this expected value by simulating a large number of scenarios and averaging the observed payoff of the bet over all scenarios.

The simulation of the scenarios is best accomplished by a computer program. Computational power is becoming cheap and programming languages more powerful and more elegant.

Object orientation in particular repesents a significant advance over procedural programming. The classes of an object oriented programming language allow us replicate the fundamental notions of our domain of application in the same order and with the same terminology as in the established theory. A considerable effort has already been invested to discover the fundamental concepts and the most efficient logical organization and we can take advantage the resulting efficiency.

Such an approach is taken here with regard to elementary probability theory. The basic notions of random variable, random vector, stochastic process, optional time, sampling a process at an optional time, conditioning on information are all replicated and the familiar terminology is maintained.

The reader who is not familiar with all of these notions should not be put off. Implementation on a finite and discrete computer entails a high degree of simplification. Everything can be explained again from scratch. For the most part all that is needed is a basic familiarity with the notion of a random variable, the Law of Large Numbers and conditioning of a random variable on information (sigma-fields). Martingales and the stochastic integral are used in the sections dealing with option pricing and hedging but only in

routine ways and a superficial familiarity with the subject is sufficient.

The concepts are implemented in Java and C++ with some degree of overlap but also essential differences. The Java language is chosen because of the enormous support which the Java platform has to offer for various purposes. The Java code examples in the book all refer to the library *martingale* which can be downloaded from http://martingale.berlios.de/Martingale.html. The Java code is more substantial and provides graphical interfaces and charting capabilities. The C++ code is limited to Probability, Stochastic Processes and the Libor market model. There are four different C++ implementations of a Libor market model with deterministic volatilities and constant correlations.

From the perspective of numerical computation the code is completely naive. There is no error analysis or attention paid to roundoff issues in floating point arithmetic. It is simply hoped that double precision will keep roundoff errors from exploding. In the case of C++ we can fall back on the type long double if doubts persist.

The code is also only minimally tested. It is therefore simply not suitable for commercial application. It is intended for academic experimentation and entertainment. The target audience are readers who are willing to study the source code and to change it according to their own preferences.

We will take minor liberties with the way in which the Java code is presented. To make the presentation more pleasant and the text more readable member functions of a class might be introduced and defined outside a class body. This helps us break up large classes into smaller pieces which can be discussed separately. Moreover not all member functions might be described in the text. Some classes are quite large containing multiple methods differing only in their parameter signatures but otherwise essentially similar. In this case only the simplest method is described in order to illuminate the principles.

Almost all code examples are given in Java while C++ is shown in case there is no Java equivalent (templates). In case a Java idiom is not available in C++ a possible workaround is indicated. The book can be read by a reader familiar with Java but not with C++. A reader with a background in C++ can regard the Java code snippets as detailed pseudo code and move to the C++ source code.

In any case the code examples are not given in complete detail. Instead the basic principles are illustrated. In particular in the later chapters dealing with American option pricing and the Libor market model fewer code examples are shown but detailed descriptions of the algorithms are provided. It is hoped that this will be enough to allow the reader to work through the

source code.

The code does not use complicated programming constructs and is limited to the simple use of class inheritance and composition and the most basic use of class templates. Here is a brief overview of the topics which will be covered:

1.1 Random Variables and Conditional Expectation

A random number generator provides samples from some probability distribution. A random variable X is much more than this. In addition to providing samples from its distribution we would expect it to be able to compute its mean, standard deviation, histograms and other statistics. One could also imagine operations on random variables (sums, products, etc.) although we will not implement these.

We should note that a random variable is usually observed in some context which provides increasing information about the distribution of X as time t increases. Naturally we will want to take advantage of this information by sampling from the distribution of X conditioned on all information available at time t.

The precise nature of this information and how we condition X on it depends on the particular random variable and is left to each particular implementation. In short we condense all this into the *abstract* (ie. undefined) method

```
public abstract double getValue(int t)
```

to be intepreted as the next random sample of X conditioned on information available at time t. All other methods are defined in terms of the method getValue(t). To do this we don't have to know how getValue(t) works. In order to allocate a concrete random variable we do have to know how this works however, that is, we have to define the method getValue(t). Time t is an integer as time proceeds in multiples of a smallest unit, the *time step*.

The definition of the method getValue(t) can be as simple as a call to a random number generator, a very fast operation. Frequently however X is a deterministic function of the path of some stochastic process. In this case a call X.getValue(t) involves the computation of an entire path of this process which usually is considerably slower than a call to a random number generator.

Quite often there is no information about X. In this case the parameter t is simply ignored when the method getValue(t) is defined. In case X is a deterministic function of the path of some stochastic process the information available at time t is the realization of this path up to time t.

We can easily condition on this information by simulating only paths which follow the current path up to time t and then branch off in a random manner. In other words a call X.getValue(t) computes the next sample of X from a branch of the current path of the underlying stochastic process where branching occurs at time t.

The expected value (mean) of X can be approximated by the arithmetic mean of a large number N of sample values of X. To condition this expectation on information available at time t we use only samples conditioned on this information. Thus a simple method to compute this expectation might read as follows:

```
public double conditionalExpectation(int t, int N)
{
    double sum=0;
    for(int j=0; j<n; j++) sum+=getValue(t);
    return sum/N;
}
```

There are a number of reasons why one could want to assemble separate random variables into a random vector. As noted above a call to the stochastic mechanism generating the samples of a random variable can be quite costly in terms of computational resources. Quite often several distinct random variables derive their samples (in different ways) from the same underlying stochastic mechanism.

In this case it is more efficient to introduce the random vector having these random variables as components and to compute samples for all components with a single call to the common stochastic generator.

If we want to compute correlations and covariances we must essentially proceed in this way. If the random variables are kept separate and sampled by separate calls to the underlying stochastic generator they will be found to be *independent*. In order to preserve correlation samples derived from the same call to the stochastic generator must be considered. These notions are considered in Chapter 2.

1.2 Stochastic Processes and Stopping Times

Stochastic processes model the development of some random phenomenon through time. The theory of stochastic processes is quite formidable. Fortunately, in order to be able read this book, it is not necessary to have studied this theory. Indeed, once simulation becomes the main objective and time is discretized into multiples of a smallest unit (the time step dt) we find ourselves in a greatly simplified context. In this context a stochastic process is a finite sequence

$$X = (X_0, X_1, \ldots, X_t, \ldots, X_T), \tag{1.1}$$

of random variables. Integer time t corresponds to continuous time $t * dt$, $t = 0, 1, \ldots, T$ and T is the *time horizon*. The random variable X_t is the state of the random phenomenon at time $t * dt$. The outcomes (samples) associated with the process X are *paths* of realized values

$$X_0 = x_0, X_1 = x_1, \ldots, X_T = x_T. \tag{1.2}$$

At any given time t the path of the process has been observed up to time t, that is, the values $X_0 = x_0, X_1 = x_1, \ldots, X_t = x_t$ have been observed and this is the information available at time t to contemplate the future evolution $X_{t+1} \ldots, X_T$. Conditioning on this information is accomplished by restricting sampling to paths which follow the realized path up to time t. These paths are *branches* of the curent path where branching occurs at time t.

Random variables τ associated with the process are often deterministic functions $\tau = f(X_0, X_1, \ldots, X_T)$ of the path of the process. If such a random variable τ takes integer values in [0,T] it is called a *random time*. The random time τ might be the time at which we need to react to some event associated with the process X.

At each time t we ask ourselves if it is now time to react, that is, if $\tau = t$. The information available to decide wether this is the case is the path of the process $X_0, X_1, \ldots, X_t$ up to time t. We can make the decision if the event $[\tau = t]$ (or more precisely its indicator function $1_{[\tau=t]}$) is a deterministic function of the observed values $X_0, X_1, \ldots, X_t$ but not of the future values $X_{t+1}, \ldots, X_T$, for all $t = 0, 1, \ldots, T$.

In this case the random time τ is called a *stopping time* or *optional time*. The random variable X_τ defined as $X_\tau = X_t$, whenever $\tau = t$, equivalently

$$X_\tau = \sum_{t=0}^{T} 1_{[\tau=t]} X_t \tag{1.3}$$

is called the process X *sampled at* τ . The value of this random variable is a crucial quantity as it represents the state of the random phenomenon at the time of our reaction.

The term stopping time arises from gambling where X_t is the size of the gamblers fortune after game number t and τ represents a strategy for exiting the game which relies only on information at hand and not on prevision of the future.

The notion of sampling a process at an optional time is quite fundamental and is considered in Chapter 3.

1.3 Trading, Options, Interest rates

Investing in the stock market beats gambling in casinos if for no other reason than that there is no irrefutable mathematical proof yet that an investor must necessarily lose at least on average. Casinos on the other hand are businesses for which profits are assured by solid mathematical theories.

Consider a market in which only two assets are traded, a risky asset S and the riskfree bond B. The prices $S(t)$, $B(t)$ follow some stochastic process and the quotient $S^B(t) = S(t)/B(t)$ is the price of the asset at time t discounted back to time $t = 0$. It is also the price of S expressed in a new numeraire, the riskfree bond. The unit of account of this new currency is one share of the risk free bond B. In the new numeraire the risk free rate of interest for short term borrowing appears to be zero.

Keeping score in discounted prices makes sense as it properly accounts for the time value of money and prevents the illusion of increasing nominal amounts of money while inflation is neglected.

Given that a liquidly traded asset S is on hand we might want to engage in a *trading strategy* holding $w(t)$ shares of the asset S at time t. The stochastic process $w(t)$ is called the *weight* of the asset S. Our portfolio is rebalanced (ie. the weight $w(t)$ adjusted) in response to events related to the price path of the asset S. Suppose that trades take place at an increasing family of random times

$$0 = \tau_0 < \tau_1 < \ldots < \tau_{n-1} < \tau_n = T, \tag{1.4}$$

where the number n of trades itself can be a random variable. The discounted *gains from trading* are of great interest to the trader. These gains can be viewed as the sum of gains from individual trades buying $w(\tau_k)$ share of S at time τ_k and selling them again at time τ_{k+1} (to assume the new position). Taking account of the fact that the price of the asset S has moved

from $S(\tau_k)$ at time τ_k to $S(\tau_{k+1})$ at time τ_{k+1} this transaction results in a discounted gain of

$$w(\tau_k)[S^B(\tau_{k+1}) - S^B(\tau_k)] \tag{1.5}$$

This computation assumes that the asset S pays no dividend and that trading is not subject to transaction costs. It is however quite easy to add these features and we will do it in Chapter 4. The total discounted gain from trading acording to our strategy is then given by the sum

$$G = \sum_{k=0}^{n-1} w(\tau_k)[S^B(\tau_{k+1}) - S^B(\tau_k)] \tag{1.6}$$

The term *gain* conceals the fact that G is a random variable which may not be positive. Other than trading in the asset itself we might try to profit by writing (and selling) an option on the asset S. For the premium which we receive up front we promise to make a payment to the holder (buyer) of the option. The size of this payment is random in nature and is a function of the price path $t \in [0, T] \mapsto S(t)$. The option contract specifies how the payment is calculated from the asset price path.

If we write such an option we cannot simply sit back and hope that the payoff will turn out to be less than the premium received. Instead we trade in the asset S following a suitable trading strategy (*hedge*) designed to provide a gain which offsets the loss from the short position in the option so that the risk (variance) of the combined position is reduced as much as possible. If this variance is small enough we can increase the odds of a profit by shifting the mean of the combined payoff upward. To do this we simply increase the price at which the option is sold. This assumes of course that a buyer can be found at such a price.

The hedge tries to replicate the discounted option payoff as the sum of the option premium and discounted gains from trading according to the hedging strategy. The central problem here is the computation of the weigths $w(t)$ of the underlying asset S in the hedging strategy. In Chapter 4 we will investigate several alternatives and compare the hedge performance in the case of a European call option.

One could derive the price at which an option is sold from an analysis of the hedging strategy which one intends to follow and this would seem to be a prudent approach. In this case there is the possibility that there are better hedging strategies which allow the option to be sold at a lower price. No unique option price can be arrived at in this way since every hedging strategy can be improved by lowering transaction costs. If transaction costs are zero the hedge can be improved by reacting more quickly to price changes in the underlying asset S, that is, by trading more often.

This leads to the ideal case in which there are no transaction costs and reaction to price changes is instantaneous, that is, trading is continuous. In this case and if the asset S satisfies a suitable technical condition the risk can be eliminated completely (that is the option payoff replicated perfectly) and a unique option price arrived at.

In fact this price at time t is the conditional expectation of the discounted option payoff conditioned on information available at time t and computed *not* in the probability which controls the realizations of asset price paths (the so called *market probability*) but in a related (and equivalent) probability (the so called *risk neutral probability*) .

Fortunately for many widely used asset price models it is known how to simulate paths under this risk neutral probability. The computation of the ideal option price then is a simple Monte Carlo computation as it is treated in Chapter 2. In reality an option cannot be sold at this price since trading in real financial markets incurs transaction costs and instantaneous reaction to price changes is impossible.

In case the technical condition alluded to above (market completeness) is not satisfied the pricing and hedging of options becomes considerably more complicated and will not be treated here.

Very few investors will be willing to put all their eggs into one basket and invest in one asset S only. In Chapter 5 markets with multiple assets are introduced and many of the notions from single asset markets generalized to this new setting.

In Chapter 6 we turn to the modelling of interest rates. Rates of interest vary with the credit worthiness of the borrower and the time period of the loan (possibly starting in the future). The system of $forward\ Libor$ rates is introduced and the simplest model (deterministic volatilities and constant correlations) developed. Some interest rate derivatives (swaps, caps, swaptions, Bermudan swaptions etc.) are considered also.

1.4 Low Discrepancy Sequences and Lattice Models

Chapter 7 gives a very brief introduction to the use of low disrepancy sequences in the computation of integrals and Monte Carlo expectations.

In Chapter 8 we introduce lattice models. The basic weakness of Monte Carlo path simulation is the fact the path sample which is generated by the simulation has no usefull structure. In particular paths do not split to accomodate the computation of conditional expectations. Such path split-

ting has to be imposed on the sample and leads to astronomical numbers of paths if repeated splits have to be carried out.

Lattices address this problem by introducing a very compact representation of a large path sample which is invariant under path splitting at a large number of times (each time step). This leads to straightforward recursive computation of conditional expectations at each node in the lattice. This is very useful for the approximation of American option prices. There is a drawback however. The lattice can only represent a statistical approximation of a true path sample. One hopes that the distribution represented by the lattice converges to the distribution of the underlying process as time steps converge to zero (and the number of nodes to infinity).

Chapter 2

Random Variables and Expectation

2.1 Monte Carlo expectation

Let X be an integrable random variable with (unknown) mean $E(X) = \mu$ and finite variance $Var(X) = \sigma^2$. Our problem is the computation of the mean μ.

If $X_1, X_2, \ldots$ is a sequence of independent random variables all with the same distribution as X we can think of the X_j as succesive independent observations of X or, equivalently, draws from the distribution of X. By the Law of Large Numbers

$$\frac{X_1 + X_2 + \ldots + X_n}{n} \to \mu \tag{2.1}$$

with probability one. Consequently

$$\mu(X, n) := \frac{X_1 + X_2 + \ldots + X_n}{n} \tag{2.2}$$

is an estimator of the mean μ of X. This estimator is itself a random variable with mean μ and variance $\sigma_n^2 = \sigma^2/n$. As a sum of independent random variables it is approximately normal (Central Limit Theorem). In cases of interest to us the sample size n will be at least 1000 and mostly considerably larger. Consequently we can regard $\mu(X, n)$ as a normal random variable without sacrificing much precision. Let $N(0, 1)$ denote a standard normal variable and $N(x) = Prob(N(0, 1) \leq x)$ be the standard normal cumulative distribution function. Then

$$Prob(|N(0, 1)| < \epsilon) = 2N(\epsilon) - 1 \tag{2.3}$$

and since $\sigma_n^{-1}(\mu(X,n)-\mu)$ is standard normal we have the probabilistic error estimate

$$\begin{aligned} Prob\left(|\mu(X,n)-\mu|<\epsilon\right) &= 2N(\epsilon/\sigma_n)-1 \quad (2.4)\\ &= 2N(\epsilon\sqrt{n}/\sigma)-1. \quad (2.5)\end{aligned}$$

The *Monte Carlo method* computes the mean $\mu = E(X)$ simply as the sample mean $\mu(X,n)$ where n is a suitably large integer. Equation 2.4 shows how the sample size n and variance σ^2 control the probability that the estimate $\mu(X,n)$ approximates the mean μ to the desired precision ϵ. As the sample size $n \uparrow \infty$ increases, $\sigma_n = \sigma/\sqrt{n} \downarrow 0$ decreases and the probability on the right of 2.4 increases to 1.

Like the mean μ which we are trying to compute, the variance $\sigma^2 = Var(X) = E(X^2) - \mu^2$ is usually itself unknown but it can be estimated by the sample variance

$$\sigma^2(X,n) := \frac{X_1^2 + X_2^2 + \ldots + X_n^2}{n} - \mu(X,n)^2. \quad (2.6)$$

With this we can rewrite 2.4 as

$$Prob\left(|\mu(X,N)-\mu|<\epsilon\right) = 2N\left(\epsilon\sqrt{n}/\sigma(X,n)\right)-1. \quad (2.7)$$

at least to good approximation. In this new equation the quantity on the right can be computed at each step in a simulation $X_1, \ldots, X_n, \ldots$ of X which is terminated as soon as the desired confidence level is reached.

Summary. Monte Carlo simulation to compute the mean of a random variable X generates independent draws $X_1, X_2, \ldots, X_n$ from the distribution of X (*observations* of X) until either the sample size n hits a predetermined ceiling or reaches a desired level of precision with a desired level of confidence (2.7) and then computes the mean $\mu = E(X)$ as $\mu = \mu(X,n)$.

If $X_1, X_2, \ldots, X_n$ are independent observations of X and $f = f(x)$ is any (deterministic) function, then $f(X_1), f(X_2), \ldots, f(X_n)$ are independent observations of $f(X)$ and consequently the mean $E[f(X)]$ can be approximated as

$$E[f(X)] = \frac{f(X_1) + f(X_2) + \ldots + f(X_n)}{n} \quad (2.8)$$

2.2 Monte Carlo Variance

Motivated by the formula $\sigma^2 = Var(X) = E(X^2) - \mu^2$ we have above estimated the variance by the sample variance

$$\sigma^2(X,n) := \frac{X_1^2 + X_2^2 + \ldots + X_n^2}{n} - \mu(X,n)^2. \quad (2.9)$$

However this estimator is biased in that $E\left[\sigma^2(X,n)\right] \neq Var(X)$. Observe that $E(X_iX_j) = E(X_i)E(X_j) = \mu^2$ for $i \neq j$ (by independence) while $E(X_iX_j) = E(X^2)$, for $i = j$, to obtain

$$E\left[\mu(X,n)^2\right] = \frac{1}{n}E(X^2) - \frac{n-1}{n}\mu^2$$

from which it follows that

$$E\left[\sigma^2(X,n)\right] = \frac{n-1}{n}\left(E(X^2) - \mu^2\right) = \frac{n-1}{n}Var(X).$$

This can be corrected by using $\frac{n}{n-1}\sigma^2(X,n)$ as an estimator of $Var(X)$ instead. For large sample sizes the difference is negligible.

2.3 Implementation

Let us now think about how the notion of a random variable should be implemented in an object oriented programming language such as Java. Section 1.1 was predicated on the ability to generate independent draws X_1, X_2,... from the distribution of X. On the computer a *random number generator* serves this purpose and such random number generators are available for a wide variety of distributions. Thus we could define

```
public abstract class RandomVariable {

    // a new draw from the distribution of X
    public abstract double getValue()
    {
       //call to appropriate random number generator
       return next random number;   }

}     //end RandomVariable
```

Exactly which random number generator is called depends on the distribution of the random variable X. A random variable such as this is nothing more than a random number generator by another name. Some improvements immediately suggest themselves.

Rarely is a random variable X observed in isolation. More likely X is observed in some context which provides additional information about X. As time t passes more and more information is available about the distribution of X and one will want to make use it by computing expectations conditioned on this information rather than unconditional expectations.

Assume for example that we are simulating some *stochastic process* $W(t),\ t = 0, 1, 2, \ldots, T$. The random variable X could be a *functional* (deterministic function) of the *path*

$$t \in [0, T] \mapsto W(t).$$

At any time t this path is already known up to time t, information which we surely do not want to disregard when contemplating probabilities involving the random variable X. This leads us to define:

```
public abstract class RandomVariable {

    // a new draw from the distribution of X conditioned on information
    // available at time t
    public abstract double getValue(int t)

     //other methods: conditional expectation, variance,...

}     //end RandomVariable
```

Time is measured in integer units even in the context of *continuous time*. A simulation of continuous time typically proceeds in discrete *time steps* dt and *discrete time* t (an integer) then corresponds to continuous time $t * dt$.

To obtain a concrete random variable we only have to define the method getValue(int t) which depends on the nature of the random variable X and the information structure at hand. If there is no context providing additional information about the distribution of X, the method getValue(int t) simply does not make use of the parameter t and consequently is independent of t.

Here is how we might define a standard normal random variable (mean zero, variance one) . To do this we make use of the standard normal random number generator STN() in the class Statistics.Random:

```
public class StandardNormalVariable extends RandomVariable{

    public abstract double getValue(int t){ return Random.STN(); }

} //end StandardNormalVariable
```

Once we have the ability to observe X conditioned on information $\mathcal{F}_t$ available at time t we can compute the conditional expectation $E_t[f(X)] = E[f(X)|\mathcal{F}_t]$ of any function of X via 2.8.

These expectations include the conditional expectation of X itself, variance (hence standard deviations), moments, central moments, characteristic

function and others. All of these depend on X only through the observations getValue(t) of X conditioned on information available at time t and so can already be implemented as member functions of the abstract class RandomVariable in perfect generality. Ordinary (unconditional) expectations are computed as conditional expectations at time $t = 0$. This agrees with the usual case in which the information available at time $t = 0$ is trivial.

It's now time to implement some member functions in the class RandomVariable. Conditional expectations are fundamental and so a variety of methods will be implemented. The simplest one generates a sample of values of X of size N and computes the average:

```
public double conditionalExpectation(int t, int N)
{
    double sum=0;
    for(int n=0; n<N; n++) sum+=getValue(t);
    return sum/N;
}
```

The ordinary expectation is then given as

```
public double expectation(int N)
{
        return conditionalExpectation(N);   }
```

Remark. The reader familiar with probability theory will remember that the conditional expectation $E_t(X)$ is itself a random variable and might wonder how we can compute it as a constant. In fact $E_t(X)$ is a random variable when seen from time $t = 0$ reflecting the uncertainty of the information that will have surfaced by time t. At time t however this uncertainty is resolved and $E_t(X)$ becomes a known constant.

In more precise language let $\mathcal{F}_t$ denote the σ-field representing the information available at time t, that is, the σ-field of all events of which it is known by time t wether they occur or not. Under the unconditional probability (at time $t = 0$) events in $\mathcal{F}_t$ may have any probability. Under the conditional probability at time t events in F_t have probability zero or one and so conditioning on this probability makes random variables constant.

When we call the routine X.conditionalExpectation(t) the parameter t is *current time* and so this conditional expectation returns a constant not a random variable.

The next method increases the sample size N until a certain level of precision is reached with a certain level of confidence (probability):

```
// Continues until precision or N=1,000,000 samples are reached.
public double conditionalExpectation(int t, double precision, double confidence)
{
      double sum=0,                    //X_1 + X_2 + ... + X_n
             sumSquares=0,             // X_1^2 + X_2^2 + ... + X_n^2
             mean,                     // current mean
             variance,                 // current sample variance
             f=-1000;

      int n=0;
      for(n=0; n<1000000; n++)
      {
           double x=getValue();        // x = X_n
           sum+=x;
           sumSquares+=x*x;

           // check for termination every 100 samples
           if(n%100==99)
           {
                mean=sum/n;
                variance=sumSquares/n;
                f=precision*Math.sqrt(N/variance);      // see (2.7)
                if(2*FinMath.N(f)>confidence)) break;
           } end if

       } // end n

       return sum/n;

}      // end conditionalExpectation
```

A variety of other methods suggests itself. Recall that the computation of an observation of X may not be as simple as a call to a random number generator. For example in the case of greatest interest to us, the random variable X is a functional of the path of some stochastic process. Information available at time t is the state of the path up to time t. To generate a new observation of X conditional on this information we have to continue the existing path from time t to the horizon. In other words we have to compute a new *branch* of this path, where branching occurs at time t.

Depending on the underlying stochastic process this can be computationally costly and slow. Thus the Monte Carlo computation of the mean of X can take hours. In this case we would like know in advance how long we will have to wait until the computation finishes, that is, we might want to report projected time to completion to a progress bar. This is possible for loops with a fixed number N of iterations.

For the same reason we should try to minimize calls to the stochastic mechanism which generates the observations of X. Thus it is more efficient to compute mean and standard deviation simultaneously in the same method from the same calls to the underlying mechanism rather than to compute mean and standard deviation in separate methods (doubling the number of these calls). The reader is invited to browse the javadoc documentation and to study the source code.

Groups of dependent observations. So far we have assumed that the observations of the random variable X are independent. From 2.4 we can see the influence of the standard deviation σ of X on the error estimate. Clearly the smaller σ the better. In an attempt to reduce the variance occasionally one does not generate pairwise independent observations of X. Instead one generates observations of X in equally sized groups where observations are independent accross distinct groups but observations within the same group are not independent.

In this case the observations of X have to be replaced with their group means (averages). These group means are again independent and have the same mean (but not the same distribution) as X. To compute the mean of X we now compute the mean of the group means.

It is hoped that the group means have significantly lower variance than the observations of X themselves even if we take into account that their number is far smaller. Obviously we must make sure that the number of sample groups is still large and sample group means identically distributed (which they are if all sample groups are generated by succesive calls to the same stochastic mechanism).

Since the arithmetic average of the group averages is equal to the arithmetic average of the observations themselves this modus operandi necessitates no adjustment as long as only means are computed. If standard deviations are computed however the group means must be introduced explicitly and their standard deviation computed. The standard deviation of the observations themselves has no significance at all and is related to the standard deviation of X in ways which depend on the nature of the dependence between observations in the same group and which is generally unknown.

For a concrete example of this procedure consider the case where X is a functional of the path of some stochastic process. The method of *antithetic paths* generates paths in groups of two paths which are mirror images of each other in an appropriate sense. Accordingly observations of X come in pairs which are averaged. We will go into more detail in the study of securities and options where this and more general methods will be employed.

Example 1. *The Hello World program of Monte Carlo simulation.* We create a random variable X of the type $X = Z^2$, where Z is standard normal. Then $E(X) = 1$ and we will check how close Monte Carlo simulation of 100,000 samples gets us to this value.

The most interesting part of this is the creation of an object of the abstract class RandomVariable. The definition of the abstract method getValue(int t) is provided right where the constructor RandomVariable() is called:

```
public class MonteCarloTest{

    public static void main(String[] args)
    {
        RandomVariable X=new RandomVariable(){

            //the definition of getValue missing in RandomVariable.
            public double getValue(int t)
            {
                double x=Random.STN();   //standard normal random number
                return x*x;      }
        }; //end X

        double mean=X.expectation(100000);
        System.out.println("True mean = 1, Monte Carlo mean = "+mean)

    } //end main

} //end MonteCarloTest
```

2.4 Empirical Distribution

Interesting random variables can be constructed from raw sets of data. Assume we have a set of values $\{ x_1, x_2, \ldots, x_N \}$ (the data) which are independent observations of a random variable X of unknown distribution. The *empirical distribution* associated with these data is the probability concentrated on the data points in which all the data points are equally likely, that is, the convex combination

$$Q = \frac{1}{N} \sum_{j=1}^{N} \delta(x_j), \tag{2.10}$$

where $\delta(x_j)$ is the point mass concentrated at the point x_j. In the absence of other information this is the best guess about the distribution of X.

You may be familiar with the notion of a *histogram* compiled from a raw set of data and covering the data range with disjoint intervals (bins).

This simplifies the empirical distribution in that data values in the same bin are no longer distinguished and a simplified distribution is arrived at where the possible "values" are the bins themselves (or associated numbers such as the interval midpoints) and the bin counts are the relative weights. In other words you can think of the empirical distribution as the "histogram" of the data set with the finest possible granularity.

The *empirical random variable* X associated with the data set is distributed according to this empirical distribution. In other words, drawing samples from X amounts to sampling the data with replacement. This random variable is already interesting. Note that there is no information to condition on and consequently the time parameter t below is ignored;

```
public class EmpiricalRandomVariable{

    int sampleSize;                 // size of the data set
    public int get_sampleSize(){ return sampleSize; }

    double[] data_set;              // the data set
    public double[] get_data_set(){ return data_set; }

    // Constructor
    public EmpiricalRandomVariable(double[] data_set)
    {
        this.data_set=data_set;
        sampleSize=data_set.length;
    }

    // the next sample of X, parameter t ingnored
    public double getValue(int t)
    {
        int k=Random.uniform_1.nextIntFromTo(0,sampleSize-1);
        return data_set[k];
    }

} // end EmpiricalRandomvariable
```

Here Random.uniform_1.nextIntFromTo(0,sampleSize - 1) delivers a uniformly distributed integer from the interval [0,sampleSize-1]. You would use an empirical random variable if all you have is a raw source of data. When dealing with an empirical random variable X we have to destinguish between the size of the underlying data set and the sizes of samples of X. Once we have the random variable X we can compute sample sets of arbitrary size. If this size exceeds the size of the underlying data set all that happens is that values are drawn repeatedly. However such random oversampling has benefits for

the resulting empirical distribution as the pictures of histograms computed in the following example show:

Example 2. We construct a data set of size N=2000 set by drawing from a standard normal distribution, then allocate the empirical random variable X associated with the sample set and display histograms of sample sizes N and 50N. Both pure and smoothed histograms are displayed. The smoothing procedure repeatedly averages neighboring bin heights (see Statistics.BasicHistogram#smoothBinHeights().

```
public class EmpiricalHistogram{

    public static void main(String[] args)
    {
        int N=2000;          // size of data set
        int nBins=100;       // number of histogram bins

        // construct a raw standard normal sample set
        double[] data_set=new double[N];
        for(int i=0; i<N; i++) data_set[i]=Random.STN();

        // allocate the empirical random variable associatd with this sample set
        RandomVariable X=new EmpiricalRandomVariable(data_set);

        // a histogram of N samples
        System.out.println("displaying histogram of N samples");
        X.displayHistogram(N,nBins);

        Thread.currentThread.sleep(3000);

        // a histogram of 50N samples
        System.out.println("displaying histogram of 50N samples");
        X.displayHistogram(50*N,nBins);

    } // end main

} // end EmpiricalHistogram
```

The reader should note that a sample of size N of X represents N random draws from the underlying data set and will thus not reproduce the data set exactly even if it is of the same size as the data set. With this caveat the improvement of the shape of the histogram from increasing the sample size (by resampling) is nonetheless surprising.

Extreme values at both tails of the distribution have been lumped together

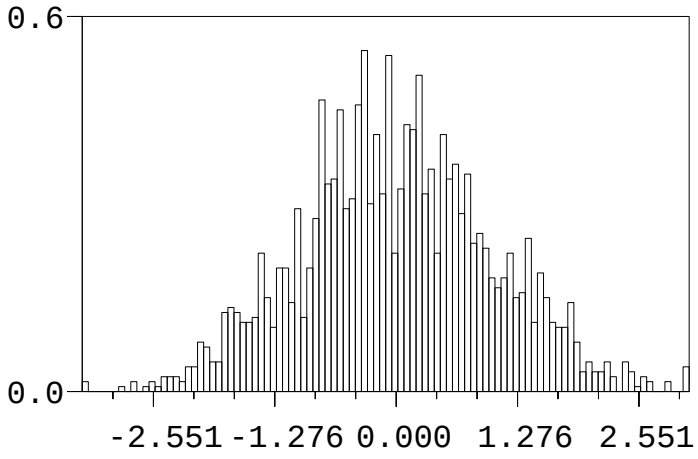


Figure 2.1: Histogram, $N = 2000$.

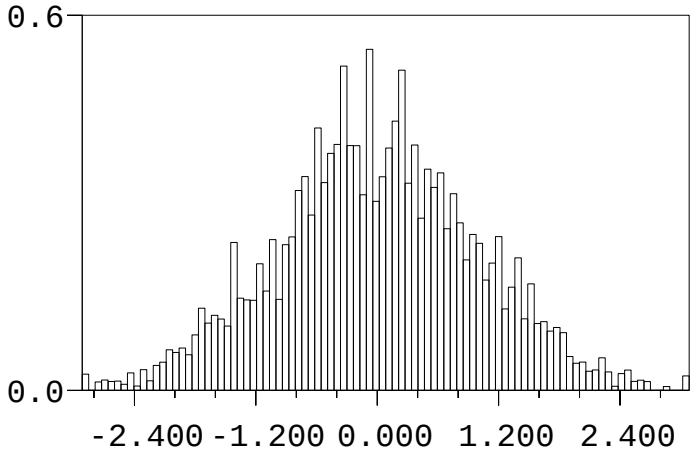


Figure 2.2: Histogram, $N = 100000$.

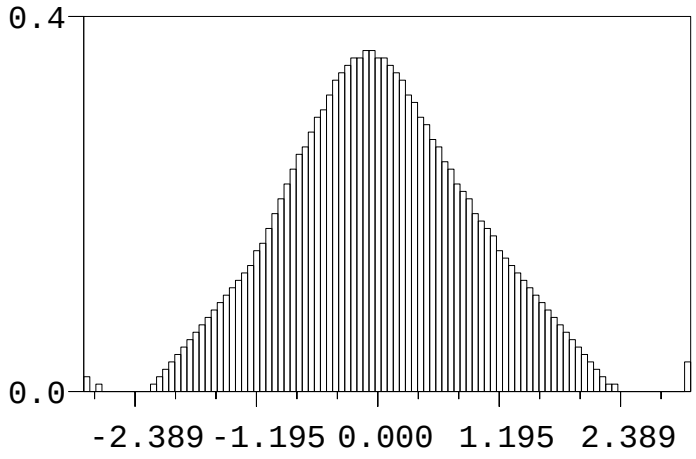


Figure 2.3: Smoothed histogram, $N = 2000$.

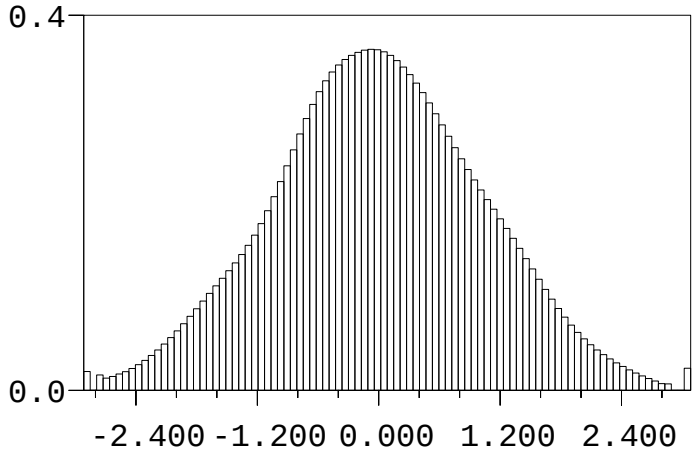


Figure 2.4: Smoothed histogram, $N = 100000$.

into the outermost displayed bins to keep the range of displayed values reasonable.

If the data set is small as in our case the results are highly sensitive to the number of bins used in the histogram. Increase the number of bins to 200 and see how the shape of the histogram deteriorates. 2000 data points are not enough to increase the resolution of the data range to 200 bins.

2.5 Random Vectors and Covariance

The class RandomVariable implements methods to compute the variance of a random variable X but no methods are included to compute the *covariance* or *correlation* of two random variables X, Y (for example in the form X.covariance(Y)).

The reason for this is simple: in mathematics computation of the covariance Cov(X,Y) assumes that the random variables X, Y are defined on the same underlying probability space $(\Omega, \mathcal{F}, P)$ and one then defines

$$Cov(X,Y) = \int_\Omega X(\omega)Y(\omega)P(d\omega) - E(X)E(Y). \tag{2.11}$$

Note how in this integral both X and Y are ervaluated at the same state of nature ω. Correspondingly, when simulated on a computer, the observations of X and Y must be derived from the same state of the underlying stochastic generator. This is very hard to ensure when the random variables X and Y are considered as separate entities.

Naively producing observations of X and Y by repeated calls to their respective stochastic generators will result in $Cov(X,Y) = 0$, even if $X = Y$, since repeated calls to one and the same stochastic generator usually produce independent results.

Thus, to preserve correlation, the random variables X and Y have to be more tightly linked. One solution is to combine X and Y into a *random vector* and to ensure that observations of the components of this random vector are all computed from the same state of the underlying stochastic generator.

In other words, covariance and correlation are best implemented as covariance and correlation of the components of a random vector. This motivates the introduction of the class

```
public abstract class RandomVector {

    int dim;        //dimension
    double[] x;     //the current sample of X (a vector)
```

```
    //constructor
    public RandomVector(int d)
    {
        dim=d;
        double[] x=new double[dim];      }

  // the next observation of X
  public abstract double[] getValue(int t);

  //other methods

}     //end RandomVector
```

A concrete subclass might now define

```
public double[] getValue(int t)
{
    // call to stochastic generator G at time t, then

    for(int j=0; j<dim; j++)
    { x[j] is computed from the same state of G for all j; }

    return x;

}     //end getValue
```

Exactly what the stochastic generator G delivers at time t depends on the context and in particular on the evolution of available information with time.

Calls to G can be quite expensive in computational terms. Consider the case in which the components X are all functionals of the path of a stochastic process. The information at time t is the realization of a path up to time t. A new state of G conditioned on information at time t is a new branch of this path, where branching occurs at time t. In other words a call to G computes this path forward from time t to the horizon.

Besides ensuring that the correlation between components is preserved this setup has an advantage over loose collections of random variables even if we are not interested in correlations: it is more economical to derive new observations of the components of X from a single call to the underlying stochastic generator G rather than considering these components in isolation and making a new call to G for each component.

It would be a bad idea to allocate the array x containing the new observation of X in the body of getValue via

```
double x=new double[dim]
...
return x;
```

since this would fill up memory with observations of X in short order. Instead we write the sample to the class member x[]. The class RandomVector is quite similar to the class RandomVariable if it is noted that expectations and standard deviations are computed component by component. Such operations on vectors are implemented in the class Statistics.Vector.java, where vectors are simply viewed as arrays of doubles. The reader is invited to browse the javadoc and read the source code.

2.6 Monte Carlo estimation of the covariance

Let (X_j, Y_j) be a sequence of independent random vectors all with the same distribution as the random vector (X, Y) (a sequence of independent observations of (X, Y)). The formula

$$Cov(X, Y) = E(XY) - E(X)E(Y)$$

leads us to estimate the covariance $Cov(X, Y)$ by the sample covariance

$$\begin{aligned} Cov(X, Y, n) &= \frac{X_1Y_1 + X_2Y_2 + \ldots + X_nY_n}{n} - \mu(X, n)\mu(Y, n), \quad \text{where} \\ \mu(X, n) &= \frac{X_1 + X_2 + \ldots + X_n}{n}. \end{aligned}$$

This neglects the fact that $\mu(X, n)$ is only an estimate for the expectation $E(X)$. In consequence the estimator $Cov(X, Y, n)$ is biased, ie. it does not satisfy $E(Cov(X, Y, n)) = Cov(X, Y)$. Indeed, observing that $E(X_iY_j) = E(X_i)E(Y_j) = E(X)E(Y)$ for $i \neq j$ (by independence) while $E(X_iY_i) = E(XY)$, for $i = j$, we obtain

$$E\left[\mu(X, n)\mu(Y, n)\right] = \frac{1}{n}E(XY) + \frac{n-1}{n}E(X)E(Y)$$

from which it follows that

$$E\left[Cov(X, Y, n)\right] = \frac{n-1}{n}\left(E(XY) - E(X)E(Y)\right) = \frac{n-1}{n}Cov(X, Y).$$

This can be corrected by using $\frac{n}{n-1}Cov(X, Y, n)$ as an estimator of $Cov(X, Y)$ instead. For large sample sizes the difference is negligible. In pseudo code

mean_X=$(X_1 + X_2 + \ldots + X_n)/n$;
mean_Y=$(Y_1 + Y_2 + \ldots + Y_n)/n$;
mean_XY=$(X_1Y_1 + X_2Y_2 + \ldots + X_nY_n)/n$;
Cov(X,Y)=mean_XY−mean_X ∗ mean_Y;

2.7 C++ implementation.

Note how Java has forced us to treat the cases of random variables and random vectors separately even though these concepts are very similar and differ only in the *type* of the observed values. In the case of a random variable these values are numbers, in the case of a random vector they are vectors.

It would be desirable to treat these two as special cases of a "random variable" with values of an arbitrary type. Such a random quantity is called a *random object* in probability theory.

C++ templates answer this very purpose. A class template can depend on a number of template parameters which are types. To obtain a class from the template these parameters are then specified at compile time (possibly in the form of type definitions).

With this we can write much more concise code while simultaneously increasing its scope. We can allow a random variable X to take values of arbitrary type RangeType. This type merely needs to support the operators +=(RangeType&) and /=(int) needed to perform the calculations in computing an average. This allows for complex valued random variables, vector valued random variables, matrix valued random variables etc.

In the case of a random vector we want to compute covariance matrices. This assumes that RangeType is a vector type and supports the subscripting operator ScalarType operator[](int i) which computes the components of a RangeType object. Here ScalarType is the type of these components.

We can choose the basic number type Real (typedef long double Real;) as the default for both RangeType and ScalarType and define

```
template⟨typename RangeType=Real, typename ScalarType=Real⟩
class RandomObject {

    // a new draw from the distribution of X
    virtual Real nextValue() = 0;

    //other methods: expectation, variance, covariance matrix...

}; //end RandomObject
```

One of the advantages of C++ class templates is the fact that a template class method is not instantiated unless it is actually used. This means that the class template RandomObject can be coded assuming all manner of features for the types RangeType, ScalarType such as operator *=(RangeType&) needed for variances, operator *=(ScalarType&) needed for covariances and a subscripting operator

```
ScalarType operator[](int i)
```

on RangeType needed for the covariance matrix and you can instantiate the template with types that do not support these features. No error will result unless you call methods which actually need them.

The type ScalarType comes into play only as the range type of the subscripting operator in case RangeType is a vector type. No such subscripting operator needs to be defined and ScalarType can become the default in case no covariances or covariance matrix are computed. With this we can define all sorts of random objects

```
typedef Complex std::complex;
typedef Vector std::vector;

// Real valued random variables (RangeType, ScalarType are the default Real)
typedef RandomObject<> RandomVariable;

// Complex valued random variables (ScalarType plays no role)
typedef RandomObject< Complex > ComplexRandomVariable;

// Real random vector (ScalarType is the default Real)
typedef RandomObject< Vector< Real > > RandomVector;

// Complex random vector
typedef RandomObject< Vector< Complex >, Complex >
ComplexRandomVector;
```

Note the absence of time t as a parameter to nextValue(), that is, there is no notion of time and information and the ability to condition on information available at time t is no longer built in. This reduces the number of class methods. It is still possible to condition on information in a suitable context by defining the method nextValue() so as to deliver observations already conditioned on that information. This is only a minor inconvenience and the C++ class PathFunctional below implements conditioning in a fairly general context.

2.8 Control Variates

From the probabilistic error estimate 2.4 it can be seen that there are two ways in which to decrease the probabilistic error: increase the sample size n or decrease the standard deviation σ of X.

The sample size n is always chosen as large as the computational budget will allow so that *variance reduction* is usually the only remaining option. Of course the random variable X has a certain variance which we cannot alter to suit our purpose. We can however replace the random variable X with any other random variable $\tilde{X}$ which is known to satisfy $E(\tilde{X}) = E(X)$ and then compute the expectation of $\tilde{X}$ instead.

Means of groups of dependent observations of X is one possible replacment for X mentioned above. In this case nothing is usually known about the degree of variance reduction. On the other hand any random variable $\tilde{X}$ satisfying $E(\tilde{X}) = E(X)$ has the form

$$\tilde{X} = X + \beta(E(Y) - Y), \tag{2.12}$$

where Y is some random variable and β a constant. In fact we can choose $\beta = 1$ and $Y = X - \tilde{X}$ in which case $E(Y) = 0$. Conversely any random variable $\tilde{X}$ of the form 2.12 satisfies $E(\tilde{X}) = E(X)$ and so can serve as a proxy for X when computing the mean of X. We can easily compute the variance of $\tilde{X}$ as

$$Var(\tilde{X}) = Var(X - \beta Y) = Var(X) - 2\beta Cov(X, Y) + \beta^2 Var(Y). \tag{2.13}$$

As a function of β this is a quadratic polynomial assuming its minimum at

$$\beta = Cov(X, Y)/Var(Y), \tag{2.14}$$

the so called *beta coefficient* (corresponding to X and Y). Note that β can be rewritten as

$$\beta = \rho(X, Y)\sigma(X)/\sigma(Y), \tag{2.15}$$

where $\sigma(X)$, $\sigma(Y)$ denote the standard deviations of X, Y respectively and $\rho(X, Y) = Cov(X, Y)/\sigma(X)\sigma(Y)$ denotes the correlation of X and Y. Entering this into 2.13 computes the variance of $\tilde{X}$ as

$$Var(\tilde{X}) = (1 - \rho^2(X, Y))Var(X) \leq Var(X). \tag{2.16}$$

Replacing X with $\tilde{X}$ defined by 2.13, 2.14 reduces the variance and the variance reduction is the greater the more closely X and Y are correlated

(or anticorrelated). The random variable Y is called a *control variate* for X.

To see how the control variate Y works to reduce the variance in X assume first that Y is highly correlated with X ($\rho(X,Y) \simeq 1$). Then $\beta > 0$ and if X overshoots its mean then so will Y and the term $\beta(E(Y) - Y)$ in (2.12) is negative and corrects the error in X. The same is true if X samples below its mean.

If X and Y are highly anticorrelated ($\rho(X,Y) \simeq -1$), then $\beta < 0$ and if X overshoots its mean Y is likely to sample below its mean and the term $\beta(E(Y) - Y)$ is again negative and corrects the error in X. The same is true if X samples below its mean.

Note carefully that we must have the mean $E(Y)$ of the control variate to be able to define $\tilde{X}$. Consequently a random variable Y is useful as a control variate for X only if it is

- highly correlated (or anticorrelated) with X and
- the mean $E(Y)$ is known or more easily computed than $E(X)$.

As we have seen in section 1.3 care must be taken to preserve the correlation of X and Y. Given a sample of X the control variate must deliver a sample of Y corresponding to this selfsame sample of X ie. produced by the same call to the underlying stochastic generator.

A practical way to ensure this is to combine the random variable X and its control variate into one entity similar to a two dimensional random vector. However we cannot simply extend the class RandomVector since expectations of random vectors are computed component by component while a control variate for X is not used by computing its expectation separately and combining it with the mean of X into a vector. This leads to:

```
public abstract class ControlledRandomVariable{

    final int nBeta=2000; // number of samples used for the beta coefficient

    public abstract double[] getValue(int t);
    public abstract double getControlVariateMean(int t);

    //other methods

}     //end ControlledRandomVariable
```

Here double[ ] is a double[2] with getValue(t)[0] being the next observation of X and getValue(t)[1] the *corresponding* observation of the control variate Y conditioned on information available at time t. In other words a concrete subclass might define:

```
public double[] getValue(int t)
{
    // call the stochastic generator G at time t
    double x=...,   //observation of X computed from the new state of G
           y=...;   //observation of Y computed from the same state of G
    return new double[] {x,y};
}
```

It is best to avoid the creation of the new doubles[] by writing to preallocated workspace. The "other methods" will primarily be composed of methods computing the expectation of X using the control variate Y but in order to do so we must first compute the beta coefficient $\beta = Cov(X,Y)/Var(Y)$ from (2.14). Here it is conditioned on information available at time t and computed from a sample of size N:

```
public double betaCoefficient(int t, int N)
{
    double sum_X=0, sum_Y=0, sum_XX=0, sum_XY=0;
    for(int n=0; n<N; n++)
    {
        double[] d=getControlledValue(t); double x=d[0], y=d[1];
        sum_X+=x; sum_Y+=y; sum_XX+=x*x; sum_XY+=x*y;      }

    return (N*sum_XY - sum_X*sum_Y)/(N*sum_XX - sum_X*sum_X);

} //end betaCoefficient
```

The unconditional version calls the conditional version at time $t = 0$ and is distinguished from the conditional version by the parameter signature.

Recall how the control variate is used in the computation of the mean of X: conditioning on all information available at time we replace the random variable X with the random variable $X_c = X + \beta(t)(E_t(Y) - Y)$ which has the same conditional mean as X but lower variance. Here Y is the control variate and $\beta(t) = Cov_t(X, Y)/Var_t(X)$ the beta coefficient (both conditioned on all information available at time t).

Thus it is useful to have a member function controlled_X which allocates this random variable. We let current time t be a parameter of controlled_X. Thus the unconditional mean of controlled_X corresponds to the conditional mean of X:

```
public RandomVariable controlled_X(int t)
{
    final double beta=betaCoefficient(t,nBeta),          // beta coefficient
                 mean_y=getControlVariateMean(t);        // control variate mean

      // allocate and return the random variable X_c = X + β(t)(E_t(Y) − Y)
    return new RandomVariable(){
    {
        public double getValue(int t)
        {
            double[] value_control_variate_pair=getControlledValue(t);

            double x=value_control_variate_pair[0],    // sample of X
                   y=value_control_variate_pair[1];    // control variate

            return x+beta * (mean_y − y);

        } // end getValue

      }; // end return new

} // end controlled_X
```

With this the conditional expectation at time t computed from a sample of size N becomes:

```
public double conditionalExpectation(int t, int N)
{
    return controlled_X(t).expectation(N);
}
```

and the ordinary expectation calls this at time $t = 0$:

```
public double expectation(int N){ return conditionalExpectation(0,N); }
```

All other methods to compute the expectation of X using its control variate follow a similar pattern. Recall that the measure of the quality of a control variate for X is the degree of correlation with X. Since in general we will have to experiment to find a suitable control variate it is useful to implement a method computing this correlation. All that is necessary is to allocate X and its control variate Y as a two dimensional random vector. We can then draw on methods from RandomVector to compute the correlation between the components. Note how the abstract class RandomVector is instantiated on the fly:

```
public double correlationWithControlVariate(int t, int N)
{
    // X and its control variate as a 2 dimensional random vector
    RandomVector X=new RandomVector(2){

       public double[] getValue(int t){ return getControlledValue(t); }

    }; // end X

    return X.conditionalCorrelation(0,1,t,N);

} // end correlationWithControlVariate
```

As usual the unconditional version calls this at time $t = 0$. Random variables will become more interesting in the context of stochastic processes. These will be introduced in the next chapter. At present our examples are admittedly somewhat contrived:

Example 2. We allocate a random variable X of the form $X = U + \epsilon N$, where U is uniform on $[0, 1]$, N standard normal and independent of U and $\epsilon = 0.1$. Clearly then $E(X) = 0.5$. As a control variate for X we use $Y = U - \epsilon N$. The correlation of X and Y is then given by $\rho = \rho(X, Y) = (1-12\epsilon^2)/(1+12\epsilon^2) = 0.78571$. Consequently $Var(\tilde{X}) = (1-\rho^2)Var(X) = 0.3826 * Var(X)$, a 72% reduction in variance.

We compute the correlation of X with its control variate and the mean of X, both with and without using the control variate, from a sample of 100 observations of X. The sample size is chosen so small to allow the control variate to show greater effect.

The uniform and standard normal random numbers are delivered by the static methods U1() and STN() of the class Statistics.Random:

```
public class CVTest1{

    public static void main(String[] args)
    {
        int nSamples=100;    // sample size

        // X without control variate
        RandomVariable X=new RandomVariable(){

            public double getValue(int t)
            {
                double U=Random.U1(),    // uniform in [0,1]
                       N=Random.STN(),   // standard normal
                       e=0.1;
                       return U+e*N;
            }// end getValue

        }; //end X


         //Monte Carlo mean without control variate
         double mean=X.expectation(nSamples);
         //messages to report the findings...

         // X with control variate
         ControlledRandomVariable Xc=new ControlledRandomVariable(){

             public double[] getControlledValue(int t)
             {
                   double U=Random.U1(), // uniform in [0,1]
                          N=Random.STN(),        // standard normal
                          e=0.1,
                          x=U+e*N,               // X
                          y=U-e*N;               // control variate Y

                   double[] result={x,y};
                   return result;

               } // end getControlledValue

               public double getControlVariateMean(int t){return 0.5;}

         }; //end Xc

         //correlation of X with control variate
         double rho=Xc.correlationWithControlVariate(20000);

         //Monte Carlo mean with control variate
         mean=Xc.expectation(nSamples);
```

```
        //messages to report the findings...

    } // end main

} // end CVTest1
```

If you run this code (Examples.ControlVariate.ControlVariateTest_1.java) you should get 0.5481 for the uncontrolled mean and 0.5051 for the controlled mean. The correlation of X with the control variate is computed as 0.7833 (from 20000 samples). This assumes that you have not tinkered with the random number generators in the class Statistics.Random. The results depend on the state of the random number generator.

If you increase the sample size you can find cases where the uncontrolled mean is closer to the true mean than the controlled mean. There is nothing wrong with that. The results are after all "random". The probabilistic error bound 2.4 provides confidence intervals but we have no information exactly where the results will actually fall. They may not even be in the confidence interval.

Exercise. Increase the sample size to 10000 and embed this computation in a loop over 50 iterations to see how often the controlled mean beats the uncontrolled mean. You might want to remove the computation of the correlation of X with its control variate.

2.8.1 C++ Implementation.

Some differences between Java and C++ now come to the surface. In C++ we can use private inheritance to implement a ControlledRandomVariable as a two dimensional random vector and this approach will not be apparent to users of the class. On the other hand we lose some handy Java idioms. For example the instantiation of an abstract class by defining the abstract methods in the body of the constructor call:

```
public abstract class AbstractClass {

    public abstract void doSomething();

};
```

```
// some method somewhere
public AbstractClass foo() {

    return new AbstractClass(args) {

        //define doSomething here
        public void doSomething(){ /* definition */ }

    }; // end return new

} // end foo
```

no longer works and we have to officially define the type TheConcreteClass which the method foo intends to return by derivation from AbstractClass. Moreover quite likely foo will be a method in a class OuterClass and we could want to define TheConcreteClass as an inner class of OuterClass:

```
class OuterClass {

    class TheConcreteClass : public AbstractClass
    { /* definition */ };

    AbstractClass* foo(){ return new TheConcreteClass(args); }

};
```

In Java TheConcreteClass is aware of its enclosing class OuterClass and can make use of class methods and fields. Not so in C++. We have to hand a pointer to OuterClass to the constructor of the TheConcreteClass and call this constructor with the this pointer of OuterClass. This is more akward than the Java solution but other features of C++ more than make up for these shortcomings.

Chapter 3

Stochastic Processes and Stopping Times

3.1 Processes and Information

A random phenomenon is observed through time and X_t is the state of the phenomenon at time t. On a computer time passes in discrete units (the size of the time step). This makes time t an integer variable and leads to a sequential stochastic process

$$X = (X_0, X_1, \ldots, X_t, \ldots, X_T), \tag{3.1}$$

where T is the *horizon*. The outcomes (samples) associated with the process X are *paths* of realized values

$$X_0 = x_0, X_1 = x_1, \ldots, X_T = x_T, \tag{3.2}$$

and it is not inappropriate to view the process X as a path valued random variable. The generation of sample paths consists of the computation of a path starting from x_0 at time $t = 0$ to the horizon $t = T$ according to the laws followed by the process.

The most elementary step in this procedure is the single *time step* which computes X_{t+1} from the values X_u, $u \leq t$. The computation of an entire path is then merely a sequence of time steps.

At any given time t the path of the process has been observed up to time t, that is, the values

$$X_0 = x_0, X_1 = x_1, \ldots, X_t = x_t, \tag{3.3}$$

have been observed and this is the information available at time t to contemplate the future evolution $X_{t+1} \ldots, X_T$. Conditioning on this information is accomplished by restricting ourselves to paths which follow the realized path up to time t. These paths are *branches* of the realized path where branching occurs at time t.

This suggests that the basic path computation is the computation of branches of an existing path, that is, the continuation of this path from the time t of branching to the horizon. A full path is then simply a path branch at time zero:

```
public abstract class StochasticProcess{

    int T;                  // time steps to horizon
    double dt;              // size of time step
    double x0;              // initial value
    double[] path;          // path array

    //constructor
    public StochasticProcess(int T, double dt, double x0)
    {
      this.T=T;
      this.dt=dt;
      this.x0=x0;
      path=new double[T+1];     // allocate the path array
      path[0]=x0;               // intialize

    } // end constructor

    // Compute path[t+1] from path[u], u≤t.
    public abstract void timeStep(int t);

    public void newPathBranch(int t){ for(int u=t;u<T;u++)timeStep(u); }
    public void newPathBranch(){ newPathBranch(0); }

    // other methods

} // end StochasticProcess
```

Clearly the repeated calls to timeStep in newPathBranch introduce some computational overhead which one may want to eliminate for fastest possible performance. Note that the above methods provide default implementations which you are free to override in any subclass with your own more efficient methods if desired.

Exactly how the time step is accomplished depends of course on the nature of the process and is defined accordingly in every concrete subclass. The

method timeStep is the only abstract method of the class StochasticProcess. All other methods can be implemented in terms of timeStep.

This means that a concrete stochastic process can be defined simply by defining the method timeStep and all the methods of StochasticProcess are immediately available. Here is how we could allocate a constant process starting at $x_0 = 5$ with T=1000 steps to the horizon and time step $dt = 1$ without officially introducing a new class extending StochasticProcess. The method timeStep is defined in the body of the constructor call:

```
int T=1000,
double dt=1,
       x0=5;

StochasticProcess constantProcess=new StochasticProcess(T,dt,x0){

    //define the timeStep
    public void timeStep(int t){ path[t+1]=path[t]; }

} // end constantProcess
```

3.2 Path functionals

A stochastic process $X = X(t)$ provides a context of time and information such that conditioning on this information has a natural interpretation and is easily implemented on a computer.

The random variables which can be conditioned are the *functionals* (deterministic functions) of the path of the process, that is, random variables H of the form

$$H = f(X) = f(X_0, X_1, \ldots, X_T), \tag{3.4}$$

where f is some (deterministic) function on R^T. With the above interpretation of information and conditioning we would define a path functional as follows:

```
public abstract class PathFunctional extends RandomVariable{

   StochasticProcess underlyingProcess;

   // constructor
   public PathFunctional(StochasticProcess underlyingProcess)
   {
        this.underlyingProcess=underlyingProcess;
   }
```

```
    // The value of the functional (this) computed from the
    // current path of the underlying process

    public abstract double valueAlongCurrentPath();

    // New sample of H (this) conditioned on information
    // available at time t.
    public double getValue(int t)
    {
        // branch the current path (the information) at time t
        underlyingProcess.newPathBranch(t);
        return valueAlongCurrentPath();
    }

} // end PathFunctional
```

A particular such functional is the maximum $X_T^* = \max_{t \leq T} X(t)$ of a process along its path which we might implement as follows:

```
public class MaximumPathFunctional extends PathFunctional{

    // constructor
    public MaximumPathFunctional(StochasticProcess underlyingProcess)
    { super(underlyingProcess); }

    // Computing the value from the current path
    public double valueAlongCurrentPath()
    {
        double[] path=underlyingProcess.get_path();        // the path
        double currentMax=underlyingProcess.get_X_0();     // starting value
        int T=underlyingProcess.get_T();                   // time steps to horizon

        for(int s=0; s<=T; s++)
        if(path[s]>currentMax)currentMax=path[s];
        return currentMax;
    }

} // end MaximumPathFunctional
```

With this we can now easily compute a histogram of the maximum of a standard Brownian motion starting at zero:

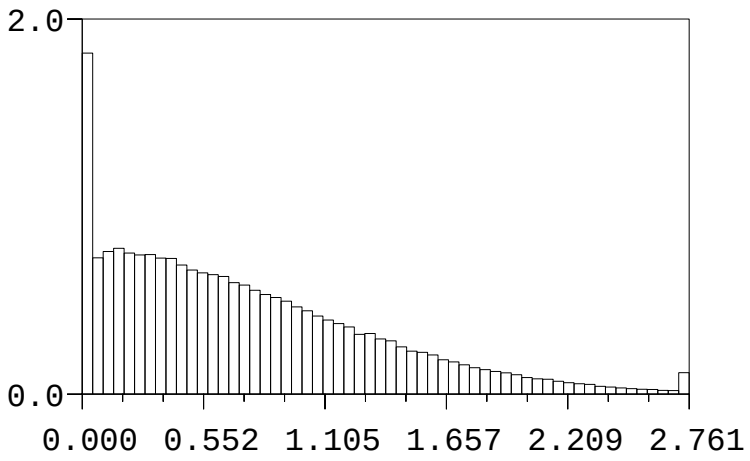


Figure 3.1: Brownian motion, path maximum.

```
public class PathFunctionalHistogram extends MaximumPathFunctional{

    // constructor, allocates maximum functional of a Brownian motion
    public PathFunctionalHistogram()
    {      super(new BrownianMotion(100,0.01,0)); }

    // Display the histogram of such a functional
    // over 200,000 paths using 100 bins.
    public static void main(String[] args)
    {
         new PathFunctionalHistogram().displayHistogram(200000,100);
    }

} // end PathFunctionalHistogram
```

Since our Brownian motion W starts at zero the path maximum is nonegative and it is zero with the exact probability with which W stays non positive along its entire path. Extreme values at the right tail of the distribution have been lumped together into the last displayed bin to limit the range of displayed values.

