Pricing Exotic Options Using C++

Submitted in partial fulfilment
of the requirements of the degree
Master of Science (Mathematical Statistics)
of Rhodes University

By Tawuya D R Nhongo

September 2006

Abstract

This document demonstrates the use of the C++ programming language as a simulation tool in the efficient pricing of exotic European options. Extensions to the basic problem of simulation pricing are undertaken including variance reduction by conditional expectation, control and antithetic variates. Ultimately we were able to produce a modularized, easily extend-able program which effectively makes use of Monte Carlo simulation techniques to price lookback, Asian and barrier exotic options. Theories of variance reduction were validated except in cases where we used control variates in combination with the other variance reduction techniques in which case we observed increased variance. Again, the main aim of this half thesis was to produce a C++ program which would produce stable pricings of exotic options.

Acknowledgements

I would like to thank my supervisor Professor Irek Szyszkowski for his constant guidance and support. I would also like to thank Tapiwa Daniel Karoro (MComm.) for his insight and advice.

Contents

1 Introduction **1**

1.1 Background 1

1.1.1 Geometric Brownian Motion 1

1.1.2 European Options 3

1.1.3 The Black-Scholes Formula 3

1.1.4 Exotic Options 6

1.1.4.1 Lookback Options 6

1.1.4.2 Barrier Options 6

1.1.4.3 Asian Options 8

1.1.5 Simulation Techniques 8

1.1.5.1 Monte-Carlo Simulation 9

1.1.6 Pricing Exotic Options using Simulation Methods 9

1.1.6.1 Improving Simulation Estimator Efficiency 9

2 Related Work **12**

2.1 Odegaard vanilla option pricing by simulation 12

2.1.1 Simulating lognormally distributed random variables 13

2.1.2 Pricing of European Call options 13

2.1.3 Improving the efficiency of simulation 14

2.1.4 Odegaard Results 18

2.2 Odegaard Monte-Carlo Pricing of options whose payoff depends on the whole price path 18

2.2.1 Simulating a *series* of lognormally distributed variables 19

2.3 Ross/Shanthikumar Efficient simulation of Barrier options 24

2.4 Ross/Shanthikumar Efficient simulation of Asian and Lookback options 26

2.5 Summary 28

3 Methodology **29**
3.1 Pillar I - Generating the data 29
3.2 Pillar II - The pricing program 29
3.2.1 Why C++ 30
3.2.2 How C++ 31
3.2.3 Pricing Exotic European options using C++ 31
3.3 Pillar III - Improve simulator efficiency 32
3.4 Summary 32

4 The Data Set(s) **33**
4.1 Generating Continuous Random Variables 33
4.1.1 The Rejection Method 33
4.1.1.1 Generating a Normal Random Variable using the Rejection Method 34
4.1.2 The Polar Method for Generating Normal Random Variables 36
4.2 Analysis and Descriptive statistics 38
4.3 Summary 38

5 Implementation **39**
5.1 Generating the Data 42
5.2 The Pricing Program 50
5.3 Variance Reduction 53

6 Data Analysis and Results **59**
6.1 Data Analysis and Results 59

7 Conclusion **62**
7.1 Conclusions 62
7.2 Contributions 63
7.3 Future Work 63

References **65**

A Additional Material **67**

Chapter 1

Introduction

The pricing of exotic options is complex because the risk neutral valuation depends on not only the contract end date price of the stock but on the stocks entire price path. New path-dependent options are created often and as such efficient and flexible pricing methods are necessary. Numerical and closed form methods are sometimes not only impractical but impossible in the establishment of exotic option prices. It is the aim of this paper to address this problem and show how simulation techniques can be used to effectively deal with the above problem. As well as this we wish to produce source code which can be easily implemented, amended and extended.

1.1 Background

Before proceeding we discuss points pertinent to the pricing of exotic European options.

1.1.1 Geometric Brownian Motion

Firstly, we discuss Geometric Brownian motion which is the behavioural pattern we assume our security/stock conforms to.

Brownian Motion with Drift

We say that $\{X_t,\ t \geq 0\}$ is a Brownian motion process with drift coefficient μ and variance parameter σ^2 if

1. $X_0 = 0$,
2. $\{X_t,\ t \geq 0\}$ has stationary and independent increments and,

3. X_t is normally distributed with mean μt variance $t\sigma^2$ for each $t > 0$ [10].

Now, let present time be time 0 and let S_y denote the price of our stock at a time y from the present. We say the collection of prices S_y, $0 \leq y < \infty$, follows *geometric Brownian motion (gBm)* with drift parameter μ and volatility parameter σ if, for all nonnegative values of y and t, the random variable

$$\frac{S_{t+y}}{S_y}$$

is independent of all prices up to time y, and if, in addition,

$$\log\left(\frac{S_{t+y}}{S_y}\right)$$

is a normal random variable with mean μt and variance $t\sigma^2$ [12]. Equivalently, if $\{Y_t,\ t \geq 0\}$ is a Brownian motion process with drift coefficient μ and variance parameter σ^2, then the process $\{X_t, t \geq 0\}$ defined by

$$X_t = \exp(Y_t)$$

is called *geometric Brownian Motion.*

So our series of stock prices is geometric Brownian motion if the ratio of the price some time t in the future to the present price has a lognormal probability distribution with parameters μt and $t\sigma^2$, this ratio being independent of all past prices. Probabilities concerning the ratio of the price a time t in the future to the present price will not depend on the present price [12].

The expected value of the price at some prospective time t depends on both geometric Brownian parameters. To see this consider that

$$\log\left(\frac{S_t}{S_0}\right) \sim (\mu t, \sigma^2 t).$$

Now,

$$S_t = s_0 e^W$$

where $s_0 = S_0$ and W is some normal random variable such that; $W \sim N(\mu t, \sigma^2 t)$.

Then,

$$E[S_t] = E\left[s_0 e^W\right]$$

$$= s_0 E\left[e^W\right]$$

$$= s_0 e^{t(\mu+\sigma^2/2)}.$$

We arrive at the final equality by manipulation of the moment-generating function of the normal distribution. We therefore find that, under geometric Brownian motion, the expected price grows at a rate $\mu + \sigma^2/2$.

1.1.2 European Options

Financial markets can be divided into two distinct species. There are the underlying stocks: shares, bonds, commodities, foreign currencies; and their derivatives, claims that promise some payment or delivery in the future contingent on an underlying stock's behaviour [1].

Plain, or vanilla, options are some of the simplest derivatives one will come across. *European style options* may be exercised only at the expiry time of the option, as opposed to *American style options* which can be exercised by the holder at any time up to expiry. Our two main vanilla options are *call* options and *put* options.We define the payoff of a call option as

$$C = (S - K)^+.$$

The function $(x)^+$ returns the maximum between x and 0. A call option is so-called because it gives one the option of calling for the stock at a specified price, known as the *exercise* or *strike* price K [12]. S is the expiry time price of the security. A put option is defined in the following way

$$P = (K - S)^+.$$

This gives the owners the option of putting a stock up for sale at a specified price K [12].

These simple options, coupled with the assumptions we've made about our stocks, can be efficiently priced using the famous *Black-Scholes Formula.*

1.1.3 The Black-Scholes Formula

The Black-Scholes formula gives the unique no-arbitrage cost, C, of a call option on a security whose price varies according to geometric Brownian motion and is given by the following

formula

$$C = S_0\Phi(\omega) - Ke^{-rt}\Phi(\omega - \sigma\sqrt{t}), \tag{1.1}$$

where

$$\omega = \frac{rt + \sigma^2 t/2 - \log(K/S_0)}{\sigma\sqrt{t}}$$

and $\Phi(x)$ is the *standard normal distribution.*

We will now prove that the above formula is indeed the cost of our call option. Under risk-neutral geometric Brownian motion, S_t/S_0 is a lognormal random variable with mean parameter $(r-\sigma^2/2)t$ and variance parameter $\sigma^2 t$ [12]. There are a number of ways of arriving at the above, one of the most popular being by the use of martingale measures (see [1], pages 83 to 98). The present value of a call option to purchase a security at time t for a particular price K is given by

$$C = e^{-rt}E\left[(S_t - K)^+\right]$$

$$= e^{-rt}E\left[\left(S_0e^W - K\right)^+\right]$$

where $W \sim N\left((r - \sigma^2/2)t,\, \sigma^2 t\right)$. So

$$\frac{W - (r - \sigma^2/2)t}{\sigma\sqrt{t}} = Z \sim N(0,1)$$

and

$$W = \sigma\sqrt{t}Z + (r - \sigma^2/2)t.$$

Thus our original call option valuation is transformed to

$$C = e^{-rt}E\left[\left(S_0e^{\sigma\sqrt{t}Z + (r-\sigma^2/2)t} - K\right)^+\right]$$

$$= e^{-rt}\int_{-\infty}^{+\infty}\left(S_0e^{\sigma\sqrt{t}z + (r-\sigma^2/2)t} - K\right)^+ \frac{1}{\sqrt{2\pi}}e^{-z^2/2}dz.$$

We want to ensure the result in the parentheses remains positive, that is

$$S_0e^{\sigma\sqrt{t}z + (r-\sigma^2/2)t} \geq K$$

$$\sigma\sqrt{t}z + (r - \sigma^2/2)t \geq \log\left(\frac{K}{S_0}\right)$$

$$z \geq \frac{\log\left(\frac{K}{S_0}\right) + \sigma^2 t/2 - rt}{\sigma\sqrt{t}} = a.$$

To achieve this we remove the superscript "+" and C becomes

$$= e^{-rt}\int_a^{+\infty}\left(S_0 e^{\sigma\sqrt{t}z+(r-\sigma^2/2)t} - K\right)\frac{1}{\sqrt{2\pi}}e^{-z^2/2}dz$$

$$= \underbrace{e^{-rt}\int_a^{+\infty} S_0 e^{\sigma\sqrt{t}z+(r-\sigma^2/2)t}\frac{1}{\sqrt{2\pi}}e^{-z^2/2}dz}_{P_1} - \underbrace{e^{-rt}\int_a^{+\infty} K\frac{1}{\sqrt{2\pi}}e^{-z/2}dz}_{P_2}$$

$$= P_1 - P_2.$$

It now remains to show that $P_1 = S_0\Phi(\omega)$ and $P_2 = Ke^{-rt}\Phi(\omega - \sigma\sqrt{t})$. Let's consider, firstly P_2

$$P_2 = Ke^{-rt}P(Z \geq a) = Ke^{-rt}\underbrace{P(Z \leq -a)}_{\Phi(-a)}$$

and

$$-a = \frac{rt - \frac{\sigma^2 t}{2} - \log\left(\frac{K}{S_0}\right)}{\sigma\sqrt{t}}$$

$$= \omega - \sigma\sqrt{t}.$$

Thus the equivalency of P_2 and the second part of Eq (1.1) is proved. Showing that P_1 and the first part of the Black-Scholes formula are equivalent is a bit more complex. After some simplification we find that

$$P_1 = \frac{1}{\sqrt{2\pi}}\int_a^{+\infty} S_0 e^{\sigma\sqrt{t}z - \frac{\sigma^2 t}{2} - \frac{z^2}{2}}dz$$

and some inspection of the exponent reveals that $\sigma\sqrt{t}z - \sigma^2 t/2 - z^2/2 = -\frac{1}{2}(z - \sigma\sqrt{t})^2$. So

$$P_1 = S_0\int_a^{+\infty}\frac{1}{\sqrt{2\pi}}e^{-\frac{1}{2}(z-\sigma\sqrt{t})^2}dz$$

$$= P\left(Z \geq \underbrace{a - \sigma\sqrt{t}}_{-\omega}\right) = P(Z \leq \omega) = \Phi(\omega)$$

and thus,

$$P_1 = S_0\Phi(\omega). \tag{1.2}$$

Now that we have the Black-Scholes formula under our belts we may proceed to discuss the core topic of this thesis; Exotic Options.

1.1.4 Exotic Options

Exotic options are also called *path-dependent* options since their payoff at exercise time is dependent on the security's price path up to expiry time. For the purposes of this thesis we will be considering three such types of option, namely lookback, barrier and Asian options. We will consider contracts of duration T, and denote the maximum and minimum process of a security price process $S = \{S_t,\ 0 \leq t \leq T\}$ as

$$M_t^S = max\left\{S_u;\ 0 \leq u \leq t\right\} \ \ and\ m_t^S = min\left\{S_u;\ 0 \leq u \leq t\right\}, \qquad 0 \leq t \leq T.$$

1.1.4.1 Lookback Options

Using risk-neutral valuation, we have that the time $t = 0$ price of a minimal lookback call option is given by

$$LC = e^{-rT}E\left[S_T - m_T^S\right].$$

while the risk -neutral valuation of a maximal lookback call option is given by

$$LC = e^{-rT}E\left[M_T^S - K\right],$$

where K is the strike price of the option.

1.1.4.2 Barrier Options

We will look at the following types of single barrier options, as defined by [14]:

- A down-and-out barrier option is worthless unless its minimum remains above some "low barrier" v. If it remains "alive" then it retains the structure of a vanilla European call with strike K. Its initial price is given by

$$D_o = e^{-rT} E\left[(S_T - K)^+ 1\left(m_T^S > v\right)\right].$$

  $1(\alpha \leq (\geq)\beta)$ is an indicator function which equals 1 if the contained inequality is true over the entire specified ranges of alpha and beta and zero otherwise.

- A down-and-in barrier option, on the other-hand, only comes to life if its minimum value went below some "low barrier" v. If this barrier was never crossed then the option remains worthless. Its initial price is given by

$$D_i = e^{-rT} E\left[(S_T - K)^+ 1\left(m_T^S \leq v\right)\right].$$

- An up-and-in barrier option is worthless unless its maximum crossed some "high barrier" v, in which case it retains the structure of a vanilla European call with strike K. Its initial, risk-neutral geometric Brownian motion, price is given by

$$U_i = e^{-rT} E\left[(S_T - K)^+ 1\left(M_T^S \geq v\right)\right].$$

- An up-and-out barrier option becomes worthless when its maximum crosses some "high barrier" v. If it remains below this barrier it retains the structure of a vanilla European call with strike K. Its initial price is

$$U_o = e^{-rT} E\left[(S_T - K)^+ 1\left(M_T^S < v\right)\right].$$

Now, if one owns both a down-and-in and a down-and-out call option, both with the same values of K and t, then exactly one option will be in play at time t (the down-and-in option if the barrier is breached and the down-and-out otherwise); hence owning both is equivalent to owning a vanilla option with exercise time t and exercise price K [12]. As a result

$$D_i + D_o = C(s, t, K),$$

where $C(s, t, K)$ is the Black-Scholes valuation of the call option given by Eq (1.1). So determining either one of D_o or D_i automatically yields the other. The same holds true for up-and-in

and up-and-out options (with the same t and K)

$$U_i + U_o = C(s, t, K).$$

Note that for up-and-in and up-and-out options the barrier value v must be greater than the exercise price K. The above formulae assume continuous observation but for most applications the barrier is only considered breached if the *end-of-day* price is lower (in the case of "down" barrier options) than v; so if the price dips below v in the middle of the trading day the barrier is not considered breached.

1.1.4.3 Asian Options

Asian options are options whose value at the time t of exercise is dependent on the *average price* of the security over at least part of the time between 0 (when the option was purchased) and the time of exercise [12]. Again, the prices we are interested in are the end-of-day ones. Thus in the case of Asian options the averages are in terms of the end of day prices. Letting N denote the number of trading days in a year we let

$$S_d(i) = S_{i/N},$$

denote the security's price at the end of trading day i. Note that N is usually taken to be 252. The most common Asian type option, *floating price,* is the one in which exercise time is at the end of n trading days, the strike is K, and the payoff at exercise time is

$$\left(\sum_{i=1}^{n} \frac{S_d(i)}{n} - K \right)^+.$$

Yet another type of Asian option is the *floating strike* Asian option whose final value is given by

$$\left(S_d(n) - \sum_{i=1}^{n} \frac{S_d(i)}{n} \right)^+.$$

Where the exercise time is at the end of trading day n.

1.1.5 Simulation Techniques

Suppose we are given a random vector $\mathbf{X} = (X_1, ..., X_n)$ which has an associated density function $f(x_1, ..., x_n)$. In addition to this let us say we are interested in computing

$$E\left[g\left(\mathbf{X}\right)\right] = \int\int\int ... \int g(x_1, ..., x_n) f(x_1, ..., x_n) dx_1 dx_2 dx_3 ... dx_n,$$

for some n-dimensional function g [10]. Sometimes it may not be analytically possible to compute the above multiple integral exactly or even to numerically approximate it within a given accuracy using quadrature methods. A choice which remains is to approximate the above expectation using simulation techniques. A popular technique is that of *Monte-Carlo* simulation.