This example is provided only to illustrate principles. In the chapters on financial applications we will see more interesting examples: *asset prices* are stochastic processes and the *gains from trading* are functionals of the asset price paths in which we will be very interested.

3.2.1 C++ Implementation

Recall that the C++ code does not implement conditioning as a feature of random variables. Instead this feature is implemented in the class PathFunctional. Indeed path functionals are exactly the random variables for which we have a context of information and a natural way of conditioning on this information. The class PathFunctional has a method conditionedAt(int t) which returns the path functional as a random object conditioned on the state of the path at time t. The use of templates allows us to have path functionals with values in arbitrary types which support the algebraic structure needed by the RangeType of RandomObject. For example we can have vector valued or matrix valued path functionals.

3.3 Stopping times

A path functional $\tau = f(X_0, X_1, \ldots, X_T)$ of the stochastic process X which takes integer values in [0,T] it is called a *random time*. It is called a *stopping time* (*optional time*) if at each time t we can decide wether $\tau = t$ on the basis of the history of the path up to time t and without using the future $X_{t+1}, \ldots, X_T$. In more mathematical parlance the indicator function $1_{[\tau=t]}$ of the event $[\tau = t]$ must be a determistic function $g_t(X_0, X_1, \ldots, X_t)$ of the path $u \in [0, t] \to X_u$.

The term "stopping time" is derived from the field of gambling. The variable X_t might represent the cumulative winnings (possibly negative) of the gambler at time t. A systematic gambler will have a rule to decide at each time t wether it is now time to quit the game or wether one should keep playing. Since most gamblers are not in a position to influence the future of the game nor equipped with the power of clairvoyance such a rule must be based on the history of the game alone.

An example of a random time which is not a stopping time is the first time τ at which the path $t \in [0, T] \to X_t$ hits its maximum. In general, at any given time $t < T$ we do not know wether the maximum has already been hit.

Natural examples of stopping times include *hitting times* (the first time a process enters a given region), *first exit times* (the first time a process leaves a given region) or, more applied, various *sell* or *buy signals* in the stock market, where stock prices are regarded as stochastic processes.

We might be watching the process X waiting until some event occurs. At time τ of the first occurence of the event ($\tau = T$ if the event fails to occur) some action is planned. The state of the process X at the time of

this action is the random variable

$$X_\tau = \sum_{t=0}^{T} 1_{[\tau=t]} X_t, \tag{3.5}$$

equivalently $X_\tau = X_t$ whenever $\tau = t$, and is called the *process X sampled at time τ* . If we have a stake in the process X the random variable X_τ is of obvious significance. Consider the case where the X_t represents the cumulative winnings of a gambler at time t and τ is a quitting rule. X_τ are the final winnings (not necessarily positive) which the gambler takes home.

Recall that time is an integer variable $t = 0, 1, \ldots, T$, where T is the horizon. If the stochastic process X is a discretization of a process evolving in continuous time, the size dt of the time step is significant. In this case discrete time t corresponds to continuous time $t * dt$. Here is the definition of a stopping time:

```
public interface StoppingTime{

    // Returns true if it is time to stop at t,
    // false otherwise.
    public abstract boolean stop(int t);

} //end StoppingTime
```

We have already seen that the fundamental path computation is the continuation of a path $s \in [0, t] \to X(s)$ from time t to the horizon T. Sometimes however we only want to continue a path to some stopping time τ rather than the horizon. In this case we also need to retrieve the time τ. This is why the following method has return type integer rather than void. It returns the time τ and also computes the path forward from time t to time $\tau \geq t$:

```
public int pathSegment(int t, StoppingTime tau)
{
    int s=t;
    while(!tau.stop(s)){ timeStep(s); s++; }
    return s;
}
```

and the usual version of computing an entirely new path up to time τ:

```
public int pathSegment(StoppingTime tau){ return pathSegment(0,tau); }
```

With this we are set to compute the random variable X_τ (the process X sampled at time τ):

```
RandomVariable bfseries sampledAt(final StoppingTime tau)
{
      return new RandomVariable(){

          // the random draw defining X_tau
          public double getValue(int t)
          {
              int s=pathSegment(t,tau);
              return path[s];
          }

      }; //end return new

} //end sampledAt
```

Let us now introduce some concrete stochastic processes:

3.4 Random Walks

Betting repeatedly one dollar on the throw of a coin the cumulative winnings X_t after game $t+1$ have the form

$$X_t = W_0 + W_1 + \ldots + W_t, \tag{3.6}$$

where the W_j are independent and identically distributed, $W_j = +1$ with probability p and $W_j = -1$ with probability $1-p$. This game is fair if $p = 1/2$ (symmetric random walk) and subfair if $p < 1/2$ (biased random walk). The case $p > 1/2$ is of interest only to the operator of the casino.

To implement this we use the static method Sign(double p) of the class Statistics.Random which delivers a random draw from $\{-1, +1\}$ equal to $+1$ with probability p and equal to -1 with probability $1-p$:

```
public class BiasedRandomWalk extends StochasticProcess{

    double p;                   // probability of success

    // constructor, time step dt=1
    public BiasedRandomWalk(int T, double x0, double p)
    { super(T,1,x0); this.p=p; }

    // the time step
    public void timeStep(int t){ path[t+1]=path[t]+Random.Sign(p); }

} //end BiasedRandomWalk
```

The case of the symmetric random walk is the special case p=1/2 but has its own implementation relying on the method Random.Sign() producing a fair draw from $\{-1, +1\}$.

Example 1. (Examples.Probability.GamblersFortune.java) Let us now check how we would fare playing this game following some stopping rules. We plan to play until we are 5 dollars ahead or 100 dollars down or until 20000 games have been played whichever comes first. Let τ denote the corresponding stopping time. Since we intend to play no more than 20000 games we can set up the random walk with horizon $T = 20000$. To make this realistic we bias the random walk with odds 0.499/0.501 against us. What we are interested in are the expected winnings $E(X_\tau)$ under this stopping rule:

```
public class GamblersFortune{

    public static void main(String[] args)
    {
        int T=20000;     //time steps to horizon
        double x0=0.0;   //initial value

        // the process of winnings
        final BiasedRandomWalk X=new BiasedRandomWalk(T,x0,0.499);

        // here's the quitting rule
        StoppingTime tau=new StoppingTime(){

            public boolean stop(int t)
            { return ((t==20000) || (X.path[t]==+5) || (X.path[t]==-100)); }

        }; //end tau

        // the winnings, X sampled at tau
        RandomVariable X_tau=X.sampledAt(tau);

        // expected winnings, sample size 50000
        double E=X_tau.expectation(50000);

        //message reporting findings...

    } //end main

} // end GamblersFortune
```

The random walk X is declared *final* since the stopping rule tau is introduced as a local class and such a class can only make use of final local variables.

Exercise. Using the above stopping rule and probability of success determine with which probability we are wiped out with losses of 100 dollars. Hint: compute this probability as the expectation $E(Y)$, where Y is the indicator function of the event $[X_\tau = -100]$. You have to define the random variable Y. The solution is in the file Examples.Probability.GamblersFortune.java.

3.5 Markov Chains

A Markov chain is a sequential stochastic process X ($dt = 1$) with range $X_t \in \{0, 1, 2, \ldots\}$. The process is completely specified by its initial state j_0 and the transition probabilities q(t,i,j), the probability at time t that a transition is made from state i to state j. It is understood that this probability only depends on time t and the current state i but not on the history of the path up to time t.

More precisely q(t,i,j) is the conditional probability at time t that the next state is j given that the current state is i. This suggests the following setup:

```
public abstract class MarkovChain extends StochasticProcess{

    // Constructor, time step dt=1
    public MarkovChain(int T, double j0){ super(T,1,j0); }

    // transition probabilities
    public abstract double q(int t, int i, int j);

    // other methods

} // end MarkovChain
```

The constructor calls the constructor for StochasticProcess which allocates and initializes the path array with x0=j0 (the initial state). T is the number of time steps to the horizon and the size of the time step dt is set equal to one.

The transition kernel q(int t, int i, int j) is the only abstract method. A Markov X chain is defined as soon as this method is defined. In order to make X concrete as a stochastic process all we have to do is define the method timeStep(int t) of StochasticProcess from the transition probabilities q(t,i,j).

The transition from one state to another is handled as follows: assume that current time is t and the current state is i. We partition the interval

[0,1) into subintervals $I_0, I_1, \ldots$ of respective lengths q(t,i,0), q(t,i,1),. . . , that is, the probabilities that the chain will transition to state 0,1,. . . . We then draw a uniform random number u from [0,1). If u falls into the interval I_j the chain moves to state j. The following method computes this state j from i and u:

```
private int j(int t, int i, double u)
{
    int j=0; double sum=0;
    while(sum<u){ sum+=q(t,i,j); j++; }
    return j−1;
} // end j
```

With this we can now define the timeStep(int t) as follows:

```
public void timeStep(int t)
{
    int i=(int)path[t];              //current state
    double u=Random.U1();            //uniform draw from [0,1)
    path[t+1]=j(t,i,u);              //next state

} // end timeStep
```

Remember that a stochastic process stores its path in an array of doubles whereas the states of a Markov chain are integers. The conversion from integer to double is not a problem and is handled automatically but the conversion from double to integer makes an explicit type cast int i=(int)path[t]; necessary.

The class MarkovChain has been designed for convenience but not for computational efficiency. If there are only a finite number N of states and the transition probabilities are independent of time t, the Markov chain is called a *stationary finite state* (SFS) *Markov chain* .

For such a chain the sampling can be sped up considerably. Assume that the possible states are $j = 0, 1, \ldots, N-1$ and set $q(i, j) = q(t, i, j)$. We introduce two more abstract methods

```
private int a(int i);
private int b(int i);
```

with the understanding that only the states j with $a(i) \leq j \leq b(i)$ can be reached from the state i in one step. Introducing these quantities as methods allows us to conveniently define them in the body of a constructor call. Since calls to the functions a(i),b(i) carry some overhead we introduce

arrays a[],b[] to cache the values $a(i)$, $b(i)$. This allows us to retrieve these values more quickly as a[i],b[i] if necessary.
Fix a state i. To sample from the conditional distribution

$$P[\, X(t+1) = j \mid X(t) = i \,] = q(i,j)$$

partition the unit interval $[0,1)$ into subintervals $\mathcal{I}(i,j)$ of length $q(i,j)$,

$$\begin{aligned} \mathcal{I}(i,j) &= [I(i,j), I(i,j) + q(i,j)), \quad \text{where} \\ I(i,j) &= \textstyle\sum_{k=a(i)}^{j-1} q(i,k), \quad j = a(i), \ldots, b(i). \end{aligned}$$

Then draw a uniform random number $u \in [0,1)$ and transition to the state j satisfying $u \in \mathcal{I}(i,j)$. To speed up computations the points of the partitions are precomputed and stored as $I(i,j) =$partition[i][j−a[i]]. Note the baseline subtraction in the second argument. The method

```
private int I(int i, int j){ return partition[i][j−a[i]]; }
```

then allows us to work with the more familiar syntax. The search for the index $j(i,u) = j$ such that $u \in \mathcal{I}(i,j)$ can then be conducted more efficiently using continued bisection as follows:

```
private int j(int t, int i, double u)
{
    int j=a[i], k=b[i]+1, m;
    while(k−j>1){ m=(k+j)/2; if(I(i,m)>u) k=m; else j=m; }
    return j;
}
```

Thus we can introduce stationary finite state Markov chains as follows:

```
public class SFSMarkovChain extends MarkovChain{

    int N;                  // number of states

    double[][] partition;   // partition of [0,1) conditional on X(t)=i

    public SFSMarkovChain(int T, int j0, double[][] Q)      // constructor
    { /* see below */ }

    public abstract double q(int i, int j);          // transition probabilities
    private abstract int a(int i);                   // bounds for states j reachable
    private abstract int b(int i);                   // from state i in one step
```

```
    private int j(int i, double u){ /* as above */ }

    public void timeStep(int t)
    {
        int i=(int)path[t];                    //current state
        double u=Random.U1();                  //uniform draw from [0,1)
        path[t+1]=j(i,u);                      //next state
    }

} // end SFSMarkovChain
```

The method timeStep(int t) overrides the corresponding method in the superclass MarkovChain and so leads to faster path computation. The constructor performs the necessary initializations. With T being the number of time steps to the horizon, j_0 the state at time $t = 0$ and N the number of states the constructor assumes the form

```
public SFSMarkovChain(int T, int j0, int N)
{
    super(T,j0);                // allocate MarkovChain
    this.N=N;                   // possible states j = 0, 1, ..., N-1
    a=new int[N];
    b=new int[N];
    partition=new double[N][];

    for(int i=0; i<N; i++)
    {
        a[i]=a(i); b[i]=b(i);
        int ni=b(i)−a(i)+2;  // number of indices a(i) ≤ j ≤ b(i) + 1
        partition[i]=new double[ni];
        double sum=0;
        for(int j=0; j<ni; j++){ partition[i][j]=sum; sum+=q(i,j+a(i)); }

     } // end for i

} // end constructor
```

Example 2. (Examples.Probability.GambersFortune1) Let us revisit our gambler who bets 1 dollar on the throw of a coin biased 0.49/0.51 against her. This time she starts with 6 dollars and intends to play until she has 10 dollars or until she is wiped out or until 2000 games have been played whichever occurs first.

Her fortune can be modelled as a Markov chain with states 0,...,10 and absorbing barriers 0 and 10, that is, transition probabilities q(t,0,0)=1 and q(t,10,10)=1. The other transition probabilities are q(t,i,i−1)=0.51 and q(t,i,i+1)=0.49. All others are zero.

We know that our chain X will eventually hit the barrier and be absorbed there. In other words, the chain settles down at an *equilibrium distribution* on the set $\{0, 1, ..., 10\}$ of states which is concentrated at the barrier $\{0, 10\}$. If T is sufficiently large this equilibrium distribution can be approximated as the vector of probabilities

$$(P(X_T = j))_{j=0,...,10} \tag{3.7}$$

and this in turn is the expectation of the random vector

$$F = (1_{[X_T=j]})_{j=0,...,10}, \tag{3.8}$$

that is, all coordinates of F are zero except $F_j = 1$ for $j = X_T$. The following program allocates the corresponding Markov chain and the random vector (3.8) and computes the equilibrium distribution as its expectation over 20000 paths. At time $T = 2000$ most paths will have hit the barrier already that is we can take $T = 2000$ as the horizon and expect to be close the the theoretical equilibrium distribution which is concentrated at 0 and 10 with the probabilities of total loss and success respectively:

```
public class GamblersFortune1{

    public static void main(String[] args)
    {
        //allocate fortune as a Markov chain
        final int T=2000;          // time steps to horizon
        double j0=6;               // initial fortune

        final double p=0.49;       // probability of success

        // allocate a MarkovChain
        final MarkovChain X=new MarkovChain(T,j0){

            // define the transition probabilities q(t,i,j)
            public double q(int t, int i, int j)
            {
                if((i==0) && (j==0)) return 1;
                if ((i==10) && (j==10)) return 1;
                if((i>0) && (i<10) && (j==i-1)) return 1-p;
                if((i>0) && (i<10) && (j==i+1)) return p;
                return 0;      // all other cases
            }

        }; // end X

        // allocate the random vector F above
        RandomVector F=new RandomVector(11){

            public double[] getValue(int t)
            {
                double[] x=new double[11], path=X.get_path();
                X.newPathBranch(t);
                x[(int)path[T]]=1;   // all other components zero
                return x;

             } // end getValue

        }; // end F

        // expectation of F over 10000 paths
        double[] Q=F.expectation(10000);

        // print the Q[j], j=0,..,10;

    } // end main

} // end GamblersFortune_1
```

The reason why some of the above variables are declared final is the fact that

local classes such as X and F above can only use local variables if they are final.

Example 3. (Examples.Probability.Urns) Consider two urns. The first urn is filled with N white balls. The second urn is filled with M black balls. The following operation is now performed repeatedly: from each urn a ball is selected at random and the two balls exchanged. This procedure leaves the number of balls in each urn unchanged.

We expect that eventually white (w) and black (b) balls will be mixed in proportion $N : M$ in both urns. To check this we let X_t denote the number of white balls in the first urn immediately before the exchange number $t+1$. Thus $X_0 = N$.

Assume that $X_t = i$. Then we have i white balls in the first urn and $N - i$ white balls in the second urn. Setting

$$p_1 = \frac{i}{N},\ q_1 = 1 - p_1 \quad \text{and} \quad p_2 = \frac{N-i}{M},\ q_2 = 1 - p_2 \tag{3.9}$$

we have the following probabilities for selection (w/b) of the pair of balls from the two urns:

$$\begin{aligned} Prob((w,w)) &= p_1 p_2, \quad Prob((b,b)) = q_1 q_2, \\ Prob((w,b)) &= p_1 q_2, \quad Prob((b,w)) = q_1 p_2. \end{aligned}$$

Consequently the possible transitions for X_t are as follows: unchanged with probability $p_1p_2 + q_1q_2$, decrease by one with probability p_1q_2, increase by one with probability q_1p_2.

Let us now allocate this Markov chain with horizon T=1000 and N=100, M=200. At the horizon we expect one third of all balls in each urn to be white. In other words we expect $E(X_T) = 100/3$. This will be checked over a sample of 2000 paths:

```
public class Urns{

    public static void main(String[] args)
    {
        final int N=100,            // balls in urn1
                  M=200,            // balls in urn2
                  T=1000,           // horizon
                  j0=100;           // initial state

        //allocate the Markov chain (number of balls in urn1)
        final MarkovChain X=new MarkovChain(T,j0){

            // define the transition probabilities
            public double q(int t, int i, int j)
            {
                double fi=(double)i,
                       p_1=fi/N, q_1=1-p_1,
                       p_2=(N-fi)/M, q_2=1-p_2;

                if((i>=0) && (i<=N) && (j==i)) return p_1 * p_2+q_1 * q_2;
                if((i>0) && (i<=N) && (j==i-1)) return p_1 * q_2;
                if((i>= 0) && (i<N) && (j==i+1)) return q_1 * p_2;
                return 0;       // all other case
            }

        }; // end X

        // allocate the random variable X_T
        RandomVariable X_T=new RandomVariable(){

            public double getValue(int t)
            {
                double[] x=X.get_path();
                X.newPathBranch(t); return x[T];
            }

        }; // end X_T

        // compute the expectation E(X_T) over a sample of 20000 paths
        double E=X_T.expectation(2000);

        // message to report the findings

    } // end main

} // end Urns
```

Our Markov chain has only finitely many states and time independent transition probabilities. Thus we could also allocate it as an SFSMarkovChain. The program Examples.Probability.Urns does both and times both computations. The improved sampling in SFSMarkovChain leads to a tenfold speedup.

3.6 Optimal Stopping

Let $X(t)$, $t = 0, 1, \ldots, T$ be a stochastic process and $\mathcal{F}_t$ the information generated by the process up to time t, that is the σ-field generated by the path

$$path(t) : u \in [0, t] \mapsto X(u), \tag{3.10}$$

and E_t denote the conditional expectation with respect to $\mathcal{F}_t$. A game is played as follows: at each time t a player can decide to stop and receive a reward $R(t)$ or to continue in hopes of a greater future reward. Here the reward $R(t)$ is assumed to be a random variable which depends only on the path of the process X up to time t, that is, a deterministic function of $path(t)$.

Let $Z(t)$ denote the expected optimal reward at time t given that the game has not been stopped yet. At time t we can either stop with reward $R(t)$ or continue and receive the expected optimal reward $E_t[Z(t+1)]$ and we suspect that the optimal strategy will stop at time t if $R(t) \geq E_t[Z(t+1)]$ and continue otherwise.

This leads us to define the process $Z(t)$ by backward induction on t as

$$\begin{aligned} Z(T) &= R(T) \quad \text{and} \\ Z(t) &= \max\{\, R(t), E_t[Z(t+1)] \,\}, \quad t < T; \end{aligned}$$

and we claim that the stopping time

$$\tau_0 = \min\{\, t \in [0, T] \mid R(t) = Z(t) \,\} \tag{3.11}$$

is optimal , that is,

$$E[R(\tau_0)] \geq E[R(\tau)], \tag{3.12}$$

for all stopping times τ with values in $\{0, 1, \ldots, T\}$. Here it is understood that $\tau_0 = T$ if the event $R(t) = Z(t)$ never happens. The proof relies on elementary properties of supermartingales (the Optional Sampling Theorem in the Appendix). Note first that

$$Z(t) \geq E_t[Z(t+1)],$$

that is, $Z(t)$ is a supermartingale. Moreover $Z(t) \geq R(t)$. It will now suffice to show that

$$E[R(\tau_0)] = Z(0),$$

since the Optional Sampling Theorem for supermartingales implies that $Z(0) \geq E[Z(\tau)] \geq E[R(\tau)]$ for all stopping time τ with values in $\{ 0, 1, \ldots, T \}$ and so $E[R(\tau_0)] \geq E[R(\tau)]$.

To set up a proof by backward induction on t we introduce the stopping times τ_t defined as

$$\tau_t = \min\{ s \geq t \mid R(s) = Z(s) \} \tag{3.13}$$

and claim that more generally

$$E_t(R(\tau_t)) = Z(t), \quad \text{for all } t \leq T. \tag{3.14}$$

This is certainly the case for $t = T$ since then $\tau_t = T$. Assume that $t < T$ and (3.14) is true if t is replaced with $t + 1$. Let A be the event $[\tau_t = t]$ and $B = A^c = [\tau_t > t]$. The set A is $\mathcal{F}_t$-measurable. This implies that random variables U, V satisfying $U = V$ on A will also satisfy $E_t(U) = E_t(V)$ on A:

$$1_A E_t(U) = E_t(1_A U) = E_t(1_A V) = 1_A E_t(V)$$

and the same is true of the set B. On the set A we have $R(t) = Z(t)$ and so $R(\tau_t) = R(t) = Z(t)$. On the set B we have $\tau_t = \tau_{t+1}$ and $R(t) < Z(t)$ and consequently $Z(t) = E_t[Z(t + 1)]$. Thus $E_t[R(\tau_t)] = Z(t)$ on A and

$$E_t[R(\tau_t)] = E_t[R(\tau_{t+1})] = E_t\left[E_{t+1}[(R(\tau_{t+1})]\right] = E_t[Z(t + 1)] = Z(t)$$

on B, by induction hypothesis. Consequently $E_t[R(\tau_t)] = Z(t)$ everywhere (more precisely with probability one).

Continuation value. In implementations it is often best to think in terms of the continuation value $CV(t)$ defined recursively as

$$\begin{aligned} CV(T) &= 0, \quad \text{and} \\ CV(t) &= E_t\left[\max\{ R(t + 1), CV(t + 1) \}\right], \quad t < T. \end{aligned} \tag{3.15}$$

An easy backward induction on t then shows that

$$Z(t) = \max\{ R(t), CV(t) \}$$

and the optimal stopping time τ_0 assumes the form

$$\tau_0 = \min\{ t \in [0, T] \mid R(t) \geq CV(t) \}$$

again with the understanding that $\tau_0 = T$ if the event $R(t) \geq CV(t)$ never happens. In other words it is optimal to stop as soon as the immediate reward is no less than the continuation value. Substituting the defining expressions for $CV(t+1), CV(t+2), ...$ until we hit $CV(T) = 0$, the continuation value at time t assumes the form

$$CV(t) = E_t[\max\{\ R(t+1), E_{t+1}[\max\{\ldots E_{T-1}[R(T)]\}]\ldots]. \tag{3.16}$$

In principle the iterated conditional expectations can be computed with the Monte Carlo method by repeatedly branching paths. Suppose we branch 10^4 times for each conditional expectation. After the fifth conditional expectation we are already computing 10^{20} path branches which themselves continue to split for each remaining conditional expectation. Such a computation is not feasible on contemporary computing equipment.

Markov chains. The situation becomes manageable however if $X(t)$ is a Markov chain. In this case conditioning on the information generated by the process X up to time t is the same as conditioning on the state $X(t)$ at time t and so all conditional expectations $E_t(\cdot)$ are deterministic functions $f(X(t))$ of the state $X(t) = 0, 1, 2, \ldots$ at time t.

Let $R(t, j)$ and $CV(t, j)$ denote the reward and continuation value at time t given that $X_t = j$. Then relation (3.15) can be rewritten as

$$CV(t, i) = \sum\nolimits_j q(t, i, j) \max\{\ R(t+1, j), CV(t+1, j)\ \} \tag{3.17}$$

where $R(t+1, j)$ is the reward for stopping at time given that $X(t+1) = j$. The sum in (3.17) needs to be computed only over all states j which can be reached from the current state i in one step (that is $q(t, i, j) \neq 0$). Let $a(t, i)$, $b(t, i)$ be such that

$$\{\ j : q(t, i, j) \neq 0\ \} \subseteq [a(t, i), b(t, i)]. \tag{3.18}$$

Then (3.17) can be rewritten as

$$CV(t, i) = \sum\nolimits_{j=a(t,i)}^{b(t,i)} q(t, i, j) \max\{\ R(t+1, j), CV(t+1, j)\ \} \tag{3.19}$$

The Markov chain X starts at time zero in some definite state $X_0 = j_0$ but as time progresses the number of possible states for $X(t)$ increases. From the quantities $b(t, i)$ above we can compute $u(t)$ such that

$$X(t) \in [0, u(t)], \tag{3.20}$$

that is, $X(t)$ must be in one of the states $j = 0, 1, \ldots, u(t)$, as follows:

$$u(0) = j_0, \quad \text{and} \tag{3.21}$$

$$u(t+1) = \max\{\, b(t,j) : j \leq u(t) \,\}. \tag{3.22}$$

Note that all these quantities depend only on the intial state $X(0) = j_0$, the transition probabilities $q(t,i,j)$ and the reward function $R(t,i)$ and so can be computed without any path simulation. These computations can therefore be relegated to the constructor of the following class:

```
public abstract class StoppableMarkovChain extends MarkovChain{

    // possible states at time t: j=0,1,...,u[t]
    int[] u;

    // [t][i]: continuation value at time t in state i
    double[][] continuationValue;

    // optimal stopping time with respect to the reward function
    StoppingTime optimalStoppingTime;

    // reward from stopping at time t in state i
    public abstract double reward(int t,int i);

    // only states j>=a(t,i) can be reached from state i at time t in one step
    public abstract int a(int t,int i);

    // only states j<=b(t,i) can be reached from state i at time t in one step
    public abstract int b(int t,int i);

    // constructor, T time steps to horizon, j0 state at time zero
    public StoppableMarkovChain(int T, double j0)
    {
        super(T,j0); // the constructor for MarkovChain

        // possible states i at time t: i=0,1,...,u[t]
        u[0]=(int)j0;
        for(int t=0; t<T; t++)
        {
            u[t+1]=b(t,0);
            for(int i=1; i<=u[t]; i++)
            if(b(t,i)>u[t+1]) u[t+1]=b(t,i);
        }

        // continuation value at time t=T
        for(int j=0; j<=u[T]; j++) continuationValue[T][j]=0;

        // continuation value at times t<T
```

```
        for(int t=T−1; t>= 0; t−−)
        for(int i=0; i<=u[t]; i++) // loop over all possible states at time t
        {
            // sum over all possible states j which can be reached
            // from state i at time t in one step
            double sum=0;
            for(int j=a(t,i); j<=b(t,i); j++)
            {
                double x=reward(t+1,j),
                       y=continuationValue[t+1][j],
                       max=(x>y)? x:y; //max{x,y}
                sum+=q(t,i,j) ∗ max;
            }
        } // end for t

        // allocate the optimal stopping time
        optimalStoppingTime=new StoppingTime(){

            public boolean stop(int t)
            {
                int i=(int)path[t]; // current state
                return(reward(t,i)>=continuationValue[t][i]);
            } // end stop

        }; // end optimalStoppingTime

    } // end constructor

} // end StoppableMarkovChain
```

3.7 Compound Poisson Process

Let X be a random variable and $N(t)$ a Poisson process. Recall that $N(t)$ can be viewed as the number of events that have occured by time t given that the number of events occuring in disjoint time intervals are independent and

$$Prob(N(t+h) - N(t) = 1 \mid \mathcal{F}_t) = \lambda h + o(h) \quad \text{and} \tag{3.23}$$

$$Prob(N(t+h) - N(t) \geq 2 \mid \mathcal{F}_t)) = o(h), \tag{3.24}$$

where $\mathcal{F}_t$ is the information generated by the process $N(t)$ up to time t and λ is a constant (the intensity of the process).

In this case the number $N(t+h) - N(t)$ of events occuring in the time interval $(t, t+h]$ is a Poisson variable with mean λh, that is, it takes values

in the nonnegative integers with probabilities

$$Prob(N(t+h) - N(t) = k) = e^{-\lambda h}\frac{(\lambda h)^k}{k!}, \quad k = 0, 1, 2, ... \tag{3.25}$$

Let $X_0, X_1, ...$ be a sequence of independent random variables all with the same distribution as X. The process

$$CP(t) = \sum_{k=0}^{N(t)} X_k \tag{3.26}$$

is called the compound Poisson process with intensity λ and *event size* X. Think of these events as insurance claims and the X_k the size of the claims. The compound Poisson process then represents the sum of all claims received by time t.

After implementing a Poisson random variable in the obvious way relying on a Poisson random number generator we can define the compound Poisson process as follows:

```
public class CompoundPoissonProcess extends StochasticProcess{

    double lambda;          // intensity
    RandomVariable X;       // event size
    RandomVariable N;       // number of events in [t, t+dt], needed for time step.

  // constructor
  public CompoundPoissonProcess
  (int T, double dt, double x0, double lambda, RandomVariable X)
  {
      super(T,dt,x0);
      this.lambda=lambda;
      this.X=X;
      N=new PoissonVariable(lambda * dt);
  } // end constructor

  public void timeStep(int t)
  {
      int n=(int)N.getValue(0);
      double sum=0;
      for(int i=0; i<n; i++) sum+=X.getValue(0);
      path[t+1]=path[t]+sum;

  } // end timeStep

} // end CompoundPoissonProcess
```

It is known that $E(N(t)) = \lambda t$ (expected number of claims) and $E(CP(t)) = \lambda t E(X)$ (expected total of claims). The insurance company collects premiums leading to a cash flow of μt and has initial capital c_0. Assuming that

claims are paid on the spot, $c_0 + (\mu - \lambda)t$ is the *expected* cash position at time t while the *actual* cash position is given by

$$c_0 + \mu t - CP(t). \tag{3.27}$$

The fundamental problem is that of *ruin* , ie. the possibility that $CP(t) > c_0 + \mu t$ when claims can no longer be paid:

Example 4. (Examples.Probability.Insurance) What is the probability of ruin before time T*dt (dt the size of the time step, T the number of time steps to the horizon) given that $X = Z^2$, where Z is standard normal (then E(X)=1), $\lambda = 100$, $\mu = (1 + 1/10)\lambda E(X)$ and $c_0 = 50$:

```
public class Insurance{

    public static void main(String[] args)
    {
        final int T=500;                    // horizon
        final double dt=0.01,               // time step
                     lambda=100,            // claim intensity
                     x0=0,                  // claims at time t=0
                     c0=30;                 // initial capital

        final RandomVariable X=new RandomVariable(){

            public double getValue(int t)
            {
                double z=Random.STN();
                return z*z;
            }

        }; // end X

        // premium rate, yes we know it's λ*1.1 here
        final double mu=1.1*lambda*X.expectation(20000);

        // aggregate claims
        final CompoundPoissonProcess
        CP=new CompoundPoissonProcess(T,dt,x0,lambda,X);
```

```
        // the time of ruin
        final StoppingTime tau=new StoppingTime(){

            public boolean stop(int t)
            {
                double[] claims=CP.get_path();
                return ((claims[t]>c_0+t*dt*mu) || (t==T));
            }

        }; // end tau

        // the indicator function of the event [τ < T] of ruin
        RandomVariable ruin=new RandomVariable(){

            public double getValue(int t)
            {
                // path computation and value of stopping time
                int s=CP.pathSegment(t,tau);
                if(s<T) return 1;
                return 0;

            } // end getValue

        }; // end ruin

        // probability of ruin, 20000 paths
        double q=ruin.expectation(20000);

        // message reporting findings

    } // end main

} // end Insurance
```

3.8 One Dimensional Brownian Motion

A one dimensional Brownian motion is a continuous stochastic process $W(t)$ with increments $W(t+h) - W(t)$ independent of the information (σ-field) generated by the process up to time t and normally distributed with mean zero and variance h, in other words

$$W(t+h) - W(t) = \sqrt{h}\, Z, \tag{3.28}$$

where Z is a standard normal variable (mean zero, variance one). This proces is fundamental in the theory of continuous martingales and has widepread applicability outside of mathematics such as for example in finance theory. Many stock market speculators have made the acquaintance of Brownian motion and one hopes without regrets.

Remember that the constructor of each subclass of StochasticProcess must call the constructor of StochasticProcess to pass the parameters T (time steps to horizon), dt (size of time step) and x_0 (initial value) so that the path array can be allocated and initialized:

```
public class BrownianMotion extends StochasticProcess{

    double sqrtdt; // compute this only once in the constructor

    // constructor
    public BrownianMotion(int T, double dt, double x0)
    {
        super(T,dt,x0);
        sqrtdt=Math.sqrt(dt);
    }

    // Random.STN() delivers a standard normal deviate
    public void timeStep(int t)
    {
        path[t+1]=path[t]+sqrtdt*Random.STN();
    }

} // end BrownianMotion
```

We will see an application of Brwonian motion in dimension two after vectorial stochastic processes have been introduced in the next section:

3.9 Vector Valued Stochastic processes

The case of a vector valued stochastic process X is completely analogous to the case of a scalar process. The path array now has the form

```
double[][] path;
```

with the understanding that path[t] is the vector X_t, that is, the state of the process at time t, represented as an array of doubles. Clearly the dimension of the process will be a member field and necessary argument to the constructor. If τ is a stopping time, then X_τ is now a random vector instead of a random variable:

```
public abstract class VectorProcess{

    int dim;          // dimension
    int T;            // time steps to horizon
    double dt;        // size of time step

    double[] x0;      // initial value (a vector)
    double[][] path; // path array, path[t]=X_t (a vector)

    // constructor (call from concrete subclass)
    public VectorProcess(int dim, int T, double dt, double[] x0)
    {
        this.dim=dim;
        this.T=T;
        this.dt=dt;
        this.x0=x0;
        path=new double[T+1][dim];   //allocate the path array
        path[0]=x0;
   } // end constructor

   // other methods

} // end VectorProcess
```

The other methods are similar to those in the case of a one dimensional process. For example here is how we sample such a process at an optional time:

```
// the random vector X_τ
RandomVector sampledAt(final StoppingTime tau)
{
    return new RandomVector(dim){

        // the random draw defining X_τ
        public double[] getValue(int t)
        {
            // compute a new path branch from time t to the time
            // s=tau of stopping and get the time s.
            int s=pathSegment(t,tau);
            return path[s];
        } // end getValue

    }; // end return new

} // end sampledAt
```

The constructor of a concrete subclass of VectorProcess must call the constructor of VectorProcess (super(T,dt,x0,dim)) to pass the parameters T,dt,x0,dim so that the path array can be properly allocated and initialized.

Hitting times. Let X be a stochastic process with values in R^d. There are some important stopping times associated with the process X. If $G \subseteq R^d$ is a region (open or closed sets are sufficient for us) then the *hitting time* τ_G is the first time at which the process enters the region G or more precisely

$$\tau_G = \inf\{\, t > 0 \mid X_t \in G \,\}.$$

The *first exit time* is the first time the process leaves the region G, that is, the hitting time of the complement G^c. If the process X has continuous paths and $X_0 \notin G$, then $X(\tau_G) \in boundary(G)$. To implement this concept we must first implement the concept of a region $G \subseteq R^d$. A point x viewed as an array of doubles is either in the region D or it is not:

```
public interface Region_nD{

    public abstract boolean isMember(double[] x);

} // end Region_nD
```

With this a hitting time needs a process X and a region G:

```
public class HittingTime_nD implements StoppingTime{

    VectorProcess X;
    Region_nD D;

    /** constructor */
    public HittingTime_nD(VectorProcess X, Region_nD D)
    {
        this.X=X;
        this.G=G;
    }

    // define the method stop()
    public boolean stop(int t)
    {
        int T=X.get_T();
        return((D.isMember(X.path[t])) || (t==T));
    }

} // end HittingTime_nD
```

3.10 Brownian Motion in Several Dimensions

A d-dimensional Brownian motion is a d-dimensional stochastic process

$$X_t = (X_t^1, X_t^2, \ldots, X_t^d)$$

such that each coordinate process X_t^j, $j = 1, \ldots, d$, is a one dimensional Brownian motion and such that the coordinate processes are independent. With the fields T,dt,x0,dim inherited from StochasticProcess it can be implemented as follows:

```
public class VectorBrownianMotion extends VectorProcess{

    double sqrtdt;   // compute this only once in the constructor

    // constructor
    public VectorBrownianMotion(int dim, int T, double dt, double[] x0)
    {
        super(dim,T,dt,x0);
        sqrtdt=Math.sqrt(dt);
    }
```

```
    // Random.STN() generates a standard normal deviate.
    public void timeStep(int t)
    {
      for(int i=0; i<dim; i++)
      path[t+1][i]=path[t][i]+sqrtdt*Random.STN();
    }

} // end VectorBrownianMotion
```

Example 5. *Dirichlet's problem.* (Examples.Probability.DirichletProblem) Let $G \subseteq R^d$ be a bounded region and h a continuous function on the boundary of G. We seek to extend h to a harmonic function on all of G, that is, we want to solve the partial differential equation

$$\nabla f = 0 \quad \text{in the interior of } G \tag{3.29}$$

$$f = h \quad \text{on the boundary of } G \tag{3.30}$$

If f is a solution of 3.29 and 3.30 and B_t a d-dimensional Brownian motion starting at some point $x \in G$ it can be shown that the process $f(B_t)$ is a martingale up to the time τ when B_t hits the boundary of G (it is not defined thereafter). In other words τ is the first exit time from G. From the Optional Sampling Theorem and the fact that $B_0 = x$ is constant it follows that

$$f(x) = E(f(B_0)) = E(f(B_\tau)) = E(h(B_\tau)). \tag{3.31}$$

Note that $f(B_\tau)) = h(B_\tau)$ since $B_\tau \in \mathit{boundary}(G)$. This suggests that one could try to solve (3.29) and (3.30) by starting a Brownian motion B_t at points $x \in G$ and computing the solution f at such a point x as $f(x) = E(h(B_\tau))$, where τ is the time at which B_t hits the boundary of G.

The following program carries out this idea in dimension $d = 2$ with G being the unit disc $x_1^2 + x_2^2 < 1$ and the boundary function $h(x_1, x_2) = x_1 x_2$. This function is harmonic on the entire plane and so the solution f of (3.29) and (3.30) is also given by the formula $f(x_1, x_2) = x_1 x_2$. As the point x we choose $x = (x_1, x_2) = (1/4, 1/4)$ and consequently $f(x) = 1/16$.

Below we see a Brownian motion launched at the origin inside the disc (Examples.Probability.DirichletDemo):

Our discrete approximation of Brownian motion does not exactly hit the boundary of the disc but crosses it and then stops. To get back to the boundary we project the first point u outside the disc back onto the circle to the point $u/||u||$. Obviously this is a source of inaccuracy.

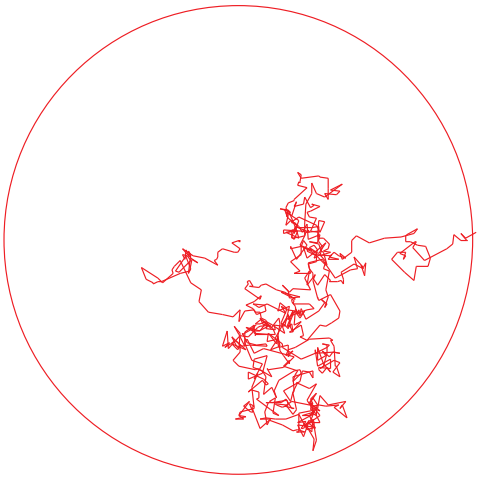


Figure 3.2: Brownian motion

```
public class DirichletProblem{

    // projects the point (u,v) radially on the unit circle
    public static double[] boundaryProjection(double[] x)
    {
        double u=x[0], v=x[1], f=Math.sqrt(u*u+v*v);
        double[] result={u/f,v/f};
        return result;
    }

    public static void main (String[] args)
    {
        int T=50000;                //time steps to horizon
        int dim=2;                  //dimension
        double dt=0.001;            //size of time step

        double[] x={0.25,0.25};     //starting point

        // allocate a two dimensional Brownian motion starting at x
        final VectorBrownianMotion B=new VectorBrownianMotion(dim,T,dt,x);

        // define the unit disc as a two dimensional region
        Region_nD disc=new Region_nD(){

            public boolean isMember(double[] z)
            { return z[0]*z[0]+z[1]*z[1]<1; }
```

```
        }; // end disc

        // allocate the hitting time for the boundary
        final FirstExitTime_nD tau=new FirstExitTime_nD(B,disc);

        // allocate the random variable h(B_tau)
        RandomVariable hB_tau=new RandomVariable(){

            public double getValue(int t)
            {
                double[] z=boundaryProjection(B.sampledAt(tau).getValue(t));
                double u=z[0], v=z[1];
                return u*v;
            } // end getValue

        }; // end hB_tau

        //compute f(x)=E(h(B_tau)) over 40000 paths
        //this should be 0.25*0.25=0.0625
        double fx=hB_tau.expectation(40000);

        // message to report the findings

    } // end main

} // end DirichletProblem
```

The computation does not slow down very much as the dimension increases since Brownian motion in higher dimensions moves out to infinity faster and thus hits the boundary more quickly. Note that the time step `dt` has to be chosen small enough to ensure accuracy when hitting the boundary and the horizon T large enough to ensure that the Brownian path does in fact reach the boundary of the disc.

The C++ implementation is somewhat more developed. It has a more accurate algorithm to determine the point where the boundary is hit and also allows for more general regions (DirichletProblem.h) in any arbitrary dimension.

3.11 Asset price processes.

Processes which model the prices in a financial market are often assumed to satisfy a dynamics of the form

$$\begin{aligned} dS_i(t) &= S_i(t)\left[\mu_i(t)dt + \nu_i(s)\cdot dW(t)\right], \\ S_i(0) &= s_i > 0, \quad t \in [0,T], \quad i = 1,\ldots,n, \end{aligned}$$

where W is a n-dimensional Brownian motion, the drift $\mu_i(t)$ and the vector $\nu_i(t)$ are deterministic (state independent), μ_i is absolutely integrable on $[0,T]$ and

$$\int_0^T \|\nu_i(t)\|^2 dt < \infty.$$

Let us write

$$\sigma_i(t) = \|\nu_i(t)\|, \quad u_i(t) = \|\nu_i(t)\|^{-1}\nu_i(t), \quad \text{and} \quad \rho_{ij}(t) = u_i(t)\cdot u_j(t).$$

Then

$$\nu_i(t) = \sigma_i(t)u_i(t) \quad \text{and} \quad \nu_i(t)\cdot\nu_j(t) = \sigma_i(t)\sigma_j(t)\rho_{ij}(t).$$

There are some standard simplifications which can now be applied. The above dynamics implies that the process S_i is positive and passing to the logarithms $Y_i = log(S_i)$ (the "returns" on the asset S_i) turns (3.32) into

$$dY_i(t) = \left(\mu_i(t) - \frac{1}{2}\sigma_i^2(t)\right)dt + \nu_i(t)\cdot dW(t) \tag{3.32}$$

where the drift term is deterministic and this implies that

$$Y_i(t) = E[Y_i(t)] + V_i(t), \quad \text{where} \quad V_i(t) = \int_0^t \nu_i(s)\cdot dW(s)$$

and $E[Y_i(t)]$ is $Y(0)$ plus the deterministic drift up to time t:

$$E[Y_i(t)] = Y(0) + \int_0^t \left(\mu_i(s) - \frac{1}{2}\sigma_i^2(s)\right)ds.$$

In concrete asset price models the functions $\mu_i(t)$ and $\sigma_i(t)$ are given explicitly and thus the quantities $E[Y_i(t)]$ can be computed by analytic formula and need not be simulated. Consequently we have to concern ourselves only with the Ito processes $V_i(t)$ which satisfy

$$dV_i(t) = \nu_i(t)\cdot dW(t). \tag{3.33}$$

From these the asset prices $S_i(t)$ are reconstructed as

$$S_i(t) = exp\left(E[Y_i(t)] + V_i(t)\right).$$

Expanding (3.33) in terms of the components of W yields

$$dV_i(t) = \sum_{k=1}^{n} \nu_{jk}(t) dW_k(t).$$

The components $W_k(s)$ of the Brownian motion will be called the *factors* and the vectors ν_i the *factor loadings*. Indeed ν_i is the vector of weights with which the factors influence the component V_i.

The determinisitic (state independent) nature of the ν_j implies that the vector process $V(t)$ is a Gaussian process (all marginal distributions multinormal). Consequently the $\nu_j(t)$ are also called *Gaussian factor loadings*.

Using the fact that the covariations of the Brownian motions W_i satisfy $d\langle W_i, W_j\rangle = \delta_{ij} ds$ and the rule

$$\left\langle \int_0^t H(s)dM(s), \int_0^t K(s)dN(s) \right\rangle = \int_0^t H(s)K(s)d\langle M, N\rangle_s$$

for the covariations of stochastic integrals with respect to scalar martingales M, N we obtain the covariations

$$\langle Y_i, Y_j\rangle_t = \langle V_i, V_j\rangle_t = \int_0^t \nu_i(s) \cdot \nu_j(s) ds = \int_0^t \sigma_i(s)\sigma_j(s)\rho_{ij}(s) ds \qquad (3.34)$$

The reader can disregard these covariations since they will be used only as a more compact way of writing the integrals on the right. The factor loadings are related to the volatility and correlation of the returns $Y_i(t)$ over any interval $[t, T]$. Indeed set

$$\Delta_i(t, T) = V_i(T) - V_i(t) = \int_t^T \nu_i(s) \cdot dW(s)$$

so that the vector $V(T)$ satisfies

$$V(T) = V(t) + \Delta(t, T).$$

To simulate the time step $t \to T$ we need the distribution of $\Delta(t, T)$ conditional on the state at time t. The deterministic nature of the factor loadings implies that the increment $\Delta(t, T)$ is in fact independent of the state at time t and has a multinormal distribution with covariance matrix $D(t, T)$ given by

$$D_{ij}(t, T) = \int_t^T \sigma_i(s)\sigma_j(s)\rho_{ij}(s) ds = \langle Y_i, Y_j\rangle_t^T.$$

This suggests the following procedure for simulating paths of the process $V(t)$ in time steps $t_j \to t_{j+1}$: at time t_j we have $V(t_j)$ and since the increment $\Delta(t_j, t_{j+1}) = V(t_{j+1}) - V(t_j)$ is independent of the state at time t we simply draw a random sample Δ from its known multinormal distribution and set

$$V(t_{j+1}) = V(t_j) + \Delta.$$

Note that this approach to path simulation entails no approximation at all. The paths are sampled precisely from the distribution of the process. This is possible whenever the distribution of the innovation (process increment) $\Delta(t, T) = V(t) - V(t)$ conditional on the state at time t is known.

To simulate a random draw from the multinormal distribution of $\Delta(t, T)$ we can proceed as follows (see Appendix B.1): we factor the covariane matrix $D(t, T)$ as

$$D(t, T) = R(t, T)R(t, T)'$$

with $R(t, T)$ upper triangular for greatest speed (Cholesky factorization) and then set

$$\Delta = R(t, T)Z(t), \quad \text{hence} \quad V(T) = V(t) + R(t, T)Z(t), \tag{3.35}$$

where $Z(t)$ is a random sample of a standard normal vector. The components of $Z(t)$ are simply populated with independent standard normal deviates. The time step for the returns Y_i then becomes

$$Y_i(T) = Y_i(t) + m_i(t, T) + \sum_{k=1}^{n} R_{ik}(t, T)Z_k(t),$$

where $m_i(t, T)$ is the deterministic drift increment

$$m_i(t, T) = \int_t^T \left(\mu_i(s) - \frac{1}{2}\sigma_i^2(s)\right) ds.$$

These increments are path independent and can be precomputed and cached to speed up the path simulation.

Implementation

Note that we do not neeed the factor loadings $\nu_i(t)$ explicitly, only the volatilities $\sigma_i(t)$ and correlations $\rho_{ij}(t)$ are needed and these become our model parameters. To simplify the implementation the correlations $\rho_{ij}(t)$ are assumned to be independent of time (as well as state):

$$\rho_{ij}(t) = \rho_{ij}.$$

There is no difficulty in principle to handle time dependent correlations. All that happens is that the covariation integrals (3.34) become more complicated. The class FactorLoading declares the abstract methods

```
Real sigma(int i, int t);
Real rho(int i, int j);
```

and then provides methods to compute the covariance matrices $D(t,T)$ and their Cholesky roots. It is a good idea to allocate an array Z storing the vectors $Z(t)$ used to drive the time steps. If we want to drive the dynamics of Y with a low discrepancy sequence all the standard normal deviates $Z_k(t)$ involved in computing a new path of Y must be derived from a single uniform vector of a low discrepancy sequence of a suitable dimension (the number of deviates $Z_k(t)$ needed to compute one path of Y). See the Section 7.1. In other words first the entire discretized path of the underlying Brownian motion $W(t)$ is computed and then we can compute the corresponding path $t \to Y(t)$.

In principle this would also be necessary if the $Z_k(t)$ are computed by calls to a uniform random number generator with subsequent conversion to standard normal deviates. In practice however the uniform random number generator delivers uniform vectors of all dimensions by making repeated calls to deliver the vector components so that the generation of the $Z_k(t)$ simply amounts to repeated calls to the uniform random number generator.

The computation of the $Z_k(t) = Z[t][k]$ is independent of the rest of the computation of the path $t \to Y(t)$ and can be handled by a separate class

```
class StochasticGenerator {

    // write the Z[t][k] for a full path into the array Z
    virtual void newWienerIncrements(Real** Z) = 0;
};
```

This class will also declare other methods to compute the $Z[t][k]$ needed for frequently encountered partial paths of the process Y (such as computing only the components Y_j, $j \geq i$ up to some time t before the horizon.

We can then define concrete subclasses MonteCarloDriver, SobolDriver which generate the $Z[t][k]$ using a Mersenne Twister or a Sobol generator (with subsequent conversion of uniform deviates to standard normal ones).

Each Ito process Y will contain a pointer to a StochasticGenerator and we can switch from Monte Carlo (uniform rng) to Quasi Monte Carlo dynamics (low discrepancy sequence) by pointing this pointer to an appropriate

object. This approach is implemented in the C++ classes ItoProcess, LiborMarketModel.

Factor reduced models.

Let $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_n \geq 0$ denote the eigenvalues λ_j of the covariance matrix $D = D(t,T)$ governing the time step (3.35) and assume that the largest r eigenvalues dominate the rest in the sense that

$$\rho = \frac{\lambda_1^2 + \lambda_2^2 + \ldots + \lambda_r^2}{\lambda_1^2 + \lambda_2^2 + \ldots + \lambda_n^2}$$

is close to one. Let U denote an orthogonal matrix of eigenvectors u_j of D (where the column u_j of U satisfies $Du_j = \lambda_j u_j$). Then the covariance matrix D can be approximated as

$$D \simeq L_r L_r',$$

where L_r is the $n \times r$ matrix with columns

$$L_r = \left(\sqrt{\lambda_1}u_1, \ldots, \sqrt{\lambda_r}u_r\right).$$

The relative approximation error in the trace norm

$$\|A\|^2 = Tr(A'A) = \sum\nolimits_{ij} a_{ij}^2$$

is given by

$$\frac{\|D - L_r L_r'\|}{\|D\|} = \sqrt{1-\rho}.$$

See Appendix, A.1, B.3. This error will be small if the sum of the first r eigenvalues is much bigger than the sum of the remaining $n-r$ eigenvalues. With this the increment $V(T) - V(t)$ can be approximated as

$$V(T) - V(t) = L_r Z_r(t), \tag{3.36}$$

and this approximation still explains $100q\%$ of the variability of $V(T) - V(t)$. Here $Z_r(t)$ is a standard normal vector of dimension r. If this reduction is applied to all covariance matrices governing all the time steps the dynamics of S can be driven by a Brownian motion in dimension $r < n$, that is the number of factors has been reduced from n to r which obviously speeds up the simulation of paths.

In practice often the largest two or three eigenvectors are already quite dominant and this leads to a substantial reduction in the number of factors.

We should note however that the matrix L_r is no longer triangular, that is, we must reduce the number of factors below $n/2$ before the benefits of the reduced dimension become apparent. Ito processes are implemented only in the C++ code (ItoProcess.h).

Chapter 4

Asset Pricing and Hedging

4.1 Single Asset Markets

We now turn to the simulation of the price processes of financial assets. In the simplest case we consider a market consisting of only two assets S and B referred to as the *risky asset* and the *risk free bond* respectively. $B(t)$ represents the value at time t of one dollar invested at time $t = 0$ into a riskless money market account and the inverse $1/B(t)$ is the discount factor which discounts cash prices from time t back to time zero. The cash price $B(t)$ is assumed to satisfy

$$B(t) = exp\left(\int_0^t r(s)ds\right), \tag{4.1}$$

where $r(t)$ is the so called *short rate process* , that is, $r(t)$ is the rate charged at time t for risk free lending over the infinitesimal interval $[t, t + dt]$. The purpose of the riskfree bond is to allow us to take proper account of the time value of money. To do this automatically we work with the *discounted price* $S(t)$ of the asset S instead of the cash price $\overline{S}(t)$. The discounted price $S(t)$ is the quotient

$$S(t) = \overline{S}(t)/B(t)$$

and is the price of the asset S at time t expressed in constant time zero dollars. The discounted price $S(t)$ can also be viewed as the price of the asset S at time t expressed in a new unit of account: one share of the riskfree bond B. When used in this role the riskfree bond B is also called the *numeraire asset* .

Switching from cash to B as the new numeraire automatically takes proper account of the time value of money, that is, eliminates interest rates

from explicit consideration. For this reason we will always work with discounted prices $S(t)$ rather than cash prices $\overline{S}(t) = S(t)B(t)$.

The processes $S(t)$, $B(t)$ are assumed to live in continuous time defined on some probability space $(\Omega, \mathcal{F}, P)$ endowed with a filtration $(\mathcal{F}_t)_{0\leq t\leq T}$. The σ-field represents the information available at time t and $E_t(\cdot)$ denotes the expectation conditioned on $\mathcal{F}_t$. The probability P is called the *market probability* and governs the realization of prices observed in the market. A second probability will be introduced below.

For the puropose of simulation continuous time will be discretized and sampled at evenly spaced times $t * dt$, $t = 0, 1, \ldots, T$, where dt is the size of the time step and T now denotes the number of time steps to the horizon. In this way time t becomes an integer variable and discrete time t corresponds to continuous time $t * dt$.

The dual use of T to denote both the continuous and the discrete time to the horizon should not cause any confusion. The context should make the meaning clear.

Risk neutral probability. Assume first that the asset S pays no dividend. The market using discounted prices consists of the two price processes: the constant process 1 and $S(t)$. If there exists a second probability Q which is equivalent to the market probability P and in which the discounted asset price $S(t)$ is a martingale, then there cannot be an arbitrage opportunity in the market of discounted prices.

We assume that such a probability exists and is uniquely determined. In many stochastic models of the discounted asset price $S(t)$ such a probability Q can found explicitly. The uniqueness is usually harder to prove and has the following consequence: the cash price $c(t)$ at time t of a random cashflow h occuring at time T is given by the expectation

$$c(t) = B(t)E_t^Q[h/B(T)].$$

This is the so called *martingale pricing formula* . The interesting point is that it is not the market probability P which is used in this. See Appendix C.1. If we replace cash prices by *discounted* prices the formula simplifies to

$$c(t) = E_t^Q(h).$$

Here h is a random cashflow at any time and $c(t)$ the value (price) of this cashflow at time t both discounted back to time zero.

Stochastic models of the asset price. Directly from its definition it follows that the riskless bond follows the dynamics

$$dB(t) = r(t)B(t)dt. \tag{4.2}$$

The discounted price $S(t)$ is assumed to follow an Ito process of the form

$$dS(t) = S(t)[(\mu(t) - r(t))dt + \sigma(t)dW(t)], \tag{4.3}$$

where $W(t)$ is a one dimensional Brownian motion under the market probability P and the stochastic processes $\mu(t)$ and $\sigma(t)$ satisfy suitable integrability conditions. The process $\mu(t)$ is called the *drift* of the asset S (before discounting) and $\sigma(t)$ the *volatility* of S (under the market probability).

The process $log(S(t))$ measures the (continuously compounded) *returns* on the asset S in excess of the riskfree rate (returns after inflation). By Ito's rule

$$d\,logS(t) = (\mu(t) - r(t) - \tfrac{1}{2}\sigma^2(t))dt + \sigma(t)dW(t). \tag{4.4}$$

Switching to the risk neutral measure Q affects only the drift but not the volatility, and since $S(t)$ is a martingale and hence driftless under Q we have

$$dS(t) = S(t)\sigma(t)dW^Q(t) \tag{4.5}$$

which, after taking logarithms, becomes

$$d\,logS(t) = -\frac{1}{2}\sigma^2(t)dt + \sigma(t)dW^Q(t). \tag{4.6}$$

Here $W^Q(t)$ is a Brownian motion under the risk neutral measure Q. It is known how this Brownian motion is related to the Brownian motion $W(t)$ under P but we do not need this relation.

Dividends. Assume now that the asset pays a *dividend* continuously with yield $q(t)$. We eliminate the dividend by reinvesting it immediately into the asset S again thereby obtaining another asset S_1 which pays no dividend.

Let us switch to the cash price $\overline{S}(t)$ as a new numeraire. In this new numeraire the cash price $\overline{S}_1(t)$ looks like a money market account with short rate $q(t)$, that is, $\overline{S}_1(t)/\overline{S}(t) = exp\left(\int_0^t q(s)ds\right)$. Since the discounted prices $S(t)$, $S_1(t)$ satisfy $S_1(t)/S(t) = \overline{S}_1(t)/\overline{S}(t)$ we have

$$S(t) = exp\left(-\int_0^t q(s)ds\right) S_1(t). \tag{4.7}$$

In particular it follows that S has the same volatility as S_1 only the drift is changed. But since S_1 is a dividend free asset, the discounted price $S_1(t)$ is a martingale under the risk neutral probability and so driftless. Consequently $dS_1(t) = S_1(t)\sigma(t)dW^Q(t)$ from which it follows that

$$dS(t) = S(t)[-q(t)dt + \sigma(t)dW^Q(t)] \tag{4.8}$$

Upon taking logarithms this assumes the form

$$d\,logS(t) = (-q(t) - \tfrac{1}{2}\sigma^2(t))dt + \sigma(t)dW^Q(t). \tag{4.9}$$

The emergence of the term "$-q(t)S(t)dt$" in 4.8 can also be understood as follows: When a dividend is disbursed the price of the asset has to be reduced by the amount of dividend paid out. Now, during the infinitesimal interval $[t, t+dt]$, $q(t)S(t)dt$ is paid out in dividends (discounted value) and this quantity must therfore be subtracted from the asset price $S(t+dt)$. Here the dividend yield $q(t)$ can be a stochastic process but we will only implement models where the dividend yield is a constant q.

Set $X(t) = exp(qt)S(t)$. An easy application of the product rule shows that $dX(t) = X(t)\sigma(t)dW^Q(t)$. Thus $X(t)$ is a martingale under the risk neutral probability Q. Let $E_t^Q(\cdot)$ denote the expectation under the probability Q conditioned on the information generated up to time t. Then the martingale condition for $X(t)$ can be written as $E_t^Q[exp(qT)S(T)] = exp(qt)S(t)$, equivalently

$$E_t^Q[S(T)] = exp(-q(T-t))S(t), \qquad t \in [0,T]. \tag{4.10}$$

The discounted price $S(T)$ at the horizon will be used as the default control variate for European options written on the asset S and equation (4.10) gives us the conditional mean of the control variate.