1.1.5.1 Monte-Carlo Simulation

To approximate $E\left[g\left(\mathbf{X}\right)\right]$, start by generating a random vector $\mathbf{X}^{(1)} = (X_1^{(1)}, ..., X_n^{(1)})$ having the joint density $f(x_1, ..., x_n)$ and then compute $Y^{(1)} = g\left(\mathbf{X}^{(1)}\right)$. Then generate a second, independent, random vector $\mathbf{X}^{(2)}$ and compute $Y^{(2)} = g\left(\mathbf{X}^{(2)}\right)$. 'Run' this step a fixed number of times, p, to generate independent and identically distributed random variables $Y^{(i)} = g\left(\mathbf{X}^{(i)}\right), i = 1, ..., p$. Now by the strong law of large numbers, we know that

$$\lim_{p\to\infty} \frac{Y^{(1)} + ... + Y^{(p)}}{p} = E\left[Y^{(i)}\right] = E\left[g(\mathbf{X})\right].$$

Thus we can use the average of the generated Y's as an estimate of $E\left[g(\mathbf{X})\right]$. This approach to estimating $E\left[g(\mathbf{X})\right]$ is known as *Monte Carlo simulation.*

1.1.6 Pricing Exotic Options using Simulation Methods

The use of simulation to price exotic options requires us to simulate the underlying security's price series, $\mathbf{X}^{(i)}$, and then use this to calculate the payoff function, $Y^{(i)} = g(\mathbf{X}^{(i)})$. See Section 1.1.5.1 for details.

1.1.6.1 Improving Simulation Estimator Efficiency

A number of ways of "improving" the implementation of Monte Carlo simulation valuations exist. These improvements result in our estimate being closer to the true value.

Control Variates

Consider our situation where we plan to use simulation to estimate X

$$\theta = E[Y].$$

Suppose that during the process of generating the value of the random variable Y, we also learn the value of some random variable V whose mean is known to be $\mu_v = E[V]$. Then instead of using just the value of Y as the estimator, we can use an estimator of the form

$$Y + c(V - \mu_v),$$

where c is a constant to be specified. This quantity also estimates θ since $E[Y + c(V - \mu_v)] = \theta$. The best estimator of this type is obtained by selecting c to be the value that makes $Var(Y + c(V - \mu_v))$ as small as possible. The value of c which minimizes $Var(Y + c(V - \mu_v))$ is

$$c^* = -\frac{Cov(Y, V)}{Var(V)}.$$

See [12] for a full derivation of the above. The variance reduction obtained when using the control variable V is $100Corr^2(Y, V)$ percent, where

$$Corr(Y, V) = \frac{Cov(Y, V)}{\sqrt{Var(Y)Var(V)}}.$$

So stronger correlations between V and Y result in greater variance reduction.

Antithetic Variables

When using this method one generates the data set $X_1, ..., X_n$ and uses it to compute Y. Then, rather than generate a second set of data, we re-use the same data with the following changes

$$X_i \Rightarrow \frac{2(r - \sigma^2/2)}{N} - X_i.$$

Where $\Rightarrow$means we assign to X_i the new value of $2(r - \sigma^2/2)/N$ minus it's old value, for each $i = 1, ..., n$. Our new value of X_i will now be negatively correlated with the old, but it will maintain normality with the same mean and variance. The value of Y based on these new values is computed , and the estimate from the simulation run is the average of the two Y values obtained [12]. It is shown in [11] that the re-use of data in this manner will result in a smaller variance than would be the case were we to generate a new set of data.

Conditional Expectation

In determining the risk-neutral payoff under geometric Brownian motion of a down-and-in barrier option we generate a series of $X_i s$ and use them to calculate the successive end-of-day prices

and resulting payoff from the option. In [12], Ross suggests we can improve on this approach by noting that for the option to come alive at least one end-of-day price must fall below the barrier. Suppose that with the generated data this barrier breach first occurs at the end of day j, with the price at the close of that day being $S_d(j) = x < v$. At this point there is time $(n-j)/N$ remaining before the option's expiry date. But this implies that the option's worth is now $C(x, (n-j)/N, K)$ [12]. We can now (i) end the simulation run once the barrier has been breached, and (ii) use the resulting Black-Scholes valuation as the estimate from this run. The resulting Black-Scholes estimator, called the *conditional expectation estimator,* can be shown to have a smaller variance than if we were to derive it following the method of Section 2.3 [12].

Chapter 2

Related Work

A number of methods have been proposed for the efficient pricing of exotic options using simulation. The basic concept is the same throughout: generate a price series and then use Monte Carlo techniques to price an option according the average of its payoffs based on the aforementioned price series. Where methods begin to differ is when it comes time to improving the *efficiency* of the estimators. The following chapter explores both simulation pricing and improving estimator efficiency according to three sources (see references [8, 12, 13]).

2.1 Odegaard vanilla option pricing by simulation

In [8], Odegaard suggests that we begin by first considering the pricing of the "plain" European call option. This is mainly for illustrative purposes since there already exists a closed form solution to this problem in the guise of the Black-Scholes equation. As stated in Section 1.1.2 at maturity a call option is worth

$$C = (S_T - K)^+ = max(0, S_T - K),$$

where S_T and K represent the terminal price of our stock and the strike price of the option, respectively. During some earlier time t the option will have the expected present value of the calls worth. Because of "risk neutrality" we can treat the (appropriately adjusted) problem as the decision of a risk neutral decision maker and if in addition we modify the expected return of the underlying asset such that this earns at the risk free rate then the time t value of our option will be

$$c_t = e^{-r(T-t)} E^* \left[max(0, S_T - K) \right],$$

Algorithm 1 Simulation of lognormally distributed random variable

```
#include <cmath> // standard mathematical functions
using namespace std;
#include "normdist.h" // definition of random number generator

double simulate_lognormal_random_variable(const double& S, // current value of variable
                    const double& r, // interest rate
                    const double& sigma, // volatility
                    const double& time) { // time to final date
  double R = (r - 0.5*pow(sigma,2))*time;
  double SD = sigma*sqrt(time);
  return S*exp(R + SD*random_normal());
};
```

where $E^*[.]$ is a transformation of the original expectation [8] with μ replaced by $r - \sigma^2/2$. Finding the price merely involves using Monte Carlo techniques on a number of simulated S_T values.

2.1.1 Simulating lognormally distributed random variables

We assumed from the beginning that our stock prices are lognormal, so we must be able to generate the appropriate random variable. Let $\widetilde{X}$ be normally distributed with a mean of zero and variance of one. If S_t conforms to the lognormal distribution then S_{t+1}, the one-time period later price, is simulated as

$$S_{t+1} = S_t e^{\left(r-\frac{1}{2}\sigma^2\right)+\sigma\widetilde{X}}.$$

More generally, if we are at time t and the terminal time is T (with time between of $T-t$) then

$$S_T = S_t e^{\left(r-\frac{1}{2}\sigma^2\right)(T-t)+\sigma\sqrt{T-t}\widetilde{X}}.$$

Odegaard provides the C++ code for simulating such a random variable, this code is presented in Algorithm 1.

2.1.2 Pricing of European Call options

In performing the Monte Carlo estimation of the price of a European call option one only needs to simulate the closing price of the underlying stock. We proceed by simulating lognormally distributed random variables, which gives us a set of observations of the terminal price S_T.

Algorithm 2 European Call Option Priced by Simulation

```
#include <cmath> // standard mathematical functions
#include <algorithm> //define max() function
using namespace std;
#include "normdist.h" // definition of random number generator

double
option_price_call_european_simulate(const double& S, // current value of stock
                    const double& K, // exercise price
                    const double& r, // interest rate
                    const double& sigma, // volatility
                    const double& time, // time to final date
                    const int& no_sims) { //number of simulations
  double R = (r - 0.5*pow(sigma,2))*time;
  double SD = sigma*sqrt(time);
  double sum_payoffs = 0.0;
  for (int n=1; n<=no_sims; n++) {
    double S_T = S*exp(R + SD*random_normal());
    sum_payoffs += max(0.0, S_T-K);
  };
  return exp(-r*time)*(sum_payoffs/double(no_sims));
};
```

Letting $S_{T,1}, S_{T,2}, \ldots, S_{T,n}$ denote the n simulated values, we can simulate $E^*\left[max(0, S_T - K)\right]$ as the average of option payoffs at maturity, discounted at the risk free rate as

$$\hat{c} = e^{-r(T-t)} \left(\sum_{i=1}^{n} max(0, S_{T,i} - K)/n \right).$$

Odegaard's C++ implementation of Monte Carlo estimation to price a European call option is shown in Algorithm 2.

2.1.3 Improving the efficiency of simulation

There are a number of ways of bettering the implementation of Monte Carlo estimation such that the estimates are closer to the true value. Odegaard [8] suggests the use of *Control* and *Antithetic* variates, both of which are discussed in Section 1.1.6.1 above.

Control Variates

Odegaard [8] recommends that when generating the set of terminal values of the underlying security you should also calculate the value of some other derivative which would use those same terminal values but which has a closed form solution. An at-the-money European call option priced using the analytical Black-Scholes formula would fulfill this end. If it turns out that our simulated value over-estimates the option price, then Odegaard suggests that it is reasonable to assume that this will also be the case for other derivatives valued using the same set of simulated terminal values. Thus we move the estimate of the price of the derivative of interest downward.

Suppose we want to value an European put option and use the price of an at-the-money European call as the control variate. We firstly estimate the price of two options using the same set of terminal values and Monte Carlo estimation

$$\hat{p}_t = e^{-r(T-t)} \left(\sum_{i=1}^{n} max(0, K - S_{T,i})/n \right)$$

and

$$\hat{c}_t = e^{-r(T-t)} \left(\sum_{i=1}^{n} max(0, S_{T,i} - K)/n \right).$$

We then calculate the Black Scholes value of the call c_t^{bs} and calculate p_t^{cv}, the estimate of the put price with a control variate adjustment, as

$$p_t^{cv} = \hat{p}_t + \left(c_t^{bs} - \hat{c}_t \right).$$

It is not only an at-the-money call which can be used as the control variate. Any other derivative which possesses a tractable analytical solution may be used. The code contained in Algorithm 3 shows the implementation of the above method.

Antithetic Variates

The other method suggested by Odegaard [8] is to use *antithetic* variates, the whole idea being that Monte Carlo works most efficiently if the simulated variables are "spread" out as closely as possible to the true distribution. In our case we are generating unit normal random variables which possess the trait of being symmetric about zero. Odegaard [8] suggests enforcing this property in our simulated terminal values. A workable way of doing this is to first simulate a unit random variable Z and then use both Z and $-Z$ to generate the lognormal random variables. The idea behind all this is shown in Algorithm 4.

Algorithm 3 Generic Monte-Carlo Pricing with Control Variates

```
#include <cmath> // standard mathematical functions
using namespace std;
#include "fin_recipes.h"
#include "payoff_black_scholes_case.h"

double
derivative_price_simulate_european_option_generic_with_control_variate(const double& S, //
current value of variable
                                const double& K, // exercise price
                                const double& r, // interest rate
                                const double& sigma, // volatility
                                const double& time, // time to final date
                                 double payoff(const double& S,
                                        const double& K),
                                const int& no_sims) { //number of simulations
  double c_bs = option_price_call_black_scholes(S,S,r,sigma,time); // price an at-the-money
derivative via Black Scholes
  double sum_payoffs = 0;
  double sum_payoffs_bs = 0;
  for (int n=0; n<no_sims; n++) {
    double S_T = simulate_lognormal_random_variable(S,r,sigma,time);
    sum_payoffs += payoff(S_T,K);
    sum_payoff_bs += payoff_call(S_T,S); //simulate at-the-money Black Scholes price
  };
  double c_sim = exp(-r*time)*(sum_payoffs/no_sims);
  double c_bs_sim = exp(-r*time)*(sum_payoffs_bs/no_sims);
  c_sim += (c_bs - c_bs_sim);
  return c_sim;
};
```

Algorithm 4 Generic Monte-Carlo Pricing with Antithetic Variates

```
#include "fin_recipes.h"
#include <cmath> // standard mathematical functions
#include "normdist.h" // definition of random number generator
using namespace std;

double
derivative_price_simulate_european_option_generic_with_antithetic_variate(const double& S,
// current value of variable
                        const double& K, // exercise price
                        const double& r, // interest rate
                        const double& sigma, // volatility
                        const double& time, // time to final date
                        const payoff(const double& S, const double& K),
                        const int& no_sims) { //number of simulations
  double R = (r - 0.5*pow(sigma,2))*time;
  double SD = sigma*sqrt(time);
  double sum_payoffs = 0.0;
  for (int n=0; n<no_sims; n++) {
    double x=random_normal();
    double S1 = S*exp(R + SD*x);
    sum_payoffs += payoff(S1,K);
    double S2 = S*exp(R + SD*(-x));
    sum_payoffs += payoff(S2,K)
  };
  return exp(-r*time)*(sum_payoffs/(2*no_sims));
};
```

2.1.4 Odegaard Results

Using the following initial values:

- S = 100.00
- K = 100.00
- r = 0.1
- sigma = 0.25
- time = 1.0
- no_sims = 50 000
- CV = control variate variance reduction
- AV = antithetic variate variance reduction

Odegaard states that **his own** programs produced the following results (with no mention of the standard errors) as output:

BLACK SCHOLES CALL OPTION PRICE = 14.9758
SIMULATED CALL OPTION PRICE = 14.995
SIMULATED CALL OPTION PRICE, CV = 14.9758
SIMULATED CALL OPTION PRICE, AV = 14.9919

BLACK SCHOLES PUT OPTION PRICE = 5.45954
SIMULATED PUT OPTION PRICE = 5.41861
SIMULATED PUT OPTION PRICE, CV = 5.42541
SIMULATED PUT OPTION PRICE, AV = 5.46043

2.2 Odegaard Monte-Carlo Pricing of options whose payoff depends on the whole price path

According to Odegaard [8], Monte Carlo simulation can be used to price a number of different types of options, the only limitation being that they must be European in nature. In the previous

section we described methods to price regular European options. When dealing with options whose payoff is dependent on the entire path of the price of the underlying security then it is not sufficient to only simulate the *terminal* price. We must incorporate the entire *sequence* of prices in our pricing procedure.

2.2.1 Simulating a *series* of lognormally distributed variables

Recall from Section 2.1.1 that we simulate a lognormal variable as follows

$$S_T = S_t e^{\left(r-\frac{1}{2}\sigma^2\right)(T-t)+\sigma\sqrt{T-t}\tilde{X}},$$

where t is the present time and T is the terminal date. To simulate a price sequence one must split this period into a series of N periods. Each period of length

$$\Delta t = \frac{T-t}{N}.$$

The C++ code in Algorithm 5 shows how one would simulate a sequence of such lognormal variables.

The above mentioned code may then be used in a **generic** routine to price *path-dependent* European options as shown in Algorithm 6.

In Algorithm 6, the method *payoff()* could be replaced by any of a number of payoff functions which accept a vector of numbers (price series) and some strike price K. Algorithm 7 shows a few methods which could fill the payoff function slot.

Control Variates

We can again utilise the Control Variate technique mentioned in Section 1.1.6.1 to improve our estimator. Just to recap, a control variate is a price for which we have both an analytical solution and find the Monte Carlo price. The difference between the two prices is a measure of the bias in the Monte Carlo estimate and is used to adjust the Monte Carlo estimate of other derivatives priced using the same random sequence [8]. We could use the analytical lookback price or the analytical solution for a *geometric* Asian option but Odegaard opts to again utilise the Black Scholes call price as a control variate (see Algorithm 8).

Algorithm 5 Simulating a sequence of lognormally distributed variables

```
#include <cmath> // standard mathematical functions
#include <vector> // vector functions
using namespace std;
#include "random.h"
#include "normdist.h" // definition of random number generator

vector<double>
simulate_lognormally_distributed_sequence(const double& S, // current value of variable
                        const double& r, // interest rate
                        const double& sigma, // volatility
                        const double& time, // time to final date
                        const int& no_steps) { //number of steps in series
  vector<double> prices(no_steps);
  double delta_t = time/no_steps;
  double R = (r - 0.5*pow(sigma,2))*delta_t;
  double SD = sigma*sqrt(delta_t);
  double S_t = S;
  for(int i=0; i<no_steps; i++) {
    S_t = S_t*exp(R + SD*random_normal());
    prices[i] = S_t;
  };
  return prices;
};
```

Algorithm 6 Generic routine for pricing path-dependent European options

```
#include <cmath> // standard mathematical functions
using namespace std;

double
derivative_price_simulate_european_option_generic(const double& S, // current value of variable
                    const double& K, // exercise price
                    const double& r, // interest rate
                    const double& sigma, // volatility
                    const double& time, // time to final date
                    double payoff(const vector<double>& prices,
                          const double& K), // user provided function
                    const int& no_steps, //number of steps in generated price sequence
                    const int& no_sims) { //number of simulations
  double sum_payoffs = 0;
  for (int n=1; n<=no_sims; n++) {
    vector<double>prices = simulate_lognormally_distributed_sequence(S,r,sigma,time,no_steps);
     sum_payoffs += payoff(prices,K);
  };
  return exp(-r*time)*(sum_payoffs/double(no_sims));
};
```

Algorithm 7 Payoff function for Arithmetic Asian, lookback call and lookback put options

```
#include <cmath>
#include <numeric>
#include <vector>
using namespace std;

double payoff_arithmetic_average_call(const vector<double>& prices, const double& K) {
  double sum = accumulate(prices.begin(), prices.end(),0.0);
  double avg = sum/prices.size();
  return max(0.0,avg - K);
};

inline double payoff_geometric_average_call(const vector<double>& prices, const double&
K) {
  double logsum = log(prices[0]);
  for (unsigned i=1; i<prices.size(); i++){ logsum += log(prices[i]); };
  double avg = exp(logsum/prices.size());
  return max(0.0,avg - K);
};

double payoff_lookback_call(const vector<double>& prices, const double& unused_variable)
{
  double m = *min_element(prices.begin(), prices.end());
  return prices.back() - m; //always positive or zero
};

double payoff_lookback_put(const vector<double>& prices, const double& unused_variable)
{
  double m = *max_element(prices.begin(), prices.end());
  return m - prices.back(); //max always larger or zero
```

Algorithm 8 Generic Exotic Monte-Carlo Pricing with Control Variates

```
#include <cmath> // standard mathematical functions
using namespace std;
#include "payoff_black_scholes_case.h"

double
derivative_price_sim_exotic_european_option_generic_with_control_variate(const double& S,
// current value of variable
                                const double& K, // exercise price
                                const double& r, // interest rate
                                const double& sigma, // volatility
                                const double& time, // time to final date
                                 double payoff(const vector<double>& prices,
                                        const double& K),
                                const int& no_steps,
                                const int& no_sims) { //number of simulations
   double c_bs = option_price_call_black_scholes(S,S,r,sigma,time); // price an at-the-money
derivative via Black Scholes
  double sum_payoffs = 0;
  double sum_payoffs_bs = 0;
  for (int n=0; n<no_sims; n++) {
    vector<double> prices = simulate_lognormally_distributed_sequence(S,r,sigma,time,no_steps);
     double S1 = prices.back();
     sum_payoffs += payoff(prices,K);
     sum_payoff_bs += payoff_call(S1,S); // simulate at-the-money Black Scholes price
  };
  double c_sim = exp(-r*time)*(sum_payoffs/no_sims);
  double c_bs_sim = exp(-r*time)*(sum_payoffs_bs/no_sims);
  c_sim += (c_bs - c_bs_sim);
  return c_sim;
};
```

2.3 Ross/Shanthikumar Efficient simulation of Barrier options

Again, suppose that our security follows risk-neutral geometric Brownian motion with nominal interest rate r; that is it follows geometric Brownian motion with variance parameter σ^2 and drift parameter μ, where $\mu = r - \sigma^2/2$. Let $S_d(i)$ denote the end of day i price of the security and let

$$X(i) = \log\left(\frac{S_d(i)}{S_d(i-1)}\right).$$

Successive daily price ratio changes are independent under geometric Brownian motion, and thus it follows that $X(1), ..., X(n)$ are independent normal random variables, each with mean μ/N and variance σ^2/N where N again denotes the number of trading days in a year. So by generating the values of n independent normal random variables having the above mean and variance, we can construct a sequence of n end-of-day prices which have the same probabilities as one which would have developed from the risk-neutral geometric Brownian motion model [13].

So if it were the case that we wanted to find the risk-neutral valuation of a down-and-out barrier option whose strike price is K, barrier value is v, initial value is $S(0) = s$, and exercise time is at the end of trading day n, Ross and Shanthikumar suggest we proceed as follows. Firstly, generate n independent normal random variables with mean μ/N and variance σ^2/N. Set them equal to $X(1), ..., X(n)$, and then determine the sequence of end-of-day prices from the equations

$$S_d(0) = s,$$

$$S_d(1) = S_d(0)e^{X(1)},$$

$$S_d(2) = S_d(1)e^{X(2)},$$

$$\vdots$$

$$S_d(i) = S_d(i-1)e^{X(i)}$$

$$\vdots$$

$$S_d(n) = S_d(n-1)e^{X(n)}.$$

In terms of these prices, let I equal 0 if an end-of-day price is ever below the barrier v, and let it equal 1 otherwise; that is

$$I = \begin{cases} 1 \; if \; S_d(i) > v \quad for \; all \; i = 1, ..., n \\ 0 \; if \; S_d(i) \leq v \quad for \; any \; i = 1, ..., n \end{cases}.$$

Then since the down-and-out call option will only be alive if $I = 1$, it follows that the time-0 value of its payoff at expiration time n is

$$\text{payoff of the down-and-out call option} = e^{-rn/N} I \left(S_d(n) - K\right)^+.$$

We would then call this payoff Y_1. Performing an additional $k-1$ 'runs' yields $Y_1, ..., Y_k$, a set of k payoff realizations. We can then use their average as an estimate of the risk-neutral geometric Brownian motion valuation of the barrier option. If we wanted to price a down-and-in call option everything would be exactly the same except that our indicator I would now be defined as

$$I = \begin{cases} 1 \; if \; S_d(i) < v \quad for \; some \; i = 1, ..., n \\ 0 \; if \; S_d(i) \geq v \quad for \; all \; i = 1, ..., n \end{cases}. \tag{2.1}$$

Ross and Shanthikumar go a step further in [13] and for our down-and-in call, let $\mathbf{X_i} = (X(1), \ldots, X(i))$ and let f be the joint density function of $\mathbf{X_n}$. Ross and Shanthikumar [13] go on to say that if g is another joint density, then the importance sampling identity yields that

$$E_f[I(S_d(n) - K)^+] = E_g\left[\frac{If(\mathbf{X_n})\left(S_d(n) - K\right)^+}{g(\mathbf{X_n})}\right].$$

Let T equal $n+1$ if the option never becomes alive, and let it equal i if the option first comes to life at the end of day i. Then, for $T \leq n$,

$$E_g\left[\frac{If(\mathbf{X_n})\left(S_d(n) - K\right)^+}{g(\mathbf{X_n})} \,|T, \mathbf{X_T}\right]$$

$$= \frac{f(\mathbf{X_T})}{g(\mathbf{X_T})} E_g\left[\frac{f(X_{T+1}, \ldots\ldots, X_n)}{g(X_{T+1}, \ldots\ldots, X_n)}(S_d(n) - K)^+ \,|T, \mathbf{X}_T\right]$$

$$= \frac{f(\mathbf{X_T})}{g(\mathbf{X_T})} E_f\left[(S_d(n) - K)^+ \,|T, \mathbf{X}_T\right]$$

$$= \frac{f(\mathbf{X_T})}{g(\mathbf{X_T})} e^{-rn/N} C\left(S_d(T), (n-T)/N, K\right).$$

Defining $C(s,t,K)$ to equal zero when $t < 0$ the preceding also holds when $T = n+1$. Hence, combining importance sampling and conditional expectation, the X_i can be generated according to a density that makes it more likely that the barrier is crossed. Once the barrier is crossed, that simulation run ends with the following estimator for the risk neutral price

$$\frac{f(\mathbf{X_T})}{g(\mathbf{X_T})} e^{-rn/N} C\left(S_d(T), (n-T)/N, K\right). \tag{2.2}$$

If we generate the X_i as normal random variables with mean $(r - \sigma^2/2)/N - b$ and variance σ^2/N, then the estimator from that run is

$$C\left(S_d(T), (n-T)/N, K\right) exp\left\{\frac{Tb^2N}{2\sigma^2} + \frac{Nb}{\sigma^2}\sum_{i=1}^{T} X_i - \frac{Tb}{\sigma^2}\left(r - \frac{\sigma^2}{2}\right)\right\}. \tag{2.3}$$

Implementation requires an appropriate choice of b, which can be arrived at empirically. However, in an importance sampling application that did not utilize the conditional expectation improvement, it was noted in [3] that the choice

$$b = \frac{(\mu - \sigma^2/2)}{N} - \frac{2\log(S(0)/v) + \log(K/S(0))}{n}$$

works well.

Ross and Shanthikumar also remark that variance reduction by conditional expectation and by importance sampling were both suggested in [2] but that they could be simultaneously used was not. Combining variance reduction techniques will be explored at the experimental stage. The estimator in Equation (2.3) has a smaller variance than the importance sampling estimator suggested in [3] and also requires less simulation time [13].

2.4 Ross/Shanthikumar Efficient simulation of Asian and Look-back options

We consider an Asian option with a floating strike price which is the average of the underlying security's end-of-day prices; that is, if the option matures at the end of n days, then the present

value of it's payoff is

$$Pe^{-rn/N} \equiv e^{-rn/N}\left(S_d(n) - \sum_{i=1}^{n}\frac{S_d(i)}{n}\right)^+$$
$$= \frac{n-1}{n}e^{-rn/N}\left(S_d(n) - \frac{\sum_{i=1}^{n-1}S_d(i)}{n-1}\right)^+.$$

Ross and Shathikumar suggest that when estimating $E\left[Pe^{-rn/N}\right]$, we first condition on the data values $\mathbf{X}_{n-1} = (X(1), \ldots, X(n-1))$ to obtain

$$E\left[Pe^{-rn/N}|\mathbf{X}_{n-1}\right] = \frac{n-1}{n}e^{-rn/N}E\left[\left(S_d(n) - \frac{\sum_{i=1}^{n-1}S_d(i)}{n-1}\right)^+|\mathbf{X}_{n-1}\right]$$
$$= \frac{n-1}{n}e^{-rn/N}E\left[\left(S_d(n-1)e^{X(n)} - \frac{\sum_{i=1}^{n-1}S_d(i)}{n-1}\right)^+|\mathbf{X}_{n-1}\right]$$
$$= \frac{n-1}{n}e^{-rn/N}C\left(S_d(n-1), \frac{1}{N}, \sum_{i=1}^{n-1}\frac{S_d(i)}{n-1}\right). \quad (2.4)$$

Hence, we can estimate $E\left[e^{-rn/N}P\right]$ by generating $\mathbf{X}_{n-1}$ to obtain $S_d(1), \ldots, S_d(n-1)$, and then using the estimator given by Equation (2.4), where $C(s,t,K)$ is the Black-Scholes risk neutral call option valuation [13].

Ross and Shanthikumar note that this estimator can be improved by noting firstly that

$$C\left(S_d(n-1), \frac{1}{N}, \sum_{i=1}^{n-1}\frac{S_d(i)}{n-1}\right) \approx \left(S_d(n-1) - \sum_{i=1}^{n-1}\frac{S_d(i)}{n-1}\right)^+.$$

As a simulation run consists of generating $X(1), \ldots, X(n-1)$, independent normal random variables with mean $(r - \sigma^2/2)/N$ and variance σ^2/N and then setting

$$S_d(i) = S(0)e^{X(1)+\ldots+X(i)}, \quad i = 1, \ldots, n-1,$$

it follows that $C\left(S_d(n-1), 1/N, \sum_{i=1}^{n-1}S_d(i)/(n-1)\right)$ will be large if the latter values of the series of normal random variables $X(1), X(2), \ldots, X(n-1)$ are amongst the largest, and small if the opposite is true. Because of this Ross and Shanthikumar suggest that one could try a control variable (see Section 1.1.6.1) of the type $\sum_{i=1}^{n-1}w_iX(i)$, where the weights w_i are increasing in i. They ([13]) however propose a better approach, which is to let the simulation itself determine

the weights, by using all of the variables $X(1), X(2), \ldots, X(n-1)$ as the control variables, so that from each run we consider the estimator

$$C\left(S_d(n-1), \frac{1}{N}, \sum_{i=1}^{n-1} \frac{S_d(i)}{n-1}\right) + \sum_{i=1}^{n-1} c_i \left(X(i) - \frac{(r-\sigma^2/2)}{N}\right).$$

$c_1, c_2, \ldots, c_{n-1}$ the values of the constants, can be calculated from the simulation runs (see [9]). An important technical point is that because the suggested control variables are independent random variables there is not much additional computation needed to determine the values of the c_i [13].

An alternative suggested by Ross and Shanthikumar is to use $e^{X(i)}$, $i = 1, \ldots, n$, rather than the $X(i)$ themselves, as control variates.

2.5 Summary

The common thread in all the related works is the emphasis on the importance of the data generation stage. The foundation of Odegaard's [8] programs is the generation of the lognormal variable(s). Ross and Shanthikumar pay homage to the significance of simulating appropriate variables when they discuss efficient pricing techniques and variance reduction; both of which in the theoretical sense rely heavily on model assumptions. The next chapter will show how we intend to use the information from the related works in making our simulation research as painless and efficient as possible.

Chapter 3

Methodology

This project comprises three pillars. Firstly we must deal with the experimental considerations involved in efficient simulation of lognormal data sets. Secondly, we must subsequently use the data sets in an efficient pricing program. And finally we must work toward improving the efficiency of the data produced by our simulating tool.

3.1 Pillar I - Generating the data

The simulation of the data set (our security's price path) warrants a chapter on its own and is addressed in Chapter 4. What can be said now is that we must be able to generate a ***series*** of prices of a security which conforms to specific lognormal constraints i.e. $N(\mu/N, \sigma^2/N)$ $where$ $\mu = r - \sigma^2/2$. The data produced must be tested to make sure it is indeed geometric Brownian motion. Tests for combined normality **and** independence can become involved with no guarantee that we would arrive at any definite conclusions. Undertaking these specific tests would take us out of the scope of this paper. Luckily there is a knock on effect which will allow us, at a later stage, to see whether our generated normal random variables are conforming to the desired distributions behaviour. Initially though we can do a superficial analysis of the data produced by our random number generator to see if on the surface it at least "looks" normal, then from there we can proceed to perform the necessary transformations to produce random daily price changes.

3.2 Pillar II - The pricing program

The use of simulation to price financial instruments is an intricate task and requires a tool which allows one to deal with these intricacies. We have chosen to use C++ for our pricing program

and there are a number of supporting reasons. The next sections will describe why and how we intend to use C++ to efficiently price exotic options.

3.2.1 Why C++

It is our intention that the source code and resulting programs be used in the future when teaching courses on exotic pricing using simulation. Because of this it is important that we use a language that will still be relevant in 5 years time. Given that C++ is the basis for many modern languages e.g. Java, Perl and Python it is very likely that it is going to be in use for some time to come. C++ also provides good cross-platform portability, so whilst I will be building my programs on a Windows XP Operating System using Dev C++, one would be able to implement the code on a Linux machine with a compiler and platform different from the one I have used with little or no code tweaking!

Many high level languages are able to compile and even run C++ code, as well as translate their own code into C++. An example of this is Matlab which is capable of translating resident m-files (Matlab source code files) into .cpp files (C++ source code files). C++ also has advantages over low level languages, for example in the case of C++ and C:

- C++ allows for many extensions to the original C language,
- C++ bridges the gap between the low-level C and higher level languages.

C++ offers the following features not found in Java:

- Pointers
- Global variables
- Multiple Inheritance
- Templates
- Operator Overloading

More importantly, it is much faster than Java and most other programming languages! From the above list we will only be using the pointer feature, this is to keep things simple and not muddy the waters with what is potentially very involved C++ theory. It is left to the curious financial programmer to see how to use any of the other features to his/her advantage when pricing financial instruments.

Now the main aim of this thesis is to build a functioning and accurate pricing program **from scratch**. This means that we will be responsible for generating the normal random variables which will be used to build geometric Brownian motion stock price movements that will in turn be used in various exotic option functions. So, given that we will be coming in on the ground floor it is important that we use a programming language which possesses speed and efficiency (in terms of the use of computer resources). C++ has been proven to provide both.

3.2.2 How C++

We intend to build a command line program which will request information from the user such as initial stock price, stock volatility, interest rate, strike price and type of option. The user provided information is then used to build a sequence of price movements and this, along with other option pertinent information, is sent to the respective payoff function which returns a payoff value based on the values of the parameters given to it. The average of these payoffs is used as the price of the option.

The program will be highly modularized so that if it turns out there is a more efficient method for producing normal random variables or pricing lookback options then that module can simply be slotted into place.