To make the above stochastic model concrete the processes $r(t)$, $\mu(t)$ and $\sigma(t)$ have to be specified. Regardless of this specification a common feature is the fact that we deal with two probability measures, the market probability and the risk neutral probability and that the price dynamics depends on the chosen probability.

Thus it will be useful to introduce a flag whichProbability which indicates if the simulation proceeds under the market probability or under the risk neutral probability.

In a simulation of any of the above stochastic dynamics the infinitesimal increment $dW(t)$ of the driving Brownian motion is replaced with the increment $W(t+dt) - W(t)$ which is independent of the history of the path $u \in [0,t] \to W(u)$ of the Brownian path up to time t. In a discretization of time t as multiples of the time step $t = 0, dt, 2dt, ..., T * dt$, this can be rewritten as

$$W(t+1) - W(t) = \sqrt{dt}Z(t) \tag{4.11}$$

where $Z(0), \ldots, Z(T-1)$ is a sequence of independent standard normal variables. Thus a simulated path is driven by an array $Z[t]$, $t < T$, of independent standard normal deviates.

The $Z[t]$ will be referred to as the *Wiener increments* driving the price path. Recall that a Brownian motion is also called a *Wiener process* and that the $Z[t]$ are the normalized increments of the process $W(t)$. The generation of these Wiener increments is computationally costly. However, once such a sequence has been generated

$$\pm Z[0], \pm Z[1], \ldots, \mp Z[T-1] \tag{4.12}$$

is another such sequence, for any of the 2^T possible distributions of signs. This suggests that we generate paths in groups of dependent paths where each group is based on the same sequence of standard normal increments subject to a certain fixed number of sign changes. These sign changes should include the path based on the increments

$$-Z[0], -Z[1], \ldots, -Z[T-1], \tag{4.13}$$

the so called *antithetic image* of the original path. In addition to save computational cost in the generation of normal deviates this device also serves to reduce the variance of random variables which are functionals of the price path. More precisely it is hoped that the means of such functionals over groups of dependent paths display significantly reduced variance when compared to the values of the functional computed from independent paths.

This topic has already been considered in the treatment of random variables in Chapter 1. Our class RandomVariable has methods for the computation of a Monte Carlo mean which take proper account of the dependence of paths in a group which is based on sign changes of the same sequence of Wiener increments. This now leads to the definition of the following basic setup:

```
public abstract class Asset{

    int T;                              // time steps to horizon
    double dt;                          // size of time step
    double S0;                          // asset price at time t = 0
    double q;                           // constant dividend yield
    boolean volatilityIsDeterministic;  // is the volatility deterministic?

    // number of sign changes before another array of standard
    // normal deviates driving the asset price path is computed
    int nSignChange;

    double[] Z;             // array of Z-increments driving the price path
    double[] B;             // price path of the risk free bond
    double[] S;             // the discounted price path S[t]=S(t*dt)

    //constructor
    public Asset(int T, int dt, double S0, double q, int nSignChange)
    { /* see below */ }

    // compute B[t+1], S[t+1] from B[u], S[u], u<=t.
    public abstract void timeStep(int whichProbability, int t);

    // other methods

} // end Asset
```

The constructor allocates the arrays and initializes fields:

```
public Asset(int T, int dt, double S0, double q)
{
    this.T=T; this.dt=dt;
    this.S0=S0; this.q=q;
    this.nSignChange=nSignChange;
    volatilityIsDeterministic=false;      // the default

    // allocate the path arrays
    Z=new double[T];
    B=new double[T+1];
    S=new double[T+1];

    // initialize the paths B(t), S(t)
    B[0]=1;
    S[0]=S0;

} // end constructor
```

One could argue that the array Z[] of Wiener increments driving the asset price path and the number nSignChange of sign changes are implementation

details which should be moved to the various concrete subclasses. After all asset price models may not even rely on Brownian motions to generate asset price paths. We have chosen the above approach for reasons of exposition. Note that a concrete subclass of Asset need not actually use the array Z[]. The path generating methods can be defined as is deemed desirable.

Path computation. The only abstract method in the class Asset is timeStep. Thus an Asset becomes concrete as soon as this method is defined. In principle path computation can be reduced to repeated calls to timeStep. This approach is implemented as the default. It obviously is not the most efficient approach (each function call carries some overhead). Moreover it is akward to keep track of Z-increments and sign changes in this case.

Methods for computing paths are intended to be overidden in concrete subclasses. The default methods are useful for quick experimentation.

Remember that the basic path computation is the continuation of a path which exists up to time t from this time t to the horizon. This computes branches of the existing path at time t. If t is current time, the path up to time t represents the available information at time t and branches of this path at time t represent future scenarios conditioned on this information.

```
public void newPathBranch(int whichProbability, int t)
{ for(int u=t; u<T; u++) timeStep(whichProbability,u); }
```

The following method newPath is intended to compute a new path not reusing the Wiener increments through sign changes (thus ensuring that the new path is independent of previous paths). Therefore we cannot simply implement it as newPathBranch(whichProbability,0) since the method newPathBranch is intended to be overidden so as to reuse Z-increments:

```
public void newPath(int whichProbability)
{ for(int t=0; t<T; t++) timeStep(whichProbability,t); }
```

Only very rarely does a stochastic model force small time steps dt on us for the sake of accuracy. More likely small time steps are taken because the price path has to be observed at all times. Small time steps slow down the simulation. Thus it is more efficient to move directly in one step from one time t to the next time $s > t$ when the price path must be observed again.

Exactly how this is accomplished with satisfactory accuracy depends on the stochastic model. This suggests that we declare such a method but not define it. On the other hand one often wants to do some quick experimentation without worrying about efficiency. Therfore we provide a convenience default implementation which reverts to individual time steps. It is intended to be overridden in concrete subclasses:

```
public void timeStep(int whichProbability, int t, int s)
{ for(int u=t; u<s; u++) timeStep(whichProbability,u); }
```

In the class ConstantVolatilityAsset for example this is overridden by a far more efficient method without loss of accuracy.

Triggers. On occasion, as the price of some asset evolves, we might be watching for some *signal* or *trigger* to initiate some action. For example a hedger might watch for the time t when a rebalancing of the hedge position is indicated or a speculator might be watching for sell or buy signals. The class trigger has been introduced for this purpose:

```
public abstract class Trigger{

    int T;      // time steps to horizon
    public Trigger(int T){ this.T=T; }   // constructor

    public abstract boolean isTriggered(int t, int s);

} // end Trigger
```

The reader might wonder why there are two times t and s instead of only one. Frequently a price path will be moved from one trigger event to the next. In this context t should be interpreted as the last time the event was triggered and isTriggered(t,s) returns true if $s > t$ is the first time the event is triggered again. For example if the event is a doubling in the discounted price of the asset S we might define

```
isTriggered(int t, int s){ return S[s]==2*S[t]; }
```

It is frequently necessary to compute a path forward from current time t to the time $s > t$ when the action is triggered. Moreover one then also wants to know at which time s this occurred:

```
public int pathSegment(int whichProbability, int t, Trigger trg)
{
    int s=t;
    do{ timeStep(whichProbability,s); s++; }
    while( (s<T) && !trg.isTriggered(t,s) );
    return s;
}
```

Note that the computation of the first time $s > t$ when the action is triggered is important but is not the primary objective here. The computation of the

path from time t to this time s which occurs as a side effect to the function call is the primary objective.

In a sense a Trigger defines a family τ_t of stopping times $\tau_t \geq t$. The method isTriggered(t,s) then returns true exactly if the event $[\tau_t = s]$ occurs. We have already encountered such families of stopping times in the analysis of the optimal stopping problem 3.6 and will do so again in the related pricing of American options 4.6.

If the short rate $r(t)$ is a stochastic process the computation of a price path involves both the price $B(t)$ of the risk free bond and the discounted price $S(t)$ of the asset. From now on the *short rate* will be assumed to be *deterministic*. The risk free bond then has only one path which can be computed by the constructor of the corresponding subclass of Asset. The path computation then simplifies to the computation of the path of the discounted asset price.

Forward price. The forward price of the asset S is the cost of delivering one share of S at time T by buying the asset immediately and holding it to the horizon T expressed in constant dollars at time $t = T$.

If S pays no dividend we simply buy one share of S at the current time t for $S(t)$ time zero dollars, equivalently, $B(T)S(t)$ dollars deliverable at time T, and hold this position until time T. If S pays a continuous dividend at the rate $q(t)$ we reinvest all dividends immediately into the asset S.

Thus if $a(t)$ shares of S are bought at time t, $a(t)exp\left(\int_t^T q(s)ds\right)$ shares are held at time T. Since we wish to deliver exactly one share of S at time T, we need to buy

$$a(t) = exp\left(-\int_t^T q(s)ds\right) \tag{4.14}$$

shares of S at time t for a cost of $a(t)B(T)S(t)$ time T dollars. The factor $a(t)$ corrects for the dividend and so is called the *dividend reduction factor*. Recall that we are restricting ourselves to the case of a constant dividend yield q. In this case the factor a assumes the form $a(t) = exp(-q(T-t))$. This motivates the following two methods:

```
public double dividendReductionFactor(int t)
{ return Math.exp(-q * (T-t) * dt); }

public double forwardPrice(int t)
{ return S[t] * B[T] * dividendReductionFactor(t); }
```

Note that the array element S[t] is the *discounted price* $S(t * dt)$ and that continuous time $T-t$ appears as $(T-t) * dt$ after discretization. At the cost of providing trivial or inefficient default implementations we have been able

to keep the number of abstract methods in the class Asset to one: timeStep is the only abstract method. We did this in order that an Asset could be allocated quickly by merely defining this method.

This also explains the following seemingly absurd definition of a method intended to return $\sigma(t)\sqrt{dt}$ whenever this is defined:

```
public double get_sigmaSqrtdt(int t)
{    System.err.println("Asset: sigma(t) * sqrt(dt) undefined in present context");
     System.exit(0);
}
```

Obviously the above code is only executed if the method is called from a subclass which does not provide a definition of this method and then it is exactly appropriate.

Some other methods are declared in the class Asset and will be discussed in the appropriate context. In the next section we turn to the implementation of the simplest concrete Asset class.

4.2 Basic Black-Scholes Asset

The simplest single asset market arises if the short rate $r(t) = r$, drift $\mu(t) = \mu$ and volatility $\sigma(t) = \sigma$ are all assumed to be constants. In this case the riskless bond $B(t) = exp(rt)$ is nonstochastic and so can be computed by the constructor (there is only one path). From (4.4) and (4.9) the discounted price $S(t)$ satisfies

$$S(s) = S(t)\ exp[\alpha(s-t) + \sigma(W(s) - W(t))] \tag{4.15}$$

where $\alpha = \mu - r - \sigma^2/2$ for the market probability and $\alpha = -q - \sigma^2/2$ for the risk neutral probability. Here $W(s) - W(t) = \sqrt{s-t}\ Z$ where Z is a standard normal random variable. In discrete time s and t are replaced with s*dt, t*dt, where dt is the size of the time step and s, t nonnegative integers (the number of the time step). Consequently it is useful to store the following constants

$$\begin{aligned} \text{sigmaSqrtdt} &= \sigma\sqrt{dt} \\ \text{marketDriftDelta} &= (\mu - r - \sigma^2/2)dt \quad \text{and} \\ \text{riskNeutralDriftDelta} &= (-q - \sigma^2/2)dt. \end{aligned}$$

The driftDelta is the change in the drift of the return process $log(S(t))$ over a single time step dt in the respective probability. Recalling that the array S[] contains the *discounted* asset price S[t]=S(t*dt) and that Random.STN() delivers a standard normal deviate the time step $s \to t$ assumes the form

```
Z=Random.STN();
S[s]=S[t]*Math.exp(driftDelta*(s-t)+sigmaSqrtdt*Math.sqrt(s-t)*Z);
```

where driftDelta is set according to the probability. Even though the time step $t \to s$ may be arbitrarily large there is no approximation involved. The ability to make large time steps if the price process has to be observed only at a few times greatly speeds up any simulation.

The case $s = t + 1$ is of particular importance since a complete path is composed of exactly these time steps. In this case we want to store the standard normal increments Z used in the time steps along with the asset price path since these increments will be used in the computation of hedge weights below. Assuming that these increments are in the array Z[] the single time step $t \to t + 1$ assumes the form

```
S[t+1]=S[t]*Math.exp(driftDelta+sigmaSqrtdt*Z[t]);
```

The array Z[] is filled with the necessary standard normal increments before a new path or path branch of the discounted asset price is computed:

```
public class ConstantVolatilityAsset extends Asset{

    int pathCounter;              // paths in current simulation of full paths
    int branchCounter;            // branches in current simulation of path-branches

    double r;                     // constant short rate
    double mu;                    // drift under market probability
    double sigma;                 // volatility

    double riskNeutralDriftDelta; // -(q + σ^2/2)dt
    double marketDriftDelta;      // (μ - r - σ^2/2)dt
    double sigmaSqrtdt;           // σ√dt

    //constructor
    public ConstantVolatilityAsset
    (int T, double dt, int nSignChange, double S0, double r, double q,
     double mu, double sigma)
    {
        super(T,dt,S0,q,nSignChange);

        this.r=r; this.mu=mu; this.sigma=sigma;
        pathCounter=0; branchCounter=0;

        riskNeutralDriftDelta=-(q+sigma*sigma/2)*dt;
        marketDriftDelta=(mu-sigma*sigma/2)*dt;
        sigmaSqrtdt=sigma*Math.sqrt(dt);

        // compute the nonstochastic riskfree bond
        for(int t=0; t<=T; t++) B[t]=Math.exp(r*t*dt);

    } //end constructor

    // other methods

} // end ConstantVolatilityAsset
```

The important other methods are of course methods to compute asset price paths or branches of such paths. Assume that we wish to generate branches of a path which exists up to time t and branching is to occur at time t. Each branch simply continues the existing path to the horizon and is computed from standard normal increments Z[u], u=t,t+1,...,T-1. A full path simulation corresponds to the case $t = 0$.

Recall also that we intend to compute paths (or branches) in groups of size nSignChange where within each group only the signs of the standard

normal increments Z[u] are changed. Thus paths within the same groups are not independent. Paths are only independent across distinct groups. This does not affect the computation of (conditional) means of random variables which are functionals of a path but it does affect the computation of standard deviations of these. This issue has already be dealt with in Chapter 1.

In the present context this makes it necessary to be able to separate distinct groups as a simulation is in progress. The variables pathCounter and branchCounter serve this purpose for paths respectively branches. Our setup does not extend to branches of branches (for this we woud need a branchBranchCounter). The reason is that we will never split paths repeatedly. In fact such a computation is not realistic with affordable computational resources. The variables pathCounter and branchCounter have to be reset to zero at the begining of each new path respectively branch simulation:

```
public void simulationInit(int t)
{
    if(t==0) pathCounter=0;      //it's a full path simulation
    else branchCounter=0;        //it's a simulation of path branches
}
```

In order to compute a path branch from time t to the horizon T we need standard normal increments Z[t],Z[t+1],...,Z[T-1]. The following routine generates these cycling through sign changes.

```
public void newWienerIncrements(int t)
{
    int m;                           //counts position in the cycle of sign changes
    if(t==0) m=pathCounter; else m=branchCounter;

    switch(m % nSignChange)
    {
        case 0 : //new standard normal increments
                 for(int u=t; u<T; u++) Z[u]=Random.STN();
                 break;
        case 1 : //increments driving the anthithetic path
                 for(int u=t; u<T; u++) Z[u]=-Z[u];
                 break;
        default: //change signs randomly
                 for(int u=t; u<T; u++) Z[u]*=Random.Sign();
    }

    if(t==0) pathCounter++; else branchCounter++;

} // end NewWienerIncrements
```

Once we have these increments a new path branch can be computed:

```
public void newPathBranch(int whichProbability, int t)
{
    double driftDelta=0;
    switch(whichProbability)
    {
        case Flag.MARKET_PROBABILITY: driftDelta=marketDriftDelta; break;
        case Flag.RISK_NEUTRAL_PROBABILITY: driftDelta=riskNeutralDriftDelta;
    }

    //compute the standard normal increments driving the branch
    newWienerIncrements(t);
    for(int u=t; u<T; u++) S[u+1]=S[u]*Math.exp(driftDelta+sigmaSqrtdt*Z[u]);

} // end newPathBranch
```

The computation of a single time step is similar. A new standard normal increment driving the time step is computed. Since some routines (hedge coefficients) make use of this increment it is stored in the array Z[]:

```
public void timeStep(int whichProbability, int t)
{
    // compute and store the new standard normal increment
    Z[t]=Random.STN();

    //choose drift according to probability
    double driftDelta=0;
    switch(whichProbability)
    {
        case Flag.MARKET_PROBABILITY: driftDelta=marketDriftDelta; break;
        case Flag.RISK_NEUTRAL_PROBABILITY: driftDelta=riskNeutralDriftDelta;
    }

    S[t+1]=S[t]*Math.exp(driftDelta+sigmaSqrtdt*Z[t]);

} // end timeStep
```

The computation of larger time steps and full paths is similar. The reader is invited to read the source code in Market.ConstantVolatilityAsset.java.

4.3 Markov Chain Approximation

The pricing of American options leads to optimal stopping problems (early option exercise). Such problems can be dealt with easily in the context of a Markov chain. In fact the class StoppableMarkovChain implements the optimal

stopping time which maximizes the expected reward from stopping the chain (see Section 2.5).

The discounted price process $S(t)$ of a ConstantVolatilityAsset is a Markov process but it is not a Markov chain even after the time domain has been discretized. In order to be able to deal with early option exercise we approximate $S(t)$ with a Markov chain by discretizing the range of $S(t)$ also. Recall that the process $S(t)$ satisfies

$$dS(t) = S(t)[-qdt + \sigma dW^Q(t)] \tag{4.16}$$

under the risk neutral probability Q, where q denotes the constant dividend yield of the asset S. Thus, for a time step of size $dt > 0$,

$$\begin{aligned} S(t+dt) - S(t) &= \int_t^{t+dt} S(u)[-qdu + \sigma dW^Q(u)] && (4.17)\\ &\cong S(t)[-qdt + \sigma(W^Q(t+dt) - W^Q(t)) && (4.18)\\ &= S(t)[-qdt + \sigma\sqrt{dt}\ Z], && (4.19) \end{aligned}$$

where $Z = (W^Q(t+dt) - W^Q(t))/\sqrt{dt}$ is standard normal and independent of the information generated up to time t. Here we have replaced $S(u)$ with the initial value $S(t)$ in the integral to obtain the approximation. Rearranging terms we can rewrite this as

$$S(t+dt) = S(t)[1 - qdt + \sigma\sqrt{dt}\ Z] \tag{4.20}$$

with Z as above. The Markov chain approximation of $S(t)$ is now obtained by subdividing the range of $S(t)$ into intervals $[jds, (j+1)ds)$, $j = 0, 1 \ldots$ and collapsing all the values in $[jds, (j+1)ds)$ to the midpoint $(j + \frac{1}{2})ds$. Here $ds > 0$ is the mesh size of the range grid.

The state $X(t) = i$ of the approximating Markov chain X is interpreted as $S(t) = (i + \frac{1}{2})ds$ and the chain transitions from state i at time t to state j at time $t + dt$ with probability

$$q(t, i, j) = Prob(\ S(t+dt) \in [jds, (j+1)ds] \mid S(t) = (i + \tfrac{1}{2})ds\ ) \tag{4.21}$$

Setting $q(t, s, a, b) = Prob(\ S(t+dt) \in [a, b] \mid S(t) = s\ )$ we can rewrite 4.21 as

$$q(t, i, j) = q(t, (i + \tfrac{1}{2})ds, jds, (j+1)ds). \tag{4.22}$$

Assume that $S(t) = s$ and set $v = s(1 - qdt)$. Then $S(t+dt) - v \cong s\sigma\sqrt{dt}\ Z$ according to 4.20 and it follows that

$$\begin{aligned} S(t+dt) \in [a, b] &\iff S(t+dt) - v \in [a - v, b - v]\\ &\iff Z \in \left[\frac{a-v}{s\sigma\sqrt{dt}}, \frac{b-v}{s\sigma\sqrt{dt}}\right] \end{aligned}$$

and so $q(t,s,a,b) = N((b-v)/s\sigma\sqrt{dt}) - N((a-v)/s\sigma\sqrt{dt})$, where N is the cumulative standard normal distribution function as usual. Entering this into 4.22 yields the transition probabilities as

$$q(t,i,j) = N(f_+) - N(f_-), \quad \text{where} \tag{4.23}$$

$$f_\pm = f_\pm(i,j) = \frac{j - i \pm 0.5}{(i+0.5)\sigma\sqrt{dt}} + q\sqrt{dt}/\sigma \tag{4.24}$$

Working with these probabilities spreads the possible values of the chain over all nonnegative integers. This can induce lengthy computations. Thus we introduce an upper and lower bounds $j_{max}(i)$, $j_{min}(i)$ for the possible states j which can be reached from state i at time t. Note that

$$Prob(\ X(t+dt) \le j \mid X(t) = i\ ) = N(f_+(i,j)) \quad \text{and} \tag{4.25}$$

$$Prob(\ X(t+dt) \ge j \mid X(t) = i\ ) = 1 - N(f_-(i,j)), \tag{4.26}$$

Residual states j for which these probabilities are small will be collapsed into a single state. Fix a small cutoff probability $\delta > 0$ and set

$$j_{min}(i) = \max\{\ j \mid N(f_+(i,j)) < \delta\ \} \quad \text{and} \tag{4.27}$$

$$j_{max}(i) = \min\{\ j \mid 1 - N(f_-(i,j)) < \delta\ \} \tag{4.28}$$

Conditional on the state $X(t) = i$ at time t all states $j \le j_{min}(i)$ are then collapsed into the state $j_{min}(i)$ and all states $j \ge j_{max}(i)$ are collapsed into the state $j_{max}(i)$, that is, the transition probabilities are adjusted as follows:

$$q(t,i,j) = \begin{cases} N(f_+(i,j)) & \text{if} \quad j = j_{min} \\ N(f_+(i,j)) - N(f_-(i,j)) & \text{if} \quad j_{min} < j < j_{max}(i) \\ 1 - N(f_-(i,j)) & \text{if} \quad j = j_{max}(i) \\ 0 & \text{if} \quad else \end{cases} \tag{4.29}$$

Observing that $N(f_+(i,j)) = N(f_-(i,j+1))$ it is then easy to check that

$$\sum_{j=0}^{\infty} q(t,i,j) = \sum_{j=j_{min(i)}}^{j_{max(i)}} q(t,i,j) = 1. \tag{4.30}$$

The transition probabilities $q(t,i,j)$ are independent of time t and so $X(t)$ is a stationary finite state Markov chain. When implemented as an SFS-MarkovChain with N possible states $j = 0, 1, \ldots, N-1$ the member functions a(i), b(i) are given by

$$a(i) = j_{min}(i) \qquad b(i) = \min\{\ N-1, j_{max}(i)\ \} \tag{4.31}$$

The class ConstantVolatilityAsset implements the method markovChain which allocates the Markov chain approximating the discounted asset price process. The approximating Markov chain is implemented as an object of class SFSMarkovChainImpl, that is, the transition probabilities $q(i, j)$ are precomputed at initialization and stored in the matrix Q of type double[][]. This matrix tends to occupy hundreds of megabytes of memory. In this regard see the exercise below.

The main method of the class MC_Asset_Test is a test program to compare the Monte Carlo price of a European call computed from the Markov chain approximation of the asset price with its analytic price. Tests show the following: the cutoff probability δ should be kept no larger than $\delta = 0.01$. The number of states nStates of the chain has to be limited or else the transition matrix Q (which is already made as small as possible only storing nonzero entries) consumes too much memory. Thus the mesh of the range grid has to be chosen larger. How large depends on the context, that is, how large the asset price $S(T)$ is likely to become which in turn depends on $S(0)$ and the volatility σ. The value ds=0.5 still yields reasonable approximations.

Exercise. The method markovChain in the class ConstantVolatilityAsset computes the Markov chain approximating the asset price as an object of class SFSMarkovChainImpl, that is, the transition probabilities $q(i, j)$ are precomputed at initialization and stored in the matrix Q of type double[ ][ ].

The idea is to speed up the retrieval of these transition probabilities. Instead of having to compute them repeatedly we simply read them from system memory. However the matrix Q is far too large to fit in the cache of the microprocessor and so resides in main memory. Access to that memory is rather slow.

Implement a second method markovChain_1 in the class ConstantVolatilityAsset which computes the approximating Markov chain as an object of type SFSMarkovChain. Although this class is abstract it can be instantiated by calling its constructor and defining all its abstract methods in the body of the constructor call.

All that is required is a slight rewrite of the method markovChain. Then expand the test program MC_Asset_Test by computing the call price using both approximating Markov chains and time the computation to see which approach is faster.

4.4 Pricing European Options

An *option* on some asset S is a financial instrument deriving a *nonnegative* payoff h in some way from the price path $t \in [0,T] \to S(t)$ of the asset (the so called *underlying*) from the present to the time T of *expiry*.

The contract can specify payouts at several points in time. Taking proper account of the time value of money these payouts can be moved forward to the time T of expiry and aggregated there as a single equivalent payment at the time of expiry. We can therefore assume the option payoff is a single payment at time T.

If the option is of *European* style the option holder does not have the ability to influence the payout in any way. If it is of *American* style the option holder can influence the payoff by *exercising* the option at any time prior to the time T of expiration. Option exercise forces the calculation of the payout from the state of the asset price path at the time of exercise in a way which is specified in the option contract. Quite often the value of the asset price at the time of exercise alone determines the payout.

Thus the payout of a European option is a deterministic function of the asset price path $t \in [0,T] \to S(t)$ alone while the payoff of an American option is a function of the asset price path and the exercise strategy of the option holder.

We shall now restrict attention to European options. Frequently the discounted payoff h depends only on the value $S(T)$ of the asset at the time T of expiration, that is $h = f(S(T))$, for some function $f = f(s)$. In this case the option is called *path independent*.

The prudent seller of an option will try to *hedge* her short position in the option by trading in the underlying and the riskfree bond with the aim to replicate the option payoff. Ideally the dynamic portfolio which is maintained requires an investment only at the time of inception and sustains itself thereafter, that is, the cost each new hedge position is exactly equal to the proceeds of the preceeding hedge position. In this case the trading strategy is called *selffinancing*.

The simplest example of such a portfolio buys one share of the asset S at time zero and holds this position to expiration without further trading. This portfolio has discounted price process $S(t)$ which is a *martingale* under the risk neutral probability Q. Under weak assumptions it can then be shown that the discounted price process of each selffinancing portfolio is a martingale in the risk neutral probability.

The option is called *replicable* if the payoff can be replicated by a self-financing dynamic portfolio. Let $C(t)$ denote the discounted price of the

option at time t. In order to avoid arbitrage (riskless profits with probability greater than zero) the option price must equal the price of the replicating portfolio at all times $t \leq T$.

Consequently the discounted option price $C(t)$ is a martingale under the risk neutral probability Q. However at the time of expiration the option price equals the payoff and so $c(T) = h$. The martingale property of the process $C(t)$ now implies that the discounted option price $C(t)$ at time t is given by the conditional expectation

$$C(t) = E_t^Q[h]. \tag{4.32}$$

The subscript t indicates that the expectation is conditioned on information available at time t, that is, the path of the asset and the riskfree bond up to time t. The superscript Q indicates that the expectation is taken with respect to the risk neutral probability Q. Note also that h is the option payoff discounted back to time zero.

Our treatment is grossly simplified and omits several subtleties. It is included here only to motivate the appearance and significance of the risk neutral probability Q and the martingale pricing formula (4.32). The reader is referred to the financial literature for the details.

The martingale pricing formula (4.32) makes the computation of the price of a European option amenable to Monte Carlo simulation: all we have to do is to implement the discounted option payoff h as an object of class RandomVariable and we can then draw on methods from the class RandomVariable to compute conditional expectations.

When the discounted option payoff is implemented as a RandomVariable, the risk neutral probability enters as follows: the payoff is computed as a deterministic function of the asset price path. In order to base samples on the risk neutral probability Q paths are generated following the dynamics of the asset price under Q. This dynamics is different from the dynamics of the asset price under the market probability in general.

Samples are conditioned on the information available at time t as follows: at time t the paths of the asset price and the riskfree bond are known up to time t. Rather than generating arbitrary paths we restrict the simulation to paths which follow the observed path up to time t (branches of the observed path where branching occurs at time t).

Such branches are computed by simply continuing the observed path from time t to the time T of option expiry. The asset price after the time of expiry is not relevant and so the horizon of the simulation can be set equal to T:

```
public abstract class Option
{
    int T;                          // time steps to horizon
    double dt;                      // size of time step
    Asset underlying;               // the underlying
    double[] C;                     // the option price path

    // constructor
    public Option(Asset asset)
    {
        underlying=asset;
        T=underlying.get_T(); dt=underlying.get_dt();
        C=new double[T+1];
    }

    // payoff aggregated to time T
    public abstract double currentDiscountedPayoff();

    // other methods

} // end Option
```

The only abstract method is currentDiscountedPayoff (the option payoff computed from the current path of the underlying asset $S(t)$ discounted back to time $t = 0$). An Option becomes concrete as soon as this method is defined. The next method allocates the discounted payoff as a RandomVariable:

```
public RandomVariable discountedPayoff( )
{
   return new RandomVariable(){

       public double getValue(int t)
       {
           underlying.newPathBranch(Flag.RISK_NEUTRAL_PROBABILITY,t);
           return currentDiscountedPayoff();
       }

   }; // end return new

} // end DiscountedPayoff
```

Here getValue(t) is a draw from the distribution of currentDiscountedPayoff conditioned on information at time t and hence is to be computed from a branch of the current asset price path where branching occurs at time t.

Having the discounted option payoff as a random variable is very useful if we want to compute discounted option prices as Monte Carlo means (under the risk neutral probability). In fact we might try to improve on this by

implementing the discounted payoff as a *controlled* random variable. The control variate cv must be correlated with the discounted option payoff and the conditional mean $E_t^Q(cv)$ must be known. See Chapter 1.

In this generality only the discounted price $cv = S(T)$ of the underlying at expiration answers this purpose. We can hope for some correlation with the discounted option payoff although the correlation may be weak in the case of a path dependent option. The conditional mean $E_t^Q(cv)$ is given by equation (4.10). The following allocates the discounted option payoff as a ControlledRandomVariable with this control variate:

```
public ControlledRandomVariable controlledDiscountedPayoff( )
{
    return new ControlledRandomVariable(){

        // the (value, control variate) pair
        public double[] getControlledValue(int t)
        {
            double[] S=underlying.get_S();    //discounted price path

            // path sample under Q conditioned on information at time t
            underlying.newPathBranch(Flag.RISK_NEUTRAL_PROBABILITY,t);

            double x=discountedPayoff(),      // value
                   cv=S[T];                   // control variate

            double[] value_control_variate_pair={ x,cv };

            return value_control_variate_pair;

        } // end getControlledValue

        // it remains only to define the control variate mean
        // conditioned on information at time t:

        public double getControlVariateMean(int t)
        {
            double[] S=underlying.get_S();
            double q=underlying.get_q();
            return S[t] * Math.exp(-q * (T-t) * dt);
        }

    }; // end return new

} // end controlledDiscountedPayoff
```

To test the quality of the control variate (how closely the discounted option payoff is correlated or anticorrelated with its control variate) we include

```
public double conditionalControlVariateCorrelation(int t, int nPaths)
{
    return controlledDiscountedPayoff().correlationWithControlVariate(t,nPaths);
}
```

This computes the above correlation conditioned on information available at time t from a sample of nPaths price paths of the riskfree bond and the underlying asset.

With this we can implement the martingale pricing formula for the discounted option price at any time t. If our default control variate for the discounted payoff does not perform well and you don't have a better one compute the option price as follows:

```
public double discountedMonteCarloPrice(int t, int nPath)
{
    underlying.simulationInit(t);
    return discountedPayoff().conditionalExpectation(t,nPath);

} // end discountedMonteCarloPrice
```

The price at time $t = 0$ is then given by

```
public double discountedMonteCarloPrice(int nPath)
{ return discountedMonteCarloPrice(0,nPath); }
```

However if we have a good control variate we use

```
public double controlledDiscountedMonteCarloPrice(int t, int nPath)
{
    underlying.simulationInit(t);
    return controlledDiscountedPayoff().conditionalExpectation(t,nPath);
}
```

Again the price at time $t = 0$ is given by

```
public double controlledDiscountedMonteCarloPrice(int nPath)
{ return controlledDiscountedMonteCarloPrice(0,nPath); }
```

All the methods from the classes RandomVariable respectively ControlledRandomVariable which compute conditional expectations can be transferred to the present context as above. In addition to these the class Option contains other methods related to hedging the option. These will be examined below.

You may wonder why one would want to compute such a price at any time t other than $t = 0$. For example if we want to simulate trading strategies involving the option and the underlying it maybe necessary to compute

the option price along with the price of the underlying. If the option price cannot be derived from the price of the underlying via an analytic formula and Monte Carlo simulation has to be used, such an endevour can be computationally expensive.

In case the option payoff depends only on the asset price $S(T)$ at expiry the computation of the martingale price can be sped up. When computing the option price at time t we can step to time T in as few time steps as accuracy will allow (often in a single time step). The abstract method Asset.timeStep(int t, int T) is intended to be implemented in this way. See for example how this is implemented in the class ConstantVolatilityAsset. The class PathIndependentOption overrides the methods of Option which compute Monte Carlo prices with these more efficient methods. The reader is invited to read the source code.

In order to make use of what we have we need a concrete option. The European call is a useful test case since an analytic formula exists for the call price against which we can test our Monte Carlo means.

4.5 European Calls

The European call with *strike* K and expiry T on the asset S has discounted payoff $h = (S(T) - K/B(T))^+$, where $x^+ = \max\{x, 0\}$. Recall that $S(T)$ is the *discounted* price of the asset S at time T.

If the discounted asset price $S(t)$ follows the dynamics (4.3), (4.5), (4.8) in Section 1 with *nonstochastic* volatility $\sigma(t)$ and short rate $r(t)$ the call price $C(t)$ can be computed as a deterministic function of the asset price $S(t)$ via an analytic formula. The formula is best expressed in terms of *forward prices*, that is, prices in constant time T dollars.

The forward price $F_Y(t)$ of any asset $Y(t)$ is the discounted price $Y(t)$ pushed forward to time T and reduced by the dividend payout over the interval $[t, T]$. In other words

$$F_Y(t) = a(t)B(T)Y(t), \tag{4.33}$$

where $a(t) = exp(-q(T-t))$ is the dividend reduction factor. The member functions dividendReductionFactor(int t) and forwardPrice(int t) in the class Asset compute these quantities. The forward price $F_C(t)$ of the call is now given by the following formula:

$$\begin{aligned} F_C(t) &= F_S(t)\, N(d_+) - K\, N(d_-), \quad \text{where} && (4.34)\\ d_\pm &= \Sigma(t)^{-1} log(F_S(t)/K) \pm \frac{1}{2}\,\Sigma(t,T) \quad \text{and} && (4.35)\\ \Sigma(t,T)^2 &= \int_t^T \sigma^2(t)dt. && (4.36) \end{aligned}$$

Here N is the cumulative standard normal distribution function and $\Sigma(t, T)$ is a measure of the aggregate volatility of the return process $log(S(t))$ over the remaining life $[t, T]$ of the call. Methods to compute the forward price $F_S(t)$ and the quantity $\Sigma(t)$ have been implemented in the class Asset. Note however that the above call price formula is only valid in the case of a nonstochastic volatility $\sigma(t)$. Introducing the function

$$\begin{aligned} \mathsf{blackScholesFunction}(Q, K, \Sigma) &= QN(d_+) - K\, N(d_-), && (4.37)\\ d_\pm &= \Sigma^{-1} log(Q/K) \pm \frac{1}{2}\Sigma && (4.38) \end{aligned}$$

(implemented in the class Statistics.FinMath) the call price assumes the form

$$F_C(t) = \mathsf{blackScholesFunction}(F_S(t), K, \Sigma(t,T)) \tag{4.39}$$

See Appendix C.4 for a derivation.

```
public class Call extends PathIndependentOption
{
      double K; // strike price

      //constructor, aaset is the underlying
      public Call(double K)
      {
        super(asset,"Call");
        this.K=K;

      } // end constructor

      //Payoff computed from current path of underlying.
      public double currentDiscountedPayoff()
      {
        double[]   S=underlying.get_S(),       //price path
                   B=underlying.get_B();       //riskfree bond
        double x=S[T]-K/B[T];
        return (x>0)? x:0;
      }

      // analytic price
      public double discountedAnalyticPrice(int t)
      {
        double     Q=underlying.forwardPrice(t),
                   Sigma=underlying.Sigma(t);
        double[]   B=underlying.get_B(); //riskfree bond
        return FinMath.blackScholesFunction(Q,K,Sigma)/B[T];
      }

      //other methods

} // end Call
```

The method discountedAnalyticPrice computes a valid price only if the volatility of the underlying is deterministic. Since there is no restriction on the underlying we need a boolean flag hasAnalyticPrice which is set by examining the type of the underlying. The class Call contains several other methods which will be introduced in the proper context.

Example 1 We compute the correlation of the default control variate with the call payoff and the Monte Carlo price of the call and compare this price to the analytic call price. The underlying will be a ConstantVolatilityAsset:

```
public class CallTest{

    public static void main(String[] args)
    {
        int T=50,                 // time steps to horizon
            nPaths=20000,         // paths in simulation
            nSignChange=5;        // path group size (irrelevant here)

        double S_0=50,            // asset price at time t = 0
               mu=0.24,           // market drift of asset
               sigma=0.31,        // volatility of asset
               q=0.04,            // dividend yield of asset
               r=0.07,            // riskfree rate
               K=55,              // call strike
               dt=0.02;           // size of time step

        ConstantVolatilityAsset
        asset=new ConstantVolatilityAsset(T,dt,nSignChange,S_0,r,q,mu,sigma);
        Call call=new Call(K,asset);

        double cv_corr=call.controlVariateCorrelation(3000);
        String Str="Call control variate correlation = "+cv_corr;
        System.out.println(Str);

        double an_price=call.discountedAnalyticPrice(0);
        Str="analytic price = "+an_price; System.out.println(Str);

        double mc_price=call.discountedMonteCarloPrice(nPaths);
        Str="Monte Carlo price = "+mc_price; System.out.println(Str);

        double cmc_price=call.controlledDiscountedMonteCarloPrice(nPaths);
        Str="Controlled Monte carlo price = "+cmc_price;
        System.out.println(Str);

    } // end main

} // end CallTest
```

4.6 American Options

American options introduce a new feature, *early exercise*. The holder of such a contract is entitled to *exercise* the option at any time on or before

the time of expiration and if the option is exercised at time t the option holder receives the *discounted* payoff $h_t \geq 0$.

Due to the nonnegativity of the payoff we can assume the option will in fact be exercised at the latest at the time of expiry. Exercise out of the money receives nothing but is not penalized.

In other words the problem of exercising and valuing American Options is nothing but the Optimal Stopping Problem with a finite time horizon and nonnegative rewards. However we will now investigate this problem in more detail.

As usual the σ-field $\mathcal{F}_t$ denotes the information available at time t and $E_t(\cdot)$ denotes the expectation $E^Q[\,\cdot \mid \mathcal{F}_t]$ under the risk neutral probability Q conditioned on all information $\mathcal{F}_t$ available at time t. All processes Z_t will be assumed to be adapted to the filtration $(\mathcal{F}_t)$, that is Z_t is $\mathcal{F}_t$-measurable (the value of Z_t known by time t).

In the case of the random variable h_t this merely means that you know what the payoff will be if you decide to exercise the option. The notation $h_t = h(t)$ will also be employed where convenient.

The option contract is specified by the process $h = (h_t)$ of discounted payoffs. In practice this process will typically be a deterministic function $h_t = \tilde{h}(X_t)$ of a *state vector process*. In the simplest case the state vector consists merely of the assets underlying the option but there could also be interest rates, exchange rates and other ingredients necessary to compute the option payoff at exercise time.

4.6.1 Price and optimal exercise in discrete time.

In simulation the processes h_t, X_t have to be discretized. Let $t = 0, 1, \ldots, T$ denote the time steps of the discretization. In order to determine the value of the American option we use backward induction from discrete time $t = T$. Let V_t denote the *discounted* value of the option at time t given that it has not been exercised before time t. Clearly we have $V_T = h_T$. Let now $t < T$ and note that we can do only one of two things at time t:

- exercise and receive h_t
- continue to hold and receive $E_t(V_{t+1})$.

Consequently

$$V_t = max\{\, h_t, E_t(V_{t+1})\,\}. \tag{4.40}$$

In the discussion of the continuation value below it will be seen that this recursion is not useful for the computation of the values V_t.

Exercise strategies and optimal exercise. Assume the current time is t and the option has not yet been exercised. The relevant exercise times are then exactly the stopping times ρ_t with values in $[t, T]$. Let $\mathcal{T}_t$ denote the family of all such stopping times ρ_t.

An exercise time $\rho_t \in \mathcal{T}_t$ decrees that the option will be exercised at time $\rho_t \geq t$ and under this strategy the option holder receives the discounted random payoff $h(\rho_t)$ with discounted value $E_t[h(\rho_t)]$ at time t.

An *exercise strategy* ρ is a sequence of exercise times $\rho = (\rho_t)$ with $\rho_t \in \mathcal{T}_t$, for all $t \in [0, T]$. Such a strategy will be called *consistent* if

$$\rho_t = \rho_{t+1} \quad \text{on the set} \quad [\rho_t > t].$$

For example if $B_t \subseteq R$ is a Borel set, for all $t \in [0, T]$, and Z_t any process then the stopping times ρ_t defined by

$$\rho_t = \min\{\, s \geq t \mid Z_s \in B_s \,\},$$

define a consistent exercise strategy $\rho = (\rho_t)$. Here it is understood that $\rho_t = T$ if the event $Z_s \in B_s$ does not happen for any time $s \in [t, T]$. From (4.40) it follows immediately that

$$E_t(V_{t+1}) \leq V_t, \quad \text{equivalently} \quad E_t(V_{t+1} - V_t) \leq 0, \quad \forall t \in [0, T]. \tag{4.41}$$

A process with this property is called a *supermartingale*. The supermartingale property implies that

$$E_t[V(\rho_t)] \leq V_t, \tag{4.42}$$

for all stopping times $\rho_t \in \mathcal{T}_t$. This is the simplest case of the Optional Sampling Theorem for sequential processes (see Appendix, D.1). Note that the supermartingale property (4.41) is the special case of (4.42) for the stopping time $\rho_t = t + 1$. Let us note that

Proposition 4.6.1 *V_t is the smallest supermartingale π_t satisfying $\pi_t \geq h_t$ for all $t \in [0, T]$.*

Proof. From the definition it is obvious that $V_t \geq h_t$ and we have already seen that V_t is a supermartingale. Let now π_t be any supermartingale satisfying $\pi_t \geq h_t$, for all $t \in [0, T]$. We have to show that $\pi_t \geq V_t$, for all $t \in [0, T]$, and use backward induction starting from $t = T$.

For $t = T$ the claim is true since $V_T = h_T$. Let now $t < T$ and assume that $\pi_{t+1} \geq V_{t+1}$. Then, using the supermartingale condition for π,

$$\pi_t \geq E_t[\pi_{t+1}] \geq E_t[V_{t+1}].$$

Since also $\pi_t \geq h_t$ (by assumption) we have $\pi_t \geq max\{\, h_t, E_t(V_{t+1})\,\} = V_t$, as desired. ∎

Optimal exercise time. Next we show that the following defines an optimal exercise time $\tau_t \in \mathcal{T}_t$ (starting at time t):

$$\tau_t = \min\{\, s \geq t \mid h_s = V_s\,\} \tag{4.43}$$

The time τ_t exercises the option as soon as option exercise pays off the current value of the option. Note that the stopping times τ_t form a consistent exercise strategy $\tau = (\tau_t)$. Let $\rho_t \in \mathcal{T}_t$ be any stopping time. Then, using $V_t \geq h_t$ and the supermartingale property of V_t we obtain $V_t \geq E_t[V(\rho_t)] \geq E_t[h(\rho_t)]$ by the Optional Sampling Theorem. Consequently

$$V_t \geq \sup_{\rho_t \in \mathcal{T}_t} E_t\left[h(\rho_t)\right] \geq E_t\left[h(\tau_t)\right]. \tag{4.44}$$

We now claim that

$$V_t = E_t\left[h(\tau_t)\right]. \tag{4.45}$$

This shows that $\rho_t = \tau_t$ is optimal among all stopping times $\rho_t \in \mathcal{T}_t$ and that equality holds in (4.44):

$$V_t = \sup_{\rho_t \in \mathcal{T}_t} E_t\left[h(\rho_t)\right] = E_t\left[h(\tau_t)\right]. \tag{4.46}$$

The proof of (4.45) will be by backward induction starting from $t = T$. For $t = T$ the equality (4.45) is clear since $\tau_T = T$. Let now $t < T$, assume that

$$V_{t+1} = E_{t+1}\left[h(\tau_{t+1})\right]$$

and employ the following trick: if α_t, β_t are $\mathcal{F}_t$ measurable and U_t is any random variable then

$$\begin{aligned} E_t\left[\alpha_t + \beta_t U_t\right] &= \alpha_t + \beta_t E_t[U_t] \\ &= \alpha_t + \beta_t E_t\left[E_{t+1}(U_t)\right]. \end{aligned}$$

By consistency $\tau_t = \tau_{t+1}$ on the event $B = [\tau_t > t]$ while $h(\tau_t) = h_t$ on the event $A = B^c = [\tau_t = t] = [h_t = V_t] = [h_t \geq E_t(V_{t+1})]$ and all these events are $\mathcal{F}_t$-measurable. Consequently

$$1_A h_t + 1_B E_t(V_{t+1}) = \max\{\, h_t, E_t(V_{t+1})\,\} = V_t$$

and so we can write

$$\begin{aligned} E_t\left[h(\tau_t)\right] &= E_t\left[1_A h(\tau_t) + 1_B h(\tau_t)\right] \\ &= E_t\left[1_A h_t + 1_B h(\tau_{t+1})\right] \\ &= 1_A h_t + 1_B E_t\left[E_{t+1}(h(\tau_{t+1}))\right] \\ &= 1_A h_t + 1_B E_t\left[V_{t+1}\right] = V_t. \end{aligned}$$

Here the second to last equality uses the induction hypothesis. Let us summarize these findings as follows:

Theorem 4.6.1 *Let V_t be the discounted value at time t of an American option with discounted payoff h_s if exercised at time s. Then*

(i) V_t is the smallest supermartingale π_t satisfying $\pi_t \geq h_t$, for all $t \in [0,T]$.

(ii) At time t the exercise time $\tau_t = \min\{\, s \geq t \mid h_s = V_s \,\}$ is optimal:

$$V_t = E_t\left[h(\tau_t)\right] = \sup_{\rho_t \in \mathcal{T}_t} E_t\left[h(\rho_t)\right]. \blacksquare$$

Continuation value. In practical computations it is often better to think in terms of the continuation value $CV(t)$ at time t defined recursively as:

$$CV(T) = 0, \quad \text{and} \tag{4.47}$$

$$CV(t) = E_t\left[\max\{\, h(t+1), CV(t+1)\,\}\right], \quad t < T; \tag{4.48}$$

A straightforward backward induction on $t = T, T-1, \ldots, 0$ shows that

$$V_t = \max\{\, h_t, CV(t)\,\}$$

in accordance with intuition. With this the optimal exercise time τ_t starting from time t becomes

$$\tau_t = \min\{\, s \geq t \mid h_t \geq CV(t)\,\}. \tag{4.49}$$

ie. we exercise the option as soon as the value from exercise is at least as great as the value from continuation and this is also in accordance with intuition.

Following the recursion (4.48) to its root presents the continuation value as an iterated conditional expectation

$$CV(t) = E_t\left[\max\left\{h(t+1), E_{t+1}\left[\max\left\{h(t+2), E_{t+2}[\ldots]\right\}\right]\right\}\right] \tag{4.50}$$

which cannot be computed for the sheer number of path branches of the state vector process which this would entail (path branches must be split at each time t at which a conditional expectation is taken).

Therefore it is not feasible to compute the optimal exercise policy from the continuation value without some sort of approximation. The following two approaches avoid the exact computation of the continuation value:

- Obtaining lower bounds for the option price from concrete exercise strategies and computing associated upper bounds.
- Regression of the continuation value on the current information (path history).

4.6.2 Duality, upper and lower bounds.

Duality methods to find upper bounds for the price of an American option appeared first in [Rog01]. We follow the development in [HK01]. If π_t is any supermartingale then

$$\psi_t := E_t\left[\sup_{s\in[t,T]}(h_s - \pi_s)\right]$$

is a supermartingale (this does not use the supermartingale property of π_t) and

$$M_t^\pi = \pi_t + \psi_t = \pi_t + E_t\left[\sup_{s\in[t,T]}(h_s - \pi_s)\right] \tag{4.51}$$

is a supermartingale satisfying $M_t^\pi \geq \pi_t + (h_t - \pi_t) = h_t$ and consequently

$$M_t^\pi \geq V_t.$$

Recall that V_t is the smallest supermartingale dominating h_t. If here $\pi_t = V_t$ (which is a supermartingale), then $\psi_t \leq 0$ (since then $\pi_s \geq h_s$) and so $M_t^\pi \leq V_t$ and consequently $M_t^\pi = V_t$. This leads to the following *dual characterization* of the price process V_t:

$$V_t = \inf_\pi M_t^\pi, \tag{4.52}$$

where the infimum is taken over all supermartingales $\pi = (\pi_t)$ and is in fact a minimum assumed at the price process $\pi_t = V_t$. Of course we cannot use $\pi = V$ to compute the unknown price process V_t. The idea is now to replace $\pi_t = V_t$ with an approximation $\pi_t \simeq V_t$ in the hope that then $M_t^\pi \simeq V_t$ giving us a close upper bound M_t^π for V_t.

All this depends on having a suitable supermartingale π_t. If U_t is any process an associated supermartingale π_t can be defined as follows

$$\begin{aligned} \pi_0 &= U_0 \quad \text{and} \\ \pi_{t+1} &= \pi_t + (U_{t+1} - U_t) - (E_t[U_{t+1} - U_t])^+ . \end{aligned} \tag{4.53}$$

The supermartingale property of π_t in the form $E_t(\pi_{t+1} - \pi_t) \leq 0$ follows from $g - g^+ \leq 0$ applied to $g = E_t(U_{t+1} - U_t)$.

If U_t is already a supermartingale then $g \leq 0$, hence $g^+ = 0$, and the recursion (4.53) becomes $\pi_{t+1} = \pi_t + (U_{t+1} - U_t)$ and so $\pi_t = U_t$.

If U_t is close to the price process V_t (which is a supermartingale) then we hope that π_t is close to U_t and hence close to V_t. We then hope that M_t^π is close to V_t also in which case it is a close upper bound for V_t.

Recall that $M_t^\pi = V_t$ if $\pi_t = V_t$. The following estimates the error $|M_t^\pi - V_t|$ in terms of the errors $|U_s - V_s|$:

Theorem 4.6.2 *Assume that the process U_t satisfies $U_t \geq h_t$ and the supermartingales π_t, M_t^π are derived from U_t as in* (4.53) *and* (4.51). *Then we have the estimate*

$$M_0^\pi \leq U_0 + E(K), \quad \textit{where} \quad K = \sum\nolimits_{t<T} (E_t[U_{t+1} - U_t])^+ . \tag{4.54}$$

Proof. The recursion (4.53) implies that $\pi_s = U_s - \sum_{j<s} (E_j [U_{j+1} - U_j])^+$ and so, using $h_s \leq U_s$,

$$h_s - \pi_s \leq U_s - \pi_s = \sum\nolimits_{j<s} (E_j [U_{j+1} - U_j])^+ \leq K,$$

for all $s \in [1, T]$. The same equality is also true for $s = 0$ since $h_0 - \pi_0 = h_0 - U_0 \leq 0$. Thus

$$\sup_{s\in[t,T]}(h_s - \pi_s) \leq K, \quad \text{for all } t \in [0, T].$$

(4.54) now follows from $M_0^\pi = U_0 + E\left[\sup_{s\in[t,T]}(h_s - \pi_s)\right]$. ∎

In a sense K is a measure of the deviation of U from the supermartingale property. It is easy to relate this deviation to the deviation of U from the supermartingale V. We do this under the assumption $h_t \leq U_t \leq V_t$.

Our estmates U_t for V_t will be the payoff of concrete exercise strategies and these estimates have this property. Write

$$\begin{aligned} U_{t+1} - U_t &= (U_{t+1} - V_{t+1}) + (V_{t+1} - V_t) + (V_t - U_t) \\ &\leq (V_{t+1} - V_t) + |V_t - U_t| \end{aligned}$$

and take the conditional expectation E_t observing that $E_t(V_{t+1} - V_t) \leq 0$ by the supermartingale property of V_t. This yields

$$(E_t [U_{t+1} - U_t])^+ \leq \|U_t - V_t\|_{L^1}.$$

It now follows from (4.54) that

$$M_0^\pi \leq U_0 + \sum_{t \leq T} \|U_t - V_t\|_{L^1}.$$

Putting all this together we obtain the following procedure for estimating the option price V_0:

Estimates of V_0. A lower bound for the price V_0 is always obtained by specifying a clever exercise policy $\rho = (\rho_t)$. A general class of policies which works well in several important examples is introduced below. However in most cases an investigation of each particular problem does suggest suitable exercise policies.

Once we have a policy ρ the discounted payoff $A_0 = h(\rho_0)$ from exercising according to ρ is a lower bound for V_0. There are now two possibilities to obtain an upper bound for V_0. Set $m_t = h(\rho_t)$ (the price process if the option is exercised according to ρ) and $U_t = \max\{m_t, h_t\}$. Then $h_t \leq U_t \leq V_t$ and the price V_0 satisfies

$$V_0 \leq U_0 + E \sum_{t<T} (E_t[U_{t+1} - U_t])^+ . \tag{4.55}$$

This upper bound has the advantage that it is completely determind by the exercise policy ρ. It has the disadvantage that it is expensive to compute (path splitting). In practice it requires subtle methods to get useful upper bounds this way. See [HK01].

The second method due to C. Rogers [Rog01] makes use of the duality (4.52) and obtains an upper bound

$$V_0 \leq M_0^\pi = \pi_0 + E \left[\sup_{t \in [0,T]} (h_t - \pi_t) \right] \tag{4.56}$$

by choosing a particular supermartingale π. In practice martingales are chosen more precisely the discounted price processes of trading strategies. We know that equality (4.56) is exact if $\pi_t = V_t$, that is, if the chosen strategy is a replicating strategy. Of course the problem of finding a perfect replicating strategy is even harder than the pricing problem. So in practice we design approximate replicating strategies (see 5.2, 5.6.2).

To ensure the martingale property the trading strategies are kept self-financing and this can easily be effected by trading in a designated asset (such as the risk free bond for example) to finance the strategy. See [JT] for an application of this approach to Bermudan swaptions.

4.6.3 Constructing exercise strategies

Recall that the optimal strategy τ satisfies

$$\tau_t = \min\{\, s \geq t \mid h_t > CV(t) \,\},$$

where $CV(t)$ is the continuation value above. A first exercise strategy $\rho = (\rho_t)$ is obtaind approximating the continuation value $CV(t)$ with the quantity

$$Q(t) = \max\{\, E_t(h_{t+1}), E_t(h_{t+2}), \ldots, E_t(h_T) \,\} \tag{4.57}$$

and then exercising as soon as $h_s \geq Q(s)$:

$$\rho_t = \min\{\, s \geq t \mid h_s \geq Q(s) \,\} \tag{4.58}$$

At each time $s \geq t$ the exercise time ρ_t compares the discounted payoff h_s from immediate exercise to the expected discounted payoffs $E_s(h_j)$ from exercise at all possible future dates $j > s$ and exercises the option if h_s exceeds all these quantities. Let us call this strategy the *pure strategy*. It is a baseline from which other strategies are derived.

This strategy is easily implemented in stochastic models of the state price process X_t: to compute the quantities $Q(t)$ we merely have to compute the conditional expectations $E_s(H)$ of the random vector $H = (h_1, h_2 \ldots, h_T)$. At worst this involves path splitting. Quite often however the payoff h_s is identical to the payoff of the coresponding European option. In this case the conditional expectation $E_t(h_s)$ is the price of the European option maturing at time s (discounted to time $t = 0$). It is then often possible to use analytic formulas and to avoid the costly Monte Carlo computation of $E_t(h_s)$. This is the case for American puts on a single Black-Scholes asset and for Bermudian swaptions.

Let us note note an important property of the strategy ρ: the strategy ρ always exercises earlier than the optimal strategy τ:

$$\rho_t \leq \tau_t, \quad t \leq T. \tag{4.59}$$

This follows from the inequality

$$CV(t) \geq Q(t). \tag{4.60}$$

For a proof of this inequality set $a \vee b = \max\{a, b\}$, observe that $CV(T) = 0$ and

$$E_t\left[CV(s)\right] = E_t\left[h_{s+1} \vee CV(s+1)\right], \quad t < s \leq T,$$

and repeatedly use the trivial inequality $E_t[f \vee g] \geq E_t[f] \vee E_t[g]$ to obtain

$$\begin{aligned} CV(t) &= E_t\left[h_{t+1} \vee CV(t+1)\right] \\ &\geq E_t\left[h_{t+1}\right] \vee E_t\left[CV(t+1)\right] = E_t\left[h_{t+1}\right] \vee E_t\left[h_{t+2} \vee CV(t+2)\right] \\ &\geq \ldots \geq E_t\left[h_{t+1}\right] \vee E_t\left[h_{t+2}\right] \vee \ldots \vee E_t\left[h_T\right] = Q(t). \end{aligned}$$

The difference between $E_t[f \vee g]$ and $E_t[f] \vee E_t[g]$ is large if f is small on a set where g is large and conversely. There is no difference at all for example if $g \geq f$. Here f is the payoff from immediate exercise and g the continuation value. For most options in practice the payoffs from immediate exercise and continuation are related, that is, with a high probability $g \geq f$ and even if $f > g$ then usually g is still close to f.

In this case we would expect the continuation value $CV(t)$ to be close to $Q(t)$ with high probability and consequently the strategy $\rho = (\rho_t)$ to be close to the optimal strategy $\tau = (\tau_t)$. This would not be the case for options specifying payoffs at different times which are completely unrelated.

Continuation and exercise regions. Fix a time t and assume our option has not been exercised yet. At time t the option is then exercised on the region $[\tau_t = t] = [h_t \geq CV(t)]$ and is held on the region $[\tau_t > t] = [h_t < CV(t)]$. These regions are events in the probability space and so are hard to visualize. Instead we project them onto the two dimensional coordinate plane by using coordinates x and y which are $\mathcal{F}_t$-measurable, that is, the values of x and y are known at time t.

The most descriptive coordinates are of course $x = CV(t)$ and $y = h_t$. In these coordinates the continuation region assumes the form $[y < x]$. Since we cannot compute $CV(t)$ we use the coordinates $x = Q(t)$ and $y = h_t$ instead. In these coordinates $[y < x]$ is the continuation region of the suboptimal pure exercise strategy while the optimal continuation region completely contains this region.

This points us to the idea of constructing better exercise strategies as follows: we enlarge the continuation region (the region below the diagonal $[y = x]$) by denting the diagonal $[y = x]$ upward and parametrize the exercise boundary as $y = k(x)$ for a function $k(x) \geq x$. Experiments with the function

$$k(x, \alpha, \beta) = \begin{cases} \beta\,(x/\beta)^{\alpha} & : \quad x < \beta \\ x & : \quad x \geq \beta \end{cases} \tag{4.61}$$

with parameters $0 < \alpha < 1$ and $\beta > 0$ have shown excellent results in several examples.

For $\alpha = 0.65$ and $\beta = 0.5$ the boundary assumes the shape seen in figure 4.1. It is a good idea to make α and β time dependent. As time t

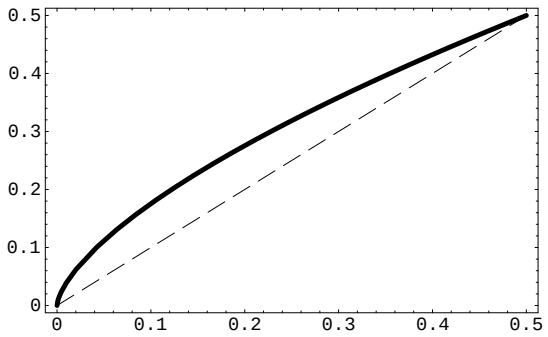


Figure 4.1: Exercise boundary

approaches the time T of expiration $Q(t)$ will approach $CV(t)$ (they are equal for $t = T - 1$) and so we might want to have $\alpha(t) \uparrow 1$ and possibly also $\beta(t) \downarrow 0$ as $t \uparrow T$. This flattens out the diagonal as expiry approaches. Note that we are approximating the continuation value $CV(t)$ as

$$CV(t) = k(Q(t), \alpha(t), \beta(t))$$

and starting from time t we exercise at the first time $s \geq t$ such that

$$h_s > k(Q(s), \alpha(s), \beta(s)) = \beta(s) \left(Q(s)/\beta(s)\right)^{\alpha(s)} . \tag{4.62}$$

Let us call this strategy *convex exercise* (from the convexity of the continuation region) while the strategy exercising as soon as $h_s > Q(s)$ will be called *pure exercise*.