The **cost** of a numerical procedures simulation is an important consideration but equally important is the **accuracy** of the method. Because different sources quote different quantities for the sufficient number of runs required to produce trustworthy output (the industry standard being 10000). We will have to use our own discretion and experiment to find what number of simulations give us the necessary combination of speed and accuracy. Gentle et al. state, *"In general, compromises must be made between simplicity of the algorithm, quality of the approximation, robustness with respect to the distribution parameters, and efficiency (generation speed, memory requirements, and setup time)"[6].*

3.2.3 Pricing Exotic European options using C++

C++ through the use of Monte Carlo simulation will use simulated price paths and associated payoff functions to arrive at the risk neutral price of various exotic options. The results of this will be compared to the results of Sections 2.1.4. This procedure will enable us to test **both** C++ as a simulator and the code we have written. Our program should be versatile and adaptable, in that brand new European-type exotic options should be almost immediately priceable.

3.3 Pillar III - Improve simulator efficiency

We may be willing to sacrifice the speed of our price generation for a gain in efficiency obtained via variance reduction methods. Increased reliability may more than compensate slower price generation. In the case of antithetic variates (see Section 1.1.6.1) we can expect to see longer simulation times because whilst new random variables will not have to be produced we will be calculating and averaging twice as many simulated payoffs. With conditional expectation (see Section 2.3) on the other hand, because the simulation is ended early the simulation process can be rightfully expected to not take as long.

We will also attempt to compare the effects of **combining** different variance reduction techniques to see if the combinatorial approach has any significant, positive repercussions on the accuracy of the simulated price.

3.4 Summary

To summarize we will be using C++ to produce a price sequence which will be subsequently used in our pricing modules. Upon getting satisfactory results we will explore the feasibility of incorporating variance reduction techniques alone and in combination with one another. In the following chapter we explore the first pillar: Generating the data.

Chapter 4

The Data Set(s)

In this statistical thesis we are not dealing with some predetermined data set and analyzing it in the hopes of unveiling some exciting, underlying information. In this project it is left to us to generate our own data set which must conform to the assumptions made when dealing with financial market data. More precisely we must simulate a security whose price evolves according to risk neutral geometric Brownian motion (see Section 1.1.1). This means that at some point we must be able to produce independent normal random variables.

4.1 Generating Continuous Random Variables

There are a number of techniques available for generating variables which follow the behavioural characteristics of certain continuous distributions. *"In computational statistics, random variate generation is usually made in two steps: (1) generating imitations of independent and identically distributed (i.i.d.) random variables having the uniform distribution over the interval (0,1) and (2) applying transformations to these i.i.d. U(0,1) random variates in order to generate (or imitate) random variates and random vectors from arbitrary distributions"* [6]. We will outline two methods, the Rejection and Polar methods.

4.1.1 The Rejection Method

If we have a method for generating a random variable having density function $g(x)$, we can use this as the foundation for generating from the continuous distribution having density function $f(x)$. We do this by firstly generating Y from g and then accepting this generated value with a

probability proportional to $f(Y)/g(Y)$. More precisely, let c be a constant such that

$$\frac{f(y)}{g(y)} \leq c \quad for\ all\ y.$$

Ross in [11] proposes the following technique for generating a random variable having density f.

THE REJECTION METHOD

1. Generate Y having density g.
2. Generate a random number U.
3. If $U \leq \frac{f(Y)}{cg(Y)}$, set $X = Y$. Otherwise, return to 1.

Thus Y is only accepted with probability $f(Y)/cg(Y)$ when our generated random U adheres to the following; $U \leq f(Y)/cg(Y)$. The following theorem can easily be proved (see [11] for details):

THEOREM

1. *The random variable generated by the rejection method has density f.*
2. *The number of iterations of the algorithm that are needed is a geometric random variable with mean c.*

4.1.1.1 Generating a Normal Random Variable using the Rejection Method

To generate Z a standard normal variable (i.e. one with mean 0 and variance 1), we firstly note that the absolute value of Z, |Z|, has a probability density function

$$f(x) = \frac{2}{\sqrt{2\pi}} e^{-x^2/2} \quad 0 < x < \infty. \tag{4.1}$$

Given that we are going to be using the rejection technique we define our known density function g, to be an exponential density function with mean 1

$$g(x) = e^{-x} \quad 0 < x < \infty.$$

So

$$\frac{f(x)}{g(x)} = \sqrt{2/\pi}e^{x-x^2/2},$$

the maximum of the above occurs at $x = 1$, so we take

$$c = \max \frac{f(x)}{g(x)} = \sqrt{2e/\pi}.$$

And because

$$\frac{f(x)}{cg(x)} = exp\left\{x - \frac{x^2}{2} - \frac{1}{2}\right\}$$
$$= exp\left\{-\frac{(x-1)^2}{2}\right\}$$

we can generate the **absolute** value of a standard normal random variable as follows:

1. Generate Y, an exponential random variable with rate 1.
2. Generate a random number U.
3. If $U \leq exp\{-(Y-1)^2/2\}$, set $X = Y$. Otherwise return to 1.

We can obtain a standard normal Z by letting Z be equally likely to be X or $-X$.

In step 3 of the above, the value of Y is accepted if $U \leq exp\{-(Y-1)^2/2\}$, which is the same as $-\log U \geq (Y-1)^2/2$, but it can be shown (see Chapter 5 of [11]) that $-\log U$ is exponential with mean 1. So we can generate an independent random normal variable as follows:

1. Generate Y_1 an exponential variable with mean 1.
2. Generate Y_2 an exponential with mean 1.
3. If $Y = Y_2 - (Y_1 - 1)^2 > 0$, go to step 4. Otherwise return to step 1.
4. Generate a random number U and set

$$Z = \begin{cases} Y_1 & if\ U \leq \frac{1}{2} \\ -Y_1 & if\ U > \frac{1}{2} \end{cases}$$

The random variables Z and Y generated by the foregoing are independent with Z being a normal with mean 0 and variance 1 and Y being exponential with rate 1 [11]. To transform our standard

normal variable into a normal random variable with mean μ and variance σ^2 we simply take $\mu + \sigma Z$. Since $c = \sqrt{2e/\pi} \approx 1.32$, the forgoing requires a geometric distributed number of iterations of step 2 with mean 1.32. In our case we want to generate a sequence of normal random variables, so we can use the exponential random variable Y of step 3 as the initial exponential (Y_1) needed in step 1 for the next iteration. Thus on average we can simulate a standard normal by generating 1.64 exponentials and computing 1.32 squares.

4.1.2 The Polar Method for Generating Normal Random Variables

Letting X and Y be independent standard normal random variables we can let R and θ denote the polar coordinates of the vector (X, Y) such that

$$R^2 = X^2 + Y^2$$

$$\tan\theta = \frac{Y}{X}.$$

Since X and Y are independent of each other, their joint density is the product of their individual densities given by

$$f(x,y) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2} \frac{1}{\sqrt{2\pi}} e^{-y^2/2}$$

$$= \frac{1}{\sqrt{2\pi}} e^{-(x^2+y^2)/2}. \tag{4.2}$$

In determining the joint density of R^2 and Θ, let's call it $g(d,\theta)$, we make the change of variables

$$d = x^2 + y^2, \qquad \theta = \tan^{-1}\left(\frac{y}{x}\right).$$

Given the Jacobian of this transformation is 2 it follows from Equation (4.2) that the joint density of R^2 and Θ is given by

$$g(d,\theta) = \frac{1}{2}\frac{1}{2\pi} e^{-d/2}, \quad 0 < d < \infty, \ 0 < \theta < 2\pi.$$

Since this is equal to the product of an exponential density with mean 2 and the uniform density on $(0, 2\pi)$ it follows R^2 and Θ are independent with

$$\begin{array}{c} R^2 \sim \textit{exponential with mean } 2 \\ \Theta \sim \textit{uniform distribution over } (0, 2\pi) \end{array}. \tag{4.3}$$

We can now generate a pair of independent standard normal variables X and Y using Equation (4.3) to first generate their polar coordinates and then transform those to rectangular coordinates. Ross outlines the following steps:

1. Generate random numbers U_1 and U_2.
2. $R^2 = -2\log U_1$ (thus R^2 is exponential with mean 2). $\Theta = 2\pi U_2$ (and thus Θ is uniform between 0 and 2π).

Now let

$$\begin{aligned} X &= R\cos\Theta = \sqrt{-2\log U_1}\cos(2\pi U_2) \\ Y &= R\sin\Theta = \sqrt{-2\log U_1}\sin(2\pi U_2) \end{aligned} \quad . \tag{4.4}$$

The above transformations being what are known as the Box-Muller transformations. Unfortunately the use of the Box-Muller transformations, Equation (4.4), to generate a pair of independent standard normals is computationally not very efficient: The reason for this is the need to compute the sine and cosine trigonometric functions [11]. There is a work-around to this time-consuming difficulty by indirectly computing the sine and cosine of a random angle. This approach would then use the following steps to generate a pair of independent standard normals

1. Generate random numbers, U_1 and U_2.
2. Set $V_1 = 2U_1 - 1,\ V_2 = 2U_2 - 1,\ S = V_1^2 + V_2^2$.
3. If $S > 1$ return to Step 1.

Return the independent standard normals

$$X = \sqrt{\frac{-2\log S}{S}}V_1,\quad Y = \sqrt{\frac{-2\log S}{S}}V_2\ .$$

The above is called the polar method and will, on average, require 2.546 random numbers, 1 logarithm, 1 square root, 1 division, and 4.546 multiplications to generate two independent unit normals.

While the polar method has the advantage of generating **two** independent normal random variables it does so requiring noticeably more calculations to get around having to compute the necessary trigonometric functions. More importantly we are interested in generating a **sequence** of normal random variables which the rejection method deals with more succinctly than the polar method in its current form would (see Section 4.1.1.1). The importance of being able to generate a sequence of normal variables is discussed in more detail in Section 5.1.

Thus because of the shortcoming of the Box-Muller polar method versus the rejection method we will be using the rejection method (Section 4.1.1) to generate normal random variables. These normal random variables will be subsequently used in simulating a geometric Brownian motion price series.

4.2 Analysis and Descriptive statistics

We will be analyzing the final output of our pricing program, the Monte Carlo prices of the options. While these prices are not going to be normally distributed they should oscillate about some "true" value. Any deviant behaviour in the price series will be a firm indicator that, assuming our algorithmic logic is correct, something is wrong at the random variable generation stage.

4.3 Summary

We now have a trusted method for producing normal random variables which are essential for producing the geometric Brownian movement we want our stocks to exhibit. What is left for us is to now build the program and gather results.

Chapter 5

Implementation

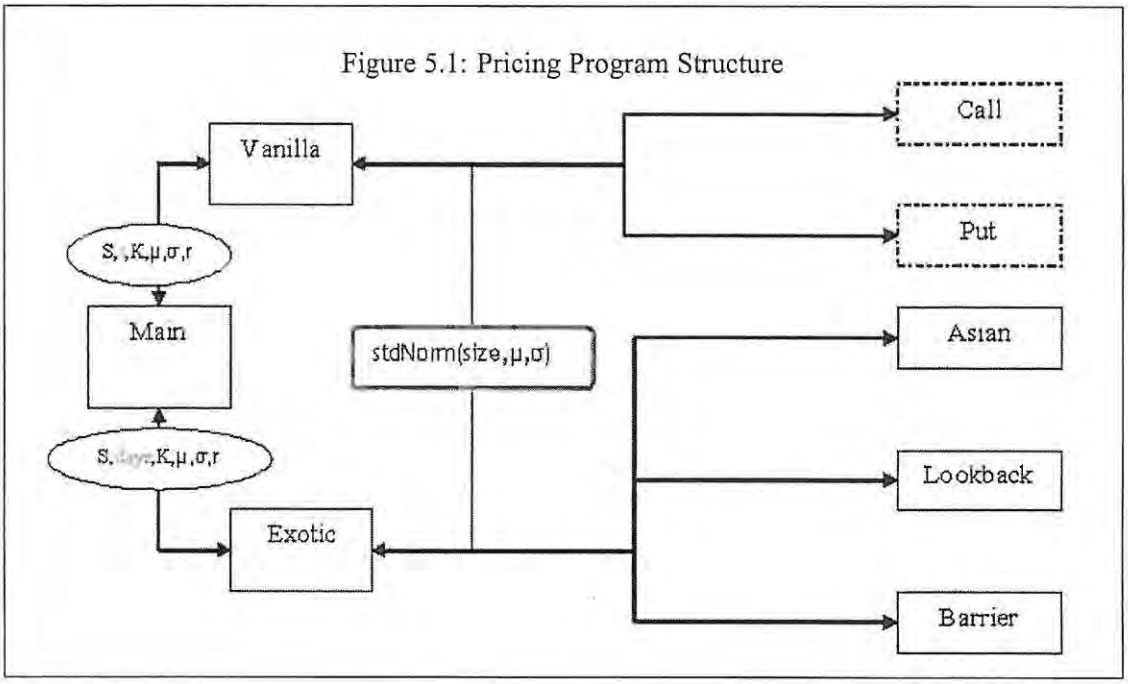
The program we built was implemented following the structure shown in Figure 5.1. Figure 5.1 will be explained in terms of the three pillars of Chapter 3. But firstly we will give a brief description of the whole program and mention some noteworthy points.

Everything begins, as is customary in C++, from the *main()* method. It is in the *main()* method that we solicit information pertinent to the option from the user. This includes:

1. Whether they would like to price a vanilla or exotic option.
2. Initial stock price.
3. Time to expiry of the option.
4. Stock's volatility.
5. Interest rate.

A couple of points must be noted; firstly, the time to expiry can be entered in days, weeks, months or a fraction of the financial year. We set the length of a financial year to two hundred and fifty two days. This information (time to expiry) must then be transformed into days in case it will be used in the pricing of an exotic option and a fraction of one year if it is to be used in the pricing of a vanilla option. The reasons for this are that in the case of a vanilla option we do not need to model the price movement of the stock up to expiry, we only need to model a single termination-time price per simulation run. In the case of an exotic option we must simulate a price for every single day leading up to expiry per simulation run. Thus if an option expires in half a year then we must simulate one hundred and twenty six end of day prices **per simulation run**. The idea is illustrated in Figure 5.2. t is the floating point placeholder for the

Figure 5.1: Pricing Program Structure



length of time in terms of a fraction of a year that a vanilla option is defined and *days* is the integer placeholder for the number of days an exotic option is defined for. The second point we must discuss refers to the statement, **per simulation run**, after some trial and error we decided that it would not be left up to the user to define how many simulation runs they would like to run. The reasoning, or fear, being that an inexperienced user may define a simulation value which is too low and result in an unreliable price being output, or too high and subsequently crash the program or take too long to be useful. In Chapter 3 Section 3.2.2 it is mentioned that the industry standard is ten thousand simulation runs. Initially we found that our program could comfortably run fifty thousand simulations. Although it did produce a price, this price was only stable for options which had a long time till expiry written into them. For exotic options where the number of days till expiry was short i.e. in the region of three to twenty days, it was found upon repeated observation of the programs output that the resulting price generated when using the fixed amount of fifty thousand simulation runs was unreliable. To cope with this we decided to make the number of simulations dynamic depending on the length of the time an option was defined for. We began with a simulation number, *numSims,* of 5 040 001 and whatever number of days, *t,* an option was defined for we divided *numSims,* minus one, by this number of days and

Figure 5.2: Getting the time

```
cout << "\nWould you like to enter the time to expiry as\n(d)Days,\n(w)Weeks,(m)Months or\n" <<
        "(f)a fraction of a trading year? ";
cin >> timeType;

/* The user is given a wide choice of timescales to choose from but at it's heart the time variable will always
be a the fraction of a full years 252 trading days */
switch(timeType){
                case 'd': cout << "\nEnter time to expiry in trading days: ";
                          cin >> t;
                          time = (double)t/N;
                          days = round(t);
                          break;

                case 'w': cout << "\nEnter time to expiry in weeks: ";
                          cin >> t;
                          time = t/52;
                          days = round(t*N/52);
                          break;

                case 'm': cout << "\nEnter time to expiry in months: ";
                          cin >> t;
                          time = t/12;
                          days = round(t*(N/12));
                          break;

                case 'f': cout << "\nEnter time to expiry as a fraction of one trading year (252 days): ";
                          cin >> t;
                          time = t;
                          days = round(t*N);
                          break;

                default: cout << "\nInvalid time scale! Please restart program.\n";
                         break;
```

used that result as our new run count *noSims*

$$noSims = (numSims - 1)/t.$$

This way shorter options went through a larger number of simulation runs making the resulting estimation more reliable. The reason we chose this particular number is that it allows one to price exotic options written for an entire financial year i.e. $t = 252$ days, and still utilise a suitable simulation run count of 20000 (twice the industry standard!). The reader should be wary of pricing long term options written beyond this date. It should be noted that for option time periods **far** beyond this limit the program does endure and produce a price but again caution should be exercised when dealing with this simulated price. This is for exotic options only, for vanilla options where we are not interested in modelling daily behaviour we simply used a fixed number of simulations fifty thousand.

From the *main()* method our program can take one of two branches and so on and so on. Rather than make this a user manual and go through each program path we will address the three major issues of this project (see Chapter 3) in the context of the possible paths the program may take. The source code used in this project is included, in it's entirety, at the end of this document in the appendix.

5.1 Generating the Data

While the *exotic()* and *vanilla()* functions use different forms of the time variable they both at some point call on the method *stdNorm()* to generate the normal random variables needed in generating price movements.

The *stdNorm()* function generates normal random variables with user defined mean and variance. This method is not internal to the C++ class libraries and was built using the rejection method presented in Section 4.1.1.1. Because this particular module is paramount to the **entire** pricing program we will examine it's full implementation in three separate parts and discuss some interesting details.

The first segment of code we will look at is shown in Figure 5.3. Firstly, one must note that the *stdNorm()* function requires one to define the length of the sequence, *seqSize*, of normal random variables. This sequence size information is quite important and has some very serious implications on the goodness of the normal random variables the method produces. This point will be discussed later on. If a mean and variance are not defined it is assumed that the user

wishes to generate a series of standard normal variables, $N(0,1)$. The first action to be performed is that of setting the seed of the built in C++ (pseudo) random number generator (RNG). For the purposes of our pricing method this conventional method of using system time to set the seed is sufficient; there are more complex and powerful ways of generating seeds but these are mainly used in high level security data encryption. The rest of the excerpt shows which variables need to be declared for use in the function and is fairly self explanatory.

From here we begin to follow the steps of the procedure laid out in Section 4.1.1.1 (see Figure 5.4). The outer "for" loop will be run for the number of random variables the user has chosen to be generated. Now, steps one and two require the generation of independent exponential random variables. The generation of random variables which follow the exponential probability density function is also not a part of the standard C++ function library so we had to do this ourselves. Ross [11] notes that we can generate X, an exponential random variable, by the assignment $X = -logU$ where U is a Uniform 0-1 random variable. The simple "trick" we employed to generate $U1$ and $U2$ is to divide *rand()*, a random number produced by the C++ (pseudo) RNG, by *RAND_MAX*, the largest possible random number the RNG can produce. This produces a random number between zero and one. It was pointed out in the final paragraph of Section 4.1.1.1 that if one is generating a sequence of normal random variables we may use the Y exponential random variable from Step 3 as the initial exponential random variable in Step 1. Step 3 is the check for whether our transformed random variable, Y, passes the rejection step test for normality. If it does pass the test the boolean placeholder *pass* is set to one (= true). But if Y fails the rejection step test then no combination of $Y1 = Y$ and $Y2$ will be valid and so we must not only check if we are performing the initial step for Step 1 but also if our Y variable failed the Step 3 test. This tidbit of information was found out after the program entered, without fail, "unexplainable" infinite loops every time it was run! From here we assign a sign, mean and standard deviation to the variable we have produced and then repeat this a number of, *seqSize - 1*, times and return the array Z which contains the number of requested normal variables.

It is very important that one note that using the sequence extension is for more than just the reduced number of exponentials which have to be generated from $U(0,1)$. It is in fact a necessity if one want to get a series of variables which do indeed exhibit normal random behaviour. The reason for this lies in the seed-setting step, *srand((unsigned)(time(NULL)))*, shown in Figure 5.3. If we were to call *stdNorm(10,0,1)* the function would return to us ten standard normal random variables, but if we were to call *stdNorm(1,0,1)* **ten times** we would be returned an array of the same ten numbers! This is because the call to the program is so fast that in the seed-setting step, which uses the system clock to set the seed, the clock would not have had enough time

Figure 5.3: Generating a Normal Random Variable I

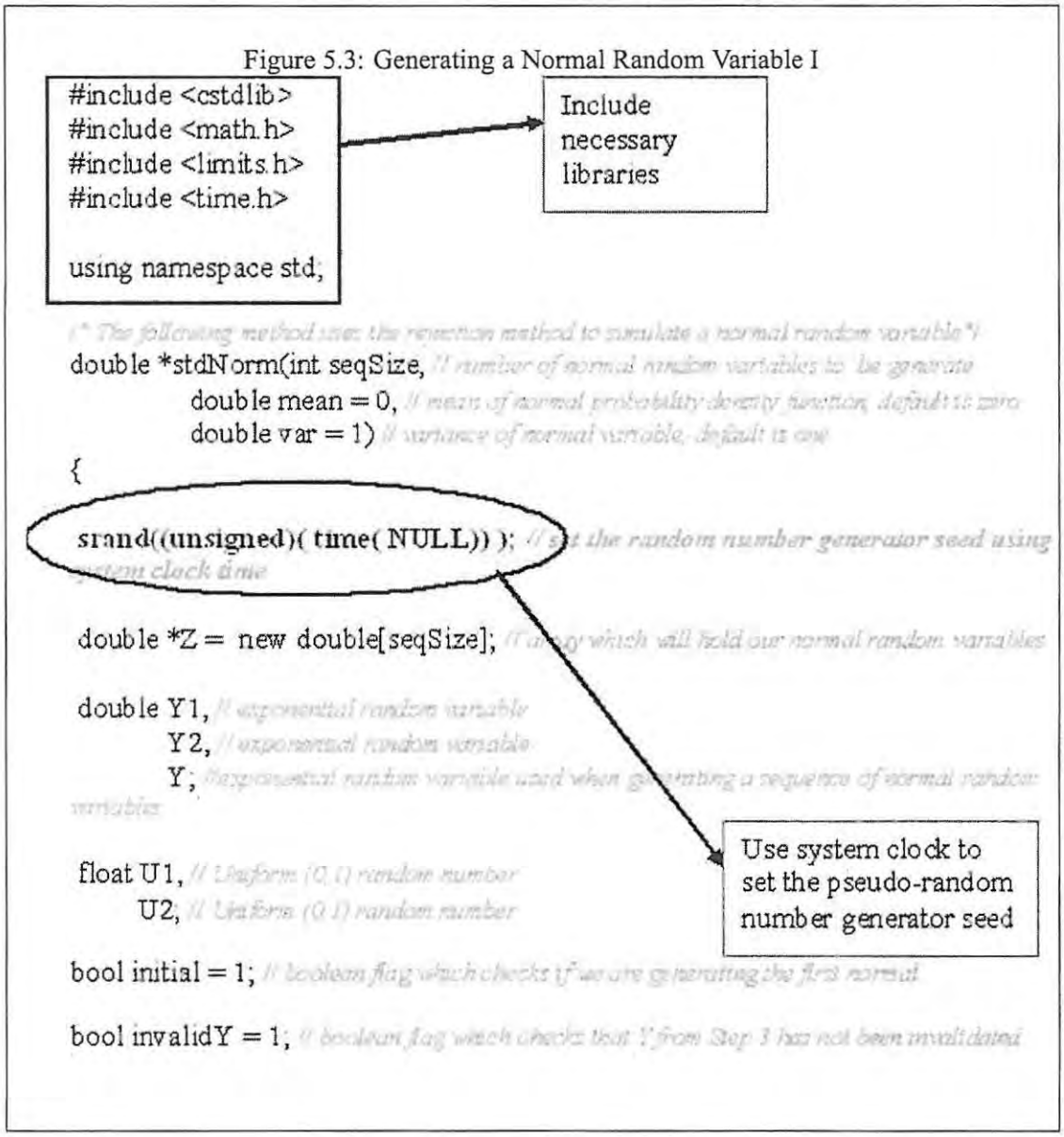


Figure 5.4: Generating a Normal Random Variable II

```
for (int f=0; f<seqSize;f++){

   bool pass = 0,

   while (pass == 0){

      if(initial == 1 || invalidY){
         U1 = (float)rand()/RAND_MAX;
         Y1 = -(double)log(U1);
      }else{
         Y1 = Y;
      }

      U2 = (float)rand()/RAND_MAX,
      Y2 = (double)-log(U2),

      if (Y2 - (double)pow(Y1 - 1,2)/2 > 0){

        Y = Y2 - pow(Y1-1,2)/2,
        pass = 1;

      }

   }
```

'Trick' to generate Unifrom random 0-1 number

Inverse transform method to generate exponential random variable

Figure 5.5: Generating a Normal Random Variable III

```
if ((double)rand()/RAND_MAX > 0.5){
  Z[f] = -Y1*pow(var,0.5) + mean;
}else{
  Z[f] = Y1*pow(var,0.5) + mean;
}

initial = 0;
}

return Z;
```

to increment. So in the millisecond it takes to call *stdNorm()* ten times the seed, and therefore the resulting normal, would not have changed and the deterministic algorithm would produce the same normal number. This lack of randomness, pseudo though it may be, wreaked havoc initially with the pricing because the underlying price movements on which the final Monte Carlo price relied were not, even superficially, random at all. So multiple calls to the *stdNorm()* function were avoided. In fact what we did was make one call to the normal generating function at the very beginning of the *exotic()* module to generate a pool of 5040001, normal random variables we could call on.

The technique we use outlined in Section 4.1.1.1 is designed to produce independent normal random variables. As a check we exported a sample of 252 normal random variables generated by this *stdNorm()* function, each having mean 0.000379167 and variance 0.00130952, to a separate statistical package (the R Foundation free statistical computing software) and examined the Quantile-Comparison Graph or Q-Q Plot. The resulting plot is shown in Figure 5.6. The plot of the points is approximately linear and falls within the defined boundaries (dashed lines in diagram). The boundaries themselves are calculated internally by R and included in the plot without any need for us to define them. We ran this same test a number of times with different groupings of differing sizes and attained the same results. Thus we can say from our tests at least that the variables being generated by our function are indeed normal in nature. One may be a bit tentative in accepting this but given that the main aim of this project is to produce a working program we will accept this widely accepted technique without too much interrogation. So we proceed under the assumption that the rejection method is sound.

Figure 5.6: Normal Quantile Comparison Plot

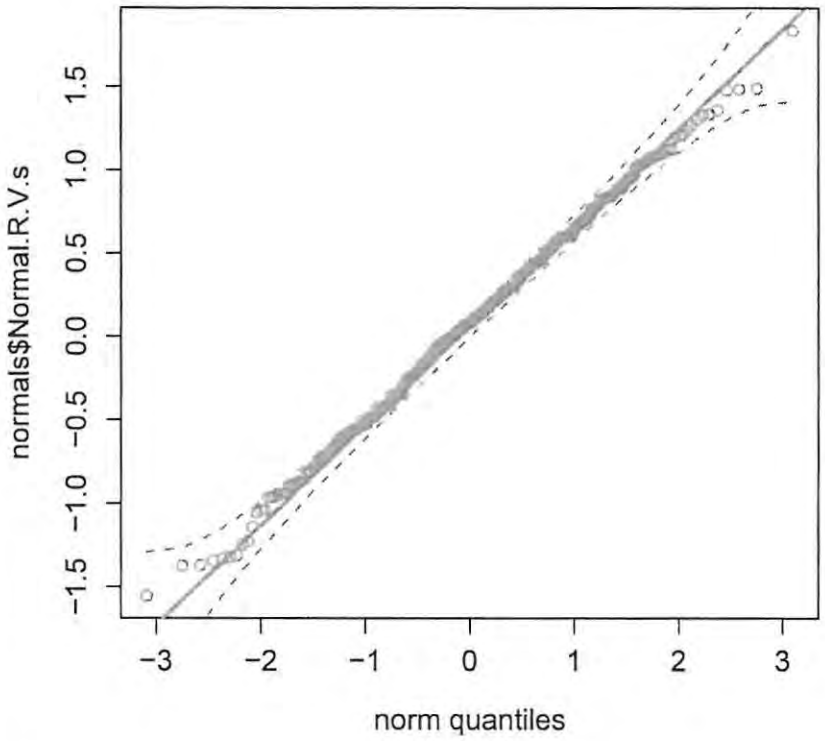


Figure 5.7: Generating the Standard Normals for Vanilla

```
double *stdNorms = new double[numSims];/* matrix which will hold ALL
                                          standard normal variables for
                                          this pricing */

/* The standard normals are created to be passed to the call() and
put() methods for subsequent use in the simulation of payoffs */
stdNorms = stdNorm(numSims,0,1);
```

So having a way to generate normal random variables we will show where exactly this function fits into our program. As we mentioned earlier there are differences in the way one goes about using simulation to arrive at a Monte Carlo estimate of the price of vanilla and exotic European options.

Vanilla Option

Under risk-neutral probabilities, S_t can be expressed as

$$S_t = s \exp\left\{(r - \sigma^2/2)t + \sigma\sqrt{t}Z\right\},$$

where Z is a standard normal random variable [12]. So given that a user has entered a time, t, to expiry and we have defined the number of simulation runs as fifty thousand, we generate fifty thousand standard normal variables, see Figure 5.7. These standard normals are subsequently passed, one by one to the payoff function of whatever vanilla option has been selected. The normal variable is used to calculate the expiry time, t, price of the stock and the payoff is returned.

Exotic Options

With exotic options things are a bit more involved. In the *vanilla()* module we were producing one simulated price per simulation. With the *exotic()* it seemed logical, after noting the "weakness" of the *stdNorm()* function, to generate for each run an array of size $t = days$, and pass this on to whichever exotic payoff function had been selected. This did seem to work reasonably well for options with a large number of days till expiry as it gave the clock and hence the *srand()* seeding function time to reset. But, for options defined on a lower number of days we faced the

Figure 5.8: Use of Standard Normals in Vanilla

```
#include <cstdlib>
#include <iostream>
#include <math.h>

using namespace std;

#include "stdNorm.h"
#include "statistics.h"

#define numSims 50000

/* The following call() method works by , firstly, simulating the time t,
termination date of contract, price of the stock and then using this in
calculating the payoff of a C(s,t,K) call contract */
double call(double s, // stock price
            double z, // normal random variable
            double t, // time to maturity as fraction of 252 day year
            double K, // strike price
            double u, // mhu
            double sig){ // volatility sigma

  double St, // Time t value of stock

  payoff=0.0; /* payoff of standard European call option, we initialize it
              to zero so if option is not in money we simply return zero */

  St = s*exp(u*t + sig*pow(t,0.5)*z);

  if (St > K){
     payoff = St - K;/*else payoff remains = 0 thus simulated => (St - K)+
  }

  return payoff;
}//call
```

same problem mentioned earlier and hit price "spikes" i.e. for the periods when the time did not change we would generate a high frequency of the same price and hence payoff. Finally we decided that the best course of action was to simply generate the entire sequence of normals; an array of size *numSims*, where *numSims* is the number of normal random variables to be used in the simulations i.e. 504001 (see the opening paragraphs of Chapter 5 for an explanation of how this array will be eventually used). The length of the stock prices array, the data set we are interested in, on the other-hand maintained a size of t and we simply overwrote the values of the array for each subsequent simulation run, this is all shown in Figure 5.9, for now you may disregard **antiNorms* and **antiStockPrices*. These are used for variance reduction purposes and will be explained later on. We used the variables shown in Figure 5.6 to test this price sequence generation algorithm. We read the variables into an array and plotted them using MS Excel, the results are shown in Figure 5.10. At this initial stage the data appears to exhibit reasonable randomness and no trend, two qualities we absolutely need our price series to exhibit. Upon generating a price sequence what was left to do was to pass the *stockPrices[]* array to the appropriate exotic payoff function for use in calculating the payoff of the option.

The data generation stage was probably the biggest hurdle because of the subtle nuances which caused such big problems in terms of price instability and the difficulty involved in tracing the source of the problem. Next we discuss the pricing program itself, this section is brief as putting the program together structurally was quite straightforward.

5.2 The Pricing Program

The high level, overall aim of this paper was to create a program which is easy to use, understand, modify and be relevant for some time to come. The ease of use of the program lies in requesting the needed information from the user with, firstly, logically ordered questions and secondly asking those questions concisely but with enough information that ambiguity is avoided. The future relevance of the program is intrinsic to the language chosen to write it in, C++. This choice of implementation language was actually chosen after a long stint with Matlab which did not provide the same flexibility when modifying the code at lower levels.