Parameter optmization To find good values for the parameters $\alpha(t)$, $\beta(t)$ generate a set of training paths of the underlying and store the values of h_t and $Q(t)$ along each path (though not the path itself) for all paths. Parameters α, β optimizing the payoff over this set of paths are computed by backward recursion on t starting from $t = T - 1$. Together with these parameters we also store for each exercise date t and each training path the optimal exercise time $s \geq t$ under the computed optimal parameters. Parameter optimization at the exercise date $t-1$ then uses this information for very fast computation of the payoff from exercise under any parameters $\alpha(t-1)$, $\beta(t-1)$.

Examples. Here are some results in the case of particular examples. The first example is a plain vanilla American put on a Black-Scholes asset S satisfying $S(0) = 50$, $\sigma = 0.4$, $r = 10\%$ and $T = 5/12$. The exact price of this put is 4.28. The price of the corresponding European option is 4.07. The

pure exercise strategy with 10 exercise dates yields 4.225. Convex exercise with recursive parameter optimization at each exercise date improves this to 4.268. The class AmericanPutCvxTrigger implements the selfcalibrating trigger and the class AmericanBlackScholesPut in the package Option contains the price computation.

The second example is the call on the maximum of 5 uncorrelated Black-Scholes assets. This is the example in [HK01]. We use the same parameter values (volatilities, time to expiration, number of exercise periods, strike). With 10 exercise periods to expiry [HK01] locates the price for the at the money call in interval $[26.15, 26.17]$. The pure exercise strategy yields 25.44 while the parameters $\alpha = 0.85$ and $\beta = 150$ with linear adjustment along the exercise dates improve this to 26.18. The corresponding European price is 23.05. The results were computed from only 10000 paths of the underlying basket (no analytic formulas) and are not very reliable.

Finally in the case of Bermudan swaptions benchmark prices were provided by Peter Jaeckel computed using a sophisticated setup (low discrepancy sequences, reduced effective dimension). For a long dated 20 year semiannual Bermudan swaption callable after 2 years Jaeckel gives a price of 0.06813. The pure exercise strategy results in 0.065 while convex exercise with parameter optimization improves this to 0.0675. Here the price of the most expensive European swaption was 0.041. This case will be discussed again below where it will be seen that exercise strategies constructed using information about the particular problem can be superior and much faster computationally.

Implementation. An American option will be written on some underlying either a single asset or a basket of assets. Let's assume it is a single asset. The payoff from exercise at time t computed from the current path of the underlying is given by the abstract method currentDiscountedpayoff:

```
public class AmericanOption {

    Asset underlying;      // the underlying

    // discounted payoff h_t from exercise at time t computed
    // from the current path of the underlying.
    public abstract double currentDiscountedPayoff(int t);

    // other methods

} // end AmericanOption
```

Here is $Q(t)$. The parameter nBranch is the number of path branches spent

on computing each conditional expectation $E_t(h_j)$, $t < j \leq T$:

```
public double Q(int t, int nBranch)
{
    // the random vector H = (h_0, h_1, ..., h_T), draws at time t only
    // compute h_{t+1}, ..., h_T. This is all we need for Q(t).
    final double[] x=new double[T+1];
    RandomVector H=new RandomVector(T+1){

        public double getValue(int t)
        {
            underlying.newPathBranch(Flag.RISK_NEUTRAL_PROBABILITY,t);
            for(int j=t+1; j<T+1; j++) x[j]=currentDiscountedPayoff(j);
            return x;
        }
    }; // end H

    // max{ E_t(h_{t+1}), E_t(h_{t+2}), ..., E_t(h_T) } = max_{j>t} y[j]
    double[] y=H.conditionalExpectation(s,nBranch);
    double max=y[t+1]; for(int j=t+2; j<=T; j++) if(y[j]>max)max=y[j];
    return max;
} // end Q
```

In case the conditional expectations $E_t(h_j)$, $t < j \leq T$, can be computed via analytic formula the random vector H will obviously be replaced with these formulas and the parameter nBranch is not needed.

Recall that the concept of a family $\rho = (\rho_t)$ of stopping times $\rho_t \geq t$ is already implemented in the class Trigger.java in the package Trigger. The method isTriggered(t,s) returns true if $\rho_t = s$ and false otherwise. With this we can now implement the exercise strategy strategy (4.58). The parameter nBranch is as above:

```
public Trigger pureExercise(final int nBranch)
{
    return new Trigger(T){

        public boolean isTriggered(int t, int s)
        {
            if(s==T) return true;
            if(currentDiscountedPayoff(s)>Q(s,nBranch)) return true;
            return false;
         }
```

```
            public boolean nextTime(int t)
            {
               int s=t;
               while(!isTriggered(t,s))
               { underlying.timeStep(Flag.RISK_NEUTRAL_PROBABILITY,s); s++; }
               return s;
            }

    }; // end return new

} // end naiveExercise
```

Note that the method nextTime(int t, int s) advances the path of the underlying asset (or basket of assets or Libor process) to the first time $s \geq t$ at which the trigger triggers ($\rho_t = s$). If exercise is any trigger then the discounted payoff $h(\rho_t)$ for the exercise strategy ρ_t corresponding to the trigger exercise can be implemented as follows:

```
public double currentDiscountedPayoff(int t, Trigger exercise)
{
    int s=exercise.nextTime(t);
    return currentDiscountedPayoff(s);
}
```

The classes Options.AmericanOption and Options.AmericanBasketOption follow this pattern. Forward prices instead of discounted prices are used in the case of Bermudan swaptions (Libor.LiborDerivatives.BermudanSwaption). All these classes implement a variety of exercise triggers. In addition there are the standalone triggers pjTrigger, cvxTrigger for Bermudan swaptions in the package Libor.LiborDerivatives which optimize their parameters using a BFGS multidimensional optimizer.

4.6.4 Recursive exercise policy optimization

All our exercise policies rely on an approximation $CV_0(t)$ of the continuation value $CV(t)$ and we exercise at time t if $h_t \geq CV_0(t)$. Let us recall how we have approximated the continuation value $CV(t)$ above. We have recognized that the quantity $Q(t)$ probably is an important feature of the continuation value which will be somewhat larger than $Q(t)$ and the discrepancy diminishes as option expiry approaches. Based on these insights we have chosen the approximation

$$CV(t) \simeq k(Q(t), \alpha(t), \beta(t)), \tag{4.63}$$

where $k = k(x, \alpha, \beta)$ is as in (4.61). This method can be generalized as follows: rather than limiting ourselves to the specific approximation (4.63) we try a more general form

$$CV(t) \simeq f(t, \alpha), \tag{4.64}$$

where the random variable $f(t, \alpha)$ depends on a parameter vector α and is $\mathcal{F}_t$-measurable, the minimal requirement which ensures that the value of $f(t, \alpha)$ is known by time t when it is needed. The random variable $f(t, \alpha)$ can incorporate any feature from the path $s \in [0, t] \to X(t)$ of the state vector process up to time t (the information at hand at time t) which we deem significant.

Except for the value of the parameter vector α (which will be optimized) we need to come up with all details of the quantity $f(t, \alpha)$. See [TR99] for a detailed discussion how such a function f might be specified in a concrete example.

The task at hand is now to find for each time t an optimal parameter vector $\alpha(t)$ and the solution uses backward recursion starting with $t = T$. For $t = T$ we have $CV(t) = 0$.

Training paths. We generate a number (typically tens of thousands) of training paths of the state vector and for each path we store all the information needed to compute the quantities $f(t, \alpha)$ for each parameter vector α. At worst we have to store the entire path $s \in [0, T] \to X(s)$ and precompute and store all computationally expensive features (such as $Q(t)$ for example) which are incorporated into $f(t, \alpha)$. This costs memory but does allow for fast computation of $f(t, \alpha)$ for each parameter α over all training paths.

For each training path and at each time t we also store the first time $s = s(t) \geq t$ at which the computed optimal parameters exercise the option and the resulting payoff. Here "optimal parameters" designates the parameters resulting from our parameter optimization. For $t = T$ we can already set $s = T$ (recall our convention $h_T \geq 0$).

Suppose optimal parameters $\alpha(s)$ have been found for $s > t$ and the corresponding exercise times set for each training path. We now want to determine the optimal parameter vector $\alpha(t)$ at time t and can proceed in one of two ways:

Maximizing the payoff. Each parameter vector α defines how we exercise at time t. With this we can compute the payoff $H(\alpha)$ from exercise starting at time t as a function of α. Morever this computation is very fast since we only have to decide wether we exercise immediately (determined by α) or later, in which case the time $s(t+1)$ of exercise is already set and the payoff

cached. We can then employ methods to maximize the payoff $H(\alpha)$. This is a computationally efficient approach and intuitively appealing since we are in fact looking for an exercise strategy which maximizes the payoff.

Recursive approximation of $CV(t)$. This method, also proposed in the literature, is less appealing. The idea is as follows: once we already have an approximation $CV(t+1) \simeq f(t+1, \alpha(t+1))$ we can compute the continuation value $CV(t)$ approximately as

$$\begin{aligned} CV(t) &= E_t\left[\max\{h_{t+1}, CV(t+1)\}\right] \\ &\simeq E_t\left[\max\{h_{t+1}, f(t+1, \alpha(t+1))\}\right]. \end{aligned}$$

Monte Carlo simulation is usually necessary but paths are split only once at time t and advanced only one time step to time $t+1$. We compute $CV(t)$ for each training path and then minimize the squared error $[CV(t) - f(t, \alpha)]^2$ over all trainig paths as a function of the parameter α. This approach is computationally much more expensive and also addresses the goal of finding an exercise strategy which maximizes the option payoff only indirectly.

The convex exercise strategy is very general but the computation of the quantity $Q(t) = \max_{s>t} E_t(h_s)$ is computationally quite expensive even if analytic formulas are available for the conditional expectations $E_t(h_s)$. It is extremely expensive if no such formulas are available.

Chapter 5

Trading And Hedging

5.1 The Gains From Trading

Betting on a single asset S is an acceptable form of gambling and the theory of finance more intellectually rewarding than the analysis of odds in lotteries or casinos. Let us therefore consider a market in which only two assets are traded, a risky asset S and the riskfree bond B. The prices $S(t)$, $B(t)$ are stochastic processes and $S(t)$ is the discounted price of the asset S at time t. We can also view $S(t)$ as the price of S expressed in a new numeraire, the riskfree bond. The unit of account of this new currency is one share of the risk free bond B. In this currency the risk free rate of interest for short term borrowing appears to be zero.

We will assume that we have the ability to borrow unlimited amounts at the risk free rate. Keeping score in discounted prices then automatically accounts for the time value of money and eliminates interest rates from explicit consideration.

A *trading strategy* is defined by the number $w(t)$ shares of the asset S held at time t (the *weight* of the asset S, a random variable). A *trade* takes place to adjust the weight $w(t)$ in response to events related to the price path of the asset S. The trades take place at an increasing family of random times

$$0 = \tau_0 < \tau_1 < \ldots < \tau_{n-1} < \tau_n = T, \tag{5.1}$$

where the number n of trades itself can be a random variable. The trade at time T liquidates the position and we then take a look at the discounted *gains from trading* which the strategy has produced. These gains can be viewed as the sum of gains from individual trades buying $w(\tau_k)$ share of S at time τ_k and selling them again at time τ_{k+1} (to assume the new position).

Taking account of the fact that the price of the asset S has moved from $S(\tau_k)$ at time τ_k to $S(\tau_{k+1})$ at time τ_{k+1} this transaction results in a discounted gain of

$$w(\tau_k)[S(\tau_{k+1}) - S(\tau_k)] \tag{5.2}$$

This computation assumes that the asset S pays no dividend and that trading is not subject to transaction costs. If on the other hand the asset S pays a continuous dividend with yield q and if each trade has fixed transaction costs ftc and proportional transaction costs ptc (cost per share) (5.2) becomes

$$w(\tau_k)[(S(\tau_{k+1}) - S(\tau_k)) + dvd(k)\ ] - trc(k), \tag{5.3}$$

where

$$dvd(k) = qS(\tau_k)(\tau_{k+1} - \tau_k) \tag{5.4}$$

is the dividend earned by one share of the asset S during the interval $[\tau_k, \tau_{k+1}]$ (closely approximated) and

$$trc(k) = |w(\tau_{k+1} - w(\tau_k)|\mathsf{ptc} + \mathsf{ftc} \tag{5.5}$$

are the transaction costs. The total (discounted) gains from trading are then given by the sum

$$G = \sum_{k=0}^{n-1} w(\tau_k)[\ \Delta S(\tau_k) + dvd(k)\ ] - trc(k), \tag{5.6}$$

where

$$\Delta S(\tau_k) = S(\tau_{k+1}) - S(\tau_k). \tag{5.7}$$

Note that G is a random variable in accordance with the uncertainty and risk surrounding every trading activity. The first question which presents itself is how we should implement the increasing sequence (5.1) of times at which the trades occur. Typically the time of a new trade is determined with reference to the time of the last trade. Quite often the size of the price change from the time of the last trade triggers the new trade ("if it goes down 15%" etc.). The class Trigger implements this concept. The method isTriggered(t,s) returns true if an event (such as a trade) is triggered at time s with reference to time t where usually t is the last time the event was triggered. The sequence (5.1) is then defined recursively as

$$\tau_0 = 0 \quad \text{and} \quad \tau_{k+1} = \min\{\ \mathsf{s} > \tau_k \mid \mathsf{isTriggered}(\tau_k, \mathsf{s})\ \} \tag{5.8}$$

and this is even easier to code by checking at each time step s if isTriggered(t,s) is true and moving t from trigger time to trigger time. With this a trading strategy consists of an asset to invest in, a trigger triggering the trades and an abstract method delivering the weights:

```
public abstract class TradingStrategy{

    double currentWeight;    // current number of shares held
    Asset asset;             // the asset invested in
    Trigger tradeTrigger;    // triggers the trades
    double fixed_trc;        // fixed transaction cost (per trade)
    double prop_trc;         // proportional transaction cost (per trade)

    int T;                   // time steps to horizon
    double dt;               // size of time step
    double[] S;              // price path of the asset
    double[] B;              // price path of the riskfree bond
    int nTrades;             // current number of trades

    // the new weight of the asset given that a trade occurs at time t
    public double abstract newWeight(int t);

    // constructor
    public TradingStrategy
    (double fixed_trc, double prop_trc, Asset asset, Trigger tradeTrigger)
    {
        this.fixed_trc=fixed_trc; this.prop_trc=prop_trc;
        this.asset=asset; this.tradeTrigger=tradeTrigger;
        T=asset.get_T(); dt=asset.get_dt();
        S=asset.get_S(); B=asset.get_B();
        nTrades=0;
        currentWeight=newWeight(0);
    }

    // other methods

} // end TradingStrategy
```

The field currentWeight is kept to allow the method newWeight to compute the new weight of the asset with reference to the last weight ("selling half" etc.). Consequently it is important to keep this variable up to date at all times even in sections of code which do not explicitly use it. It might be used through calls to newWeight.

The "other methods" alluded to above are devoted to the analysis of the discounted gains from trading. First we need a method which computes these gains along one path of the asset. This path is computed simultaneously with the trading strategy and driven by new Z-increments and is thus independent of previous paths. Thus the variable Asset.nSignChange is irrelevant. Nonetheless it must be assigned some value if we want to allocate an asset.

```
public double newDiscountedGainsFromTrading()
{
    int t=0,                                  // time of last trade
        s=0;                                  // current time

    double q=asset.get_q(),                   //dividend yield
           gain=0,                            // discounted gain from trading
           w=newWeight(t),                    // current number of shares held
           currentWeight=w;                   // update

    while(t<T)
    {
        // Currently w=newWeight(t), number of shares held at time t of last trade
        // Move the price path forward to time s of next trade
        s=asset.pathSegment(Flag.MARKET_PROBABILITY,t,tradeTrigger);

        double w_new,                         // new weight we move to at time s
               deltaS=S[s]-S[t];              // price change

        if(s<T)w_new=newWeight(s);            // adjust position
        else w_new=0;                         // sell all

        // the dividend in [t,s], 1 share:
        double dividend=q*S[t]*(s-t)*dt;

        // the transaction costs:
        double trc=Math.abs(w_new-w)*prop_trc+fixed_trc;

        gain+=w*(deltaS+dividend)-trc;

        // move state forward to time s:
        t=s; w=w_new; currentWeight=w;

    } // end while

    return gain;

} // end newDiscountedGainsFromTrading
```

This produces one sample of the discounted gains from trading viewed as a random variable. The following method returns this random variable:

```
public RandomVariable discountedGainsFromTrading()
{
    return new RandomVariable(){

        public double getValue(int t)
        { return newDiscountedGainsFromTrading(); }

    }; // end return new

} // end discountedGainsFromTrading
```

Note that the time variable t is ignored in the method getValue defining the random variable discountedGainsFromTrading. This means that we have not implemented the ability to condition on information available at time t. It would be somewhat more complicated to implement this and we will not use it. This means however that the discounted gains from trading can only be meaningfully considered at time $t = 0$.

Once we have the gains from trading as a random variable we can call on methods implemented in the class RandomVariable to compute interesting statistics associated with it. What sort of trading strategies should we pursue? A popular strategy is to increase the stakes when the position has moved against you. For an investor going long this means averaging down. For example we might initially buy 100 shares and double our position every time the asset price declines 5% from the level of the last buy. Assuming an Asset has already been allocated this strategy could be defined as follows:

```
double prcnt=5;          // percentage decline which triggers the next buy
       fixed_trc=0.1;    // fixed transaction cost per trade
       prop_trc=0.25;    // proportional transaction cost

Trigger tradeTrigger=new TriggerAtPercentDecline(asset,prcnt);

TradingStrategy doubleOrNothing=
new TradingStrategy(fixed_trc,prop_trc,asset,tradeTrigger){

    // define the weights
    public double newWeight(int t){ return 2*currentWeight; }

}; // end doubleOrNothing
```

Here TriggerAtPercentDecline is a trigger defined in the package Triggers which triggers every time the discounted asset price declines a certain percentage from the level of the last trigger event. The time parameter t is ignored since the new weight is defined with reference to the last weight. Some trad-

ing strategies however will make use of time t, most notably the strategies hedging option payoffs defined below.

Proceeding like this allocates the strategy for local use. If we want to use it more than once we could define a class DoubleOrNothing which extends the abstract class TradingStrategy and implements the above method in the class body. Some such strategies are implemented in the package TradingStrategies. The reader is invited to read the source code and browse the javadoc.

The gains from trading are not the only quantity of interest associated with a trading strategy. For example the double or nothing strategy above provides excellent returns even in the case of an asset drifting down. Undoubtedly this accounts for its appeal to money managers which later become fugitives from the law or inmates of minimum security prisons. The reason for this are other quantities of interest associated with the strategy such as the *borrowings* necessary to maintain the strategy and the *drawdown* which is experienced and must be stomached during the life of the strategy. Here we assume that the strategy is financed completely by borrowing at the riskfree rate, that is the trader does not put up any capital initially.

The doubleOrNothing strategy above executed in a bear market will balloon your position to astronomical size. Borrowings may not be availble in sufficient quantity to last until a reversal of fortune sets in.

Thus we see that it is is prudent to compute some metrics in addition to the gains from trading associated with a trading strategy. These are best all computed simultaneously and assembled into a random vector. The methods

```
public double[] newTradeStatistics(){ /* definition */ }
public RandomVector tradeStatistics(){ /* definition */ }
```

defined in the class TradingStrategy do this and the following quantities associated with a trading strategy are computed: maximum borrowing, maximum drawdown, gains from trading, return on investment and number of trades. The reader is invited to check the source code for bugs and browse the javadoc.

Results. The package TradingStrategies defines the following strategies:

- Buy and hold.
- Dollar cost averaging.
- Averaging down.
- Double or nothing.

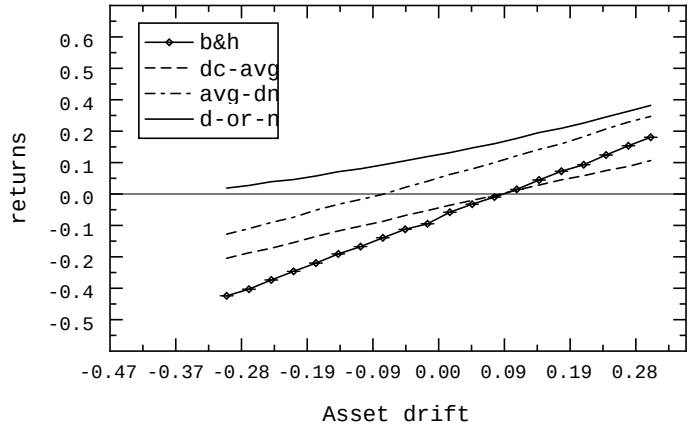


Figure 5.1: Returns

All strategies start with an initial purchase of 100 shares at time $t = 0$. The buy and hold strategy keeps this position until the time horizon, averaging down buys another 100 shares whenever the asset price declines a certain trigger percentage below the level of the last buy, dollar cost averaging buys 100 shares at regular time intervals regardless of price, and double or nothing increases the current position by a certain *factor* f whenever the asset price declines a certain trigger percentage below the level of the last buy.

How we fare depends on the asset drift μ, that is, wether we are in a bear market ($\mu < 0$) or a bull market ($\mu > 0$). To compare performance we compute some statistics as functions of the asset drift μ evaluated at points evenly spaced between extremes μ_{min} and μ_{max}.

Lets first look at the return on investment. The return of a trading strategy will be computed as the return on the maximum amount invested during the lifetime of the strategy. The gains from trading are viewed as interest on this maximum and a corresponding continuously compounded return derived. The strategy is assumed to be financed by borrowing at the risk free rate. The following parameters apply to all figures below: $S(0) = 50$, $\sigma = 0.3$, $q = 0$, $r = 0.05$, $dt = 0.02$, $T = 50$, fixed_trc $= 10$, prop_trc $= 0.25$, Trigger percentage $= 7$, dollar cost averaging, number of buys $= 5$.

In pondering the returns in figure 5.1 recall that the cash price process $S(t)$ of the constant volatility asset (no discounting) satisfies

$$S(t) = S(0)exp[(\mu - 0.5\sigma^2)t + \sigma W(t)] \tag{5.9}$$

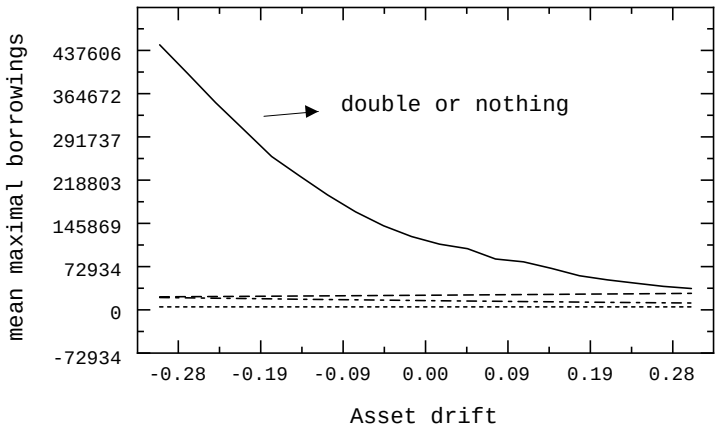


Figure 5.2: Borrowings

where μ and σ are the drift and volatility of the (cumulative) returns and $W(t)$ is a Brownian motion. Thus the expected returns of the buy and hold strategy are $\mu - 0.5\sigma^2$ and this is borne out by the simulation.

Dollar cost averaging is a rather mediocre strategy under all scenarios. Note that our definition of dollar cost averaging (buying constant numbers of shares) differs from the more usual one (investing a constant sum of money) on each buy. This latter strategy should improve returns in a bear market and decrease returns in a bull market. The reader is invited to define such a strategy and carry out the simulation.

The most striking aspect is the return profile of the double or nothing strategy. It is able to stay above water even in a severe bear market. Aggressively increasing the bets and waiting for a reversal of fortune however is the path to ruin for most who succumb to temptation.

Figure 5.2 explains why. Enormous amounts of money have to be borrowed should the drift be down. In fact this chart is completely dominated by the borrowings of the double or nothing strategy. Note that your creditors are likely to become nervous in case your fortunes turn down while your borrowings expand. For this reason it might be a good idea to to do the following

Exercise. Compute the returns from the double or nothing strategy under the assumption that you will be sold out by your broker in case borrowings exceed 200% of the intial investment. You might want to add a new field cumulativeBorrowings to the class TradingStrategy and define the method

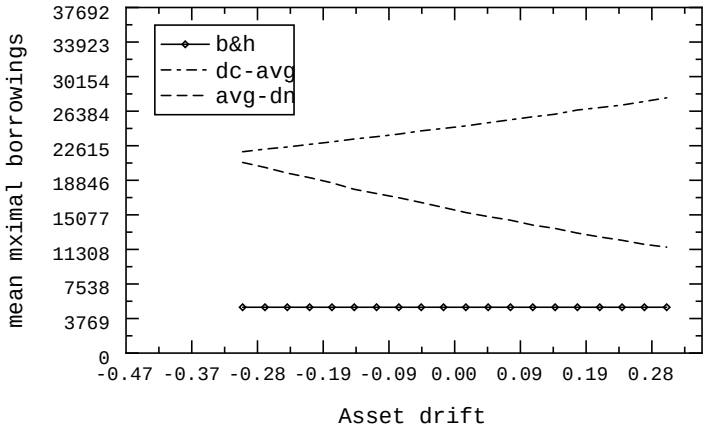


Figure 5.3: Borrowings

TradingStrategy.newWeight with reference to the state of this field.

The fact that huge funds can overcome subfair odds is already known from ordinary gambling. Needless to say the strategy is not viable in practice even if you have the funds. You must begin with a large cash hoard and a rather small position (lest your funds not suffice to support the ballooning portfolio should the drift be down). If the unknown asset drift μ turns out to be positive and volatility small you will remain underinvested to the end. At worst your stock is delisted and the strategy ends in total loss. Thus we concentrate on the other strategies. (invested funds).

Figure 5.3 gives us another look at the borrowings. A dollar cost averaging investor will need more funds in a rising market while one who averages down will need less. Other definitions of dollar cost averaging have the investor buy a varying number of shares for a fixed amount of cash at each buy. This copes with the uncertainty of the funds needed to maintain the strategy. You can implement this strategy and check its performance as an exercise.

Investors must face the unpleasantness of drawdowns (from the original investment) depicted in figure 5.4 as a percentage of invested funds (that is funds sunk into the strategy regardless of what's left). Drawdowns are important for psychological reasons. Many an investor has been prompted to sell out at an inopportune time due to the pressure of the drawdowns. Drawdowns are also alarming for creditors who in the end will make the final decisions.

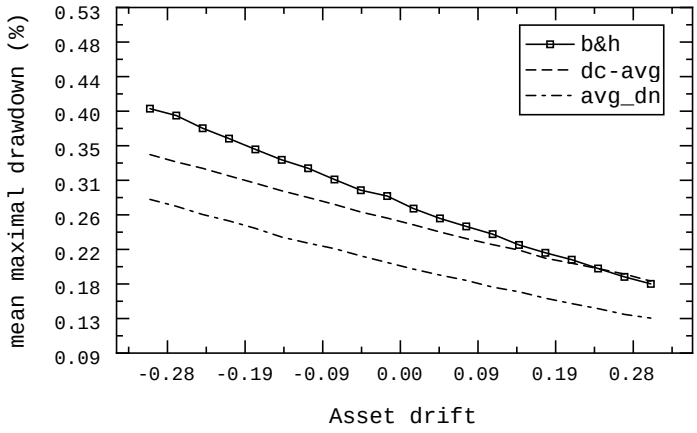


Figure 5.4: Drawdown

The graphs are computed by the class Examples.Trading.GainsFromTrading. The directory Pictures/GainsFromTrading/FunctionOfDrift contains these and others as PNGs. The colors should make the distinction between the graphs easier. We conclude this excursion with histograms of the returns from trading. These histograms are smoothed and normalized (area equal to one).

It is undeniable that averaging down produces superior returns.

5.2 Hedging European Options

Recall that we work with *discounted* prices, that is, $S(t)$ denotes the discounted price of the asset S at time t. In other words we keep score in constant time zero dollars.

Assume we have written an option with *discounted payoff* h at time T and discounted price process $C(t)$. We then effectively have a short position of one such option. To offset the risk associated with this position we trade in the underlying S according to some trading strategy (the *hedge*). The idea is to determine the weights $w(t)$ of the strategy in such a manner that the gains from trading track the loss on the option short position as closely as possible.

We assume that the discounted option price $C(t)$ is a martingale in the risk neutral probability Q and consequently

$$C(t) = E_t(h), \quad t \in [0, T],$$

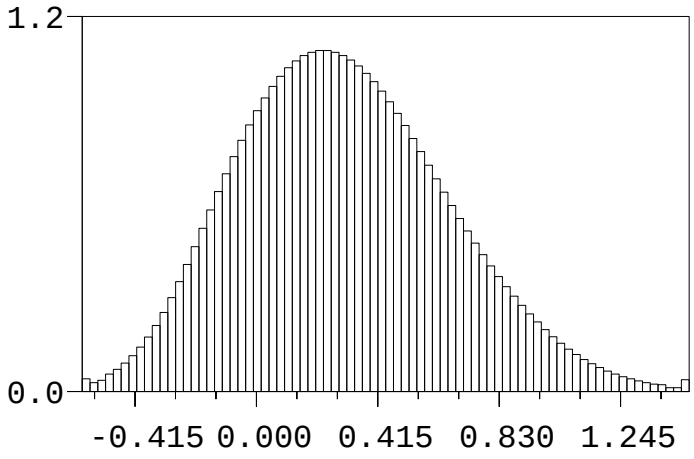


Figure 5.5: Returns: averaging down.

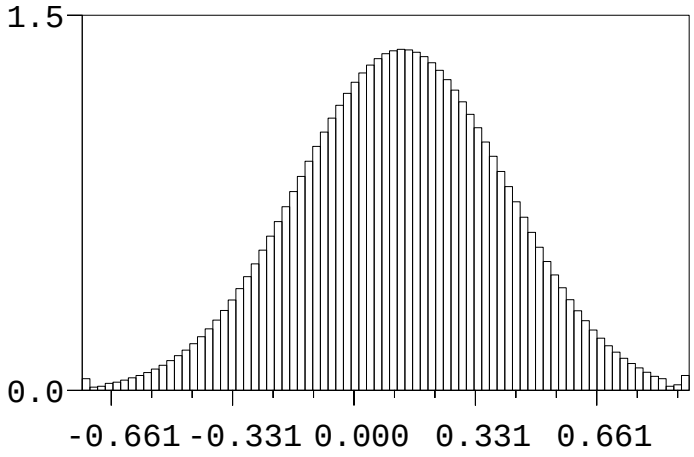


Figure 5.6: Returns: dollar cost averaging.

where the conditional expectation is taken with respect to Q.

5.2.1 Delta hedging and analytic deltas

The classic weights arise from a straightforward discretization of the continuous strategy of delta hedging. Here it is assumed that the discounted price $C(t)$ of the option is given by a known function of time and the discounted price $S(t)$ of the underlying

$$C(t) = c(t, S(t)) \tag{5.10}$$

where $c = c(t, s)$ is a continuously differentiable function. The martingale property of the discounted price $C(t)$ in the risk neutral probability Q implies that the bounded variation terms in the expansion of the stochastic differential $dC(t) = d\,c(t, S(t))$ must vanish leaving us with

$$dC(t) = w(t)dS(t), \quad \text{where} \quad w(t) = \frac{\partial c}{\partial s}(t, S(t)).$$

(Ito's formula). In integrated form

$$C(t) = C(0) + \int_0^t w(s)dS(s), \quad t \in [0, T].$$

Since the integral on the right represents the discounted gains from trading S continuously with weights $w(s)$, the continuously rebalanced hedge with weights $w(s)$ perfectly replicates the option payoff h. The weight

$$w(t) = \frac{\partial c}{\partial s}(t, S(t)) \tag{5.11}$$

is called the *analytic delta* of the option at time t. Such analytic deltas exist only if the option price is given by an analytic formula and have to be implemented differently for each concrete option.

If the hedge is not continuously rebalanced and instead the hedge weights $w(t)$ are constant on intervals $[\tau_k, \tau_{k+1})$ the discounted gains $G(T)$ from trading on $[0, T]$ assume the form

$$G(T) = \sum\nolimits_k w(\tau_k)\Delta S(\tau_k),$$

where $\Delta S(\tau_k) = S(\tau_{k+1}) - S(\tau_k)$. The error with which the hedge tracks the option is the quantity $C(0) + G(T) - C(T)$ and telescoping the difference $C(T) - C(0)$ as $\sum_k \Delta C(\tau_k)$ this can be written as

$$C(0) + G(T) - C(T) = \sum\nolimits_k \left[w(\tau_k)\Delta S(\tau_k) - \Delta C(\tau_k)\right]. \tag{5.12}$$

Although this quantity is the discounted gain from selling and hedging the option it may assume negative values and so is best thought of as the tracking error of the hedge. We are interested in minimizing this error and since the mean can always be absorbed into the option premium we are interested in minimizing the variance of the tracking error.

5.2.2 Minimum Variance Deltas

The tracking error on a single interval $[t, s)$ on which the weight $w(t)$ is kept constant is the quantity

$$e(t) = w(t)\Delta S(t) - \Delta C(t),$$

where $\Delta S(t) = S(s) - S(t)$ and $\Delta C(t) = C(s) - C(t)$. This quantity is written as a function of t only since s is understood to be the first time $s > t$ where the hedge is rebalanced. To minimize this tracking error we choose $w(t)$ so as to minimize the L^2-norm

$$\|e(t)\|_2^2 = E_t\left[e(t)^2\right]$$

(conditioned on the state at time t) over all possible weights $w(t)$ which are $\mathcal{F}_t$-measurable, that is, the value of $w(t)$ can be identitfied at time t. In the risk-neutral probability Q the discounted asset prices $S(t)$ and $C(t)$ are martingales and so $E_t(\Delta S(t)) = E_t(\Delta C(t)) = 0$. From this it follows that $E_t[e(t)] = 0$ and consequently the L^2-norm is also the variance of the hedge error on the interval $[t, s]$. This is not true for the market probability P.
Let X and Y be random variables and w a constant. Then

$$E\left[(wX - Y)^2\right] = w^2E[X^2] - 2wE[XY] + E[Y^2]$$

and this polynomial in w is minimized by $w = E(XY)/E(X^2)$. Applying this to $X = \Delta S(t)$ and $Y = \Delta C(t)$ and conditioning on the state at time t yields the *minimum variance delta* $w(t)$ as

$$w(t) = \frac{E_t\left[\Delta S(t)\Delta C(t)\right]}{E_t\left[(\Delta S(t))^2\right]}. \tag{5.13}$$

In practical simulation the conditioning on the state at time t only means that these quantities will be computed from simulated paths which are branches of the current path with branching at time t. Here we have not specified the probability (market/risk neutral) under which these quantities

will be computed. It can be shown that the above weights minimize the variance of the cumulative hedge error (5.12) for the entire hedge in the risk neutral probability. They do not in general minimize the variance of this error in the market probability.

To determine the weights which minimize the hedge error in the market probability is considerably more complicated. See [Sch93]. A minimizing hedge strategy may not exist. Even under conditions where existence can be shown the weights have to be computed by backward recursion starting with the last weight and involving iterated conditional expectations which makes them impossible to compute in practice. We shall therefore have to stick with (5.13).

Let us reflect upon the effort involved in computing the weight w directly from 5.13. In the absence of explicit formulas (the usual case) the conditional expectations $E_t[...]$ involve the simulation of a large number of branches (say 20,000) of the current path at time t. Now look at the expression $\Delta C(t) = C(s) - C(t)$ in the numerator. At time t the value $C(t)$ is already known but

$$C(s) = E_s[h] \tag{5.14}$$

has to be computed for each of these 20,000 branches by splitting each of these branches at time s into yet another 20,000 branches resulting in $N = 400,000,000$ path segments from time s to the horizon T.

This can take hours and yields only the hedge weight for one single path at one single time t. Suppose now you want to check the efficieny of this hedge policy along 1000 paths of the underlying rebalancing the hedge 15 times along each path. The above then has to be multiplied by another 15,000. This is not feasible on anything less than a supercomputer. The problem here is the numerator

$$\begin{aligned} E_t[\Delta S(t)\Delta C(t)] &= E_t[\Delta S(t)(C(s) - C(t))] & (5.15)\\ &= E_t[\Delta S(t)C(s)] - E_t[\Delta S(t)C(t)]. & (5.16) \end{aligned}$$

Assume now that we are working in the risk neutral probability Q. Since the discounted price $S(t)$ is a Q-martingale we have $E_t[\Delta S(t)] = 0$. Moreover $C(t)$ a known constant at time t and so

$$E_t[\Delta S(t)C(t)] = C(t)E_t[\Delta S(t)] = 0. \tag{5.17}$$

To simplify the first summand recall that $C(s) = E_s[h]$ and that $\Delta S(t) = S(s) - S(t)$ is a known constant at time s and so

$$E_t[\Delta S(t)C(s)] = E_t[\Delta S(t)E_s[h]] \tag{5.18}$$

$$= E_t[E_s[\Delta S(t)h]] \tag{5.19}$$
$$= E_t[\Delta S(t)h] \tag{5.20}$$

Thus we can rewrite 5.13 as

$$w(t) = \frac{E_t[\Delta S(t)h]}{E_t[\Delta S(t)\Delta S(t)]} \tag{5.21}$$

which gets rid of the iterated conditional expectations and dramatically decreases the computational effort. So far we have only made use of the fact that the discounted prices $t \to S(t)$ and $t \to C(t)$ are Q-martingales but not of the particular form of the dynamics of the underlying S. The minimum variance delta of an option can now be added to the class Option as the following method:

```
public double minimumVarianceDelta
(int whichProbability, int t, int nPath, Trigger rebalance)
{
    double sum_1=0, sum_2=0,
            deltaSt, deltaCt;

    double[] S=underlying.get_S();   //price path
    underlying.simulationInit(t);    //sets pathCounter to zero

    switch(whichProbability){

        case Flag.RISK_NEUTRAL_PROBABILITY:
        // compute delta as E_t[ΔS(t)h]/E_t[(ΔS(t))^2]

        for(int n=0; n<nPath; n++)
        {
            // move path to time s of next hedge trade to get ΔS(t) = S(s) − S(t)
            int s=underlying.pathSegment(whichProbability,t,rebalance);
            deltaSt=S[s]−S[t];
            // move path to horizon T to get h
            underlying.newPathBranch(whichProbability,s);
            double h=currentDiscountedPayoff();

            sum_1+=deltaSt * h;
            sum_2+=deltaSt * deltaSt;
        }

        return sum_1/sum_2;          //E_t[Z(t)h]/E_t[(ΔS(t))^2]
```

```
        case Flag.MARKET_PROBABILITY:
        // compute delta as E_t[ΔS(t)ΔC(t)]/E_t[ΔS(t)^2]

        for(int n=0; n<nPath; n++)
        {
            // path computation to time s of next hedge trade
            int s=underlying.pathSegment(whichProbability,t,rebalance);

            // compute C[s]
            if(hasAnalyticPrice)C[s]=discountedAnalyticPrice(s);
            else C[s]=discountedMonteCarloPrice(s,nPath);

            deltaSt=S[s]−S[t];
            deltaCt=C[s]−C[t];

            sum_1+=deltaSt * deltaCt;
            sum_2+=deltaSt * deltaSt;
        } // end for n

        return sum_1/sum_2;                    //E_t[ΔS(t)ΔC(t)]/E_t[ΔS(t)^2]

    } // end switch

    return 0; // keep the compiler happy

} // end minimumVarianceDelta
```

5.2.3 Monte Carlo Deltas

The dynamics of the discounted asset price $S(t)$ under the risk neutral probability Q has the driftless form

$$dS(t) = S(t)\sigma(t)dW^Q(t) \tag{5.22}$$

Let us now use this dynamics to simplify 5.21. Rewriting (5.22) in integral form yields

$$\Delta S(t) = S(s) - S(t) = \int_t^s S(u)\sigma(u)dW^Q(u). \tag{5.23}$$

From this and properties of the Ito integral (martingale property and form of the quadratic variation) it follows that

$$E_t[(\Delta S(t))^2] = \int_t^s S^2(u)\sigma^2(u)du \tag{5.24}$$

Replacing the integrand by its value at $t = u$ (a constant) we obtain the approximation

$$\Delta S(t) \simeq S(t)\sigma(t)[W^Q(s) - W^Q(t)] \quad \text{and} \tag{5.25}$$

$$E_t[(\Delta S(t))^2] \simeq S^2(t)\sigma^2(t)(s-t). \tag{5.26}$$

In practical computation the quantity $W^Q(s) - W^Q(t)$ has to be computed as the path $S(t) \to S(s)$ is simulated. Entering this into 5.21 and observing that $S(t)\sigma(t)$ is a known constant at time t, we obtain the *Monte Carlo delta* at time t as

$$w(t) = \frac{E_t[(W^Q(s) - W^Q(t))h]}{S(t)\sigma(t)(s-t)}. \tag{5.27}$$

Obviously our approximations are good only if $s - t$ is small. For example if rehedging takes place at each time step and dt is the size of the time step, we have $s = t + dt$, $W^Q(s) - W^Q(t) = \sqrt{dt}\ Z(t)$, where $Z(t)$ is the standard normal increment driving the time step $t \to t + dt$ ($t \to t + 1$ in discrete time) and the Monte Carlo delta assumes the form

$$w(t) = \frac{E_t[Z(t)h]}{\sigma(t)S(t)\sqrt{dt}}. \tag{5.28}$$

See also [CZ02]. For small dt this approximates the analytic delta $w(t) = (\partial c/\partial s)(t, S(t))$. We implement it with $s = t + 1$:

```
public double monteCarloDelta(int whichProbability, int t, int nPath)
{
    double sum=0, mean=0;
    double[] Z=underlying.get_Z(),   //driving standard normal increments
             S=underlying.get_S();   //discounted price path
    double sigmaSqrtdt=underlying.get_sigmaSqrtdt(t);   //σ(t)√dt

    underlying.simulationInit(t);       //sets pathCounter to zero

    // compute E_t[Z(t)h]
    for(int n=0; n<nPath; n++)
    {
        underlying.newPathBranch(whichProbability,t);
        double h=discountedPayoff();
        sum+=Z[t]*h;
    }

    mean=sum/nPath;                     // E_t[Z(t)h]
    return mean/(S[t]*sigmaSqrtdt);

} // end monteCarloDelta
```

5.2.4 Quotient Deltas

The simplest attempt to make the tracking error $e(t) = w(t)\Delta S(t) - \Delta C(t)$ on $[t, s)$ small is to chose the weight $w(t)$ so that the conditional mean $E_t[e(t)]$ is equal to zero leading to the equation $w(t)E_t[\Delta S(t)] = \Delta C(t)$. If $E_t[\Delta S(t)] \neq 0$, then

$$w(t) = \frac{E_t[\Delta C(t)]}{E_t[\Delta S(t)]} \tag{5.29}$$

is the unique weight satisfying this equation. These weights can be computed only in the market probability since in the risk neutral probability $E_t[\Delta S(t)] = 0$. They also perform better than analytic deltas at least in the case of a European call on a basic Black-Scholes asset.

5.3 Analytic approximations

There is no problem with computing the conditional expectations involved in a single hedge weight. However if we want to carry out a simulation of the hedge over tens of thousands of paths rebalancing the hedge dozens of times along each path then hundreds of thousands of such weights will have to be computed and that makes the Monte Carlo computation of conditional expectations forbidding. Instead we need analytic formulas which approximate the above weights. We give such formulas for a basic Black-Scholes asset for hedging in the market probability. More specifically it is assumed that the discounted asset price $S(t)$ follows the dynamics

$$dS(t) = S(t)\left[(\mu - r)dt + \sigma dW(t)\right],$$

where μ and σ are the (constant) drift and volatility of the asset, r is the riskfree rate and W a standard Brownian motion. It is assumed that the asset pays no dividend and that the option price $C(t)$ satisfies

$$C(t) = c(t, S(t))$$

where $c = c(t, s)$ is a twice continuously differentiable function. We then set

$$\begin{aligned} &d(t,s) = \frac{\partial c}{\partial s}, \quad D(t) = d(t, S(t)), \\ &\frac{\partial D}{\partial S}(t) = \frac{\partial d}{\partial s}(t, S(t)) \quad \text{and} \quad \frac{\partial D}{\partial t}(t) = \frac{\partial d}{\partial t}(t, S(t)). \end{aligned}$$

In other words $D(t)$ is the classical analytic delta and $\partial D/\partial t$ and $\partial D/\partial S$ are the sensitivities of this delta with respect to changes in time and price

of the underlying. We need the following functions:

$$\begin{aligned} f(\alpha,\beta,u) &= \alpha\, exp(\beta u) \\ F(\alpha,\beta) &= \int_0^{\Delta t} f(\alpha,\beta,u)du = (\alpha/\beta)[exp(\beta\Delta t)-1], \\ h(\alpha,\beta,u) &= u\, f(\alpha,\beta,u) \quad \text{and} \\ H(\alpha,\beta) &= \int_0^{\Delta t} h(\alpha,\beta,u)du = (\alpha/\beta)\left[(\beta\Delta t-1)exp(\beta\Delta t)+1\right]. \end{aligned}$$

Set

$$\begin{aligned} \kappa_n &= n(\mu-r)+\tfrac{1}{2}\sigma^2(n^2-n), \quad \text{and} \\ q &= exp(\kappa_1\Delta t) = exp[(\mu-r)\Delta t] \end{aligned}$$

and note that $\kappa_1 = \mu - r$, $\kappa_2 = 2(\mu-r)+\sigma^2$ and $\kappa_3 = 3(\mu-r)+3\sigma^2$. The derivation of the following formulas an be found in [Mey02b]:

5.3.1 Analytic minimum variance deltas

The minimum variance deltas $w(t)$ from (5.13) computed in the market probability can be approximated as

$$w(t) \simeq D(t) + \alpha\frac{\partial D}{\partial t}(t) + \beta S(t)\frac{\partial D}{\partial S}(t), \quad \text{where} \tag{5.30}$$

$$\alpha = \frac{\sigma^2 H_1 - [F_3 - 2F_2 + F_1]}{\sigma^2[q^2 exp(\sigma^2\Delta t) - 2q + 1]}, \quad \beta = \frac{3F_2 - F_3 - 2F_1}{q^2 exp(\sigma^2\Delta t) - 2q + 1},$$

$$\begin{aligned} H_j &= qH(\kappa_1+\sigma^2, \kappa_j + j\sigma^2) - H(\kappa_1,\kappa_j) \quad \text{and} \\ F_j &= qF(\kappa_1+\sigma^2, \kappa_j + j\sigma^2) - F(\kappa_1,\kappa_j). \end{aligned}$$

The constants α, β depend on μ, r, σ and Δt but not on t. This means that they can be computed in advance of a simulation run of the hedge error which greatly speeds up the computation.

5.3.2 Analytic quotient deltas

The quotient deltas $w(t)$ from (5.29) can be approximated as follows

$$\begin{aligned} w(t) &\simeq D(t) + \phi\frac{\partial D}{\partial t}(t) + \psi S(t)\frac{\partial D}{\partial S}(t), \quad \text{where} \qquad (5.31) \\ \phi &= \frac{\sigma^2 H(\kappa_1,\kappa_1) - F(\kappa_1,\kappa_3) + 2F(\kappa_1,\kappa_2) - F(\kappa_1,\kappa_1)}{\sigma^2(q-1)} \quad \text{and} \\ \psi &= \frac{3F(\kappa_1,\kappa_2) - F(\kappa_1,\kappa_3) - 2F(\kappa_1,\kappa_1)}{q-1}. \end{aligned}$$

Again the constants ϕ, ψ depend on μ, r, σ and Δt but not on t and can therefore be precomputed.

5.3.3 Formulas for European calls.

Let us collect some formulas which are useful in testing the above deltas when hedging a European call in the simple Black-Scholes model. We assume that the discounted asset price $S(t)$ follows the dynamics

$$dS(t) = S(t)\left[(\mu - r)dt + \sigma dW(t)\right]$$

under the market probability P with constant drift μ, volatility σ and short rate r. We also assume that the asset S does not pay any dividends. The *discounted* price $C(t)$ of the European call on S with strike price K is then given by $C(t) = V(t, S(t))$, where

$$\begin{aligned} &c(t,s) = sN(d_+) - K\,e^{-rT}N(d_-), \quad \text{with} \\ &d_\pm = \frac{log(s) - log(K) + rT}{\Sigma(t)} \pm \frac{1}{2}\Sigma(t) \quad \text{and} \quad \Sigma(t) = \sigma\sqrt{T-t}, \end{aligned}$$

for $t \in [0, T]$ and $s > 0$. Here $N(x)$ is the cumulative normal distribution function and so $N'(x) = (2\pi)^{-1/2}e^{-x^2/2}$. Observe that

$$\frac{1}{2}(d_+^2 - d_-^2) = \frac{1}{2}(d_+ + d_-)(d_+ - d_-) = log(se^{rT}/K)$$

which implies

$$\frac{N'(d_-)}{N'(d_+)} = se^{rT}/K, \quad \text{that is,} \quad sN'(d_+) = Ke^{-rT}N'(d_-).$$

Note also that

$$\frac{\partial d_\pm}{\partial s} = \frac{1}{s\Sigma(t)} \quad \text{and} \quad \frac{\partial d_\pm}{\partial t} = \frac{1}{2(T-t)}d_\mp$$

where the signs on the right hand side of the second equation are reversed. From this it follows that

$$\begin{aligned} &d(t,s) = \frac{\partial c}{\partial s} = N(d_+), \\ &s\frac{\partial d}{\partial s} = N'(d_+)\frac{1}{\Sigma(t)} \quad \text{and} \quad \frac{\partial d}{\partial t} = N'(d_+)\frac{d_-}{2(T-t)}. \end{aligned}$$

In case the asset pays a dividend continuously with yield q the delta must be corrected as follows:

$$d(t,s) = exp(-q(T-t))N(d_+).$$

5.4 Hedge Implementation

The following are the constituents which make up a hedge: the underlying asset, the option to be hedged and the trading strategy which hopefully will provide the gains from trading with which we intend to cover the option payoff:

```
public class Hedge
{
    Asset underlying;                  // the underlying on which the option is written
    Option option;                     // option to be hedged
    TradingStrategy hedgeStrategy;     // gains from trading hedge option payoff

    // Constructor
    public Hedge(/* all the above */){ /* initialize as usual */ }
    ...
} // end Hedge
```

The transaction costs for the hedgeStrategy have a fixed component (cost per trade) and proportional component (cost per share per trade). When choosing these parameters note that our simulation simulates the hedge of an option on *one* share of the underlying. Thus, if you typically transact in lots of 1000 shares and this costs you 12$ per trade choose fixed_trc=0.012. A reasonable value for proportional transaction costs is 0.2 (the bid ask spread alone will do that to you).

Assuming the option is sold for the martingale price (this disregards the friction in real markets) the gain (profit and loss) of hedging the option along one new path of the underlying is now easily computed as follows (remember everything is discounted to time $t = 0$):

```
public double newDiscountedHedgeGain()
{
    // price option is sold at
    double option_premium=option.get_C()[0];

    // compute a new asset price path and the corresponding gains from trading:
    double trading_gains=hedgeStrategy.newDiscountedGainsFromTrading();

    // option payoff
    double option_payoff=option.currentDiscountedPayoff();

    // hedge profit
    return trading_gains+option_premium-option_payoff;

} // end HedgeGain
```

Needless to say it is unwise to sell an option for its martingale price. This price is based on numerous assumptions such as instant reaction to price changes in the underlying, zero transaction costs and unlimited borrowing at the riskfree rate none of which is satisfied in existing markets. Instead we will try to shift the premium upward. If the trading strategy employed is unaffected (for example all delta hedges are independent of the premium received) this shifts the mean of the hedge gain by the same amount and has no other effect.

To compute statistics associated with the hedge gain we set it up as a random variable. For the sake of simplicity we disregard conditional expectations, that is, the method disregards the time parameter t:

```
public RandomVariable discountedHedgeGain()
{
    return new RandomVariable(){

        public double getValue(int t)
        {
            // this computes a new path of the underlying also:
            return newDiscountedHedgeGain();
        }
   }; // end return new
} // end DiscountedHedgeGain
```

With this setup it is now easy to compute the mean and standard variation of the hedge profit and loss. The following method computes the pair (mean, standard deviation) as a double[2]:

```
public double[] hedgeStatistics(int nPaths)
{
    nHedgeTrades=0;       //initializes the simulation
    return discountedHedgeGain.meanAndStandardDeviation(nPaths);
}
```

A hedge simulation can be extremely lengthy in particular if hedge deltas are computed under the market probability in the absence of analytic formulas for the option price. Thus the class Hedge.java contains a similar method which reports the computational progress and projects the time needed. The reader is invited to read the source code.

5.5 Hedging the Call

It is now time to examine how the various hedge weights perform in the case of a European call on a basic Black-Scholes asset. The following code

snippet shows how easy it is to set up an option hedge and compute and display a histogram of the hedge payoff. The hedge is to be rebalanced whenever the price of the underlying has moved 12% and classic analytic deltas are used as hedge weights:

```
ConstantVolatilityAsset
asset=new ConstantVolatilityAsset(T,dt,nSignChange,S_0,r,q,mu,sigma);

Call call=new Call(K,asset);

double q=0.12;                  // price percent change triggering a hedge trade
Trigger rebalance=new TriggerAtPercentChange(asset, q);

int nBranch=0;                  // number of branches per conditional expectation
                                // irrelevant since we use analytic deltas
double fixed_trc=0.02;          // fixed transaction cost
double prop_trc=0.05;           // proportional transaction cost

Hedge callHedge=new DeltaHedge(asset,call,rebalance,
                               Flag.A_DELTA,Flag.MARKET_PROBABILITY,
                               nBranch,fixed_trc,prop_trc);

int nPaths=20000;               // compute the hedge payoff along 20000 paths
int nBins=100;                  // number of histogram bins

RandomVariable callHedgeGains=callHedge.discountedHedgeGains();

callHedgeGains.displayHistogram(nPaths,nBins);
```

The parameters nBranch (number of branches spent on each conditional expectation involved in computing the deltas) and MARKET_PROBABILITY (the probability under which these conditional expectations are computed) are irrelevant here since we are using analytic deltas. Nonetheless these parameters must be passed to the hedge constructor.

Figure 5.7 is the histogram of the gains from hedging a call using analytic deltas and rebalancing the hedge whenever the price of the underlying changes 10%. The parameters are as follows: $S(0) = 50$, strike $K = 55$, time to expiration $\tau = 1$ year, drift $\mu = 0.3$, volatility $\sigma = 0.4$, dividend yield $q = 4\%$, and riskfree rate $r = 5\%$. The price of the call is 5.993. The source code is in the file Examples/CallHedgeHistogram.java.

The gains are seen on the x-axis. The values on the y-axis can be interpreted as values of the probability density of the distribution.

Compare this with the payoff from the unhedged position in Figure 5.8. This distribution has a point mass at 5.993, the call price and also the largest

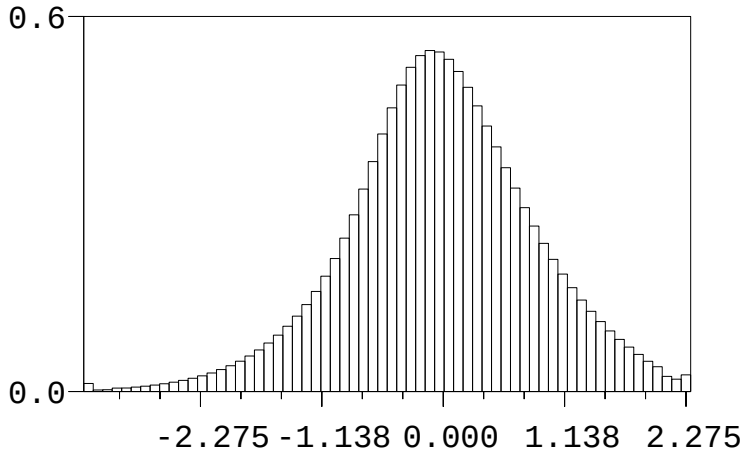


Figure 5.7: Hedged call

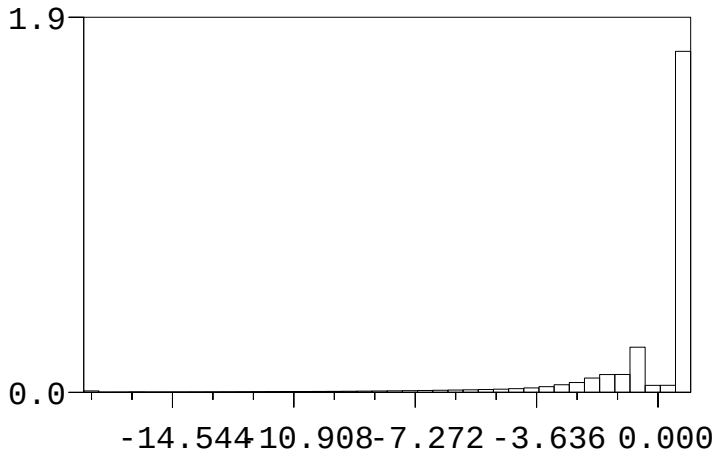


Figure 5.8: Unhedged call

possible payoff. This payoff is received with the positive probability that the call ends up out of the money.

Let us now analyze the performance of hedging a call in more detail. The classes used to do this have main methods which bring up a rudimentary graphical user interface (GUI). The values of the parameters are set by the use of sliders. To measure the quality of the hedge we report the mean and standard deviation of the payoff of the hedge as a percentage of the price of the call. The following is a brief description of the various classes. The reader wishing to inspect the source code should disregard the code which drives the graphical user interface. This code has been generated by wizards and is correspondingly ugly. In each class the code carrying out the actual computation is collected in a method called mainComputation() and listed toward the end of the class.

5.5.1 Mean and standard deviation of hedging the call

Class: Examples.Hedging.CallHedgeStatistics. To measure the performance of a hedge for the call we compute the mean and standard deviation of the gains from hedging as a percentage of the (Black-Scholes) call price. All deltas (analytic, Monte Carlo, minimum variance) are allowed. Analytic deltas are very fast and are included always. Wether or not the other deltas are included is chosen in the GUI.

Two policies for rebalancing the hedge are applied. *Periodic rebalancing* rebalances the hedge at equal time intervals with a certain number of hedge trades specified in advance (the actual number of hedge trades can differ slightly from the specified target).

Reactive rebalancing reacts to price changes in the underlying: the hedge is rebalanced as soon as the price of underlying has moved by a trigger percentage. The trigger percentage is specified by the user.

The computation of Monte Carlo and minimum variance deltas is time consuming. The program reports on estimated time to completion after the first few cycles. If you don't like what you see kill it by closing the window.

The price paths of the underlying are simulated under the market probability since this probability controls the price changes with which the hedger has to cope.

In choosing the transaction costs recall that our simulation simulates the hedge of a call on one share of the underlying. In real life the size of the transactions will be much larger. Assume that your average hedge trade involves 500 shares and that such a trade costs you 10 dollars. In this case set the fixed transaction costs to 10/500=0.02 dollars. The proportional

transaction costs (costs per share transacted) can be set much higher. Note for example the bid ask spread.

Rebalancing the hedge reacting to price changes significantly beats periodic rebalancing at equal time intervals even if the mean number of hedge trades is the same. This is also unsurprising. Would you enter a risky position and then check it every other Wednesday? This is exactly the approach of the periodic hedger.

As the call is moved more and more out of the money and hence its price decreases it becomes increasingly harder to hedge, that is, the hedge standard deviation increases as a percent of the call price. This indicates that we should study the dependence of the hedge statistics on the strike price.

5.5.2 Call hedge statistics as a function of the strike price.

Class: Examples.Hedging.DrawCHGraphs_1. Let us examine the mean and standard deviation of hedging a call as a function of the strike price K for n strikes evenly spaced in the interval $[K_{min}, K_{max}]$. Both mean and standard variation are reported as a percent of the call price. The hedge is rebalanced at each time step. Graphs of both the means and standard deviations are displayed on screen and saved as PNG files. The graph of the standard deviation shows a dramatic deterioration in the quality of the hedge as the call strike increases regardless of the type of hedge delta employed (transaction costs were set to zero).