The ease of understanding and modification are both directly tied into the modularity of the program, which is shown in Figure 5.1 and Figure 5.11. Each type of option has it's own independently functioning module, as do all supporting functions. Because the program is highly modularised, modification to a part of the structure can be done with minimal effort. This means one does not have to go mining through lines and lines of code looking for where to add or change

Figure 5.9: Generating the Normals for Exotics

```
/* For the sake of accuracy it has been found that instead
of regenerating a seed for each subsequent simulation run, we
generate one long list and draw from this uni-seeded list. Thus
we will create a single long array of normal movements. We will
maintain a stock price list of length t and simply overwrite the
values in our stockPrices array on each run but using different
normal values */

/* normals is a matrix which will contain generated independent
normals*/
double *normals = new double[numSims];
/* antiNorms is the matrix of normals negatively correlated
antithetic variables */
double *antiNorms = new double[numSims];

/* security price sequence, includes intermediate day prices as
well, every jth price is end of day price*/
double *stockPrices = new double[t];
/* We use antithetic variables and thus need a second matrix of
resultant stock prices */
double *antiStockPrices = new double[t];

/* we fill our empty normals array with normal random variables
with the appropriate mean and variance */
normals = stdNorm(numSims,mean,var);

for(int i=0;i<noSims;i++){

        /* Initialise the first entry of our array of lognormal
        stock prices */
        stockPrices[0] = S*exp(normals[i*t]);
        if(varRedAlt == 'y'){
           antiStockPrices[0] = S*exp(antiNorms[i*t]);
        }

        /* We build our sequence of prices */
        for (int x=1; x<t; x++){
            stockPrices[x]= stockPrices[x-1]*exp(normals[i*t + x]);
            if(varRedAlt == 'y'){
               antiStockPrices[x] = antiStockPrices[x-1]*exp(antiNorms[
            }
```

Figure 5.10: Plot of Stock Price Sequence

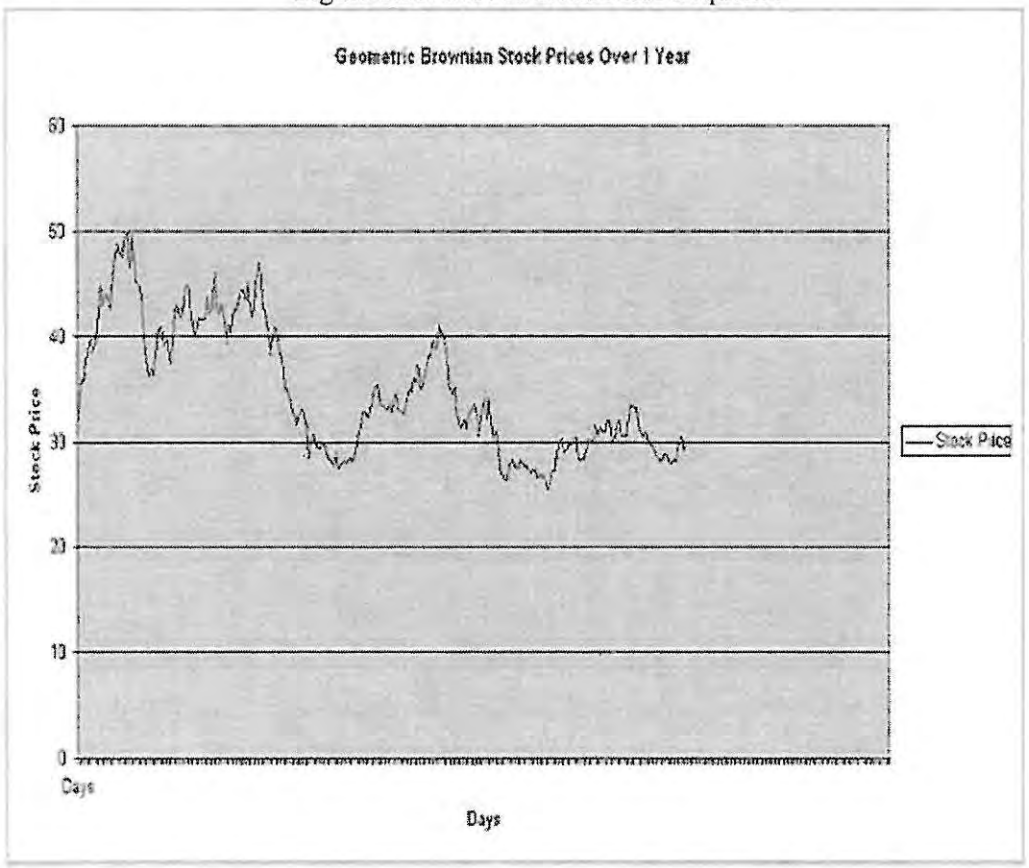
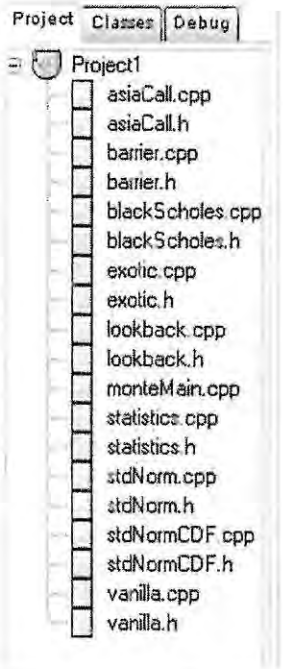


Figure 5.11: Pricing Project Structure



some payoff function and if a more efficient normal random number generator is discovered it can be easily slotted into the mix. As we stated earlier the program could be made more streamlined by the use of templates and inheritance but because it is intended for financial students and/or professionals who may not be familiar with these aspects of the language we opted not to. It was felt that too much complexity would detract from an understanding of the principal aim of the project which was the study of the mathematical finance concepts involved in pricing exotic options.

The next section details one aspect that did not fit so cleanly into the package, that of variance reduction.

5.3 Variance Reduction

Variance reduction was slightly more tricky to deal with than the other two pillars because of the nature of the procedure. For the case of barrier options we had to deal with an option-dependent variance reduction technique; Conditional Expectation (refer to Section 1.1.6.1) . So where we

Figure 5.12: Control Variate Implementation I

```
/* User should have selected which type of exotic option they want
to price */
switch(abc){
            case 'a': runPayoff[0] = asiaCall(stockPrices,t,K,r,PS);

                      if (varRedAlt == 'y'){
                         temp[0] = asiaCall(antiStockPrices,t,K,r,PS);
                         runPayoff[0] = (temp[0] + runPayoff[0])/2;
                      }

                      if (varRed1 == 'y'){
                         sumBSPayoffs += callPayoff(stockPrices[t-1],S);
                      }

                      payoff[i] = runPayoff[0];
                      sumPayoffs += payoff[i];
                      break;
```

were originally just calling the *barrier()* function we were now calling it with an added variance reduction flag which dictated the course of action the function undertook. We implemented three different types of variance reduction techniques, all of which are discussed from a theoretical point of view in Chapter 1 Section 1.1.6.1.

The first type of variance reduction technique we implemented was that of **control variates,** and required us to keep track of the simulated price of an at-the-money call option. Figure 5.12 shows where this step takes place in the case that we are pricing an Asian option and one wishes to utilise control variates. If a user has indeed selected to use control variates then the character placeholder *varRed1* is set to *'y'* indicating yes and the sum of simulated Black-Scholes payoffs, *sumBSPayoffs,* is updated. *sumBSPayoffs* will be eventually used in estimating the simulated price of an at-the-money vanilla call option, at the end of the Monte Carlo pricing procedure (Figure 5.13). The variable *bS,* shown in Figure 5.13, is the analytical Black Scholes price of an at-the-money call option, which had been calculated earlier using the parameters defined by the user for the exotic option. From the same figure *price[0]* is the simulated final price of the option we have chosen to price. The idea is that *price[0]* will be accordingly adjusted depending on whether we have been consistently under or overestimating our prices since *BSPayoff* would have been calculated from the same price series. Thus the positive or negative bias of *(bs - BSPayoff)* is used to recalibrate *price[0].*

Figure 5.13: Control Variate Implementation II

```
price[0] = sumPayoffs/noSims;

/* If the user has chosen to use control variate reduction the
following will be needed */
BSPayoff = exp(-r*t/N)*sumBSPayoffs/noSims;

if (varRed1 == 'y'){
   price[0] + (bS - BSPayoff);
}
```

For the second type of variance reduction technique we implemented **conditional expectation** reduction (Section 1.1.6.1). This particular variance reduction technique is unique to down-and-in and up-and-in barrier options. The technique relies on manipulating the actual payoff function and as such is dealt with within the *barrier()* function itself. The call to the *barrier()* function is made in the *exotic()* module (Figure 5.14), but along with the usual information we also pass the character data held in *varRed2,* which in this case indicates whether or are not to use conditional expectation variance reduction. The implementation of the actual methodology is shown in Figure 5.15. Upon breaching the barrier v the system breaks and, if variance reduction has been chosen, uses Black Scholes formula to calculate the payoff of the run. This payoff is saved and returned in *payoff[1],* the *exotic()* program checks to see if the conditional expectation flag was set and sets the payoff of the run to *runPayoff[1]* accordingly (see Figure 5.14). The multiple *if* loops in Figure 5.14 are there to deal with the different combinations of variance reduction which could be implemented together, in this case we must deal with conditional expectation in combination with antithetic variates or not and vice versa. Control variate reduction works quite independently of the other two and so does not have to be included in the myriad of *if-else* loops shown in the figure.

And finally we used **antithetic variable variance reduction.** This variance reduction technique is accessible to all option types. This method is quite straightforward but does require the declaration of extra arrays (shown in Figure 5.9) and rerunning the same exotic option payoff function with the transformed *antiStockPrices[]* matrix. An example of this implementation is shown in Figure 5.16. From the figure; *varRedAlt* is the character indicator of whether we are to use antithetic variates, if we are then the payoff function, *lookback()*, is rerun but this time with the stock prices developed from an array of normal variables which are negatively correlated

Figure 5.14: Conditional Expectation Implementation I

```
case 'b': runPayoff = barrier(stockPrices,t,K,sigma,r,S,v,UD,varRed2);

          if (varRedAlt == 'y' || varRed2 == 'y'){
             if (varRedAlt == 'y'){
                temp = barrier(antiStockPrices,t,K,sigma,r,S,v,UD,varRed2);
                runPayoff[0] = ( temp[0] + runPayoff[0] )/2;
                runPayoff[1] = ( temp[1] + runPayoff[1] )/2;
                if (varRed2 == 'y'){
                   payoff[i] = runPayoff[1];
                }else{
                   payoff[i] = runPayoff[0];
                }
             }else{
                payoff[i] = runPayoff[1];
             }
          }else{
             payoff[i] = runPayoff[0];
          }

          if (varRed1 == 'y'){
             sumSSPayoffs += callPayoff(stockPrices[t-1],S);
          }

          sumPayoffs += payoff[i];
          break;
```

Figure 5.15: Conditional Expectation Implementation II

```
if (UD == 'a' || UD == 'b'){

   /* We check to see if the barrier v is breached.
   If user has selected (a) then the option 'comes to life'
   otherwise it faces death */
   for (int x=0 ; x < t ; x++){
       T++;
       if (stock[x] < v){
          I = (int)pow(I-1,2);
          break;
       }//if
   }//for

}else{

   /* We check to see if the barrier v is breached.
   If user has selected (c) then the option 'comes to life'
   otherwise it faces death */
   for (int x=0 ; x < t ; x++){
       T++;
       if (stock[x] > v){
          I = (int)pow(I-1,2);
          break;
       }//if
   }//for

}//ifelse

if ((UD == 'a' || UD == 'c') && varRed == 'y'){
   payoff[1] = I*blackScholes(stock[T-1],
                              (double)(t-T)/N, K, sigma, r);
}

if (stock[t-1] > K){
   payoff[0] = exp(-r*t/N)*I*(stock[t-1] - K);
}

return payoff;
```

Figure 5.16: Antithetic Variates Implementation I

```
case 'c': runPayoff[0] = lookback(stockPrices,t,K,r,mM);

          if (varRedAlt == 'y'){
             temp[0] = lookback(antiStockPrices,t,K,r,mM);
             runPayoff[0] = (temp[0] + runPayoff[0])/2;
          }

          if (varRed1 == 'y'){
             sumBSPayoffs += callPayoff(stockPrices[t-1],S);
          }

          payoff[i] = runPayoff[0];
          sumPayoffs += payoff[i];
          break;
```

with our original series of normal random variables. The average of the two payoff returns is then set as the payoff for that particular run.

Running the three variance reduction techniques in tandem got a bit tricky as one can see from the multiple *if* statements of Figure 5.14 and the results of this, and the program in general, are discussed in Chapter 6. Note that the program is presented **in full** in the appendix.

Chapter 6

Data Analysis and Results

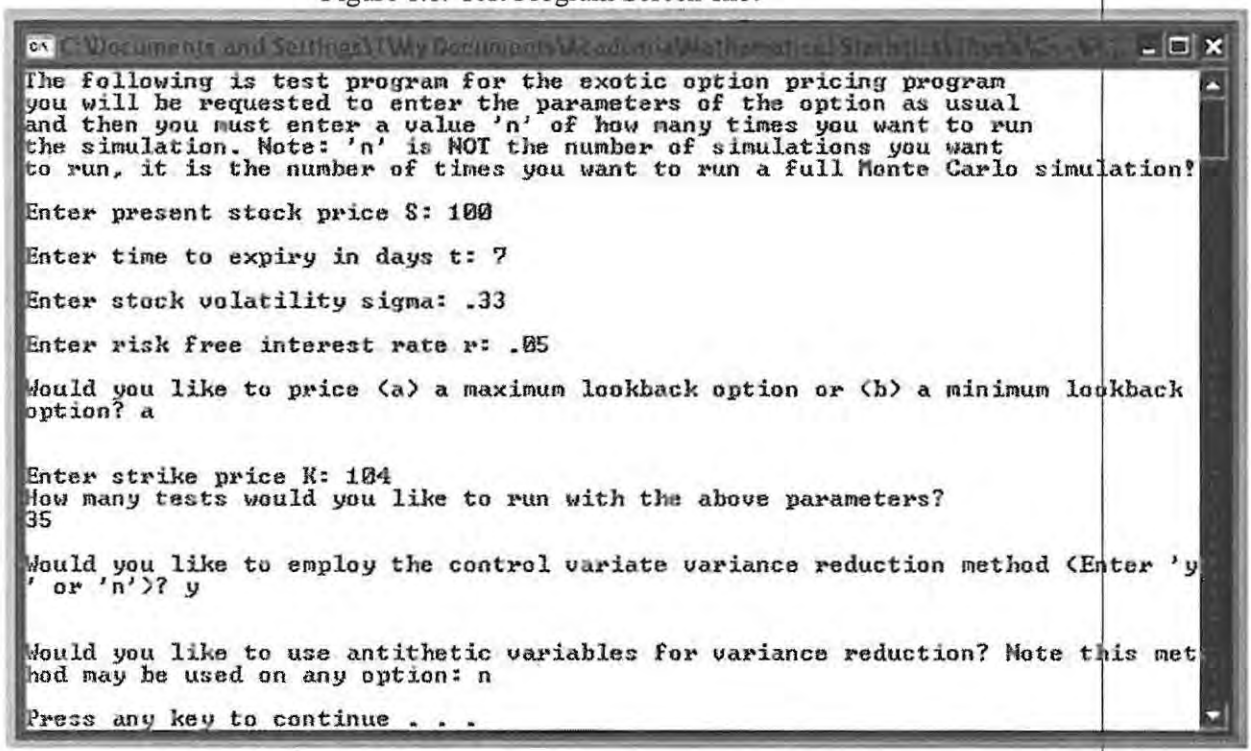
6.1 Data Analysis and Results

The main result of our project was the pricing program itself which is available in its entirety on the disk accompanying this write-up. The reader is encouraged to test this program to their hearts content or even use it as a reference for their own pricing endeavours. A second less prominent aim of our pricing project was to investigate whether Monte Carlo simulation is an optimal approach to producing stable exotic option pricings. So the focus was not necessarily to see **if** simulation estimation would work and we would indeed get some reasonable pricings, the point was to see **how well** it would work. We performed extensive checking of the main pricing program for correctness, efficiency and validity by running several diagnostic programs which we designed and are included on the CD.

The test programs allow one to define the number of times they want to run a **full** Monte Carlo estimation with the same parameters. If the reader chooses to implement variance reduction then the test programs will produce, using the same set of parameters and random variables, both non-variance reduced and variance reduced results. This is to ensure there is no biasedness due to sequence fluctuation and allows for fair comparisons when considering whether the variance reduction techniques were truly effective. The test program screen is shown in Figure 6.1.

The results from the test programs were written to a text file (.txt) which are also included on the accompanying software disk. If variance reduction is used in the testing process then the respective text file consists of two columns the first being the non-variance reduced prices and the second being that with *all* variance reduction techniques applied. Writing the results to a separate text document gives the added flexibility of later exporting and analysing the data on a separate and independent medium.

Figure 6.1: Test Program Screen-shot



The source code which is the basis of this project is included at the end of this paper and on the supplementary software disk. On the whole the program and techniques implemented were validated. In the following chapter we present the conclusions we drew from observation of our test program results.

Chapter 7

Conclusion

The aims of this project included:

- Constructing a working program which used Monte Carlo simulation techniques to efficiently price exotic options thereby,
- demonstrating the use of the C++ programming language as a simulation tool in the efficient pricing of exotic European options.
- Extending the basic problem of simulation pricing and including variance reduction by conditional expectation, control and antithetic variates.

The following sections give our thoughts on how closely our intentions were satisfied as well as some conclusions related to the process itself.

7.1 Conclusions

It turned out that a major part of our program hinged on our being able to efficiently generate normal random variables (Pillar I, see Chapter 3). We opted to use the rejection method outlined in Section 4.1.1.1. Some changes were necessary to make the technique viable using C++ but in the end it could be concluded that the **module** responsible for generating normal random variables worked (see Figure 5.6). Once this was taken care of we were indeed able to construct a modularized, easily extend-able program which effectively made use of Monte Carlo simulation techniques to price lookback, Asian and barrier exotic options.

It was mentioned that we were not only interested in testing theories of variance reduction techniques on their own but also in combination with each other. Antithetic and control variates

worked better on their own rather than in combination with each other. And while they did not work well with each other (in some cases resulting in higher variance than when not using variance reduction at all!) on their own they validated themselves as variance reduction techniques. Referring again to using the two techniques in combination it was in fact found that control variates in combination with any of the other variance reduction techniques resulted in increased variance. This was true of barrier options which had the unique characteristic of being the only option type which could implement conditional expectation variance reduction refer to Section 1.1.6.1. The barrier options were also unique in that they were the only options to exhibit an improvement in variance reduction when variance reduction combinatorics were employed. It was found for the set of barrier options results that conditional expectation in combination with antithetic variates resulted in the lowest price standard deviation. It should also be noted that because of the way that we implemented the barrier option pricing algorithm, actually the entire program, we did not reap the added benefit of having to generate fewer normal random variables since we generated that entire sequence at the very beginning of the program. But, again, it was necessary for us to follow this route of generating all the normal random variables at the beginning of the program because of the issues mentioned in Chapter 5, Section 5.1. Ultimately we validated variance reduction and even unveiled the interesting information concerned with combining the variance reduction techniques mentioned above.

7.2 Contributions

This thesis has provided an extensive list of option prices, variances and comparisons of variance reduction technique implementations. Thus results obtained independently could be compared using the table and the .txt files included on the accompanying disk to test separate programs and techniques.

The program in itself is a novel contribution to the field, and while there are some esoteric intricacies here and there in the source code it is structurally simple enough to be understood by a novice financial programmer but powerful enough to be useful to a mathematical finance maestro!

7.3 Future Work

As with all computer based endeavours the program could always be made faster, more diverse and a more user friendly GUI would aid in its adoption by individuals and groups. From a

statistical point of view a number of efficiency improving techniques were not fully utilized because of the way we had to generate our normal random variables; all at once at the beginning of the program. This means that the added advantages of conditional expectation (see [12]) could be acquired and also the slight gain in efficiency and simulation time mentioned by Ross and Shanthikumar in [13] when simulating Asian options would also be taken advantage of. It would as well be interesting to find out why exactly control variates tend to result in inflated variance about the true value when used in combination with other variance reduction techniques.

References

[1] BAXTER, M and RENNIE, A. 2002. *Financial Calculus: An Introduction to Derivative Pricing.* United Kingdom. Cambridge University Press.

[2] BOYL, P. 1977. *Options, a Monte Carlo Approach.* ***Journal of Financial Economics*** 4: 323-338.

[3] BOYL, P., BROADIE, M., & GLASSERMAN, P. 1997. *Monte Carlo methods for security pricing.* ***Journal of Economic Dynamics and Control*** 21: 1267-1321

[1] BRENT R. P.. 1993. *Fast Normal Random Number Generators for Vector Processors.* Computer Sciences Laboratory, Australian National University, Canberra, ACT 0200.

[2] ELLIS M. A. and STROUSTRUP B.. 1990. *The Annotated Reference Manual.* Addison Wesley.

[6] GENTLE, J. E., HARDLE, W. and MORI, Y. 2004. *Handbook of Computational Statistics: Concepts and Methods.* Germany. Springer.

[7] KINDERMAN, A. J. and RAMAGE J. G.. December, 1976. *Computer Generation of Normal Random Variables.* ***Journal of the American Statistical Association, Vol. 71, No. 356, pp. 893 - 896 doi:10.2307/2286857.***

[8] ODEGAARD, B. A. 2003. *Financial Numerical Recipes in C++.* http://finance-old.bi.no/~Odegaard/gcc_prog/recipes/recipes.pdf [Accessed 14 March, 2006]

[9] ROSS, S. M. 1997. *Simulation (2nd Edition).* New York. Academic Press.

[10] ROSS, S. M. 2000. *Introduction to Probability Models (7th Edition).* Burlington, Massachusetts. Harcourt Academic Press.

[11] ROSS, S.M. 2002. *Simulation (3rd Edition).* Orlando, Florida. Academic Press.

[12] ROSS, S. M. 2003. *An Elementary Introduction to Mathematical Finance (2nd Edition).* United States. Cambridge University Press.

[13] ROSS, S. M. and SHANTHIKUMAR J G. 2000. *Pricing Exotic Options: Monotonicity in Volatility and Efficient Simulation.* Cambridge University Press.

[14] SCHOUTENS, W and SYMENS, S. 2002. *The Pricing of Exotic Options by Monte-Carlo Simulations in a Levy Market with Stochastic Volatility.* Belgium. University of Antwerp.

[15] STROUSTRUP B.. 1997. *The C++ Programming Language, Third Edition.* Addison-Wesley. Soft-cover ISBN 0201889544.

Appendix A

Additional Material

Main() and Supporting methods

```
/*============================================================================
 *  This code is Copyright (C) 2006, Tawuya D R Nhongo.
 *  Permission to use this code for non-commercial purposes
 *  is hereby given, provided proper reference is made to:
 *              Nhongo, T.D.R. (2006), "Pricing Exotic Options Using C++",
 *  Statistics Department, Rhodes University, Republic of South Africa
 *      or the published version when available.
 *      This reference is still required when using modified versions of the code.
 *  This notice should be maintained in modified versions of the code.
 *      No warranty is given regarding the correctness of this code.
 *============================================================================*/
#include <cstdlib>
#include <iostream>
#include <math.h>
using namespace std;
#include "exotic.h"
#include "vanilla.h"
/* we define the variable N, which represents the total number of trading
days in a year */
#define N 252
int main()
{
  //We will need to solicit some information from our aspiring financial analyst
  double S, // present price of the stock
  sigma, // stock volatility
  r; // risk free interest rate
  double *Price = new double[2]; /* will contain Monte carlo price of whichever option is chosen and
                                    variance of our Monte Carlo price */

  char timeType; // for added flexibility diff. time frames  used when specifying option parameters
  double t; // time till option expiry, could be in days, weeks, months or fraction of year
  double time; // proportion of a trading year option is for. Used in vanilla() option method
```

```
int days; // exact number of days option is for. Used in exotic() option method

char VE = '\0'; // char placeholder of what type of option user wants to price: Vanilla or Exotic
bool valid = 0; // boolean flag which tells us whether valid character has been inputted
while (valid != 1){//check to see that either 'v' vanilla or exotic is chosen
      cout << "Would you like to price a (v) Vanilla or (e) Exotic option? ";
      cin >> VE;

      if(VE != 'v' && VE != 'e'){
            cout << "Error! Please enter either the character 'v' or 'c'\n";
      }else{
            valid = 1; //ok, VE is valid we can move on
      }
}

cout << "Enter present stock price S: ";
cin >> S;

cout << "\nWould you like to enter the time to expiry as\n(d)Days,\n(w)Weeks,(m)Months or\n" <<
        "(f)a fraction of a trading year? ";
cin >> timeType;

/* User is given choice of timescales to choose from but at heart the time variable will always
be some fraction of a full years 252 trading days */
switch(timeType){
                case 'd': cout << "\nEnter time to expiry in trading days: ";
                          cin >> t;
                          time = (double)t/N;
                          days = round(t);
                          break;

                case 'w': cout << "\nEnter time to expiry in weeks: ";
                          cin >> t;
                          time = t/52;
                          days = round(t*N/52);
                          break;

                case 'm': cout << "\nEnter time to expiry in months: ";
                          cin >> t;
                          time = t/12;
                          days = round(t*(N/12));
                          break;

                case 'f': cout << "\nEnter time to expiry as fraction of year (252 days): ";
                          cin >> t;
                          time = t;
                          days = round(t*N);
                          break;

                default: cout << "\nInvalid time scale! Please restart program.\n";
                         break;
}
```

```
    cout << "\nEnter stock volatility sigma: ";
    cin >> sigma;
    cout << "\nEnter risk free interest rate r: ";
    cin >> r;
    cout << "\n";

    double mhu = r - pow(sigma,2)/2; // mean = u = (r -sigma^2/2)
    double sig = sigma; // Standard deviation = sigma.

    /* Did user select to price an exotic or a vanilla type option?
    Whatever the choice it dictates what route we follow */
    switch (VE)
    { case 'v' : Price = vanilla(S,time,mhu,sig,r);
                 break;
      case 'e' : Price = exotic(S,days,mhu,sig,r);
                 break;
      default:   break;
    }
    cout << "The Monte-Carlo price of the option you have selected is: " << Price[0] <<
            ".\nWith variance: " << Price[1] << ".Please come again! \n";

    system("PAUSE");

    return 0;
}
/* Following functions return descriptive statistics of mean, variance and covariance of data sets */
#include <math.h>
double average(double X[],int size){
    double sum = 0.0;

    for (int i = 0; i < size; i++){
        sum += X[i];
    }

    return (double)sum/size;
}
double variance(double X[],int size){
    double varSum = 0.0;
    double Xbar = average(X,size);

    for (int i = 0; i < size; i++){
        varSum += pow(X[i]-Xbar,2);
    }

    return (double)varSum/(size-1);
}
double covariance(double X[], double Y[], int size){
    double covSum = 0.0;
    double Xbar = average(X,size);
    double Ybar = average(Y,size);
```

```
    for (int i = 0; i < size; i++){
        covSum += (X[i] - Xbar)*(Y[i] - Ybar);
    }

    return (double)covSum/(size-1);
}
#include <cstdlib>
#include <iostream>
#include <math.h>
#include <time.h>
using namespace std;
/* The following method uses the rejection method to simulate a normal random variable*/
double *stdNorm(int seqSize, // number of normal random variables to  be generated
                double mean = 0, // mean of normal probablility density function, default is zero
                double var = 1) // variance of normal variable, default is one
{

  srand((unsigned)( time( NULL)) ); // set the random number generator seed using system clock time

  double *Z =  new double[seqSize];// array which will hold our normal random variables

  double Y1, // exponential random variable
         Y2, // exponential random variable
         Y; //exponential random variable used when generating a sequence of normal random variables

  float U1, // Uniform (0,1) random number
        U2; // Uniform (0,1) random number

  bool initial = 1; // boolean flag which checks if we are generating the first normal

  bool invalidY = 1; // boolean flag which checks that Y from Step 3 has not been invalidated
  for (int f=0; f<seqSize;f++){

      bool pass = 0;// boolean flag for our Step 3 checkpoint

      while (pass == 0){
            if(initial || invalidY){
               /*Step 1: Generate Y1, an exponential random variable with rate 1*/
               U1 = (float)rand()/RAND_MAX;
               Y1 = -(double)log(U1); // Inverse transform method
            }else{
               Y1 = Y;
            }// if/else
            /*Step 2: Generate Y2, an exponential random variable with rate 1*/
            U2 = (float)rand()/RAND_MAX;
            Y2 = (double)-log(U2); // Inverse transform method
            /*Step 3: If Y2 - (Y1 - 1)^2/2 > 0, set Y = Y2 - (Y1 -1)^2/2 and go to Step 4.
            Otherwise go to Step 1*/
            if (Y2 - (double)pow(Y1 - 1,2)/2 > 0){

               Y = Y2 - pow(Y1-1,2)/2;
               pass = 1;
```

```
                invalidY = 0;
            }else {invalidY =1;}
        }//while
        /*Step 4: Generate a random number U and set Z = +-Y1 if U<=0,5 or U>0.5 respectively */
        if ((double)rand()/RAND_MAX > 0.5){
           Z[f] = -Y1*pow(var,0.5) + mean;
        }else{
           Z[f] = Y1*pow(var,0.5) + mean;
        }// if/else
        initial = 0;
    }//for
    return Z;
}//stdNorm
// Copyright 1992-2004 Datasim Component Technology BV
// Authors: Daniel Duffy, Robert Demming
//
// This file is part of the Datasim Financial Toolkit
//
// Datasim Component Technology, Schipluidenlaan 4,
// 1062 Amsterdam, The Netherlands (www.datasim.nl)
//
// Permission to modify the code and to distribute modified code is
// granted, provided the text of this NOTICE is retained, a notice that
// the code was modified is included with the above COPYRIGHT NOTICE.
//
// LICENSOR MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.
// By way of example, but not limitation, Licensor MAKES NO
// REPRESENTATIONS OR WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY
// PARTICULAR PURPOSE OR THAT THE USE OF THE LICENSED SOFTWARE
// COMPONENTS OR DOCUMENTATION WILL NOT INFRINGE ANY PATENTS,
// COPYRIGHTS, TRADEMARKS OR OTHER RIGHTS.
//
#include <math.h>
double n(double x)
{
    double A = 1.0/sqrt(2.0 * 3.1415);
    return A * exp(-x*x*0.5);
}
double stdNormCDF(double x)
{ // The approximation to the cumulative normal distribution
    double a1 = 0.4361836;
    double a2 = -0.1201676;
    double a3 = 0.9372980;
    double k = 1.0/(1.0 + (0.33267 * x));

    if (x >= 0.0){
       return 1.0 - n(x)* (a1*k + (a2*k*k) + (a3*k*k*k));
    }else{
       return 1.0 - stdNormCDF(-x);
    }
}
#include <cstdlib>
```

```
#include <iostream>
#include <math.h>
using namespace std;
#include "stdNormCDF.h"
/* We implement the famous black scholes formula => C = S(0)I(w) - Ke^-rtI(w - sigmaRoot(t)) */
double blackScholes(double s, // security price
                    double t, //time to expiry
                    double K, //strike price
                    double sigma, //stock volatility
                    double r) // risk free interest rate
{

    double C, omega;

    omega = ( r*t + pow(sigma,2)*t/2 - log(K/s) ) / (sigma*pow(t,0.5));

    C = s*stdNormCDF(omega) - K*exp(-r*t)*stdNormCDF(omega - sigma*pow(t,0.5));

    return C;
}
```

Vanilla()

```
#include <cstdlib>
#include <iostream>
#include <math.h>
using namespace std;
#include "stdNorm.h"
#include "statistics.h"
#define numSims 50000
/* The following call() method works by , firstly, simulating the time t,
termination date of contract, price of the stock and then using this in
calculating the payoff of a C(s,t,K) call contract */
double call(double s, // stock price
            double z, // normal random variable
            double t, // time to maturity as fraction of 252 day year
            double K, // strike price
            double u, // mhu
            double sig){ // volatility sigma

  double St, // Time t value of stock

  payoff=0.0; /* payoff of standard European call option, we initialize it
              to zero so if option is not in money we simply return zero */

  St = s*exp(u*t + sig*pow(t,0.5)*z);
```

```
  if (St > K){
     payoff = St - K;/*else payoff remains = 0 thus simulated => (St - K)+ */
  }
  return payoff;
}//call
double callPayoff(double s, double K){
  double payoff = 0.0;

  if (s > K){
     payoff = s - K;
  }

  return payoff;
}//call
/* Pricing a vanilla European options means that we only need to know the terminal
price S(t) of our stock, which requires us to simulate a SINGLE lognormal random
variable per simulation run */
double *vanilla(double S, // present price of stock
                double t, //time to expiry as fraction of year
                double mhu, // mean of our lognormal random variable
                double sigma, // stock volatility
                double r){ // interest rate
  double *payoff = new double[numSims]; // payoff from simulation run
  double K; // strike price of option
  double *price = new double[2]; /*price[0] => Monte Carlo simulated price of our vanilla option,
                                   price[1] => variance */

  cout << "\nEnter strike price K: ";
  cin >> K;

  char CP = '\0'; // character placeholder of type of vanilla option to price: Call or Put
  bool valid=0; // boolean flag which tells us whether valid character has been inputted

  while (valid !=1){

       cout << "Enter (c) for Call or (p) for Put: ";
       cin >> CP;

       if (CP != 'p' && CP != 'c'){
            cout << "Error: Please enter either the charcter (c) or (p)! \n";
       }else{
            valid = 1;
       }
  }

  double *stdNorms = new double[numSims];/* matrix which will hold ALL
                                            standard normal variables for
                                            this pricing */

  /* The standard normals are created to be passed to the call() and
  put() methods for subsequent use in the simulation of payoffs */
  stdNorms = stdNorm(numSims,0,1);
```

```
    double sumPayoffs = 0; // the sum of all our calls to the call() or put() methods

    double C = 0; /* Value of call option with user selected parameters. Whether pricing call or put
                     we will use this variable */

    for (int i=0 ; i<numSims ; i++){
        payoff[i] = call(S,stdNorms[i],t,K,mhu,sigma); // make call to call() here
        sumPayoffs += payoff[i];
    }
    C = exp(-r*t)*sumPayoffs/numSims; // whether we price a put or call this value is be needed

    /* If our user has decided to price a European call option then we merely set price to C otherwise,
    if the use has chosen to price a put option use the relation;
    P = K*exp(-r*t) + C - S, where P - price of put option, K - strike price, r - risk neutral interest
    rate, t - time to expiry in years, C - price of call option with same parameters and S - initial stock
    price,  */
    if (CP == 'c'){
           price[0] = C;
    }else{
           price[0] = K*exp(-r*t) + C - S;
    }

    delete [] stdNorms;

    price[1] = variance(payoff,numSims);

    return price;
}//vanilla
```