Classic analytic deltas and the (highly accurate) analytic approximations for minimum variance and market deltas are used in the hedge. The asset dynamics is

$$dS(t) = S(t)\left[(\mu - r)dt + \sigma dW(t)\right]$$

with constant drift μ, volatility σ and risk free rate r. The hedge is rebalanced at each time step and the size of the time steps is chosen to be $\Delta t = 0.05$ (fairly large). In a rough sense the size of the drift term is $(\mu - r)\Delta t$ while the size of the volatility term is $\sigma\sqrt{\Delta t}$. In other words the volatility will dominate the drift unless μ is dramatically larger than σ.

Moreover the volatility term will dominate the drift term to increasing degree as $\Delta t \to 0$. The Greek deltas (referred to as analytic deltas below) are designed for perfect replication and continuous trading ($\Delta t = 0$). In this case the drift no longer matters at all and in fact does not enter into the computation of the Greek deltas.

Minimum variance and quotient deltas on the other hand are designed to take the drift and size of the time step (inter hedge periods) into account.

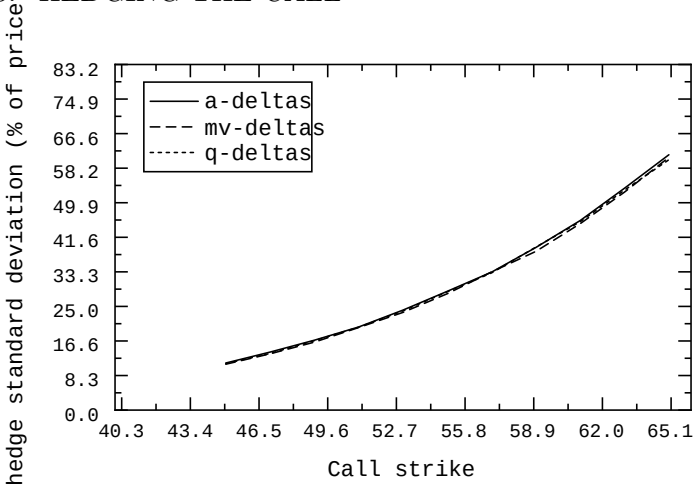


Figure 5.9: Hedge standard deviation, $\mu = 0.3$, $\sigma = 0.4$

For practical applications that means that you must correctly forecast the drift of the underlying until option expiration. Thus you must hedge with an opinion regarding the drift.

While minimum variance deltas are designed to minimize the hedge variance, quotient deltas are designed to minimize the hedge mean (over each hedge period). Let's see how the results stack up.

For high volatility low drift ($\mu = 0.3$, $\sigma = 0.4$) we expect little difference in the performance of the different weights regarding the hedge variance (Figure 5.9). Indeed there is no recognizable difference. The same is not true for the hedge *means*. Quotient deltas show the best tracking performance while minimum variance deltas err on the side of positive hedge profits.

Greek deltas on the other hand incur losses in the mean (figure 5.10). The picture changes in a high drift low volatility situation ($\mu = 0.8$, $\sigma = 0.2$).

Greek deltas now show visible performance deterioration in the hedge standard deviation while minimum variance deltas perform best (figure 5.11).

The difference is even more dramatic regarding the hedge mean (figure 5.12). Since the price of the call is now much less particularly for far out of the money calls the negative mean associated with Greek deltas is becoming a problem while minimum variance deltas show a tidy profit. Note the near perfect tracking of quotient deltas. The directory pictures/CHGraphs contains coloured PNG versions of these graphs. The reader can investigate other parameter settings by running Examples.Hedging.DrawCHGraphs_1.

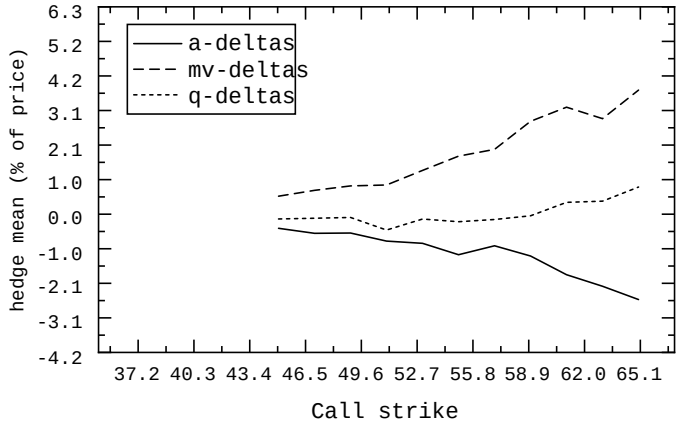


Figure 5.10: Hedge means, $\mu = 0.3$, $\sigma = 0.4$

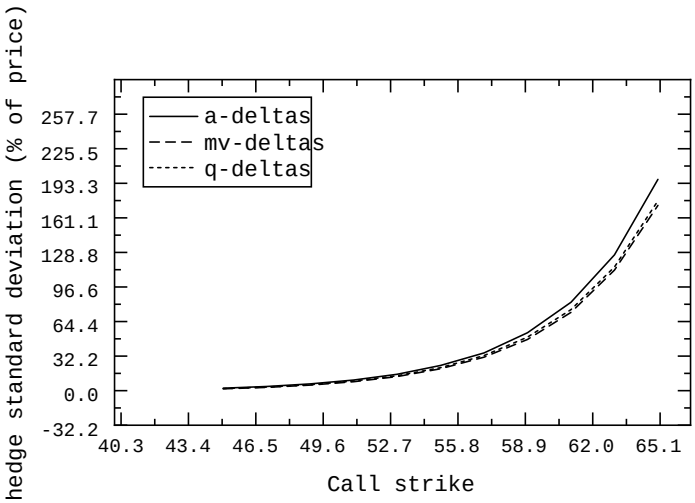


Figure 5.11: Hedge standard deviation, $\mu = 0.8$, $\sigma = 0.2$

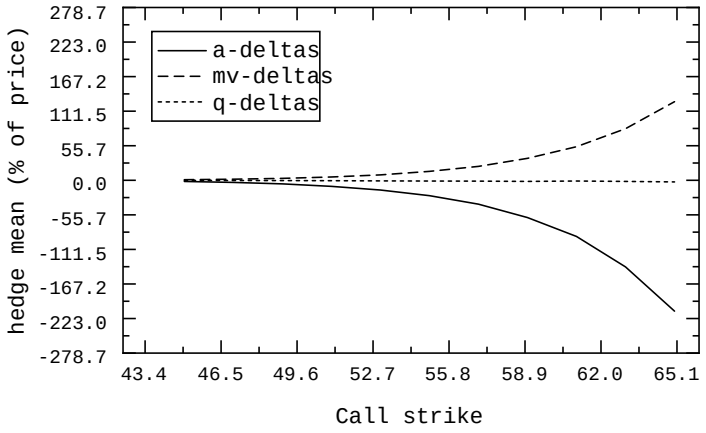


Figure 5.12: Hedge means, $\mu = 0.8$, $\sigma = 0.2$

5.5.3 Call hedge statistics as a function of the volatility hedged against.

File: Examples/DrawCHGraphs_2.java. The volatility σ of the underlying is a parameter necessary to compute the analytic deltas employed in hedging the call. In our ideal world we have the luxury of knowing the constant volatility of the underlying. In reality the future realized volatility is unknown. A hedger hedging with analytic deltas must choose the volatility σ from which these deltas are computed (hedge volatility). This brings up the interesting question how the quality of the hedge is affected by misjudging the true volatility of the underlying.

The class DrawCHGraphs_2.java computes the mean and standard variation of hedging a call as a function of the hedge volatility σ for volatilities σ evenly spaced in the interval $[\sigma_{min}, \sigma_{max}]$.

The hedge is rebalanced at each time step (figure 5.13). Unsurprisingly misjudging the true volatility decreases the quality of the hedge. The true volatility of the underlying in this example was $\sigma = 0.4$.

5.6 Baskets of Assets

Trading in markets with only one risky asset is interesting for the pricing and hedging of options written on a single asset. In general however one will trade more than one asset. Let us therefore consider a market consisting of

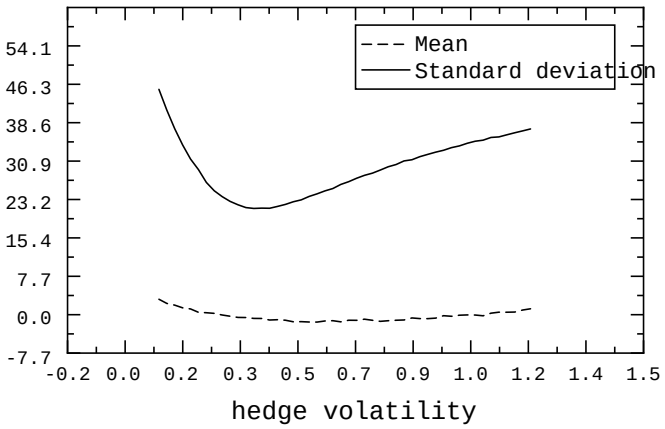


Figure 5.13: Hedge standard deviation

a riskless bond $B(t)$ satisfying

$$B(t) = exp(rt),$$

where r is the constant riskfree rate and $m+1$ risky assets $S_0, S_1, \ldots, S_m$. As before we work with *discounted* prices and let $S_j(t)$ denote the discounted price of the asset S_j at time t. Let P denote the market probability and Q the risk-neutral probability. We shall restrict ourselves to the simple Black-Scholes dynamics

$$\begin{aligned} dS_i(t) &= S_i(t)\left[(\mu_i - r)dt + \nu_i \cdot dW(t)\right] && (5.32) \\ &= S_i(t)\left[(\mu_i - r)dt + \nu_{i0}dW_0(t) + \ldots + \nu_{im}dW_m(t)\right], && (5.33) \end{aligned}$$

where $W(t) = (W_0(t), W_1(t), \ldots, W_m(t))'$ is an $m+1$-dimensional standard Brownian motion and $\nu_i = (\nu_{i0}, \nu_{i1}, \ldots, \nu_{im})' \in R^{m+1}$ a constant vector. As usual the prime denotes transposition, that is, all vectors are column vectors and the dot denotes the dot product.

The coordinate Brownian motions $W_i(t)$ are independent one dimensional Brownian motions and are called the *(risk) factors* and the vectors ν_i the *factor loadings*. More precisely ν_{ij} is the loading of the factor $W_j(t)$ in the asset S_i. Usually however we do not think in terms of the factor loadings ν_i themselves. Instead we work with the *volatilities* σ_i and *correlations* ρ_{ij} defined as

$$\sigma_i = \|\nu_i\|, \quad \text{and} \quad \rho_{ij} = u_i \cdot u_j,$$

where u_i is the unit vector $u_i = \sigma_i^{-1}\nu_i$. Writing

$$V_i(t) = u_i \cdot W(t)$$

it is easily seen that V_i is a standard one dimensional Brownian motion and

$$dV_i(t) = u_i \cdot dW(t)$$

(by linearity of the stochastic differential) and so

$$dS_i(t) = S_i(t)\left[(\mu_i - r)dt + \sigma_i dV_i(t)\right]. \tag{5.34}$$

Consequently σ_i is seen to be the (annual) volatility of S_i (more precisely of the return process $log(S_i)$). Moreover the Brownian motions V_i are no longer independent, instead the covariation processes $\langle V_i, V_j\rangle$ satisfy

$$d\langle V_i, V_j\rangle_t = \rho_{ij}dt. \tag{5.35}$$

Observing that the return processes $log(S_i)$ satisfy

$$d\,log(S_i) = (\mu_i - r - \tfrac{1}{2}\sigma_i^2)dt + \sigma_i dV_i(t) \tag{5.36}$$

it follows that the covariation processes of the returns satisfy

$$d\langle log(S_i), log(S_j)\rangle_t = \sigma_i\sigma_j\rho_{ij}dt$$

which justifies the interpretation of ρ_{ij} as the instantaneous correlation of the returns $d\,log(S_i(t)$, $d\,log(S_j(t)$. In fact in the very special case of the above dynamics it is not necessary to resort to infinitesimals or covariation processes. One can easily verify that the covariance and correlation of the random variables $log(S_i(t))$, $log(S_j(t))$ are given by

$$\begin{aligned} Cov\left[log(S_i(t), log(S_j(t)\right] &= \sigma_i\sigma_j t \quad \text{and} \\ Corr\left[log(S_i(t), log(S_j(t)\right] &= \rho_{ij}. \end{aligned}$$

If we switch to the risk neutral probability Q, then the discounted asset prices follow the dynamics

$$dS_i = S_i(t)\left[-q_i dt + \sigma_i dV_i(t)\right] \quad \text{and} \tag{5.37}$$

$$d\,log(S_i) = (-q_i - \tfrac{1}{2}\sigma_i^2)dt + \sigma_i dV_i(t), \tag{5.38}$$

where q_i denotes the continuous dividend rate of the asset S_i. We will assume that the matrix of factor loadings $\nu = (\nu_{ij})$ has full rank. This matrix however will not be used as a model parameter. Instead we work with the volatilities σ_i and correlations ρ_{ij}. Since the matrix ν of factor loadings has full rank the matrix u with columns u_i has full rank also and consequently the matrix $\rho = (\rho_{ij}) = (u_i \cdot u_j) = uu'$ is positive definite.

5.6.1 Discretization of the time step

Recall that we are usually not given the factor loadings ν_i but instead have to work from the volatilities σ_i and correlations ρ_{ij}. From the dynamics of the returns $log(S_i)$ we see that the time step $t \to s$ is carried out as follows

$$S_i(s) = S_i(t) exp\left[d_i \Delta t + \sigma_i \sqrt{\Delta t}\, Y_i(t)\right], \tag{5.39}$$

where $\Delta t = s - t$, the drift delta d_i is given by $d_i = \mu_i - r - \frac{1}{2}\sigma_i^2$ in the case of the market probability and $d_i = -q_i - \frac{1}{2}\sigma_i^2$ in the case of the risk neutral probability and $Y_i(t) = (\Delta t)^{-1/2}(V_i(s) - V_i(t))$ is a standard normal random variable. Observing that

$$Y_i(t) = u_i \cdot (\Delta t)^{-1/2}(W(s) - W(t))$$

and that $(\Delta t)^{-1/2}(W(s) - W(t))$ is a vector with independent standard normal components it follows that

$$Cov(Y_i(t), Y_j(t)) = u_i \cdot u_j = \rho_{ij}.$$

It follows that $Y(t) = (Y_0(t), \ldots, Y_m(t))'$ is a zero mean multinormal vector with covariance matrix $\rho = (\rho_{ij})$. The mean and covariance matrix completely determine the joint distribution of the multinormal vector $Y(t)$.

To drive the time step $t \to s$ such a vector can be generated as follows: the positive definite correlation matrix admits a *Cholesky factorization*

$$\rho = RR'$$

where the matrix R (the *Cholesky root* of ρ) is lower triangular and uniquely determined. If $Z(t) = (Z_0(t), \ldots, Z_m(t))'$ is a column vector of independent standard normal random variables $Z_j(t)$, then $Y(t) = RZ(t)$ is a mean zero multinormal vector with covariation matrix ρ exactly as needed (see Appendix B.1). In other words we can generate the $Y_i(t)$ as

$$Y_i(t) = \sum\nolimits_{j \le i} R_{ij} Z_j(t).$$

Note how the lower triangularity of C reduces the number of floating point multiplications thereby speeding up the simulation of paths.

Implementation This suggests the following implementation. Volatilities σ_i, correlations ρ_{ij} and initial asset prices $S_i(0)$ are model parameters.

Suppose that continuous time has been discretized into integral multiples $t * dt$, $t = 0, 1, \ldots, T$ of the time step dt and that the array `S[][]` contains the

path $t \mapsto S_i(t)$, in the array S[i][], that is $S[i][t] = S_i(t * dt)$. The Cholesky root R of the correlation matrix ρ and the quantities

$$\begin{aligned} \text{sigmaSqrtdt[i]} &= \sigma_i\sqrt{dt} \\ \text{marketDriftDelta[i]} &= \mu_i - r - \frac{1}{2}\sigma_i^2 \quad \text{and} \\ \text{riskNeutralDriftDelta[i]} &= -q_i - \frac{1}{2}\sigma_i^2 \end{aligned}$$

are all computed and stored by the constructor. The time step $t \to s$ in discrete time then assumes the form

```
double f=Math.sqrt(s-t);

for(int i=0; i<=m; i++)
{
    double  Yi=0,
            driftDelta_i=...// depending on the probability
            g=driftDelta_i*(s-t);

    for(int j=0; j<=i; j++) Yi+=R[i][j]*Random.STN();
    S[i][s]=S[i][t]*Math.exp(g+sigmaSqrtdt[i]*f*Yi);
}
```

Note that there is no approximation involved at all. The distribution of $S_i(s)$ conditioned on $S_i(t)$ is reproduced exactly. It is very useful to be able to make long time steps since we will then of course compute the path $t \to S(t)$ only at times t for which $S(t)$ is really needed.

The case $s = t+1$ is special since full paths and path branches are made up of such steps. We might want to allocate an array Z[][] of size $T \times m$ to be filled with the standard normal increments needed for the path or path branch. These can the be reused through random sign changes (*antithetic paths*). The call Random.STN() above is then replaced with Z[t][j].

5.6.2 Trading and Hedging

A trading strategy investing in the asset vector S is now a vector valued process $t \mapsto w(t) = (w_0(t), \ldots, w_m(t))' \in R^{m+1}$. The quantity $w_i(t)$ is the number of shares of the asset S_i (the *weight* of S_i) held at time t. If $w(t)$ is constant on the interval $[t, s]$, then

$$w(t) \cdot \Delta S(t) = \sum\nolimits_{j \le m} w_j(t)\Delta S_j(t),$$

where $\Delta S(t) = S(s) - S(t)$, is the *discounted* gain from holding the position on the interval $[t, s]$. Recall that we are using discounted asset prices $S_j(t)$.

The weights $w_j(t)$ must obviously be $\mathcal{F}_t$-measurable, that is, it must be possible to determine the value of $w_j(t)$ at time t.

Suppose now that we have another asset with discounted price $C(t)$ and that we want to hedge a short position of one share of C by trading in the asset vector S. We assume only that the discounted prices $S_i(t)$ and $C(t)$ are Q-martingales but make no other assumptions on the price dynamics of S and C. The hedge is rebalanced at times

$$0 = \tau_0 < \tau_1 < \ldots < \tau_n = T.$$

The τ_l can be stopping times and the weights $w(t)$ are assumed to be constant on each interval $[\tau_l, \tau_{l+1})$. Fix l and write $t = \tau_l$, $s = \tau_{l+1}$. Then the combined position which is short one share of C and long $w(t)$ shares of S shows the discounted gain

$$e(t) = w(t) \cdot \Delta S(t) - \Delta C(t)$$

on the interval $[t, s]$. This quantity is best viewed as the tracking error of the hedge over the interval $[t, s]$. It is not hard to show that the $\mathcal{F}_t$-measurable weights $w(t)$ which minimize the L^2-norm $\|e(t)\|_2^2 = E[e(t)^2]$ of the tracking error are the solutions of the system of linear equations

$$\sum\nolimits_{j \le m} A_{ij}(t) w_j(t) = b_i(t) \tag{5.40}$$

where

$$A_{ij}(t) = E_t\left[\Delta S_i(t) \Delta S_j(t)\right] \quad \text{and} \tag{5.41}$$

$$b_i(t) = E_t\left[\Delta S_i(t) \Delta C(t)\right]. \tag{5.42}$$

This is true regardless of the probability in which the weights and L^2-norm are computed. If this computation is carried out in the risk neutral probability Q then it can also be shown that the trading strategy using the above weights minimizes the variance of the cumulative hedge error over the entire interval $[0, T]$. The computation of the weights $w(t)$ can be quite laborious since many conditional expectations have to be evaluated.

If $C(t)$ is an asset of which the price dynamics is known the right hand side $b_i(t)$ can be computed by straightforward Monte Carlo simulation of the single time step $t \to t + \Delta t$. In case C is a European option written on the basket S with known (discounted) payoff

$$h = C(T)$$

but unknown price process $C(t)$ we have to compute $C(s)$ as $E_s(h)$. Observing that $\Delta S(t)$, $\Delta C(t)$ are both $\mathcal{F}_s$-measurable we can write

$$\Delta S(t)\Delta C(t) = E_s\left[\Delta S(t)(h - C(t))\right]$$

and with this $b_i(t)$ becomes

$$b_i(t) = E_t\left[\Delta S_i(t)(h - C(t))\right].$$

This quantity can be computed by Monte Carlo simulation of paths to the horizon T. If the computation is carried out in the risk-neutral probability Q, then $E_t\left[\Delta S_i(t)\right] = 0$ (martingale condition) and the expression for $Y_i(t)$ can be simplified further to

$$b_i(t) = E_t\left[\Delta S_i(t)h\right] \quad \text{(risk neutral probability } Q \text{ only)}.$$

Connection to Greeks. The weights $w(t)$ are related to the classical deltas and can be used for numerical computation of these deltas if $\Delta t = s - t$ is chosen to be sufficiently small. More precisely assume that the discounted asset price $C(t)$ satisfies

$$C(t) = c(t, S(t)) = c(t, S_0(t), \ldots, S_m(t)),$$

for some continuously differentiable function $c = c(t, s_0, s_1, \ldots, s_m)$ and write

$$\theta(t) = \frac{\partial c}{\partial t}(t, S(t)) \quad \text{and} \quad D_i(t) = \frac{\partial c}{\partial s_i}(t, S(t)).$$

In other words the $D_i(t)$ are the usual Greek deltas. A first order Taylor approximation of the function $c = c(t, s_0, \ldots, s_m)$ yields

$$\begin{aligned} \Delta C(t) &= C(t + \Delta t, S(t) + \Delta S(t)) - C(t, S(t)) && (5.43)\\ &\cong \theta(t)\Delta t + \textstyle\sum_{j \le m} D_j(t)\Delta S_j(t) && (5.44)\\ &\cong \textstyle\sum_{j \le m} D_j(t)\Delta S_j(t) && (5.45) \end{aligned}$$

where the time sensitivity $\theta(t)$ has been ignored. Here the deltas $D_i(t)$ are $\mathcal{F}_t$-measurable. Multiplying this with $\Delta S_i(t)$ and taking the conditional expectation E_t we obtain

$$\begin{aligned} E_t[\Delta S_i(t)\Delta C(t)] &\cong \textstyle\sum_{j \le m} E_t[\Delta S_i(t)\Delta S_j(t)]D_j(t) && (5.46)\\ &= \textstyle\sum_{j \le m} A_{ij}(t)D_j(t). && (5.47) \end{aligned}$$

In other words the classical deltas $\tilde{w}_i(t) = D_i(t)$ form an approximate solution of (5.40). This justifies a numerical computation of the deltas $D_i(t)$ as solutions of (5.40) for small $\Delta t = s - t$ (if these Greeks are needed) or the use of the deltas in hedging if they are available by analytic formula and time steps are small. However if time steps are large then the actual solutions $w_i(t)$ of (5.40) can visibly outperform the classical deltas $D_i(t)$ in a hedge of the asset C.

So far everything is true for any asset price dynamics not only the very restrictive dynamics (5.34). It only has to be assumed that the asset prices $S_i(t)$ and $C(t)$ are martingales under Q (see [Mey02b]). Let us now simplify (5.40) in the case of the dynamics (5.34).

Assume that the times $\tau_k = t$ and $\tau_{k+1} = t+\Delta t$ of hedge trade k, $k+1$ are constants. This case would occur for example if the hedge is rebalanced at regular time intervals regardless of price developments. In this case and for our simple asset price dynamics we can get exact formulas for the coefficients $A_{ij}(t)$ of (5.40). Setting

$$\lambda_i = exp(d_i\,\Delta t), \quad \beta_i = exp\left[\left(d_i + \tfrac{1}{2}\sigma_i^2\right)\Delta t\right]$$
$$U_i = \sigma_i\sqrt{\Delta t}\,Y_i(t) \quad \text{and} \quad U_{ij} = U_i + U_j$$

we have $\Delta S_i(t) = S_i(t)\left[\lambda_i exp(U_i) - 1\right]$ (from (5.39)) and so

$$\Delta S_i(t)\Delta S_j(t) = S_i(t)S_j(t)\left[\lambda_i\lambda_j exp(U_{ij}) - \lambda_i exp(U_i) - \lambda_j exp(U_j) + 1\right].$$

Recall that $E(\sigma Z) = \sigma^2/2$, for any standard normal variable Z. Note that U_{ij} is a mean zero normal variable with variance $(\sigma_i^2 + \sigma_j^2 + 2\sigma_i\sigma_j\rho_{ij})\Delta t$ and that all of U_i, U_j, U_{ij} are independent of the σ-field $\mathcal{F}_t$. With this

$$\begin{aligned} A_{ij}(t) &= E_t\left[\Delta S_i(t)\Delta S_j(t)\right] \\ &= S_i(t)S_j(t)\left[\beta_i\beta_j exp\left(\sigma_i\sigma_j\rho_{ij}\Delta t\right) - \beta_i - \beta_j + 1\right] \\ &= S_i(t)S_j(t)B_{ij}, \end{aligned} \tag{5.48}$$

where

$$B_{ij} = \beta_i\beta_j exp\left(\sigma_i\sigma_j\rho_{ij}\Delta t\right) - \beta_i - \beta_j + 1. \tag{5.49}$$

We can now substitute this into equation (5.40), divide by $S_i(t)$ and switch to the variables $x_j(t) = S_j(t)w_j(t)$ to reduce (5.40) to $Bx(t) = v(t)$, that is,

$$x(t) = B^{-1}v(t),$$

where the $m \times m$-matrix B with entries (5.49) is state independent and

$$v_i(t) = S_i(t)^{-1}b_i(t).$$

The matrix inverse B^{-1} can be precomputed and stored to speed up the calculation. The weights $w_i(t)$ are recovered as $w_i(t) = S_i(t)^{-1}x_i(t)$. For small Δt we can approximate the exponential $exp(x)$ in (5.49) with $1+x$ to obtain

$$B_{ij} \simeq (\beta_i - 1)(\beta_j - 1) + \beta_i\beta_j\sigma_i\sigma_j\rho_{ij}\Delta t$$

Dropping all terms which are of order $(\Delta t)^2$ we obtain $B_{ij} \simeq \sigma_i\sigma_j\rho_{ij}\Delta t$, equivalently

$$A_{ij}(t) \simeq S_i(t)S_j(t)\sigma_i\sigma_j\rho_{ij}\Delta t.$$

On the right of (5.40) we can approximate $\Delta S_i(t)$ with $S_i(t)\sigma_i\sqrt{\Delta t}\,Y_i(t)$ to write

$$b_i(t) \simeq S_i(t)\sigma_i\sqrt{\Delta t}\,E_t\left[hY_i(t)\right].$$

Enter all this into (5.40), cancel $S_i(t)\sigma_i\sqrt{\Delta t}$ and switch to the variables $x_i(t) = w_i(t)S_i(t)\sigma_i\sqrt{\Delta t}$. The linear system (5.40) becomes $\rho\, x(t) = y(t)$, that is,

$$x(t) = \rho^{-1}E_t[hY(t)]$$

The matrix inverse ρ^{-1} can be precomputed and stored. The weights $w_i(t)$ are then recovered as

$$w_i(t) \simeq \frac{x_i(t)}{S_i(t)\sigma_i\sqrt{\Delta t}}.$$

Only the vector $E_t[(hY(t)]$ has to be computed by Monte Carlo simulation. Here $Y(t) = (\Delta t)^{-1/2}\left(V(t+\Delta t) - V(t)\right)$ is the normalized increment of the driving Brownian motion $t \mapsto V_t$ over the time step $t \to t + \Delta t$.

The Greek deltas $D_i(t)$ can then be approximated with the weights $w_i(t)$ computed using a small time step Δt. For larger time steps a hedge based on deltas $D_i(t)$ does not perform as well as the actual solutions of (5.40).

Chapter 6

The Libor Market Model

Early interest rate models were models of the (instantaneous) risk free rate $r(t)$ associated with the risk free bond $B(t)$. These models tried to express some desirable properties of interest rates (such as mean reversion) and usually offered only a small number of parameters and Brownian motions to drive the interest rate dynamics. The main appeal of these models was the lack of alternatives.

A significant advance was made in [DHM92] which took up the modelling of a whole continuum of instantaneous forward rates $f(t,T)$, the rate charged at time t for lending over the infinitesimal interval $[T, T+dt]$. The crucial insight was the fact that the absence of arbitrage between zero coupon bonds enforces a relation between the drift $\mu(t,T)$ and volatility processes $\sigma(t,T)$ in the dynamics

$$df(t,T) = \mu(t,T)dt + \sigma(t,T) \cdot dW(t)$$

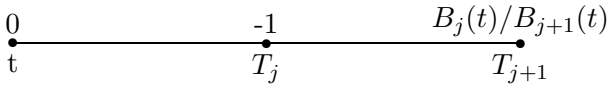
of the forward rates which allows the drifts to be computed from the volatilities. However instantaneous forward rates are not well adapted to the prevailing reality of cash flows at discrete times. A finite vector of forward rates for lending over the periods between successive cash flows is exactly what is needed in this case. The corresponding rates are called *forward Libors* and will be defined below.

After pioneering work by Brace, Musiela and Gatarek [ABM97] the dynamics of the forward Libor process found elegant treatment at the hands of Jamshidian [Jam97] and it is this treatment which we follow with some modifications and extensions.

6.1 Forward Libors

Let $0 = T_0 < T_1 < \ldots < T_n$ be a sequence of dates at which cash flows occur, referred to as the *tenor structure*, write $\delta_j = T_{j+1} - T_j$ and let $B_j(t)$ denote the cash price of the zero coupon bond maturing at time T_j. To simplify the exposition this bond is assumed to be defined on the entire interval $I = [0, T_n]$ and to be a continuous semimartingale on a filtered probability space $(\Omega, (\mathcal{F}_t)_{0 \le t \le T_n}, P)$. The probability P represents the market probability and the σ-field $\mathcal{F}_t$ the information available at time t.

Let $t \le T_j$. By trading in the zero coupon bonds B_j, B_{j+1}we can implement at time t a loan of one dollar over the future time interval $[T_j, T_{j+1}]$: at time t sell one zero coupon bond maturing at time T_j and with the proceeds $B_j(t)$ buy $B_j(t)/B_{j+1}(t)$ zero coupon bonds maturing at time T_{j+1}. With the understanding that the zero coupon bonds are redeemed at the time of maturity, this strategy induces the following cash flows:



Forward Libor $L_j(t)$ for the accrual interval $[T_j, T_{j+1}]$ is the simple rate of interest corresponding to these cash flows defined as as

$$B_j(t)/B_{j+1}(t) = 1 + \delta_j L_j(t). \tag{6.1}$$

By abuse of terminology the quantities $X_j = \delta_j L_j$ will also be referred to as forward Libors and are assumed to be strictly positive. Note the basic relation $B_{j+1}X_j = B_j - B_{j+1}$. The (vector) process

$$t \in I \mapsto X(t) = (X_0(t), X_1(t), \ldots, X_{n-1}(t))'$$

is called the *(forward) Libor process*. According to the corresponding convention about zero coupon bonds all forward Libors $X_j(t)$ are defined on the entire interval I and hence so is the process $X(t)$.

6.2 Dynamics of the forward Libor process

The next two sections assume some familiarity with the concept of the stochastic integral with respect to a continuous semimartingale and the covariation process associated with two continuous semimartingales. The reader can safely skip these sections. Only the dynamics (6.2) below will be used later.

Let us briefly review the relevant facts. If U_j, V_j, X_j, Y_j are continuous semimartingales on $[0,T]$ then the stochastic integrals $\int_0^t U_j(s)dX_j(s)$ and $\int_0^t V_j(s)dY_j(s)$ are defined for all $t \in [0,T]$ and the equality

$$\sum\nolimits_j U_j(t)dX_j(t) = \sum\nolimits_j V_j(t)dY_j(t)$$

is defined to be a shorthand for

$$\sum\nolimits_j \int_0^t U_j(s)dX_j(s) = \sum\nolimits_j \int_0^t V_j(s)dY_j(s)$$

for all $t \in [0,T]$. If the Y_j are local martingales and the X_j bounded variation processes then the the right hand side is a local martingale while the left hand side is a bounded variation process implying that both sides are zero (a continuous local martingale which is also a bounded variation process is constant).

Let X, Y be continuous semimartingales on $[0,T]$. Then the covariation process $\langle X, Y\rangle_t$ is a bounded variation process on $[0,T]$ and we have the stochastic product rule

$$d(X(t)Y(t)) = X(t)dY(t) + Y(t)dX(t) + d\langle X, Y\rangle_t.$$

If X and Y have paths contained in open sets F and G on which the functions f and g are continuously differentiable, then the processes $f(X)$ and $g(Y)$ are well defined continuous semimartingales and (as a consequence of Ito's lemma)

$$d\langle f(X), g(Y)\rangle_t = f'(X(t))g'(Y(t))d\langle X, Y\rangle_t.$$

In particular if X and Y are positive and $f(t) = g(t) = log(t)$ this assumes the form

$$d\langle X, Y\rangle_t = X(t)Y(t)d\langle log(X), log(Y)\rangle_t.$$

Let us now return to the forward Libor process of the preceeding section. The quotients $H_j = B_j/B_n$ are the forward zero coupon bond prices at the terminal date $t = T_n$. They are also the prices of the zero coupon bonds B_j

in the B_n-numeraire. We assume that these forward prices are continuous local martingales in some probability Q which is equivalent to P.

We fix such a probability and refer to it as the *(terminal) forward martingale measure*. Note that Q is a numeraire measure associated with the B_n numeraire. Obviously the cash prices $B_j(t)$ themselves cannot be assumed to be local martingales under any probability since these prices have a tendency to increase as maturity approaches.

The existence of the measure Q is equivalent to the no arbitrage condition NFLVR (no free lunch with vanishing risk) defined in [DS94] for the system of forward zero coupon bond prices B_j/B_n. The proof of this fact is one of the most fundamental results in finance and is far too complicated to be presented here. The reader should merely note that the existence of the measure Q is indeed related to the absence of arbitrage between zero coupon bonds.

We will now use the local martingale condition to derive the dynamics of the forward Libors $X_j = \delta_j L_j$ in the forward martingale measure Q. Note that

$$H_{j+1}X_j = H_j - H_{j+1}$$

is a Q-local martingale and that the zero coupon bonds and forward Libors are semimartingales under Q also. The continuous semimartingale $Z_j = log(X_j)$ then has a decomposition

$$Z_j = log(X_j) = A_j + M_j,$$

where A_j is a continuous bounded variation process and M_j a continuous Q-local martingale with $M_j(0) = 0$. Note that Z_j is the "return" process associated with X_j and that A_j and M_j can be thought of as the "drift" and "volatility" of the return Z_j respectively (the Q-local martingale M_j is driftless but of unbounded variation whereas A_j is of bounded variation but not driftless). By Ito's lemma we have

$$dX_j = e^{Z_j}dZ_j + \frac{1}{2}e^{Z_j}d\langle Z_j\rangle = X_j[dM_j + dB_j]$$

where $dB_j = dA_j + \frac{1}{2}d\langle M_j\rangle$. We have

$$\begin{aligned} d(H_{j+1}X_j) &= X_j dH_{j+1} + H_{j+1}dX_j + d\langle X_j, H_{j+1}\rangle \\ &= X_j dH_{j+1} + H_{j+1}X_j dM_j + [H_{j+1}X_j dB_j + d\langle X_j, H_{j+1}\rangle]. \end{aligned}$$

Since here $H_{j+1}dB_j + d\langle X_j, H_{j+1}\rangle$ is the differential of a bounded variation process while all the remaining terms are differentials of local martingales it follows that

$$H_{j+1}X_j dB_j + d\langle X_j, H_{j+1}\rangle = 0.$$

Note that $H_j = (1+X_j)(1+X_{j+1})\dots(1+X_{n-1})$ and so

$$log(H_{j+1}) = \sum_{k=j+1}^{n-1} log(1+X_k)$$

and that $d\langle X_j, H_{j+1}\rangle = X_j H_{j+1} d\langle log(X_j), log(H_{j+1}\rangle$ to rewrite the previous equation as

$$dB_j + \sum_{k=j+1}^{n-1} d\langle log(X_j), log(1+X_k)\rangle = 0,$$

equivalently

$$dB_j = -\sum_{k=j+1}^{n-1} \frac{1}{X_j(1+X_k)} d\langle X_j, X_k\rangle = -\sum_{k=j+1}^{n-1} \frac{X_k}{1+X_k} d\langle M_j, M_k\rangle$$

and so

$$dX_j = X_j \left[-\sum_{k=j+1}^{n-1} \frac{X_k}{1+X_k} d\langle M_j, M_k\rangle + dM_j \right].$$

This is the general dynamics of continous forward Libors and it identifies the drift of the forward Libors in terms of the "volatilities" M_j of the Libor returns Z_j.

Finally let us assume that $W = (W_1, W_2, \dots, W_m)'$ is a Brownian motion in R^m under Q and that the zero coupon bonds and hence the forward Libors are adapted to the filtration generated by W. Then the adapted Q-local martingale M_j can be represented as a stochastic integral

$$M_j(t) = \int_0^t \nu_j(s) \cdot dW(s)$$

for some integrable process $\nu_j \in L(W)$, that is, $\nu_j(t)$ is a predictable R^m-valued process satisfying

$$\int_0^{T_n} \|\nu_j(s)\|^2 ds < \infty, \quad Q\text{-almost surely.}$$

With this we have $dM_j(t) = \nu_j(t) \cdot dW(t)$ and the Libor dynamics assumes the form

$$dX_j(t) = X_j(t) \left[-\sum_{k=j+1}^{n-1} \frac{X_k(t)}{1+X_k(t)} \nu_j(t) \cdot \nu_k(t) dt + \nu_j(t) \cdot dW(t) \right] \quad (6.2)$$

The individual components $W_i(t)$ of the Brownian motion W are called the *factors* and the vector processes $\nu_j \in L(W)$ are referred to as the *factor loadings*. The numerical process $\sigma_i(t) = \|\nu_i(t)\|$ is called the (instantaneous) volatility of X_i. The factor loadings are often also called volatilities in the literature.

The absence of arbitrage between zero coupon bonds thus determines the Libor drifts from the factor loadings $\nu_j(t)$. In the next section we show that for each Brownian motion W and given factor loadings $\nu_j \in L(W)$ we can construct a solution X of (6.2) with associated zero coupon bonds B_j such that the system of forward prices B_j/B_n is arbitrage free.

6.3 Libors with prescribed factor loadings

Now we tackle the problem of the existence of a solution of (6.2).

Theorem 6.3.1 *Let Q be a probability on the filtered space $(\Omega, (\mathcal{F}_t))$, W a Brownian motion under Q, and $\nu_j \in L(W)$ a W-integrable process and $x_j > 0$, for $j < n$.*

Then there exists a solution X of (6.2) with $X_j(0) = x_j$ and such that the forward price B_j/B_n in the associated system of zero coupon bonds B_j is a Q-local martingale, for $j < n$. Moreover B_n can be chosen to be any positive continuous semimartingale satisfying

$$B_n(T_j)(1 + X_j(T_j)) \dots (1 + X_{n-1}(T_j)) = 1,$$

for all $j \leq n$.

Proof. Recall that for integrable processes $\mu \in L(dt)$ and $\nu \in L(W)$ the solution of the equation

$$dX(t) = X(t)[\mu(t)dt + \nu(t) \cdot dW(t)]$$

can be obtained explicitly as

$$X(t) = X(0)exp\left(\int_0^t \mu(s)ds\right) \mathcal{E}_t(\nu \bullet W),$$

where $\nu \bullet W$ denotes the integral process $\int_0^t \nu(s) \cdot dW(s)$ and $\mathcal{E}_t(\nu \bullet W)$ is the exponential local martingale

$$\mathcal{E}_t(\nu \bullet W) = exp\left(-\frac{1}{2}\int_0^t \|\nu(s)\|^2 ds + \int_0^t \nu(s) \cdot dW(s)\right).$$

The triangular nature of the equations (6.2) allows a straightforward solution by backward induction from $j = n-1, n-2, ...0$. For $j = n-1$ the equation (6.2) reads

$$dX_{n-1}(t) = X_{n-1}(t)\nu_{n-1}(t) \cdot dW(t)$$

with solution

$$X_{n-1}(t) = X_{n-1}(0)\mathcal{E}_t(\nu_{n-1} \bullet W).$$

Assume that the processes $X_{j+1}, \ldots, X_{n-1}$ have been constructed as solutions of (6.2) and rewrite (6.2) as $dX_j(t) = X_j(t)[\mu_j(t)dt + \nu_j(t) \cdot dW(t)]$ where the drift

$$\mu_j(t) = -\sum_{k=j+1}^{n-1} \frac{X_k(t)}{1 + X_k(t)} \nu_j(t) \cdot \nu_k(t)$$

does not depend on $X_j(t)$. Thus the solution is simply

$$X(t) = X(0)exp\left(\int_0^t \mu_j(s)ds\right) \mathcal{E}_t(\nu_j \bullet W).$$

In this manner all the processes X_j are obtained and it remains to construct the corresponding zero coupon bonds B_j and to examine wether the quotients B_j/B_n are all Q-local martingales. Indeed let $B_n(t)$ be any positive continuous semimartingale satisfying

$$B_n(T_j)(1 + X_j(T_j)) \ldots (1 + X_{n-1}(T_j)) = 1,$$

for all $j \leq n$ and define the remaining zero coupon bonds as

$$B_j(t) = B_n(t)(1 + X_j(t)) \ldots (1 + X_{n-1}(t)).$$

Then the preceeding condition for B_n is exactly the zero coupon bond condition $B_j(T_j) = 1$. Moreover

$$B_j(t)/B_{j+1}(t) = 1 + X_j(t)$$

so that the $X_j(t)$ are indeed the forward Libors associated with the zero coupon bonds $B_j(t)$. Fix $j < n$ and note that

$$Y_j := B_j/B_n = (1 + X_j) \ldots (1 + X_{n-1})$$

Taking logarithms we have $log(Y_k) = \sum_{i=j}^{n-1} log(1 + X_i)$, where

$$d\, log(1 + X_i) = \frac{1}{1 + X_i} dX_i - \frac{1}{2}\left(\frac{1}{1 + X_i}\right)^2 d\langle X_i \rangle. \tag{6.3}$$

Here $d\langle X_i\rangle_t = X_i(t)^2\|\nu_i(t)\|^2 dt$ (from (6.2)). Setting $X_i/(1+X_i) = K_i$ we can rewrite (6.3) as

$$d\,log(1+X_i) = (K_i/X_i)dX_i - \frac{1}{2}\|K_i\nu_i\|^2 dt. \tag{6.4}$$

Set

$$\gamma_i(t) = \sum_{j=i}^{n-1} K_j\nu_j(t), \quad 0 \le i < n,$$

and $\gamma_n(t) = 0$. Then (6.2) can be rewritten as

$$dX_i(t) = -X_i(t)\nu_i(t)\cdot\gamma_{i+1}(t)dt + X_i(t)\nu_i(t)\cdot dW(t)$$

and so $(K_i/X_i)dX_i = -\gamma_{i+1}\cdot K_i\nu_i dt + K_i\nu_i\cdot dW(t)$. Substituting this into (6.4) and observing that

$$K_i\nu_i(t) = \gamma_i(t) - \gamma_{i+1}(t),$$

we obtain

$$\begin{aligned} d\,log(1+X_i) &= -\gamma_{i+1}\cdot K_i\nu_i(t)dt - \frac{1}{2}\|\gamma_i-\gamma_{i+1}\|^2 dt + K_i\nu_i(t)\cdot dW(t) \\ &= \left[-\gamma_{i+1}\cdot(\gamma_i-\gamma_{i+1}) - \frac{1}{2}\|\gamma_i-\gamma_{i+1}\|^2\right]dt + K_i\nu_i(t)\cdot dW(t) \\ &= \frac{1}{2}\left[\|\gamma_{i+1}\|^2 - \|\gamma_i\|^2\right]dt + K_i\nu_i(t)\cdot dW(t). \end{aligned}$$

Summing over all $i = j,\ldots,n-1$ and observing that $\gamma_{n-1} = 0$ we obtain

$$d\,log(Y_j(t)) = -\frac{1}{2}\|\gamma_j(t)\|^2 dt + \gamma_j(t)\cdot dW(t).$$

from which it follows that

$$B_j/B_n = Y_j = Y_j(0)\mathcal{E}_t(\gamma_j \bullet W)$$

is indeed a Q-local martingale.

6.4 Choice of the factor loadings

In the greatest generality the factor loadings $\nu_i(t)$ can be any W-integrable process. However to obtain a computationally feasible Libor model this generality has to be restricted. From now on we assume that the factor loadings have the form

$$\nu_j(t) = \sigma_j(t)u_j, \tag{6.5}$$

where the u_j are *constant unit vectors* in R^n and set

$$\rho_{jk}(t) = \rho_{jk} = u_j \cdot u_k. \tag{6.6}$$

Recall also that the volatilities $\sigma_j(t)$ are assumed to be deterministic. A Libor model satisfying these assumptions is called a *Libor market model* abbreviated as LMM.

Special correlation structures. The ρ_{jk} are interpreted as the correlations of the infinitesimal log-Libor increments $dY_j(t)$, $dY_k(t)$, where $Y_j(t) = log(X_j(t))$. It follows from (6.6) that the matrix $\rho = (\rho_{jk})$ is positive semidefinite.

A concrete Libor market model does not provide the vectors u_j explicitly. Rather the matrix ρ is parametrized in terms of a small number of paramters. However this is done the resulting matrix must be positive semidefinite. There are many papers dealing with the problem of parametrizing a suitable family of correlation matrices and then calibrating the parameters to observed caplet and swaption prices. See for example [CS00], [CS99], [JR], [Bri].

The correlation matrix ρ itself is a possible parameter albeit of very high dimension. This makes model calibration very difficult since an enormous parameter space has to be searched. Consequently we restrict ourselves to special correlation structures which can be parametrized with a parameter of smaller dimension. For example Coffee and Shoenmakers [CS00] propose the following correlation structure:

$$\rho_{ij} = \frac{b_i \wedge b_j}{b_i \vee b_j} \tag{6.7}$$

where $b = (b_j)$ is a positive, nondecreasing sequence such that the sequence

$$j \mapsto b_j / b_{j+p}$$

is nondecreasing for each p. The sequence will be called the *correlation base*. Here $a \wedge b = \min\{a, b\}$ and $a \vee b = \max\{a, b\}$ as usual. Consequently $\rho_{ij} = b_i/b_j$ for $i \leq j$ and $\rho_{ij} = \rho_{ji}$.

In concrete models sequences b_j depending on only a few parameters will be chosen. For example $b_j = exp(\beta j^\alpha)$ is such a sequence.

To see that the matrix ρ in (6.7) is in fact positive semidefinite let (Z_k) be a sequence of independent standard normal variables. Set $b_{-1} = 0$,

$$a_k = \sqrt{b_k^2 - b_{k-1}^2}$$

and $Y_j = \sum_{k=0}^{j} a_k Z_k$. Then, for $i \leq j$ we have

$$Cov(Y_i, Y_j) = \sum_{k=0}^{i} a_k^2 = b_i^2 - b_{-1}^2 = b_i^2. \tag{6.8}$$

Thus $\sigma(Y_i) = b_i$ and so

$$Corr(Y_i, Y_j) = \frac{b_i^2}{b_i b_j} = \frac{b_i}{b_j} = \frac{b_i \wedge b_j}{b_i \vee b_j} = \rho_{ij}. \tag{6.9}$$

This implies that ρ is positive semidefinite. The vectors u_i and factor loadings $\nu_i(t)$ can be recovered from ρ as follows: use a Cholesky Decomposition to factor ρ as

$$\rho = uu'$$

(with u triangular) and let $u_i = r_i(u)$ be the ith row of u. Then $\rho_{ij} = u_i \cdot u_j$ and $\nu_i(t) = \sigma_i(t)u_i$. In any concrete model the log-Libor volatilities $\sigma_i(t)$ are specified as functions depending on a small number of parameters. The entire model is then determined from a small number of parameters and calibration of the model to observed market prices becomes possible.

For the correlation matrix ρ as in (6.9) ($\rho_{ij} = b_i/b_j$, $0 \leq i \leq j < n$) we can compute the Cholesky factorization explicitly. From (6.8) and (6.9) we know that

$$\rho_{ij} = \sum_{k=0}^{i} a_k^2/(b_i b_j) = \sum_{k=0}^{n-1} u_{ik} u_{jk} = (uu')_{ij}, \quad i \leq j,$$

where

$$u_{jk} = \begin{cases} a_k/b_j = b_j^{-1}\sqrt{b_k^2 - b_{k-1}^2} & : \quad k \leq j \\ 0 & : \quad k > j \end{cases}$$

Here $b_{-1} = 0$. The Coffee-Shoenmakers correlation structure is implemented in the class CS_FactorLoading.

We now proceed to examine several implementation of a Libor market model. Since the parameters of such a model are calibrated to observed caplet and swaption prices it will be necessary to derive exact or approximate analytic formulas for these prices in each of the models.

6.5 Discretization of the Libor dynamics

Every discretization of a continuous dynamics entails some approximation and with that comes approximation error. Unfortunately the Libor Market Model is very unforgiving in this respect. To understand why recall that

(6.2) is the dynamics of the Libor process in the terminal martingale measure Q at time T_n. This measure is the numeraire measure associated with the zero coupon bond $B_n(t)$ and so the value $c_t(h)$ of a random cashflow occurring at time $T = T_n$ can be computed as

$$c_t(h) = B_n(t)E_t^Q[h/B_n(T_n)] = B_n(t)E_t^Q[h]. \tag{6.10}$$

If the cashflow h occurs at any other time $T = T_j$ it has to be transported forward to time T_n before the above formula (6.10) can be applied. To transport the cashflow forward we multiply with the accrual factor

$$f = (1 + X_j(T))(1 + X_{j+1}(T)) \dots (1 + X_{n-1}(T)). \tag{6.11}$$

If the number $n - j$ of factors is large even small errors in the individual factors can compound to large overall errors. We will use caplets one third of the way out to the horizon as test cases. The payoff has to be compounded forward for a sizeable number of accrual intervals and is small in comparison to the accrual factor. Moreover we have an analytic formula for the caplet price as a benchmark.

Assume that volatilities $\sigma_i(s)$, factor loadings $\nu_i(s) = \sigma_i(s)u_i$ and correlations $\rho_{ij} = u_i \cdot u_j$ are as in the previous section and pass to the logarithms $Y_j(t) = log(X_j(t))$ in the Libor dynamics (6.2) to obtain

$$dY_i(s) = \mu_i(s)ds + \sigma_i(s)u_i \cdot dW(s) \tag{6.12}$$

where

$$\mu_i(s) = -\frac{1}{2}\sigma_i^2(s) - \sum_{j=i+1}^{n-1} F_j(s)\sigma_i(s)\sigma_j(s)\rho_{ij}, \tag{6.13}$$

and $F_j(s) = f(Y_j(s))$ with $f(y) = e^y/(1 + e^y)$.

6.5.1 Predictor-Corrector algorithm.

A discretization of this dynamics which proceeds in small time steps will be very slow because of the high dimension of the Libor process. The algorithm [CHJ] which will be employed allows us to take longer and hence fewer time steps. This speeds up the simulation of paths. Of course the size of the time steps is limited by the times at which the Libors have to be sampled. We can at most step from one sampling time to the next. A standard situation is that the Libors are sampled only at the reset times T_j. In this case one steps from time T_j to time T_{j+1}.

Such a time step is far to large if a standard Euler scheme is employed. Consider a general time step $t \to T$. Integrating (6.12) over the interval $[t, T]$ we obtain

$$Y(T) - Y(t) = m_i(t, T) + V_i(t, T), \tag{6.14}$$

where

$$m_i(t, T) = \int_t^T \mu_i(s)ds, \quad \text{and} \quad V_i(t, T) = \int_t^T \sigma_i(s)u_i \cdot dW(s) \tag{6.15}$$

The Ito integral $V_i(t) = \int_0^t \sigma_i(s)u_i \cdot dW(s)$ has already been dealt with in section 3.11. Recalling that $u_i \cdot u_j = \rho_{ij}$ the vector $V(t, T) = (V_i(t, T))$ has a multinormal distribution with mean zero and covariance matrix $C(t, T) = (C_{ij}(t, T))$ given by

$$C_{ij}(t, T) = \int_t^T \sigma_i(s)\sigma_j(s)\rho_{ij}ds$$

and can easily be sampled precisely as

$$V(t, T) = R(t, T)Z(t),$$

where $Z(t)$ is a standard normal vector and $R(t, T)$ is any matrix satisfying $C(t, T) = R(t, T)R(t, T)'$. Such matrices can be chosen to be upper or lower triangular (Cholesky decomposition) and the triangularity speeds up the computation. Moreover the matrix $C(t, T)$ and hence the marix $R(t, T)$ are state independent and hence can be precomputed and cached to speed up the simulation of the time step.

The drift step $m_i(t, T)$ will be approximated. Note first that $\int_t^T \sigma_i^2(s)ds = C_{ii}(t, T)$. Next approximate the term $F_j(s) = X_j(s)/(1 + X_j(s))$ in (6.13) with some constant f_j on the interval $[t, T]$. This yields the approximation

$$m_i(t, T) \simeq -\frac{1}{2} C_{ii}(t, T) - \sum_{j=i+1}^{n-1} f_j C_{ij}(t, T). \tag{6.16}$$

The crudest approximation uses $f_j = F_j(0) = X_j(0)/(1 + X_j(0))$. With this the drift step becomes state independent and can be precomputed and cached also resulting in very fast path computations (referred to as "X0"-paths in the Java source code documentation). Unfortunately this approximation is too crude to be useful.

The *accelerated predictor-corrector algorithm* uses this drift step approximation merely to compute *predicted values* for the Libors $X_j(T) =$

$exp(Y_j(T))$ and quantities $F_j(T) = X_j(T)/(1 + X_j(T))$. These predicted values are then used to obtain a better approximation

$$f_j = [F_j(t) + F_j(T)]/2.$$

With this the drift step (6.16) is recomputed to yield the *corrected* values for the Libors $X_j(T) = exp(Y_j(T))$. In this approach the drift (6.16) has to be computed once since the deterministic drift steps used to compute the predicted values are cached in advance.

The true *predictor-corrector algorithm* uses $f_j = F_j(t)$ to compute the drift step (6.16) used to obtain the predicted values and then proceeds as above. This is more accurate but since the drift step approximation used to compute the predicted values is no longer state independent it cannot be precomputed and cached. Consequently we have to compute the drift step twice.

In practice the accelerated predictor corrector algorithm (implemented in FastPredictorCorrectorLMM.h) performs just as well as a true predictor corrector algorithm (implemented in PredictorCorrectorLMM.h, LiborProcess.java). There is no noticeable difference but the simulation of paths is faster and the code somewhat simpler.

6.6 Caplet prices

The caplet $Cplt([T_i, T_{i+1}], \kappa)$ on $[T_i, T_{i+1}]$ with strike rate κ pays off $h = (X_i(T_i) - \kappa_i)^+$ at time T_{i+1}, where $\kappa_i = \delta_i \kappa$. Using the forward martingale measure $Q = P_{i+1}$ at time T_{i+1} this payoff can be valued as

$$c_i(t) = B_{i+1}(t) E_t^Q \left[(X_i(T_i) - \kappa_i)^+ \right]$$

since Q is the numeraire measure associated with the numeraire B_{i+1}. See Appendix C.1 and note that $B_{i+1}(T_{i+1}) = 1$. In the measure Q the quotient $B_i/B_{i+1} = 1 + X_i$ is a local martingale (traded assets divided by the numeraire) and hence so is X_i. Thus Appendix C.4 applies with $S_1 = X_i$, $S_2 = 1$, $P = P_{i+1}$ and $T = T_i$. The caplet price is given by the Black caplet formula

$$c_i(t) = B_{i+1}(t) \left[X_i(t) N(d_+) - \kappa_i N(d_-) \right], \tag{6.17}$$

where

$$d_\pm = \frac{log(X_i(t)/\kappa_i)}{\Sigma_i(t, T_i)} \pm \frac{1}{2} \Sigma_i(t, T_i).$$

and $\Sigma_i^2(t, T_i)$ is the quadratic variation of the logarithm $Y_i = log(X_i)$ on $[t, T_i]$ This is the cumulative volatility of returns on Libor X_i on the interval

$[t, T_i]$ to option expiry. Since X_i is driftless under Q and the factor loading is not affected by the switch from P_n to the measure Q we have

$$dX_i(t) = X_i(t)\sigma_i(t)dW^{i+1}(t)$$

and so

$$dY_i(t) = -\frac{1}{2}\sigma_i^2(t)dt + \sigma_i(t)dW^{i+1}(t),$$

where W^{i+1} is a Brownian motion under $Q = P^{i+1}$. It follows that

$$\Sigma_i^2(t, T_i) = \langle Y_i \rangle_t^{T_i} = \int_t^{T_i} \sigma_i^2(t)dt.$$

6.7 Swap rates and swaption prices

Let $S_{p,q}(t)$ denote the swap rate along $[T_p, T_q]$), that is, the rate at which a forward swap over the interval $[T_p, T_q]$ can be entered into at time t at no cost. Recall that this rate is computed from the zero coupon bonds B_j as

$$\begin{aligned} S_{p,q} &= (B_p - B_q)/B_{p,q} = (H_p - H_q)/H_{p,q} \quad \text{where} \\ H_{p,q} &= \textstyle\sum_{j=p}^{q-1} \delta_j H_{j+1} \end{aligned} \tag{6.18}$$

is the forward price of the annuity on $[T_p, T_q]$. Thus $S_{p,q}$ is a local martingale under the equivalent martingale measure Q associated with the numeraire $B_{p,q}$ (traded assets divided by the numeraire). According to Appendix C.1 the cash price $c(t)$ of any option with a single payoff of h at time T can be computed as

$$c(t) = S_{p,q}(t)E_t^Q\left[h/S_{p,q}(T)\right], \quad t \le T.$$

Consider in particular the forward swaption $Swpn(T, [T_p, T_q], \kappa)$, that is, the right to enter at time $T \le T_p$ into a swap with strike rate κ over the interval $[T_p, T_q]$. Here Libor is swapped against the constant rate κ. This swaption has payoff

$$h = (S_{p,q}(T) - \kappa)^+ B_{p,q}(T), \quad \text{that is,} \quad h/B_{p,q}(T) = (S_{p,q}(T) - \kappa)^+$$

at time T. Thus Appendix C.4 applies with $S_1 = S_{p,q}$, $S_2 = 1$ and $P = P_{p,q}$. An analytic approximation $c_{p,q}(t)$ to the swaption price is given as

$$\begin{aligned} c_{p,q}(t) &= B_{p,q}(t)[S_{p,q}(t)N(d_+) - \kappa N(d_-)], \quad \text{where} \\ d_\pm &= \Sigma_{p,q}(t,T)^{-1} log(S_{p,q}(t)/\kappa) \pm \frac{1}{2}\Sigma_{p,q}(t,T) \end{aligned} \tag{6.19}$$

and $\Sigma^2_{p,q}(t,T)$ is an $\mathcal{F}_t$-measurable estimate of the quadratic variation of the logarithm $Y_{p,q} = log(S_{p,q})$ on $[t,T]$:

$$\Sigma^2_{p,q}(t,T) = E_t\langle Y_{p,q}\rangle_t^T, \quad Y_{p,q} = log(S_{p,q}). \tag{6.20}$$

This is the forecast for the cumulative volatility of returns on the swap rate $S_{p,q}$ on the interval $[t,T]$ to option expiry. Here $p \geq 1$. If $p = 0$ the swaption has to be exercised immediately and the formula does not apply. Let us now find an estimate for the quadratic variation (6.20. Note that $H_m = (1+X_m)\dots(1+X_{n-1})$ and hence

$$\frac{\partial H_m}{\partial X_k} = 0,\ k < m \quad \text{and} \quad \frac{\partial H_m}{\partial X_k} = \frac{1}{1+X_k}H_m,\ k \geq m.$$

From this it follows that

$$Y_{p,q} = log(H_p - H_q) - log\left(\sum\nolimits_{j=p}^{q-1}\delta_j H_{j+1}\right) = f(X_p,\dots,X_{n-1})$$

where the function f satisfies

$$\frac{\partial f}{\partial X_k} = \frac{1}{1+X_k}\left[\frac{H_p}{H_p - H_q} - \frac{H_{p,k}}{H_{p,q}}\right],\ p \leq k < q \quad \text{and} \quad \frac{\partial f}{\partial X_k} = 0,\ k \geq q.$$

Setting

$$G_j(s) = X_j(s)\frac{\partial f}{\partial X_j}(X(s))$$

it follows that

$$\begin{aligned} d\langle Y_{p,q}\rangle_s &= \sum\nolimits_{j,k=p}^{q-1}\frac{\partial f}{\partial X_j}(X(s))\frac{\partial f}{\partial X_k}(X(s))dX_j(s)dX_k(s) \qquad (6.21)\\ &= \sum\nolimits_{j,k=p}^{q-1} G_j(s)G_k(s)\sigma_j(s)\sigma_k(s)\rho_{jk}ds. \end{aligned}$$

Set

$$C_{jk} = \int_t^T \sigma_j(s)\sigma_k(s)\rho_{jk}ds$$

let C be the matrix $C = (C_{jk})_{j,k=p}^{q-1}$ and $x = x(t)$ the $q-p$-dimensional vector with coordinates

$$x_j(t) = G_j(t) = \frac{X_j(t)}{1+X_j(t)}\left[\frac{H_p(t)}{H_p(t) - H_q(t)} - \frac{H_{p,j}(t)}{H_{p,q}(t)}\right],$$

for $p \leq j < q$. Now integrate (6.21) over $[t, T]$, approximate $x_j(s)$ with the value $x_j(t)$ at the left endpoint t and pull these quantities out of the integral. This yields the $\mathcal{F}_t$-measurable approximation

$$\Sigma^2_{p,q}(t,T) = E_t\langle Y_{p,q}\rangle^T_t \simeq \sum_{j,k=p}^{q-1} x_j(t)x_k(t)C_{jk} = (Cx, x).$$

The matrix C is positive semidefinite and hence can be factored as $C = RR'$ and we can then write

$$\Sigma_{p,q}(t,T) \simeq \sqrt{(Cx,x)} = ||R'x||.$$

This quantity is entered into formula 6.19 and yields the desired approximate analytic swaption price. This approximation is not as accurate as the analytic approximations in the driftless Libor market model introduced below. We will use it mainly a a source of synthetic data for calibration experiments. For a different approach see [Jae02], p167, and the references cited therein.

6.8 Libors without drift

The drift term in the Libor dynamics causes another problem: it is useless to try to speed up the simulation of paths by reducing the number of factors (see 3.11) since this has no effect on the computation of the drift term and that is exactly where the predictor- corrector algorithm spends most of its effort.

For this reason we want to get rid of the drift term alltogether. Instead of the Libors X_j we simulate some other variables U_j which are driftless under P_n and such that the Libors X_j are deterministic functions of the U_j. The variables

$$U_j = X_j(1 + X_{j+1})(1 + X_{j+2}) \dots (1 + X_{n-1}), \quad U_{n-1} = X_{n-1} \tag{6.22}$$

have this property. Indeed, setting $H_k = B_k/B_n$ (a P_n-local martingale), we have

$$U_j = X_j H_{j+1} = H_j - H_{j+1}$$

and so U_j is a local martingale under P_n (and hence driftless). From $H_n = 1$ and $H_j = U_j + H_{j+1}$ we obtain

$$H_j = 1 + U_j + \ldots + U_{n-1}, \quad (H_n = 1) \tag{6.23}$$

and so

$$X_j = \frac{U_j}{H_{j+1}} = \frac{U_j}{1 + U_{j+1} + \ldots + U_{n-1}}. \tag{6.24}$$

The U_j are the Libors X_j "accrued forward" to time time $t = T_n$. The relation between the X_j and the U_j is analogous to the relation between cash prices and forward prices. If the U_j are nonnegative they will follow a driftless dynamics of the form

$$dU_j(t) = U_j(t)\nu_j(t) \cdot dW(t) \tag{6.25}$$

(martingale representation theorems) and conversely the dynamics (6.25) ensures that the U_j and hence the Libors X_j are nonnegative. In order to take full advantage of the driftless nature of (6.25) we simulate the U_j with Gaussian (state independent) factor loadings ν_j. We assume that

$$\nu_j(t) = \sigma_j(t)u_j,$$

for some state independent volatility functions $\sigma_j(t)$ and (constant) unit vectors u_j and set $\rho_{ij} = u_i \cdot u_j$. As a consequence the logarithms

$$Y_j(t) = log(U_j(t))$$

which follow the dynamics

$$dY_j(t) = -\frac{1}{2}\sigma_j^2(t)dt + \nu_j(t) \cdot dW(t) \tag{6.26}$$

form a Gaussian vector process $Y(t)$. The time step $Y(t) \to Y(T)$ can be sampled precisely (see 3.11 for more details): the increment $Y(T) - Y(t)$ is independent of $\mathcal{F}_t$ (of earlier time steps) and multinormal with covariance matrix $C(t,T)$ given by

$$C_{ij}(t,T) = \int_t^T \nu_i(s) \cdot \nu_j(s)ds = \int_t^T \sigma_i(s)\sigma_j(s)\rho_{ij}ds. \tag{6.27}$$

and means

$$m_i(t,T) = -\frac{1}{2}\int_t^T \sigma_i^2(s)ds,$$

all of which are state independent and hence can be precomputed and cached. The Libors X_j themselves now have stochastic volatility. We will compute the factor loadings of the X_j from the factor loadings of the U_j below. It is a good idea to compute the accrual factors $H_j(t)$ along with the $U_j(t)$. Because of the backward recursion

$$H_n(t) = 1, \quad H_j = U_j + H_{j+1}$$

the computational effort is negligible. All other quantities of interest can easily be computed from the U_j and H_j, such as for example the Libors X_j (see (6.24), the forward price of the annuity

$$H_{p,q} = B_{p,q}/B_n = \sum_{j=p}^{q-1} \delta_j H_{j+1}$$

and the swap rate

$$S_{p,q} = (H_p - H_q)/H_{p,q}. \tag{6.28}$$

Zero coupon bonds $B_j(t)$ and the annuity $B_{p,q}$ cannot be computed at an arbitrary time t, their values are only determined at times $t = T_j$: for this t we have $B_j(t) = 1$ hence $H_j(t) = 1/B_n(t)$, that is

$$B_n(t) = 1/H_j(t), \quad t = T_j. \tag{6.29}$$

and so

$$B_i(t) = H_i(t)B_n(t) = H_i(t)/H_j(t), \quad t = T_j. \tag{6.30}$$

However we have to pay a price for this convenience and speed, the factor loadings of the Libors X_j are now state dependent. We will determine them in the next section.

6.8.1 Factor loadings of the Libors

The simulation of the U_j only pays off if we assume state independent factor loadings $\nu_j(t) = \nu_j^U(t)$ for the processes U_j. To see wether or not this is reasonable we relate these factor loadings to the factor loadings $\nu_j^X(t)$ of the foward Libors X_j. These processes follow a dynamics of the form

$$\begin{aligned} dX_j(t) &= dA_j(t) + X_j(t)\nu_j^X(t) \cdot dW(t) \quad \text{and} \\ dU_j(t) &= U_j(t)\nu_j^U(t) \cdot dW(t) \end{aligned} \tag{6.31}$$

where the process $A_j(t) = \int_0^t \mu_j(s)ds$ is the cumulative drift of X_j. All we need to know of this process is that it is of bounded variation. Now take stochastic differentials on both sides of the identity

$$X_j = \frac{U_j}{1 + U_{j+1} + \ldots + U_{n-1}} = \frac{U_j}{H_{j+1}}$$

using Ito's formula (Appendix C.5) on the right. Beause of the uniqueness of the decomposition of a semimartingale as a sum of a bounded variation

process and a local martingale the bounded variation part in Ito's formula on the right will have to be $dA_j(t)$ and so we obtain

$$dX_j(t) = dA_j(t) + \frac{1}{H_{j+1}(t)} dU_j(t) - \frac{U_j(t)}{H_{j+1}^2(t)} \sum_{k=j+1}^{n-1} dU_k(t).$$

substituting the right hand sides of (6.31) for $dX_j(t)$, $dU_j(t)$, cancelling $dA_j(t)$ and observing that $U_j/H_{j+1} = X_j$ we can rewrite this as

$$X_j(t)\nu_j^X(t) \cdot dW(t) = X_j(t) \left[\nu_j^U(t) - H_{j+1}(t)^{-1} \sum_{k=j+1}^{n-1} U_k(t)\nu_k^U(t) \right] \cdot dW(t).$$

and it follows that

$$\nu_j^X(t) = \nu_j^U(t) - H_{j+1}(t)^{-1} \sum_{k=j+1}^{n-1} U_k(t)\nu_k^U(t).$$

Strictly speaking this is an equality in the space of integrands $L(W)$, that is, equality almost surely in a suitable probability. We do not need the details here. We see that the Libor factor loading $\nu_j^X(t)$ is now state dependent. In the next two sections we will see that we can still price caplets and swaptions by approximate analytic formula.

6.8.2 Caplet prices

We have already seen in 6.6 that the price $c_i(t)$ of the caplet $Cplt([T_i, T_{i+1}], \kappa)$ with strike rate κ is given by

$$\begin{aligned} c_i(t) &\simeq B_{i+1}(t) \left[X_i(t)N(d_+) - \kappa_i N(d_-) \right], \quad \text{where} \\ d_\pm &= \Sigma_i(t, T_i)^{-1} log(X_i(t)/\kappa_i) \pm \frac{1}{2}\Sigma_i(t, T_i), \end{aligned} \tag{6.32}$$

$\kappa_i = \delta_i \kappa$ and $\Sigma_i^2(t, T_i)$ as an $\mathcal{F}_t$-measurable estimate of the quadratic variation of the logarithm $Y_i = log(X_i)$ on $[t, T_i]$:

$$\Sigma_i^2(t, T_i) \simeq \langle Y_i \rangle_t^{T_i}, \quad Y_i = log(X_i).$$

This is the forecast for the cumulative volatility of returns on Libor X_i on the interval $[t, T_i]$ to option expiry. In 6.6 this price was exact now it is a forecast because of the stochastic nature of the Libor volatilities. See Appendix (C.4. We have

$$\begin{aligned} Y_i &= log(X_i) = log(U_i) - log(1 + U_{i+1} + \ldots + U_{n-1}) \\ &:= f(U_i, U_{i+1}, \ldots, U_{n-1}). \end{aligned}$$

Let's apply Ito's formula (Appendix C.5) to get the quadratic variation of Y_i. Using

$$d\langle U_j, U_k\rangle_s = U_j(s)U_k(s)\sigma_j(s)\sigma_k(s)\rho_{jk}ds$$

and setting

$$G_j(s) = U_j(s)\frac{\partial f}{\partial U_j}(U(s))$$

we have

$$d\langle Y_i\rangle_s = \sum_{j,k=i}^{n-1} G_j(s)G_k(s)\sigma_j(s)\sigma_k(s)\rho_{jk}ds.$$

Now integrate this over $[t, T_i]$ and approximate the quantities $G_j(s)$ with the value $G_j(t)$ at the left end of the interval of integration. We can then pull the factor $G_j(t)G_k(t)$ out of the integral and obtain

$$\langle Y_i\rangle_t^{T_i} \simeq \sum_{j,k=i}^{n-1} G_j(t)G_k(t)C_{jk}, \quad \text{with} \quad C_{jk} = \int_t^{T_i} \sigma_j(s)\sigma_k(s)\rho_{jk}ds. \quad (6.33)$$

Factoring the matrix $C = (C_{jk})_{j,k=i}^{n-1}$ as $C = RR'$ (with R upper triangular for greatest speed) we can rewrite this as

$$\langle Y_i\rangle_t^{T_i} \simeq (Cx, x) = \|R'x\|^2, \quad (6.34)$$

where x is the vector $x = (G_i(t), G_{i+1}(t), \ldots, G_{n-1}(t))$. Let us note that

$$G_j(t) = \begin{cases} 1 & : \quad j = i \\ -U_j(t)/H_{i+1}(t) & : \quad i < j < n. \end{cases} \quad (6.35)$$

and that the G_j depend on i as well as j. This yields the approximate caplet price (6.32) with

$$\Sigma_i(t, T_i) = \sqrt{(Cx, x)} = \|R'x\|.$$

The approximation shows excellent agreement with Monte Carlo caplet prices.

6.8.3 Swaption prices

In 6.7 we have seen that the price of the swaption $Swpn(T, [T_p, T_q], \kappa)$ which can be exercised into a payer swap with strike rate κ along $[T_p, T_q]$ at time $T \leq T_p$ can be approximated as

$$\begin{aligned} c_{p,q}(t) &\simeq B_{p,q}(t)\left[S_{p,q}(t)N(d_+) - \kappa N(d_-)\right], \quad \text{where} \quad (6.36) \\ d_\pm &= \Sigma_{p,q}(t,T)^{-1} log(S_{p,q}(t)/\kappa) \pm \frac{1}{2}\Sigma_{p,q}(t,T). \end{aligned}$$

and $\Sigma^2_{p,q}(t,T)$ is an $\mathcal{F}_t$-measurable estimate of the quadratic variation of the swaprate logarithm $Y_{p,q} = log(S_{p,q})$ on $[t,T]$:

$$\Sigma^2_{p,q}(t,T) \simeq \langle Y_{p,q} \rangle_t^T, \quad Y_{p,q} = log(S_{p,q}).$$

This is the forecast for the cumulative volatility of returns on the swap rate $S_{p,q}$ on the interval $[t,T]$ to option expiry. Observing that

$$\sum_{j=p}^{q-1} \delta_j H_{j+1} = \alpha_q + \sum_{j=p+1}^{n-1} \alpha_j U_j,$$

where

$$\alpha_j = \begin{cases} T_j - T_p & : \quad p < j \leq q \\ T_q - T_p & : \quad q < j \end{cases}$$

and using (6.28) we obtain

$$\begin{aligned} Y_{p,q}(t) &= log\left(S_{p,q}(t)\right) \\ &= log(U_p + \ldots + U_{q-1}) - log\left(\alpha_q + \sum_{j=p+1}^{n-1} \alpha_j U_j\right) \\ &:= f(U_p, \ldots, U_{n-1}). \end{aligned}$$

Let $C = (C_{jk})_{p \leq j,k < q}$ be the matrix with coordinates

$$C_{jk} = \int_t^T \sigma_j(s)\sigma_k(s)\rho_{jk} ds$$

and factor C as $C = RR'$. As in the preceeding section it follows that

$$\langle Y_{p,q} \rangle_t^T \simeq (Cx, x) = \|R'x\|^2,$$

where x is the vector $x = (x_p, \ldots, x_{n-1})$ with components

$$x_j = U_j(t)\frac{\partial f}{\partial U_j}(U(t)) = \begin{cases} A_j(t) & : \quad j = p \\ A_j(t) - B_j(t) & : \quad p < j < q \\ -B_j(t) & : \quad j \geq q \end{cases},$$

with

$$A_j(t) = \frac{U_j(t)}{H_p(t) - H_q(t)} \quad \text{and} \quad B_j(t) = \frac{\alpha_j U_j(t)}{\sum_{j=p}^{q-1} \delta_j H_{j+1}(t)}.$$

resulting in the approximate swaption price (6.36) with

$$\Sigma_{p,q}(t,T) = \sqrt{(Cx,x)} = \|R'x\|.$$

6.8.4 Bond options

A general bond B is simply a linear combination

$$B(s) = \sum_{j=p}^{q-1} c_j B_j(s)$$

of zero coupon bonds. In fact every bond portfolio long or short or both has this form. We will now study the European call exercisable at time $T_t \leq T_p$ with strike K on this bond. Note that here t is an integer (discrete time) and that call exercise is assumed to take place at a Libor reset point. The call payoff at time T_t has the form

$$h = \left(\sum_{j=p}^{q-1} c_j B_j(T_t) - K \right)^+ .$$

Multiplying with $1/B_n(T_t) = B_t(T_t)/B_n(T_t) = H_t(T_t)$ this payoff accrued forward to time T_n assumes the form

$$h/B_n(T_t) = \left(\sum_{j=p}^{q-1} c_j H_j(T_t) - K H_t(T_t) \right)^+ .$$

Here the assets

$$S_1 = F := \sum_{j=p}^{q-1} c_j H_j \text{ (forward bond price) and } S_2 = H_t$$

are local martingales under P_n, in fact square integrable martingales. Consequently C.4 applies with $P = P_n$ and $T = T_t$. An analytic approximation for the bond option price $c(s)$ is given by

$$\begin{aligned} c(s) &= B_n(s) \left[F(s) N(d_+) - K H_t(s) N(d_-) \right] && (6.37) \\ &= B(s) N(d_+) - K B_t(s) N(d_-), \quad \text{where} && (6.38) \\ d_\pm &= \Sigma(s, T_t)^{-1} log \left(F(s)/K H_t(s) \right) \pm \frac{1}{2} \Sigma(s, T_t) \end{aligned}$$

and $\Sigma^2(s, T_t)$ is an $\mathcal{F}_t$-measurable estimate of the quadratic variation of the logarithm $Y = log(Q) = log(F) - log(H_t)$ on the interval $[s, T_t]$. This is the forecast for the cumulative volatility of returns on the quotient Q on the interval $[t, T_t]$ to option expiry. Note that $Q = F/H_t = B/B_t$ is just the forward price of the bond at the time T_t of option expiry. From (6.23) we have

$$F = q - p + b_q U_q + \ldots + b_{n-1} U_{n-1}, \tag{6.39}$$

where

$$b_j = c_p + \ldots + c_{j \wedge (q-1)}$$

and $a \wedge b = \min\{a, b\}$. It follows that

$$\begin{aligned} Y &= f(U_t, \ldots, U_{n-1}) \\ &= log\left(q - p + b_p U_p + \ldots + b_{n-1} U_{n-1}\right) - log\left(1 + U_t + \ldots + U_{n-1}\right). \end{aligned}$$

Set

$$C_{jk} = \int_s^{T_t} \sigma_j(u)\sigma_k(u)\rho_{jk}(u)du.$$

and factor the matrix $C = (C_{jk})_{t \le j,k < n}$ as $C = RR'$. As in the preceeding section it follows that a suitable estimate is given by

$$\Sigma(s, T_t) = \sqrt{(Cx, x)} = \|R'x\|,$$

where x is the vector $x = (x_t, \ldots, x_{n-1})$ with components

$$x_j = U_j(s)\frac{\partial f}{\partial U_j}(U(s)) = \begin{cases} -U_j(s)/H_t(s) & : \quad t \le j < p \\ b_j U_j(s)/F(s) - U_j(s)/H_t(s) & : \quad p \le j < n \end{cases}.$$

Note that this formula prices the bond as if the bond price at expiration were lognormal. Moreover the variance increases with the time to option expiration. These assumptions have several weaknesses: the lognormal distribution has positive mass on all of $(0, +\infty)$ but the bond price is bounded above. For short dated bonds the variance of the bond price will decrease as time approaches the maturity of the bond.

An option on a zero coupon bond which expires shortly before bond maturity is the worst case for these assumptions. The bond price will have very little variance and certainly not exceed one. The formula was tested in a few such cases for at the money options (strike price is the cash price of the bond) and worked very well. This seems to be due to cancellation effects in the tails of the lognormal distribution which places too much mass far below the strike price but makes up for it by placing mass at bond prices bigger than one.

Obviously this cancellation will not work if the strike price is moved up or down. For example the formula will report a positive price for a call on the zero coupon bond with strike price bigger than one. We are interested in analytic bond option formulas since they allow us to calibrate a Libor market model for hedging a bond portfolio. If you do that you need to be aware of the limitations of this formula. In other words decide which options you will use for calibration and then test how the formula performs on these options in a Libor market model with a factor loading that roughly reflects market conditions.

6.9 Implementation

The reader can find an implementations in the Java package Libor.LiborProcess as well as the C++ files LiborMarketModel.h, PredictorCorrectorLMM.h, FastPredictorCorrectorLMM.h, DriftlessLMM.h.

The Libor dynamics steps directly from one point T_j in the tenor structure to the next point T_{j+1}. No intermediate Libors are computed.

The abstract class FactorLoading encapsulates the volatility and correlation structure and lists abstract methods to compute the covariance integrals $C(t,T)$ and their Cholesky roots $R(t,T)$ which are needed to drive the time step $t \to T$ of the Libor path.

Every concrete factor loading depends on the choice of the concrete volatility and correlation structure. Three different structures are implemented. The Coffee-Shoenmakers factor loading is discussed in detail in the section on calibration below.

6.10 Zero coupon bonds

The Libors $L_j(t)$ do not determine even the zero coupon bonds $B_j(t) = B(t,T_j)$ which mature at points on the tenor structure completely. Only the ratios B_j/B_k are determined. Combining this with $B_j(T_j) = 1$ we see that the zero coupon bonds $B_j(t) = B(t,T_j)$ are determined at all times $t = T_k < T_j$.

A rough approximation for the zero coupon bond $B(t,T)$ can be computed from the Libors in our model if we follow the convention that Libor is earned fractionally on subintervals of any accrual interval. This is in accordance with the fact that Libor is a simple rate of interest.

Set $X_j(t) = \delta_j L_j(t)$ as above and assume that $i \geq j$ and the times t, T satisfy

$$T_{j-1} \leq t \leq T_j < T_i \leq T < T_{i+1}.$$

The factor $1/B(t,T)$ compounds from time t to time T. At time t Libor for the interval $[T_{j-1},T_j]$ is already set and so compounding from time t to T_j is accomplished by the factor $1 + \delta_{j-1}^{-1}(T_j - t)X_{j-1}(T_{j-1})$.

Likewise compounding (at time t) from time T_k to time T_{k+1} is accomplished by the factor $1 + X_k(t)$. Finally the factor $1 + \delta_i^{-1}(T - T_i)X_i(t)$ compounds from time T_i to time T. Putting all this together yields

$$1/B(t,T) = [1+\delta_{j-1}^{-1}(T_j-t)X_{j-1}(T_{j-1})] \times A \times [1+\delta_i^{-1}(T-T_i)X_i(t)], \quad (6.40)$$

where $A = [1 + X_j(t)][1 + X_{j+1}(t)] \dots [1 + X_{i-1}(t)]$. In case $T_{j-1} \le t \le T \le T_j$, that is, both times t, T are in the same accrual interval we set

$$1/B(t,T) = 1 + \delta_{j-1}^{-1}(T-t)X_{j-1}(T_{j-1}).$$

These formulas preserve the relation $B_j(t)/B_{j+1}(t) = 1 + X_j(t)$ and the zero coupon bond condition $B(T,T) = 1$.

In our implementation [Mey02a] Libor $X_k(t)$ is only computed at times $t = T_j \le T_k$. Consequently t is replaced with the nearest T_j in the computation of $B(t,T)$. This is rather crude and causes jumps in the computed zero coupon bond path prices at times t when the nearest T_j jumps, see Examples.Libor.BondPaths.java.

6.11 Model Calibration

In this section we will study in detail how a Libor market model is calibrated to caplet and swaption prices. We will consider both the calibration of a predictor-corrector and a driftless model. To get a better grip on the problem we parametrize the factor loading in terms of a deterministic *volatility surface* $\sigma(t,T)$ and constant correlation matrix $\rho = (\rho_{ij})$.

6.11.1 Volatility surface

More precisely we assume that the volatility functions $\sigma_j(t)$ have the form

$$\sigma_i(t) = c_i\sigma(t,T_i)$$

where $\sigma(t,T) > 0$ is a deterministic function (the volatility surface) and the $c_j > 0$ are scaling factors. Set

$$B_{ij}(t,T) = \int_t^T \sigma(s,T_i)\sigma(s,T_j)dt. \tag{6.41}$$

With this the covariation matrix $C = C(t,T)$ given by

$$C_{ij}(t,T) = \int_t^T \sigma_i(s)\sigma_j(s)\rho_{ij}dt$$

assumes the form

$$C_{ij} = c_ic_j\rho_{ij}B_{ij}. \tag{6.42}$$

Any abstraction of the concept of a volatility surface will take the quantity $\sigma(t,T)$ and integrals $\int_t^T \sigma(u,T_i)\sigma(u,T_j)dt$ as the fundamental features. See the class VolSurface in VolatilityAndCorrelation.h. Three volatility surfaces are implemented:

Jaeckel-Rebonato volatilities

This surface depends on four parameters a, b, c, d and is given by

$$\sigma(t,T) = d + (a + b(T-t))e^{-c(T-t)}.$$

M-volatilities

These volatilities depend on 2 parameters a, d and are defined as

$$\sigma(t,T) = g(1-t/T), \quad \text{where} \quad g(t) = 1 + ate^{-t/d}.$$

The constant volatility surface $\sigma(t,T) = 1$ is a special case but it is useful to implement it separetely.

6.11.2 Correlations

The correlation matrix ρ must of course be symmetric and positive semidefinite. A general parametrization of all such matrices is difficult and requires a high number of parameters. See [JR]. Therefore we take another approach. We try to isolate some properties which the correlations of the log-Libors $Y_j = log(X_j)$ (in the case of the predictor-corrector model) or the $Y_j = log(U_j)$ (in the case of the driftless model) should have and then find a low dimensional parametrization conforming to these requirements. In general we confine ourselves to correlations of the form

$$\rho_{ij} = b_i/b_j, \quad 1 \leq i \leq j < n.$$

where b_j is an increasing sequence [CS00]. The sequence b is called the *correlation base*. We have already seen in 6.4 that the matrix ρ is then positive semidefinite. Libor L_0 is set at time $T_0 = 0$ and hence is a constant. With this in mind only the correlations ρ_{ij} for $0 < i, j < n$ will actually be used in the simulation.

Coffee-Shoenmakers correlations

Coffee and Shoenmakers [CS00] postulate some desirable properties for log-Libor correlations and derive a suitable parametrization of the correlation base b. We follow these ideas with slight modifications. We have already seen in 6.4 that the matrix ρ is positive semidefinite. The correlation base $b = (b_1, \ldots, b_{n-1})$ should ensure the following properties of the log-Libor correlations:

- $\rho_{ij} \leq 1$, equivalently $b_i \leq b_j$, $i \leq j$.
- $i \mapsto \rho_{ii+1} = b_i/b_{i+1}$ is nondecreasing.

Clearly we may assume that $b_1 = 1$. Setting $\zeta_i = log(b_i)$ these conditions translate to $\zeta_1 = 0$ and

$$\begin{aligned} \zeta_i &\leq \zeta_{i+1}, \quad \text{and} && (6.43) \\ \zeta_i &\geq \tfrac{1}{2}(\zeta_{i-1} + \zeta_{i+1}), \quad 1 < i < n-1, && (6.44) \end{aligned}$$

and can be satisified by setting

$$\zeta_i = -f\left(\frac{i-1}{n-2}\right) \quad \text{ie.} \quad b_i = exp\left[-f\left(\frac{i-1}{n-2}\right)\right], \quad 1 \leq i \leq n-1, \quad (6.45)$$

for some decreasing and convex function $f = f(x)$, $x \in [0, 1]$. The convexity of f is satisfied if $f''(x) \geq 0$ on $[0, 1]$ and the decreasing property then follows

from $f'(1) < 0$. A suitably simple function $f(x)$ is obtained by requiring that

$$f''(x) = \alpha + (\beta - \alpha)x.$$

The parameters α, β are then the derivatives $f''(0)$, $f''(1)$ which control the concavity of the sequence ζ_i. As i increases the values of the correlations $\rho_{i,i+1}$, $\rho_{i-1,i}$, $\rho_{i+1,i+2}$ move closer to each other. It is thus desirable that the sequence

$$i \mapsto log\rho_{i,i+1} - \frac{1}{2}(log\rho_{i-1,i} + log\rho_{i+1,i+2}) \tag{6.46}$$

be decreasing. In terms of the funtion f we would thus like to see $f''(x)$ decreasing and so we want to have

$$\alpha \geq \beta \geq 0.$$

If the number n of Libors is large, then the concavity of (6.46) should become small, in other words we want $\beta = f''(1)$ to be small. Recall that $\zeta_1 = 0$ and so $f(0) = 0$. Integration then yields

$$\begin{aligned} f'(x) &= c + \alpha x + (\beta - \alpha)x^2/2 \quad \text{and} \\ f(x) &= cx + ax^2 + bx^3, \quad \text{where} \quad a = \alpha/2 \text{ and } b = (\beta - \alpha)/6. \end{aligned}$$

Let us replace the parameter c with the correlation

$$\rho_\infty := \rho_{1,n-1} = b_1/b_{n-1} = 1/b_{n-1}.$$

Observing that $log(\rho_\infty) = -\zeta_{n-1} = f(1) = a + b + c$ we have

$$c = log(\rho_\infty) - (a + b).$$

Recall that we must satisfy the condition $f'(1) = c+2a+3b < 0$, equivalently $a+2b < -log(\rho_\infty)$ ie. $\alpha/6+\beta/3 < -log(\rho_\infty)$. Thus we obtain the following *CS-Correlations.*

$$\rho_{ij} = b_i/b_j, \quad 1 \leq i < j < n, \quad \text{with} \tag{6.47}$$

$$b_i = exp\left[-f\left(\frac{i-1}{n-2}\right)\right], \quad \text{where} \quad f(x) = cx + ax^2 + bx^3, \tag{6.48}$$

$$a = \frac{\alpha}{2},\ b = \frac{\beta - \alpha}{6} \text{ and } c = log(\rho_\infty) - (a + b).$$

where the parameters $\alpha, \beta, \rho_\infty$ must satisfy

$$\alpha \geq \beta \geq 0 \text{ and } \frac{\alpha}{6} + \frac{\beta}{3} < -log(\rho_\infty). \tag{6.49}$$

The parameters α, β control the concavity of the sequence $\zeta_i = log(b_i)$ and β should be chosen small, the smaller the larger the number n of Libors.

Jaeckel-Rebonato correlations

Setting $b_j = exp(\beta T_j)$ we obtain a special case of the correlations suggested by Jaeckel and Rebonato, see [Jae02], 12.4, page 165. This results in correlations

$$\rho_{ij} = exp(\beta(T_j - T_i)), \quad i \leq j.$$

Correlations and Volatility surfaces are implemented in VolatilityAndCorrelations.h,cc and can be combined arbitrarily to produce a number of factor loadings for the Libor market model.

6.11.3 Calibration

Note that we need at most 7 parameters $a, b, c, d, \alpha, \beta, \rho_\infty$ for both the volatility surface and the correlation matrix. Combining these with the scaling factors c_j we obtain an $n+6$ dimensional parameter vector x

$$\begin{aligned} &x_j = c_j, \quad 1 \leq j < n, \\ &x_n = a,\ x_{n+1} = b,\ x_{n+2} = c,\ x_{n+3} = d, \\ &x_{n+4} = \alpha,\ x_{n+5} = \beta,\ x_{n+6} = \rho_\infty. \end{aligned}$$

Depending on the volatility surface and correlation matrix in actual use some of these parameters may have no influence on the factor loading. In order to calibrate the parameter vector to the market prices of caplets and swaptions we need an objective function which minimizes the calibration error as a function of the parameter vector x. We take the usual function

$$error(x) = \sum (market\_price \ - \ calibrated\_price(x))^2,$$

where the sum extends over all instruments used in the calibration. An alternative would be to use implied and calibrated volatilities instead. We also must decide which instruments we use as the basis for calibration. In our implementation we use all at the money caplets

$$Caplet(i) = Cplt([T_i, T_{i+1}], \kappa_i), \quad \text{with} \quad \kappa_i = L_i(0)$$

and all coterminal at the money swaptions

$$Swaption(i) = Swpn([T_i, T_n], \kappa_i), \quad \text{with} \quad \kappa_i = S_{pq}(0),\ p = i, q = n.$$

Note that these swaptions all exercise into a swap terminating at the horizon T_n. This decision is somewhat arbitrary designed to keep the implementation particularly simple.

With this we have a constrained optimization problem: the parameters must be positive and, in the case of Coffee-Shoenmakers correlations we must have

$$\alpha/6 + \beta/3 + log(\rho_\infty) > 0.$$

In our own implementation we use a crude quasi random search: starting from a reasonable initial guess we center a search rectangle at the current optimal parameter vector and search this rectangle with a Sobol sequence. After a certain number of points have been evaluated we move to search rectangle to the next optimum, contract it and cut the number of search points in half. The search stops when the number of search points reaches zero.

To make his work it is important to find initial values for the scaling factors c_j which approximate the caplet prices well. The details depend on the type of Libor market model which is being calibrated, that is wether we are calibrating a predictor-corrector model or a driftless model. In each case we convert the observed caplet prices to implied aggregate volatilties $\overline{\Sigma}_i(0, T_i)$ to expiry by solving the caplet price formula (6.6) or (6.8.2) for the aggregate volatility $\Sigma_i(0, T_i)$. Here the term aggregate volatility denotes cumulative volatility to expiry as opposed to annualized volatility $\overline{\sigma}_i$. the two are related as

$$\overline{\Sigma}_i(0, T_i) = \overline{\sigma}_i \sqrt{T_i}.$$

Calibration of a predictor-corrector model

In the predictor-corrector Libor market model the theoretical aggregate volatility $\Sigma_i(0, T_i)$ to be used in the caplet pricing formula is given by

$$\Sigma_i^2(0, T_i) = C_{ii} = c_i^2 B_{ii}, \quad \text{where} \quad B_{ii} = \int_0^{T_i} \sigma^2(t, T_i) dt$$

depends only on the volatility surface. Note how the correlations do not appear. If we equate this with the implied aggregate volatility

$$c_i^2 B_{ii} = \overline{\Sigma}_i^2(0, T_i)$$

we obtain the scaling factor c_i which exactly reproduces the market observed caplet price. This is a good first guess that applies equally well to all correlations.

Calibration of a driftless model

In the case of a driftless Libor market model the situation is complicated by the fact that the correlations appear in the caplet price formula and that these correlations are themselves varied in the course of the calibration. One can take two approaches: recompute the scaling factors c_j for each new set of correlations to match caplet prices exactly or start with a reasonable guess for the correlations and compute a corresponding set of scaling factors as an initial guess. Then let the search routine do the rest.

Matching caplet prices exactly is obviously quite restrictive and unlikeley to produce optimal results. However it does reduce the dimension of the optimization problem to the number of parameters for the volatility surface and correlations and this is the approach followed in our implementation (class LiborCalibrator.h).

Let us now assume that we have some correlations and we want to compute the coresponding saling factors c_j which match the market observed caplet prices exactly. Set

$$B_{ij} = \int_0^{T_i} \sigma(t, T_i)\sigma(t, T_j)dt.$$

With this the covariation matrix C in the caplet volatility estimate (6.33) in a driftless Libor market model (at time $t = 0$) assumes the form

$$C_{ij} = c_i c_j \rho_{ij} B_{ij}.$$

We will see that the scaling factors c_i can be computed by backward recursion starting from $i = n-1$, that is, c_i is computed from $c_{i+1}, \ldots, c_{n-1}$. Fix an index $i < n$ and set $G_j = G_j(0)$ and $x_j = c_j G_j$, $i \le j < n$, where the G_j are the quantities from (6.35) in section (6.8.2). Note that the G_j depend on i also and $G_i = 1$. Consequently $c_i = x_i$. With this we can rewrite (6.33) (at time $t = 0$) as

$$\Sigma_i^2(0, T_i) = E\langle Y_i\rangle_0^{T_i} \simeq \sum\nolimits_{j,k=i}^{n-1} x_j x_k \rho_{jk} B_{jk} \tag{6.50}$$

Solve formula (6.8.2) for the quantity $\Sigma_i(0, T_i)$ to obtain the market implied aggregate volatility to expiry $\overline{\Sigma}_i(0, T_i)$. We now have to determine the c_j such that

$$\overline{\Sigma}_i^2(0, T_i) = \sum\nolimits_{j,k=i}^{n-1} x_j x_k \rho_{jk} B_{jk} \quad \text{with} \quad x_j = c_j G_j. \tag{6.51}$$

The quantity on the right depends only on the factors $c_i, c_{i+1}, \ldots, c_{n-1}$. Consequently c_i can be computed from $c_{i+1}, \ldots, c_{n-1}$. For $i = n-1$ equation

(6.51) assumes the form

$$x_{n-1}^2 B_{n-1,n-1} = \overline{\Sigma}_{n-1}^2(0, T_{n-1})$$

while for $i < n-1$ we have

$$\begin{aligned} &ux_i^2 + 2vx_i + w = \overline{\Sigma}_i^2(0, T_i), \quad \text{where} \\ &u = B_{ii}, \quad v = \sum\nolimits_{k=i+1}^{n-1} x_k \rho_{ik} B_{ik} \quad \text{and} \quad w = \sum\nolimits_{j,k=i+1}^{n-1} x_j x_k \rho_{jk} B_{jk} \end{aligned}$$

The solution (which must be positive) is

$$x_i = u^{-1}\left[-v + \sqrt{v^2 - u\left(w - \overline{\Sigma}_i^2(0, T_i)\right)}\right].$$

From this c_i can be obtained as $c_i = x_i$. The quantities v and w only depend on $c_{i+1}, \ldots, c_{n-1}$. Note that the correlations ρ_{ij} make their appearance. These correlations will also vary during calibration. There are two possible approaches to this problem: recompute the scaling factors for each new set of correlations or start with a reasonable guess for the correlations, use these to get initial values for the c_j and then let the search routine do the rest.

6.12 Monte Carlo in the Libor market model

We have derived the dynamics of the Libor process only in the forward martingale measure Q at the terminal date $t = T_n$. This probability Q is the numeraire measure associated with the zero coupon bond B_n maturing at $t = T_n$. In this probability the Libor dynamics driven by a Brownian motion W asumes the form

$$dX_j(t) = X_j(t) \left[- \sum_{k=j+1}^{n-1} \frac{X_k(t)}{1 + X_k(t)} \nu_j(t) \cdot \nu_k(t) + \nu_j(t) \cdot dW(t) \right]. \quad (6.52)$$

Simulations using this dynamics can therefore be used only to compute expectations $E^Q(\cdot)$ with respect to the terminal measure Q. If a Libor derivative has a single payoff h at time $t = T_n$ then its cash price $c(t)$ can be computed from an E^Q-expectation via the formula

$$c(t) = B_n(t) E_t^Q(h). \quad (6.53)$$

If h induces cash flows at several points in time we transport all cash flows forward to the horizon and aggregate them to a single payoff h at time $t = T_n$. Likewise there is a numeraire measure P_j associated with the zero coupon bond B_j maturing at time $t = T_j$. It is called the *forward martingale measure at* $t = T_j$ and defined by the density

$$g_j = \frac{dP_j}{dQ} = c(B_j/B_n)(T_n) = cB_j(T_n),$$

where c is the normalizing factor $B_n(0)/B_j(0)$. This means that $E^{P_j}(h) = E^Q(g_j h)$ for all measurable, nonnegative functions h (and also others of course). Recall our convention that all zero coupon bonds live to time T_n. One can then show that the quotient B_k/B_j is a P_j-local martingale, for each $k \leq n$. In the new probability P_j the martingale pricing formula (6.53) assumes the form

$$c(t) = B_j(t) E_t^Q(h) \quad (6.54)$$

for a cash flow h which occurs at time $t = T_j$. See Appendix C.1 and note $B_j(T_j) = 1$. With this terminology the terminal forward martingale measure Q is the probability P_n.

For example the ith caplet $Cpl([T_i, T_{i+1}], k)$ with strike rate k has payoff $\delta_i(L_i(T_i) - k)^+$ at time $t = T_{i+1}$ and so its price $Cpl_i(0)$ at time zero is most conveniently computed using the forward martingale measure P_{i+1} at time $T = T_{i+1}$ as

$$Cpl_i(0) = \delta_i B_{i+1}(0) E^{P_{i+1}}[(L_i(T_i) - k)^+] \quad (6.55)$$

but this formula cannot be used in Monte Carlo simulation unless we switch to the P_{i+1}-dynamics of L_i. If we want to use the P_n-dynamics (6.52) instead we need to push the payoff forward from time $t = T_{i+1}$ to the equivalent payoff $(L_i(T_i) - k)^+ / B_n(T_{i+1})$ at time $t = T_n$ and rewrite (6.55) as

$$Cpl_i(0) = \delta_i B_n(0) E^{P_n}[(L_i(T_i) - k)^+ / B_n(T_{i+1})]. \tag{6.56}$$

The payoff $(L_i(T_i) - k)^+ / B_n(T_{i+1})$ at time T_n can then be simulated under the P_n-dynamics (6.52) and the expectation in (6.56) approximated as an ordinary arithmetic average over simulated payoffs. Since in general there are several cash flows occuring at different points in time there is no single probability P_j which is preferred and consequently we standardize on the terminal forward martingale measure $Q = P_n$.

Pushing the caplet payoff forward to time T_n involves all Libors L_j, $j = i + 1, \ldots, n - 1$ and comparison of the Monte Carlo simulated caplet prices under the P_n-dynamics to Black caplet prices (involving only $B_{i+1}(0)$) provides a good test of the correctness of the Libor model implementation.

6.13 Control variates

The high dimensionality of the Libor process makes the simulation of paths very slow. Control variates allow us to reduce the number of paths which have to be generated to reach a desired level of precision. A control variate Y for a random variable X is used together with its expctation $E(Y)$. Consequently we must be able to compute the expectation $E(Y)$ significantly faster than the expectation $E(X)$. The best possible case occurs if $E(Y)$ can be computed by analytic formula. However the control variate Y must also be highly correlated (or anticorrelated) with the random variable X. Recall that $P_n = Q$ denotes the forward martingale measure at time $t = T_n$.

6.13.1 Control variates for general Libor derivatives

Assume that a Libor derivative has payoff h at time $T = T_n$

$$h = f(V) \quad \text{with} \quad V = (X_0(T_0), X_1(T_1), \ldots, X_{n-1}(T_{n-1})) \in R^n \tag{6.57}$$

(other payoffs are handled in a similar manner). We obtain a highly correlated control variate Y if we compute Y as

$$Y = f(U) \quad \text{with} \quad U = (\tilde{X}_0(T_0), \tilde{X}_1(T_1), \ldots, \tilde{X}_{n-1}(T_{n-1})) \in R^n$$

where the process $\tilde{X}(t)$ approximates the process $X(t)$ path by path and the distribution of the vector U is known. In this case the mean $E^{P_n}(Y)$ can be computed by sampling directly from the distribution of U which is much faster than Libor path simulation. Once this mean has been obtained we can use Y as a control variate for h and compute the expectation $E^{P_n}(h)$ by simulating a smaller number of Libor paths.

This means of course that both paths $t \to X(t)$ and $t \to \tilde{X}(t)$ driven by the same underlying Brownian path are computed side by side. In our implementation this simply means that both paths are driven by the same sequence of standard normal increments.

As a suitable process $\tilde{X}$ we could use the Gaussian approximation for the Libor process derived in Section 2 by linearizing the Libor drift. In fact even the less sophisticated $X0$-approximation works well enough. This replaces the drift

$$\mu_i(t) = -\sum_{k=j+1}^{n-1} \frac{X_k(t)}{1+X_k(t)} \sigma_i(t)\sigma_j(t)\rho_{ij}$$

with the deterministic drift term

$$\tilde{\mu}_i(t) = -\sum_{k=j+1}^{n-1} \frac{X_k(0)}{1+X_k(0)} \sigma_i(t)\sigma_j(t)\rho_{ij}$$

and thus approximate the Libor process $X(t)$ with the process $\tilde{X}(t)$ defined by

$$d\tilde{X}_i(t) = \tilde{X}_i(t)[\tilde{\mu}_(i)(t)dt + \sigma_i(t)u_i \cdot dW(t)], \quad \tilde{X}(0) = X(0).$$

Setting $Z_i(t) = log(\tilde{X}_i(t))$ we have

$$Y = f(U) \quad \text{with} \quad U = (exp(Z_0(T_0)), \ldots, exp(Z_{n-1}(T_{n-1})) \in R^n,$$

where the process $Z(t)$ satisfies

$$dZ_i(t) = \lambda_i(t)dt + \sigma_i(t)u_i \cdot dW(t) \quad \text{with} \quad \lambda_i(t) = \tilde{\mu}_i(t) - \tfrac{1}{2}\sigma_i^2(t).$$

Since the drift $m_i(t)$ is deterministic the vector

$$log(U) := (Z_0(T_0), \ldots, Z_{n-1}(T_{n-1}))$$

is multinormal with mean vector $m = (m_i)$ and covariance matrix $C = (C_{ij})$ given by

$$m_i = \int_0^{T_i} \lambda_i(t)dt \quad \text{and} \quad C_{ij} = \int_0^{T_i} \sigma_i(t)\sigma_j(t)\rho_{ij}dt, \ i \le j.$$

Consequently sampling from the distribution of $log(U)$ is very fast and such samples lead directly to samples from the distribution of Y. The mean $E^{P_n}(Y)$ can therefore be computed to high precision with a small fraction of the effort needed to compute the mean $E(X)$.

6.13.2 Control variates for special Libor derivatives

We now turn to control variates Y for special Libor derivatives with the property that the mean $E^{P_n}(Y)$ can be computed analytically. These derivatives are all implemented in the package [Mey02a] and each has a method which computes the correlation of the derivative payoff with the control variate. The observed correlations are quite good but do obviously depend on the choice of the various parameters.

Control variates for caplets

Libor $L_i(T_i)$ is highly correlated with the caplet payoff $(L_i(T_i)-k)^+$ at time T_{i+1}. Recall however that we are simulating under the terminal martingale measure P_n and hence must work with the forward transported payoff

$$X = (L_i(T_i) - k)^+ / B_n(T_{i+1}) = (L_i(T_i) - k)^+ B_{i+1}(T_{i+1}) / B_n(T_{i+1}).$$

This would suggest the control variate $Y_0 = L_i(T_i)B_{i+1}(T_{i+1})/B_n(T_{i+1})$. Unfortunately the expectation $E^{P_n}(Y)$ is not easily computed. We do know however that

$$L_i(t)B_{i+1}(t)/B_n(t) = \delta_i^{-1}(B_i - B_{i+1})/B_n$$

is a P_n-martingale (traded assets divided by the numeraire). Consequently the closely related random variable $Y = L_i(T_i)B_{i+1}(T_i)/B_n(T_i)$ satisfies

$$E^{P_n}(Y) = L_i(0)B_{i+1}(0)/B_n(0)$$

and is thus useful as a control variate for the forward transported caplet payoff.

Control variates for swaptions

The forward swaption $Swpn(T, [T_p, T_q], \kappa)$-exercisable at time $T \leq T_p$ with strike rate κ exerices into a payer swap on $[T_p, T_q]$ with strike rate κ. If $S_{p,q}(t) = (B_p(t) - B_q(t))/B_{p,q}(t)$ denotes the swap rate for this swap the

swaption payoff at time T is the quantity $B_{p,q}(T)(S_{p,q}(T)-\kappa)^+$ and this payoff is obviously highly correlated with

$$Y_0 = B_{p,q}(T)S_{p,q}(T) = B_p(T) - B_q(T)$$

and so the forward tansported payoff $Y_0/B_n(T)$ highly correlated with

$$Y = (B_p(T) - B_q(T))/B_n(T).$$

Since $(B_p - B_q)/B_n$ is a P_n-martingale (traded assets divided by the numeraire) the mean of Y is given by

$$E^{P_n}(Y) = (B_p(0) - B_q(0))/B_n(0).$$

The reverse floater

Let $K_1 > K_2$. The $[T_p, T_q]$-reverse floater RF receives Libor $L_j(T_j)$ while paying $\max\{ K_1 - L_j(T_j), K_2 \}$ resulting in the cash flow

$$C(j) = \delta_j[L_j(T_j) - \max\{ K_1 - L_j(T_j), K_2 \}] \tag{6.58}$$

at times T_{j+1}, $j = p, \ldots, q-1$. Set $\kappa = K_1 - K_2$, $x = K_1 - K_2 - L_j(T_j)$ and recall that $(-x)^+ = x^-$ and $x^+ - x^- = x$ and so $(-x)^+ = x^- = x^+ - x$ to rewrite (6.58) as

$$\begin{aligned} C(j)/\delta_j &= L_j(T_j) - K_2 - \max\{ \kappa - L_j(T_j), 0 \} \\ &= L_j(T_j) - K_2 - (\kappa - L_j(T_j))^+ \\ &= L_j(T_j) - K_2 - [(L_j(T_j) - \kappa)^+ - (L_j(T_j) - \kappa] \\ &= 2L_j(T_j) - K_1 - (L_j(T_j) - \kappa)^+. \end{aligned}$$

and so

$$\begin{aligned} C(j) &= \delta_j(2L_j(T_j) - K_1) - \delta_j(L_j(T_j) - \kappa)^+ \\ &= 2X_j(T_j) - \delta_j K_1 - \delta_j(L_j(T_j) - \kappa)^+. \end{aligned} \tag{6.59}$$

Here $\delta_j(L_j(T_j)-\kappa)^+$ is the payoff of the caplet $Cpl([T_j, T_{j+1}], \kappa)$ with strike rate κ and $L_j(t)$ is a P_{j+1}-martingale. Evaluation of $C(j)$ in the P_{j+1}-numeraire measure thus yields the (cash) price $C_t(j)$ at time t as

$$\begin{aligned} C_t(j) &= B_{j+1}(t)E_t^{P_{j+1}}[C(j) \mid \mathcal{F}_t] \\ &= 2X_j(t)B_{j+1}(t) - K_1\delta_j B_{j+1}(t) - Cpl_t([T_j, T_{j+1}], \kappa) \\ &= 2(B_j(t) - B_{j+1}(t)) - K_1\delta_j B_{j+1}(t) - Cpl_t([T_j, T_{j+1}], \kappa). \end{aligned}$$

and summation over $p \leq j < q$ yields the price RF_t of the reverse floater as

$$RF_t = 2(B_p(t) - B_q(t)) - K_1 B_{p,q}(t) - \sum_{j=p}^{q-1} Cpl_t([T_j, T_{j+1}], \kappa)].$$

Obviously the cash flow (6.59) is highly correlated with $X_j(T_j) = \delta_j L_j(T_j)$ and so, as a control variate for (6.59) we take forward transported X-Libor

$$Y_j = X_j(T_j)B_{j+1}(T_j)/B_n(T_j) = (B_j(T_j) - B_{j+1}(T_j))/B_n(T_j).$$

Since $X_j B_{j+1}/B_n = (B_j - B_{j+1})/B_n$ is a P_n-martingale we have $E^{P_n}(Y_j) = X_j(0)B_{j+1}(0)/B_n(0)$. Consequently for the reverse floater RF we use as a control variate the sum

$$Y = \sum_{j=p}^{q-1} X_j(T_j)B_{j+1}(T_j)/B_n(T_j)$$

with mean $E(Y) = \sum_{j=p}^{q-1} X_j(0)B_{j+1}(0)/B_n(0)$. Since $X_j B_{j+1} = B_j - B_{j+1}$ this sum collapses to

$$E(Y) = (B_p(0) - B_q(0))/B_n(0).$$

The callable reverse floater

Let $K_1 > K_2$. The callable $[T_p, T_q]$-reverse floater CRF is the option to enter into the $[T_p, T_q]$-reverse floater RF at time T_p (at no cost). This option is only exercised if the value RF_{T_p} of the reverse floater at time T_p is positive and thus has payoff

$$h = RF_{T_p}^+$$

at time T_p. This quantity can be computed by Monte Carlo simulation. As a control variate we use the same as for the reverse floater except that all Libors are transported forward from time T_p of exercise. Since $X_j B_{j+1} = B_j - B_{j+1}$ this simplifies the control variate Y to

$$\begin{aligned} Y &= \sum_{j=p}^{q-1}(B_j(T_p) - B_{j+1}(T_p))/B_n(T_p) && (6.60) \\ &= (B_p(T_p) - B_q(T_p))/B_n(T_p) = (1 - B_q(T_p))/B_n(T_p). && (6.61) \end{aligned}$$

with mean $(B_p(0) - B_q(0))/B_n(0)$ as above. Note that $(B_p(t) - B_q(t))/B_n(t)$ is a P_n-martingale.

The trigger swap

The trigger swap $trswp([T_p, T_q], \kappa, K)$ initiates a payer swap $swp(([T_j, T_q], \kappa)$ at the first time $j \in \{p, \ldots, q-1\}$ such that $L_j(T_j) > K$. Here K is called the trigger level and κ the strike level of the trigger swap. This is a nasty derivative for which the Monte Carlo mean converges very slowly. However the caps $cap([T_p, T_q], k)$, where $k = \kappa$ or $k = K$ provide closely correlated control variates with analytic mean.

6.14 Bermudan swaptions

The owner of the Bermudan swaption $\mathsf{bswpn}(\kappa, [T_p, T_n])$ is entitled to enter into a swap which swaps Libor against a fixed coupon with strike rate κ at any date T_j, where $p \leq j < n$. In the case of a payer swaption the fixed coupon is paid and Libor received while it is the other way around in the case of a receiver swaption.

Recall that paying off a debt now is equivalent to paying off the debt later while paying Libor in the interim. Thus such a contract occurs naturally as follows: suppose that a bond has coupon dates T_j and the issuer has the right to cancel the bond by prepayment of the principal at any date T_j, where $p \leq j < n$. Such prepayment is equivalent to swapping Libor against the fixed coupon rate (paying Libor and receiving fixed), that is, the exercise of a Bermudan receiver swaption. Consequently the prepayment option is a Bermudan receiver swaption.

The valuation of Bermudan options has already been dealt with in Section 4.6 using the general convex exercise strategy based on an approximation of the continuation value $CV(t)$ involving the quantities

$$Q(t) = \max_{s>t} E_t(h_s),$$

where h_s is the discounted payoff from option exercise at time t. In the case of Bermudan swaptions it is easier to work with forward payoffs h_t instead since the dynamics of the underlying Libor process is modeled in the forward martingale measure at the terminal date T_n. The convex exercise strategy then exercises at time t if

$$h_t > \beta(t) \left[Q(t)/\alpha(t)\right]^{\beta(t)}. \tag{6.62}$$

The parameters $\alpha(t)$, $\beta(t)$ are computed by recursive optimization starting with $t = T_{n-2}$ and moving backward to $t = T_p$. Exercise at time $t = T_{n-1}$ depends only on the difference $L_t(T_t) - \kappa$. A payer swaption exercises if this

difference is positive. A receiver swaption exercises if it is negative. The strategy is implemented in the class LiborDerivatives.CvxTrigger.

The conditional expectation $E_t(h_s)$ is now the forward price of the European swaption swpn$(\kappa, [T_s, T_n])$ exercisable at time T_s into a swap terminating at time T_n. For this we have a very good analytic approximation but the computation is expensive. Consequently the computation of the quantities $Q(t)$ is expensive.

This prompts us to search for other exercise strategies. Let us assume we are dealing with a payer swaption. P. Jaeckel [Jae02] approaches the problem as follows: at any given time $t = T_j$ we only have to make the decision between immediate exercise and continuation. Immediate exercise leads to a deterministic payoff $\delta_j(L_j(t) - \kappa)$ and an expected future payoff which is strongly correlated with the swap rate $S_{j+1,n}(t)$ (for a swap along $[T_{j+1}, T_n]$). This suggests that we parametrize the exercise boundary at time $t = T_j$ in terms of the two coordinates $x = L_j(t)$ and $y = S_{j+1,n}(t)$. To determine what the boundary looks like in these coordinates Jaeckel implements a nonrecombining lattice for forward Libors and then recursively computes for each node wether the swaption is exercised at this node or not.

Since the lattice represents a simplified probabilistic model with finitely many states (Libor paths) all of which are processed by the recursive computation, the continuation value (in the simplified model) can be determined accurately at each node of the lattice.

Once this is accomplished we can fix a date $t = T_j$ and go through all nodes in the lattice at time t printing for each node the point (x, y) with $x = L_j(t)$ and $y = S_{j+1,n}(t)$ and coloring the points red in case of exercise and black in case of continuation. If red and black dots are strongly mixed there is no easy way to separate exercise from continuation based on the statistics x and y. If on the other hand red and black dots form two regions which are well separated we can try to parametrize the exercise boundary in terms of x and y alone.

In Chapter 8 we will see how fully recombining two and three factor lattices can be implemented for a driftless Libor market model with constant volatilities. Using such a lattice the function plotBermudanExercise() implemented in ExercisePlot.h carries out this computation in less than a minute.

Some results are contained in the file ExerciseBoundary.tar.gz which can be downloaded from http://martingale.berlios.de/Martingale.html. Inspection shows that the exercise region is well separated from the continuation region

and the boundary can be parametrized as the graph

$$x = p_1 \frac{S_{j+1,n}(0)}{y - p_2} + p_3 \tag{6.63}$$

(the numerator is a normalization factor) depending on three parameters $p_j = p_j(t)$. The parameters are then optimized recursively as in the case of the convex strategy (Section 4.6.4). Exercise at time t occurs if

$$L_j(t) = x > p_1 \frac{S_{j+1,n}(0)}{y - p_2} + p_3 = p_1 \frac{S_{j+1,n}(0)}{S_{j+1,n}(t) - p_2} + p_3.$$

While the two approaches are similar the approach in [Jae02] makes use of additional information about the problem extracted from a computation of all possible scenarios in a simplified model (the lattice). Consequently we can hope that a more precise parametrization of the exercise boundary is obtained. In addition the statistics $x = L_j(t)$ and $y = S_{j+1,n}(t)$ involve much less computational effort than a computation of $Q(t)$.

This approach yields very satisfactory results. The class PjTrigger implements the corresponding exercise strategy. Experiments show that triggers based on the parametrization (6.63) yield slightly better (higher) payoffs than convex exercise (6.62) and the computation is about 10 times faster.

Chapter 7

The Quasi Monte Carlo Method

7.1 Expectations with low discrepancy sequences

The Quasi Monte Carlo method (QMC) is usually formulated as a method to compute an integral

$$I = \int_Q f(x)dx \tag{7.1}$$

where $Q = [0,1]^d$ is the unit cube in R^d and dx denotes the uniform distribution on Q. As the integral I is the expectation $I = E(f(X))$, where X is a uniformly distributed random variable on Q the Monte Carlo method evaluates the integral as

$$I \simeq N^{-1} \sum_{j=1}^{N} f(x(j)) := S_N(f,x) \tag{7.2}$$

where $x = (x(j))_{j\geq 1} \subseteq Q$ is a uniformly distributed sequence and the points

$$x(j) = (x_1(j), x_2(j), \ldots, x_d(j))$$

in this sequence are constructed by simply making repeated calls to a one dimensional uniform random number generator and filling the coordinates with the returned values. If the concrete sequence of points $x(j)$ is replaced with a sequence $X(j)$ of independent uniformly distributed random variables on Q then the Law of Large numbers guarentees convergence

$$S_N(f,x) \to I = \int_Q f(x)dx \tag{7.3}$$

with probability one and the Central Limit Theorem provides the standard deviation $C/\sqrt{N}$, where $C = \sigma(f(X))$, as a probabilistic error bound. Such a bound only allows us to make inference about the probability with which the error exceeds any given size.

For any concrete sequence of points $x(j)$ (such as one delivered by a uniform random number generator) an application of the Strong Law of Large Numbers is not really appropriate and it is not clear how useful a probabilistic error bound is in such a situation. Nothing on our computer is truly random. What we need instead is a property which guarentees the convergence (7.3) for a reasonably large class of functions.

We call the sequence $x(j)$ *equidistributed* in Q if it satisfies (7.3) for all continuous functions f on Q. The term *uniformly distributed* is also used in the literature but should not be confused with the probabilistic term. In this terminology our random number generator should be able to deliver an equidistributed sequence in dimension d.

The problem here is the dimension. The larger the dimension the harder it is to generate equidistributed sequences. The best current uniform random number generator, the Mersenne Twister, is known to deliver equidistributed sequences up to dimension 623.

Quasi Monte Carlo methods replace the uniformly distributed random sequence $x(j)$ (or rather the output of the random number generator) with nonrandom so called *low discrepancy sequences* which are constructed with the explicit goal to fill the cube Q in a regular and efficient manner. Such sequences avoid the wastefull clustering observed in random sequences and the construction is highly nontrivial. Instead of the requirement

$$S_N(f, x) \to I$$

a low discrepancy sequence satisfies the more stringent requirement that this convergence occurs with a certain speed uniformly over a suitable class of test functions defined on Q, namely the indicator functions $f = 1_R$, where $R \subseteq Q$ is a subrectangle with lower left corner at the origin:

$$R = [0, b_1) \times [0, b_2) \times \ldots \times [0, b_d). \tag{7.4}$$

Let $\mathcal{R}$ denote the family of all such functions $f = 1_R$. The function $f = 1_R$ can be identified with the vector $b = (b_1, \ldots, b_d) \in Q$ and the approximation error is the given as

$$D_N(x, b) = D_N(x, f) = \left| S_N(f, x) - \int_Q f(u)du \right|$$

The *star discrepancy* is the sup-norm

$$D_N^*(x) = \sup_{f\in\mathcal{R}} D_N(x,f) \tag{7.5}$$

and measures this error uniformly over the test class $\mathcal{R}$. The definition of the star discrepancy implies the estimate

$$\left|S_N(f,x) - \int_Q f(u)du\right| \le D_N^*(x),$$

for the test functions $f \in \mathcal{R}$ and this estimate extends from test functions in $f \in \mathcal{R}$ to more general functions to provide several *deterministic* error bounds

$$\left|S_N(f,x) - \int_Q f(u)du\right| \le V(f)D_N^*(x),$$

where $V(f)$ is any one of several types of variation of f on Q (see [Nie92]) and

$$\left|S_N(f,x) - \int_Q f(u)du\right| \le \omega(f, D_N^*(x)),$$

if f is continuous on Q and

$$\omega(f,\delta) = \sup\{\, |f(u)-f(v)| \,:\, u,v\in Q,\ |u-v|<\delta\,\}$$

is the modulus of continuity of f. It is clear how such estimates are more useful than mere convergence. One can show that the sequence x is equidistributed in Q if $D_N^*(x) \to 0$, as $N \uparrow \infty$. Unfortunately when we compute an integral we cannot let $N \uparrow \infty$. Instead we have to make do with a fixed number N which often is not very large.