Exotic()

```
#include <cstdlib>
#include <iostream>
#include <math.h>
using namespace std;
#include "statistics.h"
#include "asiaCall.h"
#include "barrier.h"
#include "lookback.h"
#include "blackScholes.h"
#include "vanilla.h"
#include "stdNorm.h"
#define N 252
#define numSims 5040001 /* to start with we set our numsimulations to 5040000 because
                           the actual number of simulations we will run is this number
                           divided the number of days the users option has been defined for.
                           We use this particular number because it is at this level that even if
                           an option is written for an entire year we still generate 20000
                           simulations which is adaquate. */
/* Exotic option pricing requires us to generate lognormal variables for not only
```

```
the time t price of our stock but indeed lognormal variables for everyday of the
options lifetime. When pricing exotic options we are concerned with end of day prices;
it is these prices which are used in calculating the average, barrier or maximum
of an option for Asian, Barrier and Lookback options respectively. */
double *exotic(double S, // initial stock price
               int t, //days to expiry
               double mhu, // mean mhu
               double sigma, // security volatility
               double r){ //risk free interest rate


  int noSims = round((numSims-1)/t);
  /* Our mean and variance have to be adjusted accordingly */
  double mean = mhu/N;
  double var = pow(sigma,2)/N;

  double K = -1; //strike price, if an option does not use this then it's passed as -1

  double sumPayoffs = 0; // sum of simulated payoffs


  double *price = new double[2];/* Returned pointer will contain Monte Carlo price and
                                   variance of simulation */

  /* Now for the sake of accuracy it has been found that instead
  of regenerating a seed for each subsequent simulation run, we
  generate one long list and draw from this uni-seeded list. Thus
  we will create a single long array of normal movements. We will
  maintain a stock price list of length t and simply overwrite the
  values in our stockPrices array on each run but using different
  normal values */

  /* normals is a matrix which will contain generated independent
  normals*/
  double *normals = new double[numSims];
  /* antiNorms is the matrix of normals negatively correlated
  antithetic variables */
  double *antiNorms = new double[numSims];
  /* security price sequence, includes intermediate day prices as
  well, every jth price is end of day price*/
  double *stockPrices = new double[t];
  /* We use antithetic variables and thus need a second matrix of
  resultant stock prices */
  double *antiStockPrices = new double[t];

  /* we fill our empty normals array with normal random variables
  with the appropriate mean and variance */
  normals = stdNorm(numSims,mean,var);

  char abc; // character placeholder of type of exotic user wants to price: Asian, barrier or lookback
  bool validOp = 0; // boolean flag which tells us whether valid character has been inputted
```

```
while (validOp != 1){
      cout << "Would you like to price:\n(a) Asian,\n(b) Barrier or\n(c) Lookback option? \n";
      cin >> abc;

      if (abc != 'a' && abc != 'b' && abc != 'c'){
              cout << "Please choose any of characters a, b or c\n";
              abc = '\0';
      }else{
              validOp = 1;
      }
}

/* Before simulation a number of flags unique to each type of option need to be set */

/* For Asian option the user must set flag PS for whether the option is floating Price or
floating Strike. This value is passed to the function */
char PS;
bool validPS = 0;

/* Secondly, with barrier must know whether option is up or Down and in that case is it in-Out */
char UD;
bool validUD = 0;

/* We also need to check that we have a valid barrier v */
double v;
int validV;

/* And finally we need a flag for whether we are pricing a maximal or minimal lookback option */
char mM;
bool validmM = 0;

switch(abc){
            case 'a': while (validPS != 1) {
                        cout << "Would you like to price (a) floating strike or (b) floating price o
                        cin >> PS;
                        cout << "\n";
                        if (PS != 'a' && PS != 'b'){
                           cout << "Please enter either character (a) or (b) \n";
                        }else{
                           validPS = 1;
                        }

                        if (PS == 'b'){
                           cout << "\nEnter strike price K: ";
                           cin >> K;
                        }
                    }
                    break;

            case 'b': while (validUD != 1){
                        cout << "Would you like to price a\n(a) " <<
                                "Down-and-in,\n(b) Down-and-out,\n(c)" <<
```

```
                        " Up-and-in or \n(d) Up-and-out\ncall option? \n";
                cin >> UD;
                cout << "\n";

                if (UD != 'a' && UD != 'b' && UD != 'c' && UD != 'd'){
                        cout << "Enter a, b, c or d\n";
                }else{
                        validUD = 1;
                }
        }

        cout << "\nEnter strike price K: ";
        cin >> K;

        if (UD == 'a' || UD == 'b'){
                while (validV != 1){
                        cout << "Please enter a barrier value v: ";
                        cin >> v;
                        if (v>=S){
                           cout << "v MUST be less than initial stock price!";
                        }else{
                           validV =1;
                        }//ifelse
                }//while
        }else{
                while (validV != 1){
                        cout << "Please enter a barrier value v: \n";
                        cin >> v;

                        if (v<=K){
                           cout << "v MUST be greater than the strike price K!";
                        }else{
                           validV =1;
                        }//ifelse
                }//while
        }
        break;

case 'c': while (validmM != 1) {
                cout << "Would you like to price (a) a maximum" <<
                        " lookback option or (b) a minimum lookback option? ";
                cin >> mM;
                cout << "\n";
                if (mM != 'a' && mM != 'b'){
                   cout << "You must pick (a) or (b)\n";
                }else{
                   validmM = 1;
                }
                if (mM == 'a') {
                   cout << "\nEnter strike price K: ";
                   cin >> K;
                }
```

```
                    }
                    break;

            default: break;

}//switch

/* The user has a choice of three variance reduction techniques:
     1) Control variates
     2) If he/she has chosen to price up/down-and-in option then conditional expectation offered
     3) A third option is that of antithetic variables which is available for all options */

/* The variance Reduction of control variates is offered and set at this stage */
char varRed1 = '\0';
bool validVR1=0;

/* If we are indeed going to be using control variates we are going to need a few extra variables
to be declared and set */
double bS = blackScholes(S,(double)t/N,S,sigma,r); // price of an at-the-money Black Scholes
double sumBSPayoffs = 0.0;
double BSPayoff;

/* Option of using control variates as means of reducing variance and increasing price efficiency,
in some cases use of variance reduction techniques can be costly in terms of simulation time */
while (validVR1 !=1) {
     cout << "\nWould you like to employ the control variate variance reduction method (Enter 'y' or 'n
     cin >> varRed1;
     cout << "\n";
     if (varRed1 != 'y' && varRed1 != 'n'){
        cout << "Please enter either the character (y) or (n) \n";
     }else{
        validVR1 = 1;
     }
}

char varRedAlt = '\0';
bool validVRA=0;

while (validVRA != 1){
     cout << "\nAntithetic variables for variance reduction? Note this method " <<
             "may be used on any option: ";
     cin >> varRedAlt;
     cout << '\n';

     if (varRedAlt != 'y' && varRedAlt != 'n'){
        cout << "Please enter either the character 'y' or 'n' \n";
     }else{
        validVRA = 1;
     }
}

char varRed2 = 'n';
```

```
bool validVR2=0;

/* NB: in some cases use of variance reduction techniques is costly in terms of simulation time. */
if (UD == 'a' || UD == 'c'){
   while (validVR2 != 1){
      if (UD == 'a' || UD == 'c'){
         cout << "\nEmploy conditional expectation variance reduction (Enter 'y' or 'n')? ";
         cin >> varRed2;
         cout << "\n";
      }

      if (varRed2 != 'y' && varRed2 != 'n'){
         cout << "Please enter either the character (y) or (n) \n";
      }else{
         validVR2 = 1;
      }
   }
}


double *payoff = new double[noSims], // payoffs of ALL runs, used in control variance reduction
       *runPayoff = new double[2], // payoff from single simulation run
       *temp = new double[2]; // temporary placeholder for payoff given antithetic stockprices

if(varRedAlt == 'y'){
   /* As soon as we know that antithetic variables are going to be used we can build the sequence
   of negatively correlated normal variables */
   double anti = 2*(r - pow(sigma,2)/2)/N;
   for(int r = 0; r < numSims; r++){
          antiNorms[r] = anti - normals[r];
   }
}

for(int i=0;i<noSims;i++){

        /* Initialise the first entry of our array of lognormal stock prices */
        stockPrices[0] = S*exp(normals[i*t]);
        if(varRedAlt == 'y'){
           antiStockPrices[0] = S*exp(antiNorms[i*t]);
        }
        /* We build our sequence of prices */
        for (int x=1; x<t; x++){
            stockPrices[x]= stockPrices[x-1]*exp(normals[i*t + x]);
            if(varRedAlt == 'y'){
               antiStockPrices[x] = antiStockPrices[x-1]*exp(antiNorms[i*t + x]);
            }
        }

        /* User should have selected which type of exotic option they want to price */
        switch(abc){
                    case 'a': runPayoff[0] = asiaCall(stockPrices,t,K,r,PS);
```

```
            if (varRedAlt == 'y'){
               temp[0] = asiaCall(antiStockPrices,t,K,r,PS);
               runPayoff[0] = (temp[0] + runPayoff[0])/2;
            }

            if (varRed1 == 'y'){
               sumBSPayoffs += callPayoff(stockPrices[t-1],S);
            }

            payoff[i] = runPayoff[0];
            sumPayoffs += payoff[i];
            break;
case 'b': runPayoff = barrier(stockPrices,t,K,sigma,r,S,v,UD,varRed2);

            if (varRedAlt == 'y' || varRed2 == 'y'){
               if (varRedAlt == 'y'){
                  temp = barrier(antiStockPrices,t,K,sigma,r,S,v,UD,varRed2);
                  runPayoff[0] = ( temp[0] + runPayoff[0] )/2;
                  runPayoff[1] = ( temp[1] + runPayoff[1] )/2;
                  if (varRed2 == 'y'){
                     payoff[i] = runPayoff[1];
                  }else{
                     payoff[i] = runPayoff[0];
                  }
               }else{
                  payoff[i] = runPayoff[1];
               }
            }else{
               payoff[i] = runPayoff[0];
            }

            if (varRed1 == 'y'){
               sumBSPayoffs += callPayoff(stockPrices[t-1],S);
            }

            sumPayoffs += payoff[i];
            break;
case 'c': runPayoff[0] = lookback(stockPrices,t,K,r,mM);
            if (varRedAlt == 'y'){
               temp[0] = lookback(antiStockPrices,t,K,r,mM);
               runPayoff[0] = (temp[0] + runPayoff[0])/2;
            }

            if (varRed1 == 'y'){
               sumBSPayoffs += callPayoff(stockPrices[t-1],S);
            }

            payoff[i] = runPayoff[0];
            sumPayoffs += payoff[i];
            break;
Default:    cout << "Error! Select (a), (b) or (c)";
            system("EXIT");
```

```
                                    break;
            }//switch
        }//for

        price[0] = sumPayoffs/noSims;

        /* If the user has chosen to use control variate reduction the following will be needed */
        BSPayoff = exp(-r*t/N)*sumBSPayoffs/noSims;

        if (varRed1 == 'y'){
           price[0] + (bS - BSPayoff);
        }

        price[1] = variance(payoff,noSims);

        delete [] normals;
        delete [] stockPrices;

        /*Finally, return Monte Carlo simulated price of option along with the variance of the simulation*/

        return price;
    }//exotic
    #include <cstdlib>
    #include <iostream>
    #include <math.h>
    using namespace std;
    #include "stdNorm.h"
    #define N 252
    /* Function which returns the Monte Carlo simulated price of an exotic asian Option
    whose payoff is either:
          1) difference between the average of all prices leading up to
          the expiry date and a strike price K -> (mean(St) - K)+ or,
          2) difference between terminal price ST and the average of all
          prices leading up to the expiry date -> (ST - mean(St))+
    It also returns the sum of the stock prices of the run for use later if one opts to
    incorporate control variates as well */
    double asiaCall(double *stock, int t, double K, double r, char PS){

        double payoff=0;
        double sum=0, arithAvg=0, ST;

        /* When dealing with an Asian call option the relevant prices are the end of day ones */
        for (int x=0 ; x < t ; x++ ){
            sum += stock[x];
        }//for

        /* This represents the sequences terminal price */
        ST = stock[t-1];
        /*Arithmetic average of stock prices*/
        arithAvg = sum/t;
        if (PS == 'b' && arithAvg > K){ // floating price Asian call option
           payoff = exp(-r*t/N)*(arithAvg - K);
```

```
    }

    if (PS == 'a' && ST > arithAvg){ // floating strike Asian call option
       payoff = exp(-r*t/N)*(ST - arithAvg);
    }

    return payoff;
}//asiaCall
#include <cstdlib>
#include <iostream>
#include <math.h>
using namespace std;
#include "stdNorm.h"
#include "blackScholes.h"
#define N 252
/* Function which returns Monte Carlo price of barrier options whose payoff is same as a
Euro call option conditional on where barrier v is breached or not. Should be noted that variance
reduction technique used does indeed reduce variance but does not reap added intended benefit of reduced
total number of normal random variables generated because we generate entire sequence
regardless of option type beforehand */
double *barrier(double *stock, int t, double K, double sigma, double r, double s, double v, char UD,
                char varRed){
    double *payoff = new double[2];

    /* Default value is -1 for payoff[1] to indicate that variance reduction was not selected */
    payoff[1] = -1;
    /*Default value for our non-variance reduced payoff is nought */
    payoff[0] = 0;

    /* For this variance reduction we need flag T to keep track of where exactly option breaches
    barrier to become alive */
    int T = 0;
    /* 'I' below acts as an indicator to let us know if the barrier has been breached */
    int I;

    if(UD == 'a' || UD == 'c'){ I = 0;} else { I = 1;} // Options begin dead or alive

    if (UD == 'a' || UD == 'b'){

       /* We check to see if the barrier v is breached.
       If user has selected (a) then the option 'comes to life'
       otherwise it faces death */
       for (int x=0 ; x < t ; x++){
           T++;
           if (stock[x] < v){
              I = (int)pow(I-1,2);
              break;
           }//if
       }//for
    }else{

       /* We check to see if the barrier v is breached.
```

```
        If user has selected (c) then the option 'comes to life'
        otherwise it faces death */
        for (int x=0 ; x < t ; x++){
            T++;
            if (stock[x] > v){
               I = (int)pow(I-1,2);
               break;
            }//if
        }//for

    }//ifelse

    if (varRed == 'y'){
       payoff[1] = I*blackScholes(stock[T-1],
                                  (double)(t-T)/N, K, sigma, r);
    }

    if (stock[t-1] > K){
       payoff[0] = exp(-r*t/N)*I*(stock[t-1] - K);
    }
    return payoff;
}//barrier
#include <cstdlib>
#include <iostream>
#include <math.h>
using namespace std;
#include "stdNorm.h"
#define N 252
/*Function which returns the Monte Carlo simulated price of a lookback
  option whose payoff is
  1)Difference between the maximum end-of-day  price and a strike price K -> (max(St) - K)+ or,
  2)Difference between stocks closing day price and minimum end-of-day price -> (ST - min(St))+ */
double lookback(double *stock, int t, double K, double r, char mM){
    double payoff = 0, ST;

    double max = stock[0];
    double min = stock[0];

    switch(mM){
              case 'a': for (int x = 0; x < t; x++){ // Maximal lookback option
                           if(stock[x] > max){
                              max = stock[x];
                           }
                        }
                        if (max > K){
                           payoff = exp(-r*t/N)*(max - K);
                        }//if
                        break;

              case 'b': for (int x = 0; x < t; x++){ // Minimal lookback option
                           if(stock[x] < min){
                              min = stock[x];
```

```
                        }
                    }

                    /* The following is the stocks terminal price */
                    ST = stock[t-1];

                    if (ST > min){
                       payoff = exp(-r*t/N)*(ST - min);
                    }//if
                    break;

            default:  break;
    }//switch
    return payoff;
}//lookback
```