No explicit bounds on the size of D_N (control over the speed of convergence) are required for equidistribution. By contrast the sequence x is called a *low discrepancy sequence* if the star discrepancies $D_N^*(x)$, $N = 1, 2, \ldots$ satisfy

$$D_N^*(x) \le \frac{C(log(N))^d}{N}, \quad N \ge 1.$$

Values for the constant C are known for many concrete constructions (see [Nie92]). This accelerates the speed of convergence quite a bit over the probabilistic bound $C/\sqrt{N}$. The larger the dimension d the longer it takes for $(log(N))^d/N$ to fall below $1/\sqrt{N}$. This accounts for the widespread belief that low discrepancy sequences lose their edge over uniform random sequences as the dimension increases.

Three low discrepancy sequences, Halton, Sobol and Niederreiter-Xing are implemented in our package. The best comparison of these sequences

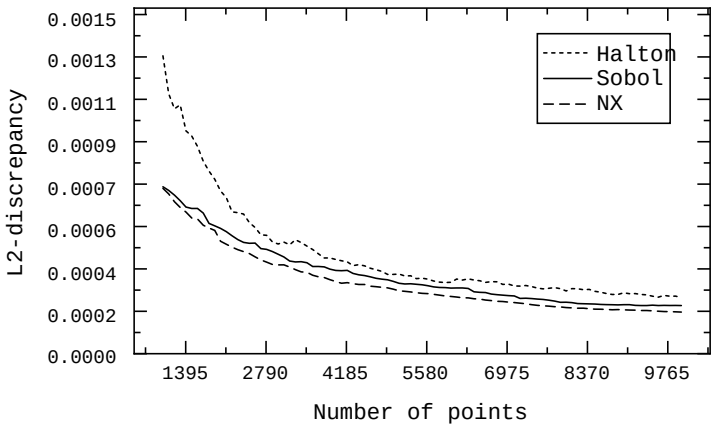


Figure 7.1: L^2-discrepancies in dimension 10.

would be a plot of the star-discrepancies as a function of the number N of points in the sequence. Unfortunately the star discrepancies are difficult to compute. The related L^2-discrepancy $D_N^{(2)}$ is obtained by replacing the sup-norm in (7.5) with an L^2-norm:

$$D_N^{(2)}(x) = \left(\int_Q D_N(x,b)^2 db \right)^{\frac{1}{2}}$$

and is computed much more easily (see [Jae02]). Figure 7.1 contains a plot of the L^2-discrepancy of these three sequences in dimension 10 plotted as a function of the number N of points of the sequence with $N \in [1000, 10000]$.

We have mostly used the Monte Carlo method in a seemingly different context. A stochastic process $X(t)$, $t \in [0,T]$ is given and we want to compute the expectation of $E(H)$ of some functional H (deterministic function) of the path $t \in [0,T] \mapsto X(t)$. For example H could be the payoff of a derivative written on X, where X is a vector of asset prices.

In a simulation this path is replaced with a discretization and the time steps in the discretized path are driven by random numbers $z_1, z_2, \ldots, z_M$ drawn from a suitable distribution. In our case the z_j were always (independent) standard normal numbers and were derived from uniform random numbers $x_1, x_2, \ldots, x_M$ via the inverse normal cumulative distribution function

$$z_j = N^{-1}(x_j). \tag{7.6}$$

With this the simulated path functional H becomes a deterministic function

$$H = h(x) = h(x_1, x_2, \ldots, x_M)$$

of the vector $x = (x_1, x_2, \ldots, x_M)$ of random numbers needed to compute a value of H and the Monte Carlo expectation $E(H)$ computed in the simulation is the integral

$$E(H) = \int_Q h(x)dx.$$

The functional relation between h and the vector x may not be known explicitly but is completely determined by the path generation algorithm. We are now in the situation where low discrepancy sequences can be applied: the vectors $x = (x_1, \ldots, x_M)$ of uniform random numbers derived from the uniform random number generator used to compute each new value of H are replaced with points $x = (x_1, \ldots, x_M)$ from an M-dimensional low discrepancy sequence. In our examples where the paths of the process X are driven by Brownian motions the coordinates of the low discrepancy points are converted to normal deviates via (7.6).

The dimension M is the number of normal deviates needed to simulate a path of X or more precisely to compute a new value of the functional H. It is easy to overlook this issue of dimensionality. In the course of a simulation one simply makes repeated calls to the uniform random number generator converts uniform deviates to normal deviates, uses these to compute a path of the process X, computes a new value for the functional H from the path and repeats this procedure.

How large can this dimension be for example in the case of a Libor derivative H? Here X is the forward Libor process. With quarterly accrual and a ten year life the number of Libors involved is $n = 40$. For a single time step evolving p Libors we need p normal deviates. 39 Libors make the first step, 38 the second step and only one Libor makes the last step. This assumes that the simulated Libor paths step directly from time T_j to time T_{j+1} on the tenor structure without intermediate steps. In this case the dimension M is $n(n-1)/2 = 780$. This is already larger than the guarenteed dimension 623 of the Mersenne Twister. If more time steps are taken this dimension increases dramatically.

Contrast this with the situation in the driftless Libor market model 6.8. Here the distribution of the process $U(t)$ of the state variables $U_j(t)$ is known (the logarithms form a Gaussian process). Consequently it may not be necessary to compute the expectation of a path functional h by path simulation. Instead if we can sample directly from the joint distribution

of the $U_k(t_k)$ which determine the functional h. Quite often only a few of these are needed and the expectation $E[h]$ can then be computed as a low dimensional Gaussian integral.

Example. Consider for example the case of an asset $S(t)$ which follows a geometric Brownian motion with constant volatility σ:

$$dS(t) = S(t)[\mu(t)dt + \sigma dW(t)] \tag{7.7}$$

($W(t)$ a standard Brownian motion) and assume that interest rates are zero. Then the asset is already discounted and hence, in the risk neutral probability, follows the driftless dynamics

$$dS(t) = S(t)\sigma dW(t) \tag{7.8}$$

with the known solution

$$S(t) = exp[-\sigma^2 t/2 + \sigma W(t)] \tag{7.9}$$

Thus a time step $S(t) \mapsto S(t+\Delta t)$ can be simulated without approximation using the equation

$$\begin{aligned} S(t+\Delta t) &= S(t)exp\left[-\sigma^2\Delta t/2 + \sigma(W(t+\Delta t) - W(t))\right] \\ &= S(t)exp\left[-\sigma^2\Delta t/2 + \sigma\sqrt{\Delta t}Z(t)\right], \end{aligned}$$

where $Z(t) = (W(t+\Delta t) - W(t))/\sqrt{\Delta t}$ is standard normal (the normal deviate driving the time step). Moreover increments $Z(t)$, $Z(s)$ corresponding to nonoverlapping time intervals $[t+\Delta t)$, $[s+\Delta s)$ are independent. In this way a path $t \mapsto S(t)$ is driven by a multinormal vector $z = (z_1, z_2, \ldots, z_n)$ of increments z_j, where the increment z_j drives the time step $S(t_{j-1}) \mapsto S(t_j)$. In particular we can write $S(T)$ as

$$S(T) = S(0)exp\left[-\sigma^2 T/2 + \sigma\sqrt{\Delta t}\,(z_1 + z_2 + \ldots + z_n)\right] \tag{7.10}$$

if the path simulation proceeds in equal time steps $t_j - t_{j-1} = \Delta t$ for some fixed positive number Δt. Now let the functional $H = H(t \mapsto S(t))$ be the payoff $H = (S(T) - K)^+$ of the vanilla call on S with strike K expiring at time T. Then we have the explicit formula

$$H(z) = \left(S(0)exp\left[-\sigma^2 T/2 + \sigma\sqrt{\Delta t}\,(z_1 + z_2 + \ldots + z_n)\right] - K\right)^+ \tag{7.11}$$

With this the computation of the call price $E(H)$ (recall that we are operating under the risk neutral probability) can be phrased as an integration problem in any arbitrary dimension n (the number of time steps to the horizon). This makes it useful to experiment and to compare the results of MC and QMC simulation with each other and also with the known analytic call price. The inverse normal cumulative distribution function N^{-1} is used to transform uniform $x \in Q \subseteq R^n$ to standard multinormal $z = \phi(x) \in R^n$. Apply N^{-1} to each coordinate of x:

$$z = \phi(x) = (N^{-1}(x_1), N^{-1}(x_2), \ldots, N^{-1}(x_n)). \tag{7.12}$$

Now we are ready for the following experiment: set S(0)=50, $T = 1$ (time to expiry), $r = 0$ (riskfree rate), $\sigma = 0.4$ and let $C(K)$ denote the (analytic) price of the vanilla call with strike K expiring at time T.

Let $QMC(K)$ denote the call price estimated by QMC simulation of N=1000 paths. Here the asset paths are driven by $z = \phi(x)$ quasinormal vectors (of increments), where $x \in Q \subseteq R^n$ moves through the Sobol sequence.

Let $MC(K)$ denote call price estimated by MC simulation of the same number N of paths driven by standard normal vectors z derived from the Mersenne Twister also using N^{-1}.

The Sobol estimate $QMC(K)$ is deterministic (we always run through the same Sobol points $x(1),\ldots,x(N)$). The Monte Carlo estimate $MC(K)$ however is "random". The result of the simulation depends on the state (seed) of the Mersenne Twister at the beginning of the simulation. Repeated Monte Carlo simulations will show variability. For any given simulation some seeds work better than others but of course it is not known which seeds work well. Thus we can view the Monte Carlo computed price $MC(K)$ as a random variable.

Consequently it makes sense to consider histograms of Monte Carlo computed values $MC(K)$ or the probability with which the Monte Carlo estimate $MC(K)$ beats the Sobol estimate $QMC(K)$. A histogram in dimension 10 with the error for both the Sobol and the Halton sequence superimposed is seen above for the following parpameters: asset $S(0) = 50$, volatility $\sigma = 0.4$, time to expiry $T = 1$.

The histogram is based on 10000 Monte Carlo runs with 10000 paths each. The Sobol sequence beats the Halton sequence (both yield deterministic results) while considerable uncertainty (dependence on the initial seed) remains with the Mersenne Twister.

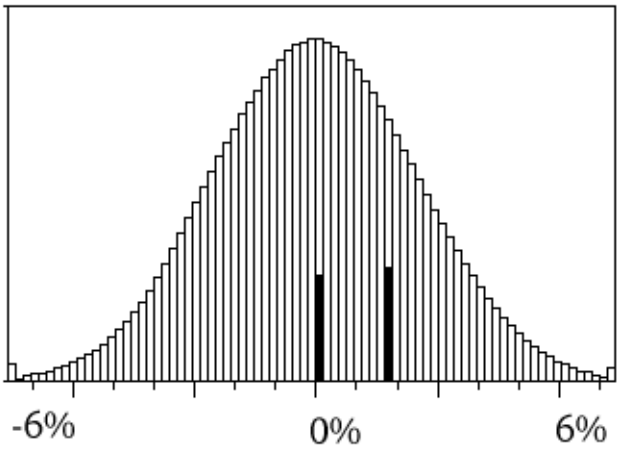


Figure 7.2: Relative error (%).

Next we see a graph plotting the probability that the Monte Carlo estimate is closer to the true value than the Sobol estimate (also derived from 1000 Monte Carlo runs with N paths each) and the relative error $|C(K) - QMC(K)|/C(K)$ of the Sobol estimate both as a function of the strike K for 10 strikes $K = 40, 45, 50, 55, ..., 90$:

At very low dimensions QMC easily beats MC but then the efficiency of QMC trails off. Interestingly at very high dimensions (1600,2000 were tested) QMC becomes much more accurate again and also easily beats MC with high probability. The explanation seems to be that the Mersenne Twister has problems delivering equidistributed vectors in high dimensions. This argues in favour of Quasi Monte Carlo in very high dimensions.

PNG files of Graphs and histograms in other dimensions can be found in the directory Pictures/QMCversusMC. These have been computed by the classes Examples.Pricing.QMCversusMC_1,2.

The Java class Market.ConstantVolatitilityAssetQMC implements a basic Black-Scholes asset with a dynamics driven by low discrepancy sequences. The C++ implementations of the Libor market models PredictorCorrectorLMM.h, PCLMM.h, LognormalLMM.h, BondLMM.h all have the ability to use a dynamics based on the Sobol sequence built in.

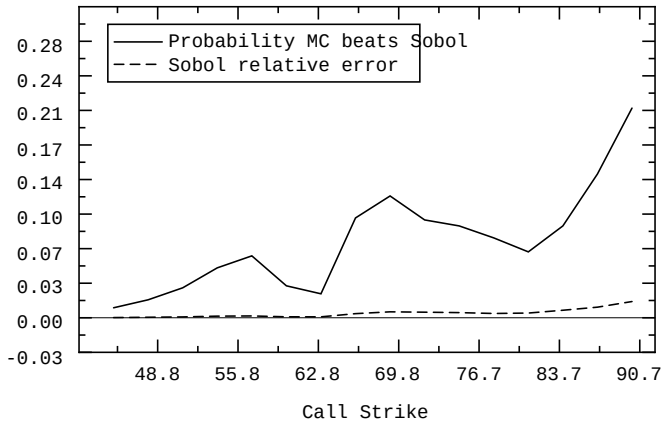


Figure 7.3: Probability that MT beats Sobol.

Chapter 8

Lattice methods

The state space of a stochastic process is the set of all paths. This set is very large and carries a complicated probability distribution. A Monte Carlo simulation generates a raw sample of paths from the distribution of the process. The principle shortcoming of this approach is the fact that the path sample has very little useful structure. For example it cannot be used to compute the conditional expectations $E_t[H]$ of a path functional H at any time other than time zero. If we want to compute the conditional expectation $E_t[H]$ we must average over paths which split at time t and an arbitrary path sample does not have this property.

Of course we can always enforce the path splitting at any point in time. If this is carried out repeatedly the number of paths will quickly become unmanageable. Lattices are an attempt to find a compact representation of a large set of paths which is invariant under path splitting at a large number of times.

8.1 Lattices for a single variable.

Consider the case of a single asset S in state $S = S(0)$ at time zero and following the dynamics

$$dS(t) = S(t)\left[\mu(t)dt + \sigma dW(t)\right],$$

where W is Brownian motion, the drift $\mu(t)$ is deterministic and the volatility σ constant. The case of nonconstant volatility will be dealt with in the next sction. Passing to the returns $Y(t) = log(S(t))$ this assumes the form

$$dY(t) = \left(\mu(t) - \frac{1}{2}\sigma^2\right)dt + \sigma(t)dW(t),$$

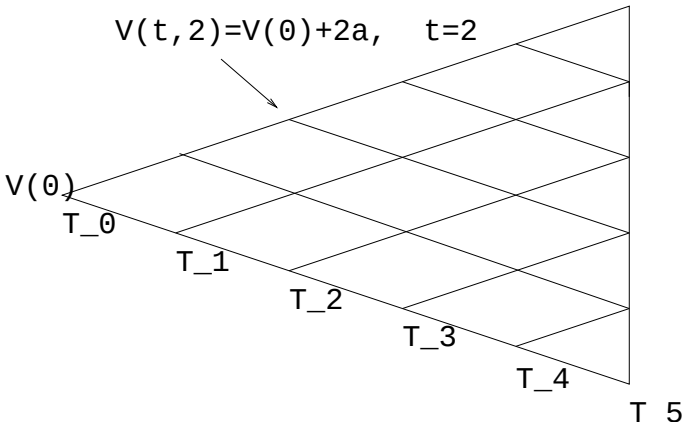


Figure 8.1: lattice

with deterministic drift term. From this it follows that

$$Y(t) = E[Y(t)] + V(t), \quad \text{where} \quad V(t) = \int_0^t \sigma dW(s) = \sigma W(t).$$

and $E[Y(t)]$ is $Y(0)$ plus the cumulative drift up to time t. This quantity is nonstochastic and can be dealt with separately. Thus we need to model only the variable V. From this S is reconstructed as $S(t) = exp\,(E[Y(t)] + V(t))$. Our lattice will sample V at equidistant times

$$0 = T_0 < T_1 < \ldots < T_t < \ldots T_n$$

with $\delta = T_{t+1} - T_t$. Here $t = 0, 1, \ldots, n$ plays the role of discrete time. Note that V follows an *additive* random walk. At each time step $T_t \to T_{t+1}$ we allow V to tick up by an amount a or down by an amount b with probabilities p, q respectively. We will choose $a = b$. As a consequence an up move cancels a previous down move and conversely. The lattice is *recombining*. This greatly limits the number of possible states at discrete time t. Indeed we must have $V(T_t) = V(0) + ja$, where $j \in \{-t, \ldots, t\}$.
This makes for a small number of nodes, see figure 8.1. Let us now compute the transition probabilities. Fix t and set $\Delta = V(T_{t+1}) - V(T_t)$. Then Δ is a mean zero normal variable with variance δ. Moreover

$$\Delta = \begin{cases} a & : \text{ with probability } p \\ -a & : \text{ with probability } q \end{cases}.$$

We have three variables at our disposal and with these we can match the conditions $E[\Delta] = 0$ and $E[\Delta^2] = \delta$ resulting in the equations

$$p + q = 1, \quad a(p - q) = 0, \quad a^2(p + q) = \delta.$$

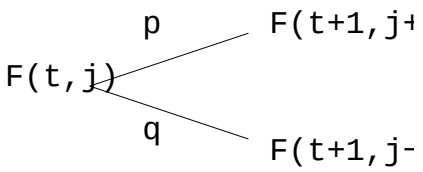


Figure 8.2: $E_t[H]$

with solution

$$p = q = 1/2, \quad a = \sqrt{\delta}.$$

Here everything is pleasantly simple due to the fact that the volatility σ does not depend on time t. In the next section we will deal with time dependent volatility functions $\sigma(t)$ in a lattice for two assets. Let $n(t, j)$ denote the node at time t at which $V(T_t)$ has the value

$$V(T_t) = V(t, j) := V(0) + ja.$$

The lattice has only $(n + 1)(n + 2)/2$ nodes but represents 2^n asset price paths, namely all paths which follow edges in this lattice. Moreover the set of these paths is invariant under path splitting at any time t. This leads to a recursive computation of conditional expectations along nodes in the lattice as follows: let H be an functional (deterministic function) of the path of S. Our simple asset S has the Markov property: the information generated by the path of S up to time t is merely the value $S(t)$ equivalently the value $V(t)$. Consequently the conditional expectation $E_t[H]$ is a deterministic function $F_t(V(t))$ of $V(t)$. Let

$$F(t, j) = F_t(V(t, j)) = F_t(V(0) + ja)$$

denote the value of $E_t[H]$ at the node $n(t, j)$.
Then the Double Expectation Theorem

$$E_t[H] = E_t[E_{t+1}[H]]$$

evaluated at the node $n(t, j)$ assumes the form

$$F(t, j) = pF(t + 1, j + 1) + qF(t + 1, j - 1),$$

where p, q are the probabilities of an up move respectively down move from the node $n(t, j)$ (figure 8.2). If the value of the functional is known by time $t = n$ we can populate the nodes at time $t = n$ (continuous time T_n) with the corresponding values of H and then compute the conditional expectations $E_t[H]$ by backward induction starting from $t = n$ at each node in the tree.

8.1.1 Lattice for two variables

Assume we have two assets $S_j(t)$ satisfying the dynamics

$$dS_j(t) = S_j(t)\left[\mu_j(t)dt + \sigma_j(t)u_j(t)\cdot dW(t)\right], \quad j = 1, 2,$$

where W is a d-dimensional Brownian motion and the drift $\mu_j(t)$ and volatility $\sigma_j(t) \geq 0$ as well as the unit vector $u_j(t)$ are deterministic for all $t \in [0, T]$. Set $\rho_{ij}(t) = u_i(t)\cdot u_j(t)$ and assume that

$$|\rho_{12}(t)| \leq \rho < 1, \tag{8.1}$$

for some constant ρ. The bound ρ measures the degree to which the assets S_1, S_2 are decorrelated. We will see below that a high degree of decorrelation is desirable in the construction. Passing to the returns $Y_j = log(S_j)$ simplifies the dynamics to

$$dY_j(t) = \left(\mu_j(t) - \frac{1}{2}\sigma_j^2(t)\right)dt + \sigma_j(t)u_j(s)\cdot dW(t).$$

Centering the variable $Y_j(t)$ yields $Y_j(t) = E[Y_j(t)] + V_j(t)$ where

$$V_j(t) = \int_0^t \sigma_j(s)u_j(s)\cdot dW(s) \tag{8.2}$$

is the volatility part of Y_j and

$$E[Y_j(t)] = Y(0) + \int_0^t (\mu_j(s) - \sigma_j^2(s)/2)ds. \tag{8.3}$$

This term is deterministic and can be handled separately and so we will build a lattice for the variables V_j since we can then reconstruct the S_j as

$$S_j(t) = exp\left(E[Y_j(t)] + V_j(t)\right).$$

The lattice will sample the variables V_1, V_2 at times

$$0 = T_0 < \ldots < T_t < \ldots T_n = T.$$

Note that here $t = 0, 1, \ldots, n$ plays the role of discrete time. Fix $t < n$, consider the time step $T_t \to T_{t+1}$. The variables V_j follow a *additive* random walks and hence will be allowed to tick up or down by an amount $A_j = A_j(t)$. Set

$$\Delta_j = \Delta_j(t) = V_j(T_{t+1}) - V_j(T_t).$$

The vector $\Delta(t)$ is multinormal with mean zero and covariance matrix $D = D(t)$ given by

$$D_{ij} = D_{ij}(t) = \int_{T_t}^{T_{t+1}} \sigma_i(s)\sigma_j(s)\rho_{ij}(s)ds.$$

In particular the quantity $\sqrt{D_{jj}(t)}$ is the volatility of V_j on the interval $[T_t, T_{t+1}]$. We now allow transitions

$$\begin{pmatrix} \Delta_1 \\ \Delta_2 \end{pmatrix} = \begin{pmatrix} A_1 \\ A_2 \end{pmatrix}, \begin{pmatrix} A_1 \\ -A_2 \end{pmatrix}, \begin{pmatrix} A_1 \\ 0 \end{pmatrix}, \begin{pmatrix} -A_1 \\ A_2 \end{pmatrix}, \begin{pmatrix} -A_1 \\ -A_2 \end{pmatrix}, \begin{pmatrix} -A_1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ A_2 \end{pmatrix}, \begin{pmatrix} 0 \\ -A_2 \end{pmatrix}$$

Let us index the transition probabilities as $p_{i,j}$ with $i, j \in \{-1, 0.1\}$ as follows: the first index i corresponds to V_1, the second index j to V_2, the values $1, -1, 0$ correspond to an uptick, a downtick and no movement respectively. For example $p_{1,-1}$ denotes the probability that V_1 ticks up while V_2 ticks down. The tick sizes $A_j = A_j(t)$ will be dependent on t and will be chosen below.

To derive the transition probabilities p_{ij} let us first match the martingale condition $E[\Delta_j] = 0$ (probability of uptick = probability of downtick = q_j). This yields

$$\begin{aligned} &\sum p_{i,j} = 1 \\ &p_{1,1} + p_{1,-1} + p_{1,0} = p_{-1,1} + p_{-1,-1} + p_{-1,0} := q_1 \\ &p_{1,1} + p_{-1,1} + p_{0,1} = p_{1,-1} + p_{-1,-1} + p_{0,-1} := q_2 \end{aligned}$$

Next we match the covariances $E[\Delta_i \Delta_j] = D_{ij}$ to obtain

$$\begin{aligned} &2A_1^2 q_1 = D_{11} \\ &2A_2^2 q_2 = D_{22} \\ &A_1 A_2[(p_{1,1} + p_{-1,-1}) - (p_{1,-1} + p_{-1,1})] = D_{12} \end{aligned}$$

Finally we match the mixed moments $E[\Delta_i^2 \Delta_j] = E\left[\Delta_i \Delta_j^2\right] = 0$. This yields the equations

$$\begin{aligned} &A_1^2 A_2 [(p_{1,1} + p_{-1,1}) - (p_{1,-1} + p_{-1,-1})] = 0 \\ &A_1 A_2^2 [(p_{1,1} + p_{1,-1}) - (p_{-1,1} + p_{-1,-1})] = 0 \end{aligned}$$

Let us write

$$Q_{ij} = D_{ij}/A_iA_j.$$

With this the equations become

$$\begin{aligned}
&\sum p_{i,j} = 1\\
&p_{1,1} + p_{1,-1} + p_{1,0} = p_{-1,1} + p_{-1,-1} + p_{-1,0} := Q_{11}/2\\
&p_{1,1} + p_{-1,1} + p_{0,1} = p_{1,-1} + p_{-1,-1} + p_{0,-1} := Q_{22}/2\\
&(p_{1,1} + p_{-1,-1}) - (p_{1,-1} + p_{-1,1}) = Q_{12}\\
&p_{1,1} + p_{-1,1} = p_{1,-1} + p_{-1,-1}\\
&p_{1,1} + p_{1,-1} = p_{-1,1} + p_{-1,-1}
\end{aligned}$$

Adding the last two equations yields $p_{1,1} = p_{-1,-1}$ whence $p_{-1,1} = p_{1,-1}$. This implies $p_{1,0} = p_{-1,0}$ and $p_{0,1} = p_{0,-1}$. One now finds that

$$\begin{aligned}
p_{1,0} = p_{-1,0} &= \frac{1}{2}(1 - Q_{22})\\
p_{0,1} = p_{0,-1} &= \frac{1}{2}(1 - Q_{11})\\
p_{1,-1} = p_{-1,1} &= \frac{1}{4}(Q_{11} + Q_{22} - Q_{12} - 1)\\
p_{1,1} = p_{-1,-1} &= \frac{1}{4}(Q_{11} + Q_{22} + Q_{12} - 1)
\end{aligned}$$

From this we see that we must have

$$Q_{11}, Q_{22} \leq 1. \tag{8.4}$$

Moreover we must also have

$$Q_{11} + Q_{22} \geq 1 + Q_{12}. \tag{8.5}$$

Wether or not this is satisfied depends on the choice of the tick sizes A_j. Obviously we must choose $A_j \geq \sqrt{Q_{jj}}$ but not too large. This suggests that we simply choose $A_j = \sqrt{Q_{jj}}$. With this choice (8.5) is indeed satisfied (as we shall see below) but there are other considerations.

We need to limit the number of possible states of the vector $V(t) = (V_1(t), V_2(t)$. To do this we choose a numbers $a_1, a_2 > 0$ which are independent time t and require that the time dependent tick $A_j = A_j(t)$ be integral multiples

$$A_j(t) = k_j(t)a_j$$

of a_j. Here $k_j = k_j(t)$ is chosen such that

$$(k_j - 1)a_j \leq \sqrt{D_{jj}} < k_j a_j, \quad \text{that is,} \quad k_j = 1 + \left[\sqrt{D_{jj}}/a_j\right], \tag{8.6}$$

where $[x]$ denotes the greatest integer less than or equal to x as usual. Note that $D_{jj} = D_{jj}(t)$ depends on t and hence so does the factor k_j. With this (8.4) is automatically satisfied. Since $V_j(s)$ moves up or down in ticks of size $k_j(s)a_j$ it follows that $V_j(t) = V_j(0) + ka_j$, where

$$|k| \leq K_j(t) = \sum_{s<m} k_j(s)$$

This limits the number of possible states at time t to

$$(2K_1(t) + 1)(2K_2(t) + 1). \tag{8.7}$$

As we will see below this number can be quite large. However k is limited to numbers which can be represented in the form

$$k = \sum_{s<t} \epsilon_s k_j(s)$$

with $\epsilon_s \in \{-1, 0, 1\}$. In the Libor market model it often happens that $k_j(s) = k_j > 1$ is constant on $[0, t]$ for all but the last several values of t. This then limits k to multiples of k_j for all such times t and limits the number of states further. The upper bound (8.7) for the number of states at time t depends on the tick sizes a_j as

$$K_j(t) = m + \sum_{s<t} \left[\sqrt{D_{jj}(s)}/a_j\right].$$

Obviously this bound (and in fact the number of states) increases as $a_j \downarrow 0$. Consequently we must find out how small the $a_j > 0$ have to be made.

Recall that a_j is not dependent on time t to ensure the recombining property of the lattice. Fix t and write $D_{ij} = D_{ij}(t)$, $Q_{jj} = Q_{jj}(t)$ and $k_j = k_j(t)$. We must satisfy the inequality (8.5). From the Cauchy-Schwartz inequality and the inequality relating the geometric and arithmetic mean we have

$$Q_{12} \leq \rho\sqrt{Q_{11}}\sqrt{Q_{22}} \leq \frac{\rho}{2}(Q_{11} + Q_{22})$$

with ρ as in (8.1) and so

$$Q_{11} + Q_{22} - (1 + Q_{12}) \geq (1 - \rho/2)(Q_{11} + Q_{22}) - 1$$

and it will suffice to ensure that

$$Q_{11} + Q_{22} \geq \frac{1}{1-\rho/2}$$

and for this it will suffice to have

$$Q_{jj} \geq \frac{1}{2}\,\frac{1}{1-\rho/2} = \frac{1}{2-\rho}, \quad j = 1, 2.$$

Recalling that $Q_{jj} = D_{jj}/A_j^2$, $A_j = k_j a_j$ and $(k_j - 1)^2 a_j^2 \leq D_{jj} < k_j^2 a_j^2$ it will suffice to have

$$\left(1 - \frac{1}{k_j}\right)^2 \geq \frac{1}{2-\rho},$$

that is

$$\frac{1}{k_j} \leq 1 - \frac{1}{\sqrt{2-\rho}} := \epsilon, \quad j = 1, 2.$$

Since $1/k_j < a_j/\sqrt{D_{jj}}$ this will be satisfied if

$$a_j \leq \epsilon\sqrt{D_{jj}(t)}, \quad j = 1, 2,$$

and this inequality must be satisfied for all t. If ρ is close to one the estimates for the number of possible states become so weak that even a number of hundreds of millions of states cannot be ruled out.

The number of nodes depends on the degree of variation of the quantities $D_{jj}(t)$ with t. If these quantities are constant then we can choose $a_j = \sqrt{D_{jj}}$ and have $k_j(t) = 1$ and $K_j(t) = 2t + 1$ leading to a small number of nodes. On the other hand if the quantities $D_{jj}(t)$ are highly variable, then so will be the $k_j(t)$ leading to a huge number of nodes.

From the definition of ϵ it is clear that we want ρ to be as small as possible. In case the variables V_1, V_2 are highly correlated we simply pass to the variables

$$\overline{V}_1 = V_1 - V_2, \quad \overline{V}_2 = V_2$$

and construct the lattice for $\overline{V}_1$, $\overline{V}_2$.

Arbitrage

Assume that the assets S_j are martingales. In practice this means that the S_j are prices in some numeraire evolved in the corresponding numeraire measure. The assets S_j then follow a driftless dynamics, that is, $\mu_j = 0$.

Recall that it is the martingale condition which ensures the absence of arbitrage in the market. Naturally we want to know wether the lattice preserves the martingale condition

$$E_t[S_j(t+1)] = S_j(t) \quad \text{equivalently} \quad E_t\left[S_j(t+1)/S_j(t)\right] = 1.$$

Our lattice does not evolve the S_j directly. Instead we pass to the returns $Y_j = log(S_j)$ and decompose these into a deterministic drift

$$A_j(t) = -\frac{1}{2}\int_0^t \sigma_j^2(s)ds \tag{8.8}$$

and the martingale V_j (the volatility part). The drift for Y_j is derived from Ito's formula which applies to continuous time processes. By construction the lattice preserves the martingale condition for the V_j but the use of the continuous time drifts in the reconstruction

$$S_j = exp(Y_j), \quad Y_j(t) = E[Y_j(t)] + V_j(t) = Y_j(0) + A_j(t) + V_j(t) \tag{8.9}$$

does not preserve the martingale property of the S_j. If the continuous time drift A_j is used we take the view that the lattice is an approximation of the arbitrage free continuous time dynamics but this approximation need not itself be arbitrage free.

To see what drifts B_j for Y_j have to be used to preserve the martingale property of the S_j in the lattice write

$$\Delta Y_j(t) = Y_j(t+1) - Y_j(t) = \Delta B_j(t) + \Delta V_j(t)$$

and so, taking exponentials,

$$S_j(t+1)/S_j(t) = exp\left(\Delta B_j(t)\right) exp\left(\Delta V_j(t)\right).$$

Applying the conditional expectation E_t we see that the martingale condition for S_j assumes the form

$$1 = exp\left(\Delta B_j(t)\right) E_t\left[exp\left(\Delta V_j(t)\right)\right].$$

which yields the drift increment $\Delta B_j(t)$ as

$$\Delta B_j(t) = -log\left(E_t\left[exp\left(\Delta V_j(t)\right)\right]\right). \tag{8.10}$$

In the case of our lattices this quantity is easily seen to be deterministic. Observing that $B_j(0) = 0$ we can obtain the arbitrage free drift $B_j(t)$ of $Y_j(t)$ in the lattice as

$$B(t) = \sum_{s<t} \Delta B_j(t).$$

Constant volatilities and correlations

In case the volatility functions $\sigma_j(t)$, correlations $\rho_{ij}(t)$ and the time steps $\delta_j = T_{j+1} - T_j$ are constant

$$\sigma_j(t) = \sigma_j, \quad \rho_{ij}(t) = \rho_{ij}, \quad \delta_j = \delta$$

the preceeding discussion simplifies considerably. It then follows that

$$D_{ij} = \delta \sigma_i \sigma_j \rho_{ij}$$

and we can take

$$a_j = \sqrt{D_{jj}} = \sigma_i \sqrt{\delta}$$

and so $Q_{ij} = D_{ij}/a_i a_j = \rho_{ij}$. From this it follows that the transition probabilities satisfy $p_{i,0} = p_{0,i} = 0$ and

$$p_{1,1} = p_{-1,-1} = \frac{1}{4}(1 + \rho_{12}), \quad \text{and} \quad p_{1,-1} = p_{-1,1} = \frac{1}{4}(1 - \rho_{12}).$$

Thus V_1 and V_2 move in the same direction with probability $(1 + \rho_{12})/2$ and in opposite directions with probability $(1 - \rho_{12})/2$ in accordance with intuition. Moreover we have $k_j(t) = 1$ and $K_j(t) = t$ which greatly reduces the number of nodes.

However in this case we can reduce the simulation to even more primitive stochastic components. The dynamics of the V_j assumes the form

$$dV_j(t) = \sigma_j u_j \cdot dW(t), \quad V_j(0) = 0,$$

from which it follows that

$$V_j(t) = \sigma_j u_j \cdot W(t). \tag{8.11}$$

Here we have $\rho_{ij} = u_i \cdot u_j$, that is , $\rho = RR'$, where R is the 2×2 matrix with rows u_i. In concrete models the correlation matrix ρ is usually a model parameter and the matrix R can be computed from ρ using the Cholesky factorization of ρ.

Writing $W(t) = (Z_1(t), Z_2(t))$, where Z_1, Z_2 are independent one dimensional Brownian motions (8.11) assumes the form

$$\begin{aligned} V_1(t) &= \sigma_1 \left[R_{11} Z_1(t) + R_{12} Z_2(t) \right] \\ V_2(t) &= \sigma_2 \left[R_{21} Z_1(t) + R_{22} Z_2(t) \right] \end{aligned}$$

so that we have to simulate only the variables $Z_j(t)$ and for these the lattice is particularly simple: the correlation $\rho_{12} = 0$ and hence the transition probabilities are

$$p_{1,1} = p_{-1,-1} = p_{1,-1} = p_{-1,1} = \frac{1}{4}$$

and all other probabilities are zero. A generalization of this case are stochastic models for the assets $S_j(t)$ in which the volatility parts V_j are given as deterministic functions

$$V_j(t) = f_j(X(t), Y(t))$$

of processes $X(t)$, $Y(t)$ with a simple dynamics which can easily be modelled in a lattice. If such a lattice is built without being based on an arbitrage free continuous time model it is crucial to preserve the martingale condition for asset prices relative to a numeraire by computing the correct drifts for the $Y_j = log(S_j)$ in the lattice.

8.1.2 Lattice for n variables

Assume now that we are faced with $n > 2$ assets S_j and corresponding variables V_j as above. Since the number of states in the lattice explodes as the number of variables increases there is no hope to build a lattice which models the evolution of all variables V_j.

The lattice will have to be limited to 3 variables at most. Moreover we have to assume that the volatility functions $\sigma_j(t)$, correlations $\rho_{ij}(t)$ and the time steps $\delta_j = T_{j+1} - T_j$ are all constant

$$\sigma_j(t) = \sigma_j, \quad \rho_{ij}(t) = \rho_{ij}, \quad \delta_j = \delta.$$

In this case we can proceed as in (8.1.1) with an *approximate* rank three factorization of the correlation matrix

$$\rho \simeq RR',$$

where R is an approximate root of rank 3 of ρ, that is,

$$R = (\sqrt{\lambda_1}h_1, \sqrt{\lambda_2}h_2, \sqrt{\lambda_3}h_3)$$

with λ_j being the three largest eigenvalues of ρ and the h_j corresponding orthonormal eigenvectors. See Appendix (A.1). This yields

$$V_j(t) \simeq \sigma_j \left[R_{j1}Z_1(t) + R_{j2}Z_2(t) + R_{j3}Z_3(t)\right] \tag{8.12}$$

where the state variables $Z_j(t)$ are independent one dimensional Brownian motions. Here the annualized volatility on the right hand side is

$$\sigma_j \sqrt{R_{j1}^2 + R_{j2}^2 + R_{j3}^2} < \sigma_j.$$

If we want to preserve volatilities we must rescale the rows of R to unit norm. This will also reestablish the unit correlations $1 = \rho_{jj} = (RR')_{jj}$. However it does diminish the quality of the approximation $\rho \simeq RR'$.

The lattice is built for the variables $Z_j(t)$, $j = 1, 2, 3$ and is particularly simple. The V_j are then obtained from the Z_j using (8.12) and the S_j are obtained using the continuous time drifts (8.8) and (8.9). If the S_j are martingales the use of the continuous time drifts in the lattice does not preserve the martingale property. The lattice is then merely an approximation of the continuous time arbitrage free dynamics of the assets S_j.

Implementation

Let us consider the implementation of a three factor lattice in which the time step and the volatility functions are all constant. In this case the lattice evolves the independent standard Brownian motions $Z_j(t)$, $j = 1, 2, 3$, and all other quantities are deterministic functions of these. The variables Z_j start out with values $Z_j(0) = 0$ and tick up and down independently with ticks of size $a = \sqrt{\delta}$ where δ is the size of the time step. At time t they can be in any state

$$Z_1 = ia \quad Z_2 = ja, \quad Z_3 = ka \quad \text{with} \quad |i|, |j|, |k| \leq K(t) = t + 1.$$

This suggests that we index the nodes by quadruples (t, i, j, k). Transition probabilities are indexed as triples $prob(p, q, r)$ with $p, q, r = \pm 1$ where the indices p,q,r correspond to Z_1, Z_2, Z_3 respectively and the sign signifies an up move or a down move. There are eight such probabilities all equal to 1/8. Each node has eight edges which we keep in a standard list and which are constructed in a loop

```
for(int p=-1; p<2; p+=2)
for(int q=-1; q<2; q+=2)
for(int r=-1; r<2; r+=2){ /* loop body */}
```

The indices then correspond exactly to up and down ticks and the node $node(t, i, j, k)$ will be connected to the node

$$node(t + 1, i + p, j + q, k + r)$$

and this node may already exist. Each new node can be reached from several nodes at time t (the recombining property) and we have to avoid to allocate a new node for the same state several times. One way to deal with this is to allocate a three dimensional array `NewNodes[i][j][k]` of pointers to nodes with $|i|, |j|, |k| \leq t+1$, that is, containing pointers to all possible new nodes and initialized with zero (null pointer). Each new node `node(t+1,i,j,k)` is then registered with this matrix by setting

```
NewNodes(i,j,k)=&node(t+1,i,j,k)
```

The existence of the node can be queried with

```
if(NewNodes(i,j,k)){ /* exists already, don't allocate it */ }
```

The matrices `NewNodes` are large but only temporary.

Number of nodes. The lattice for the variables $Z_j(t)$ has a very simple structure which allows us to compute the number of nodes in the lattice. If r denotes the number of factors then the lattice has $(t+1)^r$ nodes at discrete time t and $N = 1 + 2^r + \ldots + (t+1)^r$ nodes up to time t. In case $r = 2, 3$ this works out to be

$$N = \begin{cases} (t+1)(t+2)(2t+3)/6 & : \quad r = 2 \\ [(t+1)(t+2)/2]^2 & : \quad r = 3 \end{cases} \tag{8.13}$$

With 1GB main memory we can tolerate about 5.3 million nodes corresponding to 250 time steps in a two factor lattice and 3.5 million nodes corresponding to 60 time steps in a three factor model.

Lattice for the Libor market model

The driftless Libor market model 6.8 is amenable to this approach. The assets S_j are the forward transported Libors U_j. We have to assume constant volatility functions $\sigma_j(t)$ for the U_j. The volatilities of the Libors X_j themselves are stochastic. Two and three factor lattices are implemented in the classes LmmLattice2F, LmmLattice3F.

Chapter 9

Utility Maximization

9.1 Introduction

In this chapter we investigate trading according to risk preferences specified by a utility function.

9.1.1 Utility of Money

An informed agent taking part in a lottery must have a way to specify preferences between the random payoffs $X \geq 0$ for which tickets can be purchased. A *utility function*

$$U : [0, +\infty) \to [-\infty, +\infty)$$

is a convenient way to define these preferences. The payoff X is preferred to the payoff Y if and only if

$$E[U(X)] \geq E[U(Y)].$$

In case of equality the agent is indifferent to a choice between X and Y. The quantity $E[U(X)]$ is interpreted as the *expected utility* of the payoff X. With this interpretation the value $U(x)$ is the utility of x dollars (let $X = x$ be a constant) and the function $U(x)$ can be assumed to be *increasing*. If U is differentiable then

$$U(x+1) - U(x) = U'(\xi) \simeq U'(x), \quad \text{for some } \xi \in (x, x+1)$$

is the utility of an additional dollar given that we already have x dollars. The derivative $U'(x)$ is called the *marginal utility of wealth* at wealth level x and

is assumed to be a *decreasing* function of x. In other words it is assumed that we care less and less about each additional dollar as our wealth increases. The utility function $U(x)$ is then *concave* and this property has convenient mathematical consequences. For example it will be used below to establish the existence of certain maxima on noncompact sets.

Let X be a nonconstant (uncertain) random payoff. Because of the indifference between payoffs of the same expected utility an agent with utility function $U(x)$ is willing to pay up to x dollars for the payoff X where x is given by

$$U(x) = E[U(X)].$$

Since U is concave $E[U(X)] < U(E[X])$ (Jensen's inequality) and the increasing property of U implies that $x < E[X]$. In other words an agent with a concave utility function U is willing to pay less than the expected value $E[X]$ for an uncertain payoff X. Consequently such an agent is called *risk averse*.

The thriving insurance industry presents evidence that most people are risk averse. Only risk averse agents are willing to buy insurance against loss for an amount greater than the expected value of the loss while insurance companies have to charge premiums higher than this expected value (the Law of Large Numbers).

Let us now fix a utility function $U : [0, +\infty) \to [-\infty, +\infty)$ with the following properties:

- U is increasing, continuous on $[0, +\infty)$ and differentiable on $(0, +\infty)$.
- The derivative $U'(x)$ is (strictly) decreasing.
- $lim_{x\downarrow 0+} U'(x) = +\infty$ and $lim_{x\uparrow\infty} U'(x) = 0$.

Then U is strictly concave and U' an invertible function from $(0, +\infty)$ onto itself with decreasing inverse function $I = (U')^{-1} : (0, +\infty) \to (0, +\infty)$.

Fix $y > 0$ and set $g(x) := U(x) - yx$, $x > 0$. Then $g'(x) = U'(x) - y$ is decreasing and $g'(x) = 0$ for $x = I(y)$. It follows that $g(x)$ increases for $x \leq I(y)$ and decreases for $x \geq I(y)$ and consequently has a global maximum at $x = I(y)$. In other words we have

$$U(x) - yx \leq U(I(y)) - yI(y), \quad x \geq 0, y > 0. \tag{9.1}$$

Recall that the graph of the inverse function I is obtained from the graph $y = U(x)$ by reflection about $y = x$ and consequently I inherits the following properties from U:

$$I(y) \downarrow 0 \text{ for } y \uparrow +\infty \quad \text{and} \quad I(y) \uparrow +\infty \text{ for } y \downarrow 0. \tag{9.2}$$

9.1.2 Utility and Trading

Trading in a market

$$S(t) = (S_1(t), \ldots, S_n(t)), \quad t \in [0, T],$$

is a particular lottery. Entering the market with x dollars and trading according to the strategy

$$H(t) = (H_1(t), \ldots, H_n(t))$$

(holding $H_j(t)$ shares of $S_j(t)$ at time $t \in [0, T]$) we buy a ticket to the payoff

$$X = x + \int_0^T H(u)dS(u) \tag{9.3}$$

the so called *terminal wealth* of our trading strategy. Recall that the stochastic integral

$$G(t) = \int_0^t H(u)dS(u) \tag{9.4}$$

represents the gains (or losses) from trading according to H and consequently

$$C(t) = x + \int_0^t H(u)dS(u) = x + G(t) \tag{9.5}$$

our wealth at any time $t \in [0, T]$. Assuming that we do not wish to inject funds after initiation of the trading strategy we can admit only strategies H satisfying $C(t) \geq 0$, for all $t \in [0, T]$. Such a strategy will be called *admissible*. With this the family of admissible trading strategies depends on the initial capital x and this is the only significance of x.

Let P denote the market probability, that is the probability governing the realizations of prices observed in the market while Q denotes a probability equivalent to P in which the asset prices $S_j(t)$ are local martingales. This means that the S_j are not cash prices but prices in a numeraire for which Q is a numeraire measure. For example the numeraire could be the riskless bond in which case we are dealing with prices discounted back to time $t = 0$.

We are interested only in the terminal payoff $X = C(T) = x + G(T)$ and want to maximize the expected utility

$$E^P[U(X)]. \tag{9.6}$$

Starting from initial capital x we would like to find an admissible trading strategy H which maximizes the (9.6). To solve this problem we take a

pragmatic approach. Rather than identifying the trading strategy H we look for a payoff X_0 which maximizes (9.6) among all payoffs X of the form (9.3), that is, among all payoffs X which can be reached from initial capital x by trading according to an admissible strategy H. To obtain a simple characterization of these payoffs X we assume that the market S is *complete* (details below).

Once an optimal payoff X_0 is identified it can be viewed as a derivative and we can use trading strategies described in Chapter 5 to replicate X_0 at least approximately. For example we might trade in the assets S with weights minimizing the hedge variance between trades. This is a reasonable approach for discrete hedging of X_0. More sophisticated methods (beyond the scope of this book) are needed to find continuous strategies which exactly replicate X_0.

Let us now investigate which payoffs X can be replicated by admissible trading strategies starting from capital x. If H is an admissible strategy, then the process (9.5) is a nonnegative local martingale and hence a supermartingale in the martingale probability Q. In particular the mean $E^Q[C(t)]$ is a decreasing function of t and so

$$E^Q[X] = E^Q[C(T)] \leq E^Q[C(0)] = x. \tag{9.7}$$

Conversely let F_T denote the sigma field generated by the process $S(t)$ up to time T. Call the market S is *complete* if every F_T-measurable, Q-integrable payoff $X \geq 0$ satisfying (9.7) has the form

$$X = x + \int_0^T H(u)dS(u)$$

for some admissible strategy H. This simplifies the discussion considerably and we assume from now on that the market S is complete. With this our optimization problem can be restated as follows:

$$Maximize\ E^P[U(X)]\ subject\ to\ X \geq 0\ and\ E^Q[X] \leq x. \tag{9.8}$$

The assumption of completeness has eliminated the price process S from explicit consideration and reduced the problem to a problem about random variables only. The process S enters into the picture only through the equivalent local martingale measure Q. In general (S a locally bounded semimartingale) the existence of such a probability Q is equivalent to a topological no arbitrage condition (the condition NFLVR of [DS94]). Market completeness is then equivalent to the uniqueness of the equivalent local martingale probability Q. The proofs of these fundamental results are quite

complicated. In the case of a market S driven by a Brownian motion W, that is, satisfying a dynamics

$$dS(t) = \mu(t)S(t)dt + \sigma(t)S(t)dW(t)$$

with $\mu \in L^1(dt)$ and $\sigma \in L^1(W)$, the existence of Q is related to the existence of a suitable market price of risk process relating the drift vector μ to the volatility matrix σ, see Theorem 4.2 on page 12 of [KS98]. Assuming that the dimensions of S and W are equal, market completeness is equivalent to the invertibility of the volatility matrix σ. See Theorem 6.6 on page 24 of [KS98].

9.2 Maximization of Terminal Utility

Let $\mathcal{X}(x) := \{ X \geq 0 : E^Q[X] \leq x \}$ denote the set of all payoffs which can be reached by admissible trading with initial capital x. It is not clear that the functional $E^P[U(X)]$ assumes a maximum on $\mathcal{X}(x)$ but let us assume that we have a maximum at $X_0 \in \mathcal{X}(x)$, that is

$$E^P[U(X)] \leq E^P[U(X_0)], \quad \forall X \in \mathcal{X}(x). \tag{9.9}$$

We will try to determine X_0 from this condition and then intend to check wether (9.9) is indeed satisfied. Consequently our determination of the random variable X_0 does not have to be fully rigorous. We will use the following simple fact:

Let ϕ, ψ be linear functionals on a vector space B. If $\psi(f) = 0$ implies $\phi(f) = 0$, that is, $\phi = 0$ on $ker(\psi)$, then $\phi = \lambda\psi$ for some constant λ.

This is clear if $\psi = 0$. If $\psi \neq 0$ choose $f_0 \in B$ with $\psi(f_0) = 1$ and note that then $f - \psi(f)f_0 \in ker(\psi)$ for all $f \in B$.

Now fix $\epsilon > 0$ and let B_ϵ denote the vector space of all bounded random variables which vanish outside the set $[X_0 \geq \epsilon]$ and let

$$f \in B_\epsilon \quad \text{with} \quad E^Q[f] = 0.$$

Then $X(t) = X_0 + tf \in \mathcal{X}(x)$ for sufficiently small t (more precisely for $|t| < \|f\|_\infty$ which ensures that $X(t) \geq 0$). Consequently the function

$$g(t) := E^P[U(X(t))]$$

is defined on a neighborhood of zero and has a maximum at zero. Using the differentiability of U and differentiating under the integral sign (don't worry

about justification) we obtain

$$0 = g'(0) = E^P[U'(X_0)f].$$

Thus the linear functionals $\phi(f) := E^P[U'(X_0)f]$ and $\psi(f) := E^Q[f]$ on B_ϵ satisfy

$$\psi(f) = 0 \quad \Rightarrow \quad \phi(f) = 0$$

and from this it follows that there is a constant λ such that $\phi = \lambda\psi$. Let Z denote the Radon-Nikodym derivative

$$Z = dQ/dP$$

of the measures P and Q on $\mathcal{F}_T$. Then $Z > 0$, P-almost surely (by equivalence of P and Q), and

$$E^Q[f] = E^P[Zf], \quad \forall f \in L^1(Q). \tag{9.10}$$

With this the equality $\phi = \lambda\psi$ can be rewritten as

$$E^P[(U'(X_0) - \lambda Z)f] = 0, \quad f \in B_\epsilon.$$

From this it follows that $U'(X_0) - \lambda Z = 0$, equivalently $X_0 = I(\lambda Z)$, P-almost surely on the set $[X_0 \geq \epsilon]$. Since here $\epsilon > 0$ was arbitrary we must have $X_0 = I(\lambda Z)$, P-almost surely on the set $[X_0 > 0]$.

With this motivation we try to maximize (9.8) with a random variable X_0 of the form $X_0 = I(\lambda Z)$ with $\lambda > 0$. Since U is increasing a maximizer X_0 will have to satisfy $E^Q[X_0] = x$. This allows us to determine λ. Obviously we now need the condition:

$$\rho := \inf\{\, \lambda > 0 \,:\, E^Q[I(\lambda Z)] < \infty \,\} < +\infty, \tag{9.11}$$

equivalently $E^Q[I(\lambda Z)] < \infty$ for some $\lambda > 0$. Otherwise the set $\mathcal{X}(x)$ won't contain a random variable of the form $X_0 = I(\lambda Z)$ with $\lambda > 0$. Given (9.11) we can apply the Dominated Convergence Theorem and (9.2) to show that the function

$$h(\lambda) := E^Q[I(\lambda Z)] < \infty$$

is finite and continuous for all $\lambda > \rho$ and that $h(\lambda) \downarrow 0$ for $\lambda \uparrow +\infty$ and $h(\lambda) \uparrow +\infty$ for $\lambda \downarrow 0$. Consequently there is a unique constant $\lambda_0 > 0$ such that

$$E^Q[X_0] = x \quad \text{for } X_0 = I(\lambda_0 Z).$$

We now claim that

Theorem 1 *X_0 maximizes the expected utility $E^P[U(X)]$ for $X \in \mathcal{X}(x)$.*

Proof. Let $X \in \mathcal{X}(x)$ and note that $E^Q[X] \leq x$ while $E^Q[X_0] = x$ and so $E^Q[X_0 - X] \geq 0$. Using (9.1) with $x = X$ and $y = \lambda_0 Z$ (thus $I(y) = X_0$) we obtain

$$U(X) - \lambda_0 ZX \leq U(X_0) - \lambda_0 ZX_0,$$

P-almost surely. Rearrange this as

$$U(X_0) - U(X) \geq \lambda_0 ZX_0 - \lambda_0 ZX.$$

Now take the expectation E^P and use (9.10) to obtain

$$E^P\left[U(X_0) - U(X)\right] \geq \lambda_0 E^Q[X_0 - X] \geq 0. \quad \blacksquare$$

Appendix A

Matrices

We collect some facts and introduce notation used in the text.

A.1 Matrix pseudo square roots

Let C be a positive definite $n \times n$ matrix. Then there exists a uniquely determined upper triangular matrix $n \times n$ R and a uniquely determined lower trinagular matrix L such that

$$C = RR' = LL'.$$

The matrices R and L are pseudo square roots of the matrix C. The algorithm to compute R and L is called the Cholesky factorization of C and is implemented in the class Matrix.h. If the matrix C is only positive semidefinite we can compute pseudo square roots of C from the eigenvalues and eigenvectors of C as follows: Let

$$\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_n$$

be the eigenvalues of C (which are nonnegative), and $\{u_1, u_2, \ldots, u_n\}$ an orthonormal basis of R^n consisting of eigenvectors u_j of C satisfying $Cu_j = \lambda_j u_j$. Let $\Lambda = diag(\lambda_j)$ denote the diagonal matrix with entries $\Lambda_{jj} = \lambda_j$ down the diagonal, set

$$\sqrt{\Lambda} = diag(\sqrt{\lambda_j})$$

and let U be the $n \times n$ matrix with columns $c_j(U) = u_j$. Then

$$CU = (Cu_1, Cu_2, \ldots, Cu_n) = (\lambda_1 u_1, \lambda_2 u_2, \ldots, \lambda_n u_n) = U\Lambda$$

(recall that multiplication by a diagonal matrix on the right multiplies the columns) and so

$$C = U\Lambda U' = RR' \tag{A.1}$$

where

$$R = U\sqrt{\Lambda} = (\sqrt{\lambda_1}u_1, \sqrt{\lambda_2}u_2, \ldots, \sqrt{\lambda_n}u_n).$$

Note also that

$$R'R = \sqrt{\Lambda}U'U\sqrt{\Lambda} = \Lambda \tag{A.2}$$

because of the orthogonality of U. If

$$\lambda_1 \geq \lambda_2 \geq \ldots \lambda_r > \lambda_{r+1} = \ldots = \lambda_n = 0 \tag{A.3}$$

then we can drop the zero columns of R and write

$$C = R_r R_r',$$

where R_r is the $n \times r$ matrix $R_r = (\sqrt{\lambda_1}u_1, \sqrt{\lambda_2}u_2, \ldots, \sqrt{\lambda_r}u_r)$. Even if (A.3) is not true we can *approximate* the matrix C with the r-factor factorization

$$C \simeq R_r R_r'.$$

The matrix R_r is called the (approximate) *rank r root* of C. To see how accurate this approximation is set $C_r = R_r R_r'$ and let Λ_0, Λ_1 denote the diagonal matrices resulting from Λ by setting the first r respectively the remaining $n - r$ diagonal elements (eigenvalues) equal to zero. Then $\Lambda = \Lambda_0 + \Lambda_1$ and $R_r R_r' = U\Lambda_1 U'$ from which it follows that

$$C - C_r = RR' - R_r R_r' = U(\Lambda - \Lambda_1)U' = U\Lambda_0 U'.$$

For any matrix A let

$$\|A\|^2 = Tr(A'A) = \sum\nolimits_{ij} a_{ij}^2$$

denote the trace norm of A. Then $\|UA\| = \|A\| = \|A'\|$ for each unitary matrix U. This implies that

$$\|C - C_r\|^2 = \|\Lambda_0\|^2 = \lambda_{r+1}^2 + \ldots + \lambda_n^2.$$

Consequently the relative error in the rank two approximation of C is given by

$$\frac{\|C - C_r\|}{\|C\|} = \sqrt{\frac{\lambda_{r+1}^2 + \ldots + \lambda_n^2}{\lambda_1^2 + \ldots + \lambda_n^2}}.$$

This error will be small if the sum of the first r eigenvalues is much bigger than the sum of the remaining $n-r$ eigenvalues. An analysis such as this is useful if we want to reduce matrix sizes for speedier simulation. See (B.1.

Optimality. Next we show that all positive definite rank r matrices D satisfy

$$\|C-D\|^2 \geq \lambda_1^2 + \ldots + \lambda_r^2 = \|C-C_r\|^2. \tag{A.4}$$

In other words: C_r is the optimal approximant in the trace norm. To see this we need the following inequality:

Theorem 1 *Let Q be an $n \times n$ matrix with nonnegative entries and V denote the set of all vectors $x \in R^n$ satisfying*

$$x_1 \geq x_2 \geq \ldots \geq x_n \geq 0.$$

If Q satisfies $\sum_j Q_{ij} \leq 1$, for all i, and $\sum_i Q_{ij} \leq 1$, for all j, then

$$(Qx, y) \leq (x, y), \quad \forall x, y \in V. \tag{A.5}$$

Remark. The brackets denote the inner product as usual. The special case of permutation matrices yields the rearrangement inequality

$$x_1y_1 + \ldots + x_ny_n \geq x_{\pi(1)}y_1 + \ldots + x_{\pi(n)}y_n, \quad x, y \in V,$$

where π is any permutation of $\{1, 2, \ldots, n\}$.

Proof. Note that V is closed under addition and multiplication with nonnegative scalars. Fix $x \in V$. If the inequality (A.5) is satisfied for two vectors $y = z, w \in V$ then it is also satisfied for all linear combinations $y = az + bw$ with nonnegative coefficients a,b. Each vector $y \in V$ can be written as a linear combination with nonnegative coefficients

$$y = y_n f_1 + (y_{n-1} - y_n)f_2 + \ldots + (y_1 - y_2)f_n$$

of the vectors

$$f_k = e_1 + e_2 + \ldots + e_k = (1, 1, \ldots, 1, 0, \ldots, 0) \in V,$$

where the e_k form the standard basis of R^n. Consequently it suffices to verify (A.5) for the vectors $y = f_k$, $k = 1, 2 \ldots, n$, that is,

$$\sum_{i=1}^{k}\sum_{j=1}^{n} Q_{ij}x_j \leq \sum_{j=1}^{k} x_j, \quad \forall x \in V,\ k = 1, 2, \ldots, n.$$

Fix k. The same reasoning as above shows that we have to check this inequality only for the vectors $x = f_m$, $m = 1, 2 \ldots, n$. For $x = f_m$ the inequality assumes the form

$$\sum_{i=1}^{k} \sum_{j=1}^{m} Q_{ij} \leq k \wedge m.$$

Interchanging the order of summation if necessary we may assume $k \leq m$ and the inequality now follows from the assumptions on Q. ∎

With this we can proceed to the proof of (A.4). Let $C = U\Lambda U'$ be as above and choose a unitary matrix V such that $D = VMV'$ with $M = diag(\mu_j)$ a diagonal matrix with $\mu_{r+1} = \ldots = \mu_n = 0$. Set $W = UV$. Then W is unitary and

$$\|C - D\|^2 = \|U(C - D)U'\|^2 = \|\Lambda - WMW'\|^2 = Tr(X'X),$$

where $X = \Lambda - WMW'$ is symmetric. It follows that

$$\|C - D\|^2 = Tr(\Lambda^2) - 2Tr(\Lambda WMW') + Tr(WM^2W'). \tag{A.6}$$

Here $Tr(\Lambda^2) = \sum_{j=1}^{n} \lambda_j^2$ and $Tr(WM^2W') = Tr(M^2) = \sum_{j=1}^{r} \mu_j^2$. Finally

$$(WMW')_{ii} = \sum_{j=1}^{r} W_{ij}^2 \mu_j$$

and so

$$Tr(\Lambda WMW') = \sum_{i=1}^{n} \sum_{j=1}^{r} W_{ij}^2 \lambda_i \mu_j = (Q\lambda, \mu),$$

where the matrix Q with entries $Q_{ij} = W_{ij}^2$ satisfies the assumptions of (A.5). The vectors $\lambda = (\lambda_j)$ and $\mu = (\mu_j)$ have nonnegative decreasing components and it follows that

$$Tr(\Lambda WMW') \leq (\lambda, \mu) = \sum\nolimits_{j=1}^{r} \lambda_j \mu_j.$$

Entering this into equation (A.6) we obtain

$$\|C - D\|^2 \geq \sum_{j=1}^{r} (\lambda_j - \mu_j)^2 + \sum_{j=r+1}^{n} \lambda_j^2$$

as desired. ∎

If x, y are vectors in R^n we let (x, y) or $x \cdot y$ denote the dot product of x and y, that is,

$$(x, y) = x \cdot y = \sum_{i=1}^{n} x_i y_i.$$

Suppose we have a factorization $C = RR'$. By definition of the matrix transpose we hve the identity

$$\sum_{i,j=1}^{n} C_{ij} x_i x_j = (Cx, x) = (RR'x, x) = (R'x, R'x) = ||R'x||^2.$$

Let e_i, $1 \leq i \leq n$ denote the standard basis of R^n (coordinate i is one, all other coordinates are zero). Then $x = \sum_{i=1}^{n} x_i e_i$. Fix $1 \leq p < q \leq n$ and write

$$z := x_{p,q} := \sum_{i=p}^{q} x_i e_i.$$

Then we can write

$$\sum_{i,j=p}^{q} C_{ij} x_i x_j = \sum_{i,j=1}^{n} C_{ij} z_i z_j = ||R'z||^2 = ||R'x_{p,q}||^2. \tag{A.7}$$

A.2 Matrix exponentials

Let A be any (not necessarily symmetric) square matrix. The following approximation can be used to implement the matrix exponential $exp(A)$. Let $||\cdot||$ be any submultiplicative matrix norm. Then $||exp(A)|| \leq exp(||A||)$. Choose n such that $2^n > ||A||$ and set $B = 2^{-n}A$. Then $exp(A) = exp(B)^{2^n}$ which can be computed by squaring $exp(B)$ n times. Note that $||B|| \leq 1$. We need to compute $exp(B)$. Set $U = 2^{-6}B$. Then $||U|| \leq 1/64 := f$ and we can compute $exp(B)$ as $exp(U)^{2^6}$ that is, by squaring $exp(U)$ six times:

$$exp(B) = [exp(U)]^{2^6} \simeq \left[I + U + U^2/2! + \ldots + U^k/k!\right]^{2^6}, \tag{A.8}$$

where k will be chosen below. Set

$$E = exp(U) \quad \text{and} \quad F = I + U + U^2/2! + \ldots + U^k/k!.$$

Then $exp(B) = E^{64}$ and E and F commute and consequently the error $err = E^{64} - F^{64}$ in (A.8) satisfies

$$E^{64} - F^{64} = (E - F)(E^{63} + E^{62}F + \ldots + EF^{62} + F^{63})$$

Since $||E||, ||F|| \leq exp\,(||U||) \leq exp(f)$ it follows that

$$||err|| \leq 64 exp(63f)||E - F||.$$

Here

$$\begin{aligned} ||E-F|| &= ||\sum_{j=k+1}^{\infty} U^j/j!|| \\ &\leq \sum_{j=k+1}^{\infty} f^j/j! \\ &\leq [1+f+f^2/2!+\ldots]f^{k+1}/(k+1)! \\ &\leq exp(f)f^{k+1}/(k+1)!. \end{aligned}$$

from which it follows that

$$||err|| \leq 64exp(64f)f^{k+1}/(k+1)! = e\, f^k/(k+1)!. \tag{A.9}$$

For $k = 8$, the error estimate on the right is less than 2.67e-20. We now compute $exp(A)$ as

$$exp(A) = E^{2^{n+6}} \simeq F^{2^{n+6}}.$$

Note that we do not have an estimate for the error $||exp(A) - F^{2^{n+6}}||$ merely for the error $||exp(B) - F^{2^6}||$ to within close to machine precision. It is hoped that the subsequent n repeated squares do not lose too much precision.

To compute the approximation (A.8) for $exp(B)$ we need 7 matrix multiplications to obtain $I + U + U^2/2! + \ldots + U^k/k!$ and then 6 repeated squares to raise this to power 2^6. We then have to perform $[log_2(||A||)]+1$ additional repeated squares for a total of $[log_2(||A||)] + 14$ matrix multiplications. Obviously the smallest submultiplicative matrix norm is desired here, however in practice we must stick with what we can easily compute and this is the norm

$$||A||^2 = \sum_{ij} A_{ij}^2.$$

Appendix B

Multinormal Random Vectors

The distribution of a multinormal vector Y is completely determined by its mean y and covariance matrix C which is symmetric and nonnegative definite. This section treats the simulation of Y and the conditioning of Y on any one of its components. We assume that the covariance matrix has full rank and hence is positive definite.

B.1 Simulation

If the Z_j, $1 \leq j \leq n$, are independent standard normal ($N(0,1)$) random variables, then $Z = (Z_1, Z_2, \ldots, Z_n)$ is an n-dimensional standard normal vector:

$$E(Z_j^2) = 1, \quad E(Z_i Z_j) = 0, \ i \neq j. \tag{B.1}$$

Such a vector can be simulated by simulating independent draws $X_j \in (0,1)$, from a uniform distribution on $(0,1)$ and setting $Z_j = N^{-1}(X_j)$, where N^{-1} is the inverse cumulative normal distribution function.

Once we have a standard normal vector Z we can get a general $N(y, C)$-vector $Y = (Y_1, Y_2, \ldots, Y_n)$ with mean $y_j = E(Y_j)$ and covariance matrix C, $C_{ij} = E[(Y_i - y_i)(Y_j - y_j)]$ as follows: factor the positive definite covariance matrix C as

$$C = RR'$$

where R is upper triangular (*Cholesky factorization* of C). We claim that then the vector

$$Y \quad = \quad y + RZ, \quad \text{that is,} \tag{B.2}$$

$$Y_i \quad = \quad y_i + \sum_{k=i}^{n} R_{ik} Z_k,$$

is multinormal with distribution $N(y, C)$. Since a sum of independent normal variables is normal, every linear combination of components of Y is a normal variable and it follows that Y is multinormal. Let r_i denote row i of R. From (B.1) it follows that $E(Y) = y$ and, for $i \leq j$,

$$E[(Y_i - y_i)(Y_j - y_j)] = \sum_{k=j}^{n} R_{ik} R_{jk} = r_i \cdot r_j = (RR')_{ij} = C_{ij}$$

as desired. Here only the relation $C = RR'$ has been used. Conversely if we have a multinormal vector Y with mean y and covariance matrix $C = RR'$ we can write

$$Y \stackrel{d}{=} y + RZ = y + Z_1 c_1(R) + Z_2 c_2(R) + \ldots + Z_n c_n(R), \tag{B.3}$$

where Z is a standard normal vector, $\stackrel{d}{=}$ denotes equality in distribution and $c_j(R)$ denotes column j of R.

B.2 Conditioning

Let us now investigate what happens if the multinormal vector Y is conditioned on the last component Y_n. We want to determine the conditional distribution of the vector $(Y_1, Y_2, \ldots, Y_{n-1})$ given that $Y_n = u$. It will turn out that this distribution is again multinormal with new means and covariances which will be determined. To see this write

$$\begin{aligned} Y_1 &\stackrel{d}{=} y_1 + R_{11} Z_1 + R_{12} Z_2 + \ldots + R_{1n} Z_n \\ Y_2 &\stackrel{d}{=} y_2 + R_{22} Z_2 + \ldots + R_{2n} Z_n \\ &\ldots \\ Y_n &\stackrel{d}{=} y_n + R_{nn} Z_n \end{aligned} \tag{B.4}$$

where the upper triangular matrix R factors the covariance matrix C of Y as $C = RR'$, the Z_j are independent standard normal and $\stackrel{d}{=}$ denotes equality in distribution. The factorization and triangularity imply that $R_{nn} = \sqrt{C_{nn}}$ and $R_{1n} R_{nn} = C_{1n}$ and so

$$R_{1n}/R_{nn} = C_{1n}/C_{nn}.$$

Now note that

$$Y_n \stackrel{d}{=} u \quad \Longleftrightarrow \quad Z_n \stackrel{d}{=} R_{nn}^{-1}(u - y_n).$$

Substitute this into (B.4) and observe that conditioning Z_n has no effect on the other Z_j by independence. Consequently, conditional on $Y_n = u$, we have

$$\begin{aligned} Y_1 &\stackrel{d}{=} \hat{y}_1 + R_{11}Z_1 + R_{12}Z_2 + \ldots + R_{1,n-1}Z_{n-1} \\ Y_2 &\stackrel{d}{=} \hat{y}_2 + R_{22}Z_2 + \ldots + R_{2,n-1}Z_n \\ &\ldots \\ Y_{n-1} &\stackrel{d}{=} \hat{y}_{n-1} + R_{n-1,n-1}Z_{n-1} \end{aligned}$$

with independent standard normal vectors Z_j. From this it follows that the Y_j are again multinormal with conditional means

$$\hat{y}_j = E(Y_j|Y_n = u) = y_j + (u - y_n)C_{jn}/C_{nn}$$

and conditional covariances

$$E(Y_iY_j|Y_n = u) - \hat{y}_i\hat{y}_j = C_{ij} - R_{in}R_{jn}.$$

More generally assume that we want to condition on the last $j+1$ variables $Y_{n-j}, \ldots, Y_n$ and are interested only in the conditional expectation (rather than the whole conditional distribution). Write

$$\begin{aligned} Y_a &= (Y_1, \ldots, Y_{n-j-1}), \quad y_a = (y_1, \ldots, y_{n-j-1}), \\ Y_b &= (Y_{n-j}, \ldots, Y_n), \quad y_b = (y_{n-j}, \ldots, y_n), \end{aligned}$$

apply the condtional expectation $E[\cdot|Y_b]$ to (B.4) and note that $Z_1, \ldots, Z_{n-j-1}$ are mean zero variables which are independent of Y_b. Here we are using that we are conditioning on the last $j+1$ variables. By independence the conditional mean of $Z_1, \ldots, Z_{n-j-1}$ will also be zero and thus (B.4) becomes

$$\begin{aligned} Y_1 &\stackrel{d}{=} y_1 + R_{1j}Z_j + \ldots + R_{1n}Z_n \\ Y_2 &\stackrel{d}{=} y_2 + R_{2j}Z_j + \ldots + R_{2n}Z_n \\ &\ldots \\ Y_n &\stackrel{d}{=} y_n + R_{nn}Z_n \end{aligned}$$

This equality holds in the mean $E[\cdot|Y_b]$. Let A, B be the matrices

$$A = (R_{ik})_{i=1,k=j}^{i=n-j-1,k=n}, \quad B = (R_{ik})_{j \le i \le k \le n}.$$

Solving for Z from the last $j+1$ equations and substituting into the first $n-j-1$ equations we obtain

$$E\left[Y_a|Y_b\right] = y_a + AB^{-1}Y_b.$$

If we want to condition on other coordinates of Y we simply commute them into the last places and switch the corresponding rows in the covariance matrix of Y.

B.3 Factor analysis

A second approach to simulating an n-dimensional normal vector Y is to diagonalize the covariance matrix C of Y. This reveals more information about Y than the Cholesky fatorization $C = RR'$. Moreover we do not have to assume that the covariance matrix has full rank. Let

$$\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_n$$

be the eigenvalues of C (which are nonnegative) and U an orthogonal matrix such that

$$UCU' = \Lambda := diag(\lambda_j)$$

is a diagonal matrix. Then the λ_j are the eigenvalues of C and the columns $u_j = col_j(U)$ an orthonormal basis of associated eigenvectors: $Cu_j = \lambda_j u_j$. We have seen in (A.1) that the matrix L given by

$$L = U\sqrt{\Lambda} = (\sqrt{\lambda_1}u_1, \sqrt{\lambda_2}u_2, \ldots, \sqrt{\lambda_n}u_n).$$

is a pseudo square root of C:

$$C = LL'.$$

Let $y = E(Y)$. Because of $C = LL'$ we can write

$$Y \stackrel{d}{=} LZ = Z_1\sqrt{\lambda_1}u_1 + Z_2\sqrt{\lambda_2}u_2 + \ldots + Z_n\sqrt{\lambda_n}u_n \tag{B.5}$$

(equality in distribution) with independent standard normal variables Z_j. Note that

$$\|Y\|^2 \stackrel{d}{=} \lambda_1 Z_1^2 + \lambda_2 Z_2^2 + \ldots + \lambda_n Z_n^2$$

by orthogonality of U and since $E(Z_j^2) = 1$ the variance $E(\|Y\|^2)$ satisfies

$$E(\|Y\|^2) = \lambda_1 + \lambda_2 + \ldots + \lambda_n. \tag{B.6}$$

In case all eigenvalues are nonzero the random vector Y depends on n orthogonal standard normal *factors* Z_j. However if

$$\lambda_1 \geq \lambda_2 \geq \ldots \lambda_r > \lambda_{r+1} = \ldots = \lambda_n = 0 \tag{B.7}$$

then (B.5) becomes

$$Y \stackrel{d}{=} Z_1\sqrt{\lambda_1}u_1 + Z_2\sqrt{\lambda_2}u_2 + \ldots + Z_r\sqrt{\lambda_r}u_r$$

and Y depends only on $r < n$ such factors. Here r is the rank of the covariance matrix C which now allows the factorization $C = L_r L_r'$ where L_r is the $n \times r$ matrix

$$L_r = (\sqrt{\lambda_1}u_1, \sqrt{\lambda_2}u_2, \ldots, \sqrt{\lambda_r}u_r). \tag{B.8}$$

Even if (B.7) is not true we can *approximate* the random vector Y with the r-factor approximation

$$Y_r = Z_1\sqrt{\lambda_1}u_1 + Z_2\sqrt{\lambda_2}u_2 + \ldots + Z_r\sqrt{\lambda_r}u_r.$$

Setting $W_r = Z_{r+1}\sqrt{\lambda_{r+1}}u_2 + \ldots + Z_n\sqrt{\lambda_n}u_n$ we have the orthogonal decomposition

$$Y \stackrel{d}{=} Y_r + W_r \quad \text{with} \quad Y_r \cdot W_r = 0$$

and so

$$\|Y\|^2 \stackrel{d}{=} \|Y_r\|^2 + \|W_r\|^2$$

Consequently the r-factor approximation Y_r of Y explains $100q\%$ of the variability of Y where

$$q = \frac{E\|Y_r\|^2}{E\|Y\|^2} = \frac{\lambda_1 + \lambda_2 + \ldots + \lambda_r}{\lambda_1 + \lambda_2 + \ldots + \lambda_n}.$$

The variable Y_r has covariance matrix $C_r = L_r L_r'$ and this matrix approximates the covariance matrix C of Y with relative error

$$\frac{\|C - C_r\|}{\|C\|} = \sqrt{\frac{\lambda_{r+1}^2 + \ldots + \lambda_n^2}{\lambda_1^2 + \ldots + \lambda_n^2}}.$$

See (A.1). This error will be small if the sum of the first r eigenvalues is much bigger than the sum of the remaining $n - r$ eigenvalues. An analysis such as this is useful if we want to limit the number of factors for example to speed up simulation or reduce the dimension in $P_Y(dy)$-integrals:

B.3.1 Integration with respect to P_Y

The dimensional reduction corresponding to zero eigenvalues of the covariance matrix also takes place when integrating with respect to the distribution P_Y of Y. Let

$$n_r(z) = (2\pi)^{-r/2} e^{-\frac{1}{2}\|z\|^2}, \quad z \in R^r,$$

denote the the r-dimensional standard normal density. The r-factor approximation Y_r of Y satisfies $Y_r = L_r Z_r$, with L_r as in (B.8) and Z_r a standard normal vector in R^r. This implies that $P_{Y_r} = L_r(P_{Z_r})$ (image measure) where $P_{Z_r}(dz) = n_r(z)dz$. If $f : R^n \to R$ is measurable and nonnegative the image measure theorem now implies that

$$\begin{aligned} \int_{R^n} f(y) dP_{Y_r}(dy) &= \int_{R^r} (f \circ L_r)(z) dP_{Z_r}(dz) \\ &= \int_{R^r} f(\rho_1 \cdot z, \rho_2 \cdot z, \ldots, \rho_n \cdot z) n_r(z) dz, \end{aligned} \tag{B.9}$$

where $\rho_j = r_j(L_r)$ is the jth row of L_r, $j = 1, \ldots, n$. In particular this integral has been reduced from dimension n to dimension r.

Appendix C

Martingale Pricing

Here is an informal review of martingale pricing of options without entering into technical details.

C.1 Numeraire measure

Cash prices $S_j(t)$, $t \in [0,T]$, of traded assets are not in general local martingales because of growth and inflationary drift inherent in the economy. This drift can often be eliminated by replacing cash with a numeraire which is itself a traded asset subject to this drift.

More formally assume that the market consists of assets $S_j(t)$ which are semimartingales in some probability P on the filtered probablity space $(\Omega, (\mathcal{F}_t)_{t\in[0,T]})$. Fix an asset $B = S_0(t)$ as the numeraire and assume that the market of relative prices $S_j^B(t) = S_j(t)/B(t)$ (prices in the numeraire B) is arbitrage free. Under suitable technical assumptions we can switch to an equivalent probability P_B in which the relative prices are local martingales. A rigorous investigation of the relation between the absence of arbitrage and the existence of local martingale measures P_B is completely beyond the scope of this book ([DS94]).

An *equivalent numeraire probability* associated with the numeraire B is any probability P_B which is equivalent to the original probability and in which all the relative prices $S_j^B(t)$ are local martingales.

In many concrete models of asset prices one fixes a numeraire (often the risk free bond) and finds an explicit numeraire measure associated with this numeraire. Frequently the relative prices $S_j^B(t)$ are in fact *martingales* in the numeraire probability. Let us assume that this is the case. It then follows that the relative price of any *selffinancing* dynamic portfolio trading in the

assets $S_j(t)$ is a P_B-martingale also. This is trivial for static portfolios (linear combinations of assets S_j) and uses properties of the stochastic integral in the case of dynamic portfolios.

In particular the relative price $c^B(t)$ of any option which can be replicated by trading in the assets S_j will be a P_B-martingale also. Any trading strategy can be made selffinancing by investing excess cash in a designated asset such as the riskfree bond for example. In the case of a cash shortfall this means that cash is generated by a short sale of this asset. In other words, the strategy is financed by trading in the designated asset.

Suppose this option expires at time T with cash payoff h. Then the option cash price $c(t)$ satisfies $c(T) = h$ and hence $c^B(T) = h/B(T)$. Since $c^B(t)$ is a P_B-martingale it follows that

$$c^B(t) = E_t^{P_B}[h/B(T)],$$

where $E_t[\cdot]$ denotes the conditional expectation in the probability P_B conditioning on all information $\mathcal{F}_t$ generated by time t. This is the so called *martingale pricing formula*. It is predicated on the assumption that the option can be replicated by trading in the assets S_j since this implies the martingale property of the option price process from the martingale property of the underlying asset prices (in the numeraire B). We assume generally that this assumption is satisfied. In many market models it is the case for all options with payoff which is a deterministic function of the asset price path (market completeness). The cash price $c(t)$ of the option then assumes the form

$$c(t) = B(t)E_t^{P_B}[h/B(T)],\ t \leq T.$$

All kinds of numeraires are useful. For example $B(t)$ could be the money market account in which case the relative prices $c^B(t)$ are discounted prices. If on the other hand if $B(t)$ is the zero coupon bond maturing at time T, then the relative prices $c^B(t)$ are forward prices at time T.

It is often more natural to work with these relative prices than with cash prices. For example discounted prices or forward prices at a fixed date eliminate interest rates from explicit consideration. To switch back to cash prices we then only need to be able to observe the price of the numeraire asset in the market and to multiply with this observed price.

C.2 Change of numeraire.

Assume that P_A is a numeraire measure associated with the numeraire A and that B is another numeraire (that is positive semimartingale). If $B^A = B/A$

is a P_A-martingale we can define a numeraire probability P_B associated with the numeraire B which is equivalent to P_A as follows: on measurable sets D one defines

$$P_B(D) = E^{P_A}[f1_D], \quad \text{that is} \quad E^{P_B}[1_D] = E^{P_A}[f1_D].$$

with density $f = cB(T)/A(T)$ normalized such that P_B is a probability measure. Since B/A is a P_A-martingale by assumption we obtain $c = A(0)/B(0)$ (processes at time zero are assumed to be constants). The martingale property of B/A the yields

$$f_t := E_t^{P_A}[f] = cB(t)/A(t).$$

The defining relation for P_B extends from indicator functions $h = 1_D$ to non-negative functions h and finally to all functions h for which the expectation on the left is defined as

$$E^{P_B}[h] = E^{P_A}[fh]$$

One now verifies that for conditional expectations

$$E_t^{P_B}[h] = E_t^{P_A}[fh]/f_t \tag{C.1}$$

(Bayes theorem, the denominator on the right is necessary for normalization, let h=1)). For a proof multiply with f_t and pull f_t into the expectation on the left to transform this into

$$E_t^{P_B}[f_t h] = E_t^{P_A}[fh]$$

equivalently

$$E^{P_B}[f_t hk] = E^{P_B}\left[E_t^{P_A}[fh]k\right] = E^{P_B}\left[E_t^{P_A}[fhk]\right]$$

for all $\mathcal{F}_t$-measurable indicator functions k. By defintion of P_B this is equivalent with

$$E^{P_A}[ff_t hk] = E^{P_A}\left[fE_t^{P_A}[fhk]\right]$$

Now condition the inside of the expectation on the right on $\mathcal{F}_t$. Using the Double Expectation Theorem the right hand side becomes $E^{P_A}\left[f_t E_t^{P_A}[fhk]\right]$. Move f_t into the conditional expectation on the right to obtain the left hand side again using the Double Expectation Theorem.

Recall that $f_t = cB(t)/A(t)$ and replace h with $h/B(T)$ in (C.1) to obtain

$$B(t)E_t^{P_B}\left[h/B(T)\right] = A(t)E_t^{P_A}\left[h/A(T)\right]. \tag{C.2}$$

This is the *symmetric numeraire change formula*. Using this with $h = S(T)$ it follows that $S(t)/A(t)$ is a P_A-martingale if and only if $S(t)/B(t)$ is a P_B-martingale. By standard extension this remains true if "martingale" is repalced with "local martingale".

The numeraire measures P_A, P_B associated with different numeraires are equivalent and hence the switch from one numeraire to the other has no effect on the quadratic variation (the volatility) of a continuous price process $S(t)$. Only the bounded variation part (the drift) is affected. It is possible to analyze exactly how the drift terms are related by the switch from the numeraire A to the numeraire B (Girsanov's theorem). The details of this are necessary if we deal with a general continous semimartingale and we want to switch from the dynamics in the P_A-measure to the dynamics in the P_B-measure.

Frequently the whole point of a numeraire change is to switch to a numeraire in which the processes under consideration are local martingales and hence *driftless*. In this case we do not have to know the details of the change of numeraire technique. See C.4 below for an application to the option exchange assets.

C.3 Exponential integrals

In this section we derive some formulas for the means of exponentials of quadratic polynomials of normal variables and related variables. Recall first that the cumulative normal distribution function $N(x)$ is defined as

$$N(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-t^2/2} dt$$

and satisfies $N(+\infty) = 1$. From this we derive some integrals which are needed in the sequel. Assume first that $a > 0$ and note that

$$ax^2 + bx + c = \left(\sqrt{a}\, x + \frac{b}{2\sqrt{a}}\right)^2 + D \quad \text{where} \quad D = c - \frac{b^2}{4a}.$$

It follows that

$$\frac{1}{\sqrt{2\pi}} \int_{-\infty}^{+\infty} e^{-\frac{1}{2}(at^2+bt+c)} dt = \frac{1}{\sqrt{a}} e^{-D/2}. \tag{C.3}$$

Indeed, denoting the integral on the left with I we can write

$$I = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{+\infty} exp\left\{-\frac{1}{2}\left[\left(t\sqrt{a} + \frac{b}{2\sqrt{a}}\right)^2 + D\right]\right\} dt$$

$$= \frac{1}{\sqrt{a}}\frac{1}{\sqrt{2\pi}}\int_{-\infty}^{+\infty} e^{-D/2}e^{-u^2/2}du = \frac{1}{\sqrt{a}}e^{-D/2},$$

where the substitution $u = t\sqrt{a} + \frac{b}{2\sqrt{a}}$ has been employed. Now we relate this to expectations of exponentials of standard normal variables X. First we claim that

$$f(b) := E\left(e^{-(aX+b)^2}e^{\alpha X}\right) = \frac{1}{\sqrt{1+2a^2}}e^{-\frac{1}{2}D} \tag{C.4}$$

where

$$-\frac{1}{2}D = \frac{\alpha^2}{2} - \frac{(b+a\alpha)^2}{1+2a^2}.$$

Indeed denoting the expectation on the left with E and using (C.3) we can write

$$\begin{aligned} E &= \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{+\infty} e^{-(ax+b)^2}e^{\alpha x}e^{-x^2/2}dx \\ &= \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{+\infty} exp\left\{-\frac{1}{2}[(1+2a^2)x^2 + 2(2ab-\alpha)x + 2b^2]\right\}dx \\ &= \frac{1}{\sqrt{1+2a^2}}e^{-\frac{1}{2}D} \end{aligned}$$

where

$$D = 2b^2 - \frac{(2ab-\alpha)^2}{1+2a^2} = 2\,\frac{(b+a\alpha)^2}{1+2a^2} - \alpha^2,$$

as desired. Let us note the special case $E\left(e^{\alpha X}\right) = e^{\alpha^2/2}$. Set

$$k = \alpha - 2ab \quad \text{and} \quad q = 1+2a^2$$

and note that

$$\frac{d}{d\alpha}\left(-\frac{1}{2}D\right) = \frac{k}{q}.$$

Differente (C.4) with resepct to α commuting the derivative with the expectation to obtain

$$E\left(Xe^{-(aX+b)^2}e^{\alpha X}\right) = f'(\alpha) = k\,\frac{e^{-D/2}}{q\sqrt{q}}. \tag{C.5}$$

Denote the right hand side with $g(\alpha)$, and differentiate with respect to α again observing that $dk/d\alpha = 1$ to obtain

$$E\left(X^2e^{-(aX+b)^2}e^{\alpha X}\right) = g'(\alpha) = \frac{e^{-D/2}}{q^2\sqrt{q}}\,[q+k^2]. \tag{C.6}$$

Putting all this together we finally have the general formula

$$E\left((AX^2+BX+C)e^{-(aX+b)^2}e^{\alpha X}\right)=\frac{e^{-D/2}}{q^2\sqrt{q}}\left[A(q+k^2)+Bqk+Cq^2\right] \quad \text{(C.7)}$$

Let us summarize the three special cases of this formula which we actually need

$$E\left(e^{-(aX+b)^2}e^{\alpha X}\right) = \frac{e^{-D/2}}{\sqrt{q}} \quad \text{(C.8)}$$

$$E\left((aX+B)e^{-(aX+b)^2}e^{\alpha X}\right) = \frac{e^{-D/2}}{q\sqrt{q}}[ak+Bq] \quad \text{(C.9)}$$

$$E\left((aX+B)^2e^{-(aX+b)^2}e^{\alpha X}\right) = \frac{e^{-D/2}}{q\sqrt{q}}[(ak+Bq)^2+a^2q] \quad \text{(C.10)}$$

Completing the square in the exponent one finds that

$$\frac{1}{\sqrt{2\pi}}\int_{-\infty}^{x} exp\left[-\frac{1}{2}(t^2+2mt+n)\right]dt = e^{(m^2-n)/2}\,N(x+m). \quad \text{(C.11)}$$

Next we claim

$$E\left(e^{aX+b}N(\alpha X+\beta)\right)=e^{\frac{a^2}{2}+b}N\left(\frac{\beta+a\alpha}{\sqrt{1+\alpha^2}}\right). \quad \text{(C.12)}$$

It will suffice to show this for $b=0$. Set

$$f(u)=E\left(e^{aX}N(\alpha X+u)\right)$$

note that $f(-\infty)=0$, differentiate with respect to u and commute the derivative with the expectation to obtain

$$\begin{aligned} f'(u) &= \frac{1}{\sqrt{2\pi}}E\left(e^{aX}e^{-\frac{1}{2}(\alpha X+u)^2}\right) \\ &= \frac{1}{\sqrt{2\pi}}\frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\infty}e^{ax}e^{-\frac{1}{2}(\alpha x+u)^2}e^{-\frac{1}{2}x^2}dx \\ &= \frac{1}{\sqrt{2\pi}}\frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\infty}e^{-\frac{1}{2}((1+\alpha^2)x^2+2(\alpha u-a)x+u^2)}dx. \end{aligned}$$

Using (C.3) this becomes

$$f'(u)=\frac{1}{\sqrt{2\pi}}\frac{1}{\sqrt{1+\alpha^2}}e^{-D/2},$$

where

$$D(u) = u^2 - \frac{(\alpha u - a)^2}{1+\alpha^2} = \left(\frac{u}{\sqrt{1+\alpha^2}}\right)^2 + \frac{2a\alpha}{\sqrt{1+\alpha^2}}\,\frac{u}{\sqrt{1+\alpha^2}} - \frac{a^2}{1+\alpha^2}.$$

It follows that

$$\begin{aligned} f(\beta) &= \int_{-\infty}^{\beta} f'(u)du = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\beta}\frac{1}{\sqrt{1+\alpha^2}}e^{-\frac{1}{2}D(u)}du \\ &= \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\frac{\beta}{\sqrt{1+\alpha^2}}} exp\left[-\frac{1}{2}\left(t^2 + \frac{2a\alpha}{\sqrt{1+\alpha^2}}\,t - \frac{a^2}{1+\alpha^2}\right)\right]du, \end{aligned}$$

where the substitution $t = u/\sqrt{1+\alpha^2}$ has been employed. Use (C.11) with $n = -\frac{a^2}{1+\alpha^2}$ and $m = \frac{a\alpha}{\sqrt{1+\alpha^2}}$ and observe that $m^2 - n = a^2$ to obtain

$$f(\beta) = e^{a^2/2}N\left(\frac{\beta + a\alpha}{\sqrt{1+\alpha^2}}\right),$$

as desired.
Let $a > 0$ and Y be a standard normal variable. A straightforward computation with the normal density shows that

$$E\left[e^{aY} - K\right]^+ = e^{a^2/2}N\left(\frac{a^2 - log(K)}{a}\right) - K\,N\left(\frac{-log(K)}{a}\right). \qquad \text{(C.13)}$$

Finally if Y, Z are any jointly normal variables with means $y = E(Y)$, $z = E(Z)$ and covariance matrix C we want to compute the expectation $E\left[e^Y - Ke^Z\right]^+$. Write

$$\begin{aligned} Y &= y + aU + bW \\ Z &= z + cW \end{aligned}$$

where U, W are independent standard normal variables and the upper triangular matrix

$$R = \begin{pmatrix} a & b \\ 0 & c \end{pmatrix}$$

satisfies $C = RR'$, that is

$$a^2 + b^2 = C_{11}, \quad bc = C_{12}, \quad c^2 = C_{22}.$$

With this

$$E\left(e^Y - Ke^Z\right)^+ = E\left(e^{y+aU+bW} - Ke^{z+cW}\right)^+. \qquad \text{(C.14)}$$

Conditioning on $W = w$ we can write

$$E\left(e^Y - Ke^Z\right)^+ = \int E\left[(e^Y - Ke^Z)^+ | W = w\right] P_W(dw), \tag{C.15}$$

where U is still standard normal, $Y = y + bw + aU$ and $Z = z + cw$ and so

$$E\left[(e^Y - Ke^Z)^+ | W = w\right] = e^{y+bw} E\left[e^{aU} - Ke^{z-y+(c-b)w}\right]^+.$$

Using (C.13) we obtain

$$\begin{aligned} &E\left[(e^Y - Ke^Z)^+ | W = w\right] = \\ &e^{y+bw+a^2/2} N\left(-\frac{c-b}{a}w + \frac{a^2 - log(K) + y - z}{a}\right) - \\ &Ke^{z+cw} N\left(-\frac{c-b}{a}w + \frac{-log(K) + y - z}{a}\right) \end{aligned}$$

Integrating this with respect to $P_W(dw)$ it follows that

$$\begin{aligned} &E\left[e^Y - Ke^Z\right]^+ = \\ &e^{y+a^2/2} E\left[e^{bW} N\left(-\frac{c-b}{a}W + \frac{a^2 - log(K) + y - z}{a}\right)\right] - \\ &Ke^{z+cW} E\left[N\left(-\frac{c-b}{a}W + \frac{-log(K) + y - z}{a}\right)\right]. \end{aligned}$$

Using (C.12) this evaluates

$$\begin{aligned} E\left[e^Y - Ke^Z\right]^+ &= e^{y+(a^2+b^2)/2} N\left(\frac{a^2 - log(K) + y - z}{\sqrt{a^2 + (c-b)^2}}\right) - \\ &\quad Ke^{z+c^2/2} N\left(\frac{-log(K) + y - z}{\sqrt{a^2 + (c-b)^2}}\right). \end{aligned}$$

Here $a^2 + (c-b)^2 = C_{11} + C_{22} - 2C_{12}$ and we have

Proposition C.3.1 *Assume that Y, Z are jointly normal variables with means $y = E(Y)$, $z = E(Z)$ and covariance matrix C. Then*

$$E\left[e^Y - Ke^Z\right]^+ = f(y, z, C), \tag{C.16}$$

where the function f of the mean and covariance matrix is given by

$$\begin{aligned} f(y, z, C) &= e^{y+C_{11}/2} N\left(\frac{C_{11} - C_{12} - log(K) + y - z}{\sqrt{C_{11} + C_{22} - 2C_{12}}}\right) - \\ &\quad Ke^{z+C_{22}/2} N\left(\frac{C_{12} - C_{22} - log(K) + y - z}{\sqrt{C_{11} + C_{22} - 2C_{12}}}\right). \end{aligned}$$

C.4 Option to exchange assests

Let us now examine the option to exchange assets with a view of deriving analytic approximations to the price in the case of stochastic volatility and covariation. Suppose that $S_1(0)$ and $S_2(0)$ are constants and that $S_1(t)$, $S_2(t)$ are positive continuous local martingales in the probability P. We can then regard P as a numeraire measure associated with the constant numeraire 1. The option to receive one share of S_1 in exchange for K shares of S_2 at time T has payoff

$$h = (S_1(T) - KS_2(T))^+ . \tag{C.17}$$

We are now interested in to obtain an analytic approximation for the expectation $E_t^P[h]$. To do this we assume that S_2 is actually a martingale under P. We can then regard S_2 as a new numeraire with associated equivalent numeraire probability P_2 in which $Q = S_1/S_2$ is a local martingale and which satisfies

$$E_t^P[h] = S_2(t)E_t^{P_2}[h/S_2(T)] = S_2(t)E_t^{P_2}\left[(exp(Y(T)) - K)^+\right] . \tag{C.18}$$

where $Y(t) = log(Q(t))$. This is the symmetric numeraire change formula (C.2)) where the numeraire associated with P is the constant 1. In case $S_1 = S$, $S_2 = 1$ this is the payoff of the call on S with strike price K, $Q = S$ and $P_2 = P$.

The following computation of the conditional expectation (C.18) uses some heavy machinery but makes it very clear which role the quadratic variation of the logarithm $Y = log(Q)$ (the returns on Q) plays in the determination of the option price. The main ingredient is the representation of a continuous local martingale as a time change of a suitable Brownian motion.

In our application to caplet, swaption and bond option prices it will be useful to write the pricing formula in terms of the quadratic variation of Y since we can then use rules from the stochastic calculus to compute this quantity in the concrete cases below.

The more usual approach has to assume that volatility is deterministic from the start but we can then identify the quadratic variation in the formula and argue the general case by analogy. This is indicated at the end of this section.

The process $Y(t)$ is a continuous semimartingale in the probability P_2 and so has a decomposition

$$Y(t) = A(t) + M(t)$$

where $M(t)$ is a local martingale and $A(t)$ a bounded variation process with $A(0) = 0$ under P_2. Using Ito's formula on the identity $Q(t) = exp(Y(t))$ we obtain

$$\begin{aligned} dQ(t) &= Q(t)dY(t) + \frac{1}{2}Q(t)d\langle Y\rangle_t \\ &= S(t)dM(t) + Q(t)dA(t) + \frac{1}{2}Q(t)d\langle Y\rangle_t \end{aligned}$$

Here $dQ(t)$ and $Q(t)dM(t)$ are the differentials of local martingales while the remaining terms are the differentials of bounded variation processes. Since the local martingale on the right cannot have a nonconstant bounded variation part it follows that

$$Q(t)dA(t) + \frac{1}{2}Q(t)d\langle Y\rangle_t = 0$$

and hence the differential $2dA(t) - d\langle Y\rangle_t$ vanishes, that is, the process $2A(t) - \langle Y\rangle_t$ is constant. Because of $A(0) = 0$ this process vanishes at time $t = 0$ and it follows that

$$A(t) = -\frac{1}{2}\langle Y\rangle_t. \tag{C.19}$$

Thus

$$Y(t) = -\frac{1}{2}\langle Y\rangle_t + M(t)$$

Now the local martingale $M(t)$ can be written as a time change

$$M(t) = M(0) + W(\tau(t)) = Y(0) + W(\tau(t))$$

of a suitable P_2-Brownian motion W where

$$\tau(t) = \langle M\rangle_t = \langle Y\rangle_t. \tag{C.20}$$

This is a nontrivial martingale representation theorem ([KS96], Theorem 4.6). Note that $M(0) = Y(0)$ is constant. The logarithm $Y(t)$ is the process of "returns" and the quadratic variation $\tau(t)$ the cumulative "volatility" of returns of the quotient Q on $[0, t]$. We have

$$Y(t) = Y(0) - \frac{1}{2}\langle Y\rangle_t + W(\tau(t)).$$

Replace t with $T > t$ and subtract the two identities to obtain

$$Y(T) = Y(t) - \frac{1}{2}\langle Y\rangle_t^T + [W(\tau(T)) - W(\tau(t))]. \tag{C.21}$$

Now assume that the volatility $\tau(s)$ is nonstochastic and condition on $\mathcal{F}_t$. Given $\mathcal{F}_t$ and hence $Y(t)$ the variable $Y(T)$ is normal with mean $Y(t) - 2^{-1}\langle Y\rangle_t^T$ and variance $\Sigma^2(t,T) = \tau(T) - \tau(t) = \langle Y\rangle_t^T$. With this the usual computation shows that the expectation (C.18) is given by

$$\begin{aligned} E_t^P[h] &= S_1(t)N(d_+) - KS_2(t)N(d_-), \quad \text{where} \qquad \text{(C.22)} \\ d_\pm &= \Sigma(t,T)^{-1} log(Q(t)/K) \pm \frac{1}{2}\Sigma(t,T) \quad \text{and} \\ \Sigma(t,T) &= \sqrt{\langle Y\rangle_t^T}. \qquad \text{(C.23)} \end{aligned}$$

Note carefully that we have assumed that the S_j are local martingales under P. Forward prices under the forward martingale measure P and discounted prices under the spot martingale measure P have this property but cash prices do not. Since P can be regarded as a numeraire measure associated with the numeraire 1 the expectation (C.22) is then also the martingale price of the option to exchange assets.

Of course the above reasoning is only correct if the volatility $\tau(s) = \langle Y\rangle_s$ is deterministic. Otherwise $W(\tau(T)) - W(\tau(t))$ need not be a normal variable. However equation (C.22) can be used as an approximation in the case of stochastic volatility. Wether or not it is a good approximation depends on the volatility $\tau(s)$. We do have to make an adjustment though: the quantity $\Sigma(t,T)$ to be used in (C.22) must be known by time t (when the formula is used), that is, it must be $\mathcal{F}_t$-measurable. More rigorously: since the left hand side of (C.22) is $\mathcal{F}_t$-measurable so must be the right hand side.

We can think of $\Sigma(t,T)$ as a forecast at time t for the volatility of returns on the quotient $Q = S_1/S_2$ on the interval $[t,T]$ to option expiry.

In short: to get an approximation for $E_t^P[h]$ under stochastic volatility we use (C.22) where $\Sigma^2(t,T)$ is an $\mathcal{F}_t$-measurable estimate of the quadratic variation $\langle Y\rangle_t^T$ (a constant in case $t = 0$). There is no reason to believe that this works in general but it does work in the cases which we have in mind (approximate caplet, swaption and bond option prices).

We should note that the volatility $\tau(s) = \langle Y\rangle_s$ does not change if we switch from some probability P to an equivalent probability P_0. This is useful since P_0 will be the probability associated with a change of numeraire designed to make the S_j local martingales. We can then still compute the volatility $\tau(s)$ in the original probability P. We will apply this in cases where Y is a deterministic function of other state variables and we will use Ito's formula to estimate the quadratic variation $\langle Y\rangle_t^T$, see C.5.

Simplified derivation in case of deterministic volatility. The usual approach assumes that Q follows a dynamics

$$dQ(t) = Q(t)\sigma(t) \cdot dW(t)$$

where W is a Brownian motion under P_2 and the volatility $\sigma(t)$ is deterministic. The dynamics is driftless since Q is a local martingale under P_2. From this it follows that the logarithm $Y = log(Q)$ satisfies

$$dY(s) = -\frac{1}{2}\sigma^2(s)dt + \sigma(s) \cdot dW(s).$$

Integration over the interval $[t, T]$ yields

$$Y(T) = Y(t) - \frac{1}{2}\Sigma^2(t, T) + \int_t^T \sigma(s) \cdot dW(s),$$

where

$$\Sigma^2(t, T) = \int_t^T \sigma^2(s)ds.$$

Since $\sigma(s)$ is deterministic the integral $\int_t^T \sigma(s) \cdot dW(s)$ is a mean zero normal variable with variance $\Sigma^2(t, T)$. With this information the expectation (C.22) can be computed.

Finally we have to note only that the quadratic variation $\langle Y \rangle$ satisfies $d\langle Y \rangle_t = \sigma^2(t)dt$ from which it follows that $\Sigma^2(t, T)$ is the quadratic variation of Y on the interval $[t, T]$.

C.5 Ito's formula

This is a brief review of the manipulation of stochastic differentials and Ito's formula for Ito integrals:

$$X(t) = \int_0^t \nu(s)dW(s),$$

where W is an n-dimensional Brownian motion and $\nu(s)$ a W-integrable $n \times n$ matrix valued process with rows $\nu_i(s)$. Let

$$X_i(t) = \int_0^t \nu_i(s) \cdot dW(s)$$

denote the components of X. Then the covariation $\langle X_i, X_j \rangle$ satisfies

$$d\langle X_i, X_j \rangle_s = \nu_i(s) \cdot \nu_j(s)ds.$$

Let $f = f(x), x \in R^n$ be a twice continuously differentiable function. Then Ito's formula states that

$$f(X(t)) = A(t) + \sum_{i=1}^{n} \int_0^t \frac{\partial f}{\partial x_i}(X(s)) dX_i(s),$$

where $A(t)$ is a continuous bounded variation process, in fact

$$A(t) = f(X(0)) + \frac{1}{2} \sum_{i,j=1}^{n} \int_0^t \frac{\partial^2 f}{\partial x_i^2}(X(s))\ \nu_i(s) \cdot \nu_j(s) ds.$$

The continuous bounded variation process A does not contribute to the quadratic variation of X and using the rule

$$\begin{aligned} \left\langle \int_0^t H_i(s) dX_i(s), \int_0^t H_j(s) dX_j(s) \right\rangle &= \int_0^t H_i(s) H_j(s) d\langle X_i, X_j \rangle_s \\ &= \int_0^t H_i(s) H_j(s)\ \nu_i(s) \cdot \nu_j(s) ds \end{aligned}$$

(from the defining property of the stochastic integral) combined with the bilinearity of the covariation bracket $\langle \cdot, \cdot \rangle$ we obtain

$$\langle f(X) \rangle_t = \langle f(X), f(X) \rangle_t = \int_0^t \sum\nolimits_{i,j=1}^{n} \frac{\partial f}{\partial x_i}(X(s)) \frac{\partial f}{\partial x_j}(X(s))\ \nu_i(s) \cdot \nu_j(s) ds.$$

Appendix D

Optional Sampling Theorem

D.1 Optional Sampling Theorem

Let X_t, $t = 0, 1, \ldots, T$, be a sequence of random variables adapted to the filtration $(\mathcal{F}_t)_{t\in[0,T]}$, that is, the $\mathcal{F}_t$ form an increasing chain of σ-fields and each X_t is F_t-measurable. Then the sequence X is called a supermartingale (with respect to the filtration $(\mathcal{F}_t)$) if it satisfies $E_t\,(X_{t+1}) \leq X_t$, for all $t < T$, equivalently

$$E\,[1_A(X_{t+1} - X_t)] \leq 0, \quad \text{for all } t < T \text{ and } A \in \mathcal{F}_t. \tag{D.1}$$

Recall also that an optional time τ (for the filtration $(\mathcal{F}_t)$) is a random variable with values in $\{\,0, 1, \ldots, T\,\}$ such that the event $[\tau \leq t]$ is $\mathcal{F}_t$-measurable, for each $t \in [0, T]$.

Theorem 1 *Assume that X_t, $t = 0, 1, \ldots, T$, is a supermartingale and let τ and ρ be optional times with values in $[0, T]$ satisfying $\rho \geq \tau$. Then $E(X_\rho) \leq E(X_\tau)$.*

Proof. Recall that X_ρ is defind as $(X_\rho)(\omega) = X_{\rho(\omega)}(\omega)$, for each state ω in the probability space. For such a state ω we have

$$(X_\rho - X_\tau)(\omega) = \sum_{t=\tau(\omega)}^{\rho(\omega)-1} X_{t+1}(\omega) - X_t(\omega).$$

This can be rewritten as

$$X_\rho - X_\tau = \sum\nolimits_{t<T} 1_{[\tau\leq t<\rho]}(X_{t+1} - X_t).$$

Here the event $[\tau \leq t < \rho] = [\tau \leq t] \cap [\rho \leq t]^c$ is $\mathcal{F}_t$-measurable, for each $t < T$. The result now follows by taking expectations and using (D.1). ∎

Appendix E

Notes on the C++ code

We conclude with some notes on the C++ code and a survey of the C++ classes.

E.1 Templates

We have already seen in (2.7) how the straightforward use of templates can reduce the size of the code while increasing its scope. Templates can also be essential if we have to avoid virtual functions because of performance considerations. A virtual function call carries a small overhead. Usually the function itself does enough work to make this overhead insignificant by comparison.

On occasion however this is not the case. Suppose for example that we want to design a matrix class. Only the minimum amount of memory will be allocated for each matrix type. For example an upper triangular matrix of dimension n will allocate a triangular array of Reals

```
Real** data=new Real*[n];
for(int i=0;i<n;i++)data[i]=new Real[n-i];
```

and define a subscripting operator

```
Real& operator()(int i, int j){ return data[i][j-i]; }
```

A square matrix by comparison will allocate a rectangular array

```
Real** data=new Real*[n];
for(int i=0;i<n;i++)data[i]=new Real[n];
```

and define a subscripting operator

```
Real& operator()(int i, int j){ return data[i][j]; }
```

With the exception of these differences these matrix classes can share a large amount of code. In a classical inheritance hierarchy both classes UpperTriangularMatrix and SquareMatrix would inherit from a common base Matrix which factors out the common code and implements it in terms of a few pure virtual functions which account for the differences of the concrete subclsses. The subscripting operator

```
class Matrix {

virtual Real& operator()(int i, int j) = 0;

};
```

is one of these virtual functions. Unfortunately this operator does very little and is performance critical. Looking at the above definitions it is a prime candidate for inlining but virtual functions cannot be inlined since it is not known at compile time which code actually will be called upon.

Consequently the above approach is not a good idea. Instead we make Matrix a class template which inherits from its template parameter

```
template<class MatrixBase>
class Matrix : public MatrixBase {

// class body

};
```

The characteristics which are unique to each matrix type are defined in the template parameter MatrixBase such as

```
class Square {

    int n;          // dimension
    Real** data;

    Square(int dim) : n(dim), data(new Real*[dim])
    {
        for(int i=0;i<n;i++)data[i]=new Real[n];
    }

    Real& operator()(int i, int j){ return data[i][j]; }

};
```

```
class UpperTriangular {

    int n;          // dimension
    Real** data;

    UpperTriangular(int dim) : n(dim), data(new Real*[dim])
    {
        for(int i=0;i<n;i++)data[i]=new Real[n-i];
    }

    Real& operator()(int i, int j){ return data[i][j-i]; }

};
```

followed by definitions

```
typedef Matrix<Square> SquareMatrix;
typedef Matrix<UpperTriangular> UpperTriangularMatrix;
```

The classes SquareMatrix and UpperTriangularMatrix define concrete types without a common base class and the subscripting operator is nonvirtual and will be inlined.

Function templates have yet another very useful feature: a template function is instantiated only if it used (called) somewhere. If this does not occur the building blocks (classes and class members) referred to in the definition of the function template need not be declared anywhere.

To make clear how this can be an advantage let us look at an example: suppose we want to implement various options which can be priced either with Monte Carlo path simulation or in a lattice. To enable Monte Carlo pricing the option must implement methods to compute a new path of the underlying assets and a payoff along this path:

```
void newPath();
Real payoffAlongCurrentPath();
```

To enable lattice pricing the option has to implement a method computing the payoff at a node of suitable type

```
Real payoffAtNode(Node* node);
```

There is an immediate difficulty: different options will use nodes of different types so that Node must be the interface to all types of nodes. This interface has to be sufficiently rich to allow all types of options to compute their payoff

relying only on functions provided by the interface. All concrete node types will then have to implement the entire interface and that means that each node type will have to implement functions which are not appropriate for its type. An akward way of dealing with this is to provide empty default implementations in the interface class Node which do nothing but inform the user that the function is not implemented and then kill the program:

```
Real
Node::bondPrice(Bond* bond)
{
    cout << "Node::bondPrice(Bond*): << endl;
    cout << "Not implemented in this generality. Terminating.;
    exit(1);
}
```

These functions can then be overridden selectively as appropriate. With this we might now try some free standing functions defined in a file called Pricing.cc:

```
Real
monteCarloPrice(Option* theOption, int nPath)
{
    Real sum=0.0;
    for(int i=0;i<nPath;i++){

        theOption→newPath();
        sum+=theOption→payoffAlongCurrentPath();
    }
    return sum/nPath;
}


Real
latticePrice(Lattice* theLattice, Option* theOption)
{
    // process all nodes of universal base type Node
    Node* currentNode = ...;
    Real payoff = theOption→payoffAtNode(node);
    ....
}
```

When the compiler sees this it will look for definitions of the classes Option, Lattice and Node and of all the member functions which are called above such as:

```
Option::newPath();
Option::payoffAlongCurrentPath();
Option::payoffAtNode(Node* node);
```

as well as any member functions of Lattice and Node which are referred to and not shown explicitly. Presumably all this will be defined in headers Lattice.h, Option.h, Node.h and these headers will have to be included in Pricing.cc.

The problem here is that we have to have common interfaces Lattice, Option and Node which mold all the different and dissimilar types of lattices, options and nodes into one shape and this will inevitably cause much akwardness. The solution is to parametrize the above functions by the type of Option and Lattice and make them function templates instead:

```
template<class Option>
Real monteCarloPrice<Option>(Option* theOption, int nPath)
{
    Real sum=0.0;
    for(int i=0;i<nPath;i++){

        theOption→newPath();
        sum+=theOption→payoffAlongCurrentPath();
    }
    return sum/nPath;
}


template<class Lattice, class Option>
Real latticePrice<Lattice,Option>(Lattice* theLattice, Option* theOption)
{
    // the type of node used in theLattice
    typedef typename Lattice::NodeType NodeType;
    // process all nodes of the relevant type NodeType
    NodeType* currentNode = ...;
    Real payoff = theOption→payoffAtNode(node);
    ....

}
```

Here Lattice and Option are merely typenames signifying unspecified types. No classes Lattice and Option or any other classes with member functions which are called in the body of monteCarloPrice and latticePrice need be defined anywhere. Nothing has to be included in Pricing.cc.

Of course if such classes are not defined then the function templates monteCarloPrice and latticePrice will be quite useless. The advantage of the new approach is that we can define lattice classes, option classes and node

classes (not necessarily called Lattice, Option and Node) which do not conform to common interfaces and have no common base classes. These classes do not have to implement the full functionality needed to instantiate both monteCarloPrice and latticePrice. Instead we can implement functionality selectively. For example we can define a type of lattice

```
class EquityLattice {

    typedef EquityNode NodeType;
    ....

};
```

using nodes of type EquityNode and an option

```
class EquityCall {

    payoffAtNode(EquityNode* node){....}
    ....

};
```

which knows how to compute its payoff at an EquityNode (instead of a general Node). If the classes EquityLattice and EquityCall provide the members referred to in the body of latticePrice we can make a function call

```
latticePrice(theLattice,theOption);
```

with theLattice of type EquityLattice* and theOption of type EquityCall*. The compiler deduces the types of the theLattice and theOption automatically, substitutes EquityLattice for Lattice and EquityCall for Option, looks for the definition of the classes EquityLattice and EquityCall and checks wether all class members referred to in the body of latticePrice are defined by these classes. If everything is found the function latticePrice is instantiated.

The type EquityOption does not have to declare any of the member functions needed in the body of monteCarloPrice.

This provides us with an enormous degree of flexibility. We can define options which have a Monte Carlo price but no lattice price, options which have a lattice price but no Monte Carlo price and options which have both. Lattices can be built using dissimilar types of nodes, the various options can work with some node types but not others and the various types of lattices, nodes respectively options do not have to conform to common interfaces or have common base classes. Yet combinations of them can be used to

instantiate the function latticPrice to compute the price of a particular type of option in a particular type of lattice using a particular type of node. All that's necessary is that the concrete types provide the building blocks used in latticePrice.

The use of templates in program design is highly developed. The interested reader is referred to the books [JV02] and [Ale01] and the papers http://www.oonumerics.org/blitz/papers/.

E.2 The C++ classes

The C++ code focuses almost entirely on implementations of the Libor Market Model and the pricing of interest rate derivatives. Prices are computed only at time zero and Monte Carlo pricing is selfcontained and does not use the classes RandomVariable or ControlledRandomVariable. Routines for computing Monte Carlo prices and prices in a lattice are defined in the file Pricing.h.

Four types of Libor Market Models (LMMs) are implemented which differ in the type of algorithm used in the simulation: PredictorCorrectorLMM, FastPredictorCorrectorLMM (see 6.5.1) and DriftlessLMM and LowFactorDriftlessLMM (see 6.8). Driftless simulation involves no approximation and is nearly 3 time faster than predictor-corrector simulation and is thus the preferred approach. The class LiborMarketModel is the interface to these concrete types of LMMs.

The information about the factor loadings (6.4) is encapsulated in the class LiborFactorLoading. In the case of a predictor-corrector simulation these are the factor loadings (3.11) of the Libor logarithms $log(X_j)$ while in the case of a driftless simulation they are the factor loadings of the state variables $log(U_j)$ (6.8).

The factor loadings in turn depend on the volatility surface and volatility scaling factors c_j (6.11.1) and the log-Libor correlations (6.11.2). Three different types of volatility surface are implemented in the file VolatilityAndCorrelation.h: constant (CONST), Jaeckel-Rebonato (JR) and our own (M). Two types of correlations are implemented: Jaeckel-Rebonato (JR) and Coffee-Shoenmakers (CS). These can be combined arbitrarily to yield six types of Libor factor loadings.

In addition to the factor loadings only the initial Libors $X_j(0$ and tenor structure $(T_j)_{j=0}^n$ are necessary to fully specify a Libor market model. For simplicity this information is also aggregated in the class LiborFactorLoading.

To make it easy to set up tests the classes LiborFactorLoading and LiborMar-

ketModel contain static methods sample() which return a LiborFactorLoading respectively LiborMarketModel with all initial Libors initialized as $X_j(0) = 0.04$, randomly initialized volatility scaling factors c_j and volatility surface and correlation structure as indicated in the parameter list. Every type of Libor market model can be configured and then be used to compute option prices.

The following options are implemented: caplets, swaptions, calls on bonds and Bermudan swaptions. It is easy to implement others (see for example the Java code). Monte Carlo pricing is available in all LMMs and lattice pricing in two and three factor lattices is available in driftless LMMs with a constant volatility surface. Approximate analytic prices are computed for all options except Bermudan swaptions.

Pricing routines which compute prices of at the money options are implemented in the file TestLMM.h. These are based on the sample LMMs returned by the function LiborMarketModel::sample(parameters). If an analytic price is implemented a relative error is reported as a deviation from the analytic price. Note however that the analytic price is itself approximate so that any other of the prices may be closer to the real price.

Lattices and Monte Carlo pricing are set up so that they can be applied to options on other types of underlyings (asset baskets for example) but no such options are implemented.

In addition to this there are classes implementing random objects, random variables and vectors as well as stochastic processes, and stopping times. Examples are driftless Ito processes (especially Brownian motion in any dimension). The most interesting application here is the solution of the Dirichlet problem on a Euclidean region in any dimension using Brownian motion.

Bibliography

[ABM97] D. Gatarek A. Brace and M. Musiela. The market model of interest rate dynamics. *Mathematical Finance*, 7(2):127–155, 1997.

[Ale01] A. Alexandrescu. *Modern C++ Design*. Addison Wesley, 2001.

[Bri] D. Brigo. A note on correlation and rank reduction.

[CHJ] P. Jaeckel C.J. Hunter and M. Joshi. Drift approximation in a forward rate based LIBOR market model.

[CS99] B. Coffey and J. Schoenmakers. Libor rate models, related derivatives and model calibration, 1999.

[CS00] B. Coffey and J. Schoenmakers. Stable implied calibration of a multi-factor LIBOR model via a semi-parametric correlation structure, 2000.

[CZ02] Goukasian Cvitanic and Zapatero. Hedging with monte carlo simulation, 2002.

[DHM92] R. Jarrow D. Heath and A. Morton. Bond pricing and the term structure of interest rates, a new methodology. *Econometrica*, 60(1):77–105, 1992.

[DS94] F Delbaen and W Schachermayer. A general version of the fundamental theorem of asset pricing. *Mathematische Annalen*, 300:463–520, 1994.

[HK01] M Haugh and L Kogan. Pricing American options: A duality approach. 2001.

[Jae02] P. Jaeckel. *Monte Carlo Methods in Finance*. Wiley, 2002.

[Jam97] F. Jamshidan. Libor and swap market models and measures. *Finance and Stochastics*, 1:293–330, 1997.

[JR] P. Jaeckel and R. Rebonato. The most general methodology to create a valid correlation matrix for risk management and option pricing purposes.

[JT] M. Joshi and J. Theis. Bounding Bermudan swaptions in a swap-rate market model.

[JV02] N. Josuttis and D. Vandevoorde. *C++ Templates: The Complete Guide*. Addison Wesley, 2002.

[KS96] I. Karatzas and S. Shreve. *Brownian Motion and Stochastic Calculus*. Springer, 1996.

[KS98] I. Karatzas and S. Shreve. *Methods of Mathematical Finance*. Springer, 1998.

[Mey02a] M. J. Meyer. http://martingale.berlios.de/Martingale.html, 2002.

[Mey02b] M. J. Meyer. Weights for discrete hedging. *preprint*, 2002.

[Nie92] H. Niederreiter. *Random Number Generation And Quasi Monte Carlo Methods*. Siam, 1992.

[Rog01] L Rogers. Monte carlo valuation of American options, 2001.

[Sch93] M. Schweitzer. Variance optimal hedging in discrete time. *working paper, University of Bonn*, 1993.

[TR99] J. Tsitsiklis and B. Van Roy. Optimal stopping of Markov processes, Hilbert space theory. *IEEE Transacions on Automatic Control*, 44:1840–1851, 1999.

Index