

Final Year Project Report

Full Unit - Final Report

Value at Risk

Nishant Mittal

A report submitted in part fulfilment of the degree of

BSc. Computer Science

Supervisor: Professor V. Vovk



Department of Computer Science
Royal Holloway, University of London

March 26, 2014

Declaration

This report has been prepared on the basis of my own work. Where other published and unpublished source materials have been used, these have been acknowledged.

Word Count: approx. 20,000

Student Name: Nishant Mittal

Date of Submission: 26/03/2014

Signature:

Table of Contents

Abstract	5
1 Introduction	6
1.1 Project Goals	7
2 Theory	8
2.1 Concept of Value at Risk	8
2.2 Literature Review	8
2.3 Volatility	9
2.4 VaR Measure	14
2.5 Model Building	14
2.6 Historical Simulation	15
2.7 Monte Carlo Simulation	16
2.8 Comparison of VaR Estimation Approaches	19
3 Options	20
3.1 Black–Scholes Option Pricing Formula	21
3.2 Binomial Tree Option Pricing Model	21
3.3 Monte Carlo Option Pricing	25
3.4 Historical Simulation	25
3.5 Stochastic Volatility Models	25
4 Criticisms of Value at Risk	26
5 Model Testing	27
5.1 Backtesting	27
5.2 Stress Testing	28
5.3 p-Values	30
6 Program Design	32

6.1	Initial Command Line Interface	32
6.2	Graphical User Interface	32
6.3	Usability	35
7	Software Engineering	45
7.1	UML Class Diagram and Class Descriptions	45
7.2	Role of Classes in Program	46
7.3	Implementation of Algorithms	54
7.4	Agile Development	54
7.5	Good code design	55
7.6	Unit Testing	56
7.7	Software versioning and revision control	56
7.8	MVC Design Pattern	57
7.9	Facade and Singleton Design Pattern	59
8	Professional Issues	61
8.1	Plagiarism	61
8.2	Usability	62
9	Technologies Used	63
9.1	Java Programming Language	63
9.2	Development of Code	63
9.3	Knowledge From Previous Courses	64
10	Evaluation	65
10.1	Experience of Project	65
10.2	Achievements	66
10.3	Enhancements	66
10.4	Conclusion	67
	Appendices	68
A	Program Usage Instructions	69
A.1	Requirements	69
A.2	Launching the program	69
A.3	User Manual	70

B	Program Listing	80
B.1	EWMA	80
B.2	VaR for multiple assets using Monte Carlo Simulation	81
B.3	Option pricing using Historical Simulation	83
C	Backtesting Results	85
C.1	Using GARCH(1,1) to estimate volatility of returns	85
C.2	Using Standard Formula to estimate volatility of returns	86
C.3	Using EWMA to estimate volatility of returns	86
	Bibliography	88

Abstract

The report outlines the journey of the implementation of a **Value at Risk** Final Year Project. It covers the concept of Value at Risk, its uses in the financial world and the various theoretical components which can be implemented in a program to compute the Value at Risk of a portfolio of assets and options. This report also contains information about the tools used in the implementation of the program and the software engineering practices followed in order to keep the source code easy to maintain, modify and update. Professional issues concerned with the project are discussed alongside a reflection on how the project progressed from start to finish.

The **Value at Risk** project involved writing a program which would take in some data about a set of stocks and options, the portfolio, and would tell the user how much of their investment is at risk of loss over the next few days. Knowing the amount of risk in an investment is key to making decisions about what to do with the assets you hold in the financial world. Therefore, Value at Risk has many uses and applications across the financial industry.

The implementation of the program was rather enjoyable as the individual components were small enough to be written within a day but came together to form a useful piece of software.

The theory behind Value at Risk was sourced mainly from a book [1] and supplemented with information from online resources which are listed in the bibliography section 10.4 of this document.

Chapter 1: Introduction

As part of the final year of my Computer Science degree, I've decided to embark on writing a program to compute the Value at Risk of a portfolio consisting of an arbitrary number of stocks and options. I chose to do the Value at Risk project because I felt it would challenge my programming skills, allow me to use skills and knowledge acquired from the rest of my degree and complement my studies in the Computational Finance course.

During the implementation of the program I had the opportunity to learn new theory from various sources about how Value at Risk is calculated for different assets and used well-known industry standard tools with good software engineering processes to build up a series of algorithms and functions which come together to form the whole project.

After my Year in Industry, I was looking forward to continuing the cycle of regular software development and applying my experience and knowledge from my work experience to my project. It was a good experience for me to work individually on this project as it involved discipline, dedication and honesty about my work which are good values to take into future employment. I was able to make use of well-known libraries and resources to aid my software development as well as a lot of knowledge from previous modules undertaken on my degree.

I feel that having the opportunity to work on my own project in which I had an interest, being able to use the knowledge and experience from past courses and working in industry respectively and having a supervisor who is experienced and knowledgeable in the subject of this project made my final year project a useful and enjoyable experience.

This report starts with the theory behind Value at Risk (VaR) which includes an explanation of the VaR measure and the key components which form its calculation. Different approaches to estimating VaR are then described and compared. The theory also contains a description of option contracts and how they can be valued alongside assets to estimate the overall VaR of a portfolio containing assets and options. This section ends with a discussion about some criticisms of VaR from industry experts.

Following the theory behind VaR, the next chapter describes how a VaR model can be tested for reliability using Backtesting and Stress testing techniques and discusses the results from these techniques being applied to the VaR model constructed for this project.

The report also discusses the design of the program associated with this project and the software engineering aspects of implementing that program. There is an explanation of the individual program classes which form the graphical program and the thinking behind some of the technical decisions made during its implementation.

The report ends with a discussion on the professional issues concerned with this project, the technologies used when implementing the software for this project and an evaluation of the project experience from the student's perspective.

The appendices contain a copy of the user manual for the VaR estimating program and some source code listing which is referred to from within some of the theory sections.

Finally, a bibliography of all referenced sources of information is also present.

1.1 Project Goals

Aims: Write a program for computing Value at Risk. Check its performance using backtesting.

Background: It is a very difficult and important problem to estimate the riskiness of a portfolio of securities. Value at Risk (VaR) is a tool for measuring financial risk. The goal of this project is to implement some of the methodologies for VaR computation and test it on real financial data.

Early Deliverables

- Simple proof-of-concept program for computing Value at Risk for a portfolio consisting of 1 or 2 stocks using two different methods: model-building and historical simulation.
- Different ways of computing Value at Risk will be back tested and the statistical significance of the results analyzed.
- The report will describe the program (software engineering, algorithms etc.).
- The report will also discuss different ways of estimating the variance of returns and the covariance between returns of two different securities (using the standard formula, exponentially weighted moving average, or GARCH(1,1)).

Final Deliverables

- The program should be extended by allowing derivatives (such as options) in the portfolio and using Monte Carlo simulation.
- Allow an arbitrary number of stocks (using Cholesky decomposition).
- The final program will have a full object-oriented design, with a full implementation life cycle using modern software engineering principles.
- Ideally, it will have a graphical user interface.
- The final report will describe: the theory behind the algorithms.
- The final report will describe: the implementation issues necessary to apply the theory.
- The final report will describe: the software engineering process involved in generating your software.
- The final report will describe: computational experiments with different data sets, methods, and parameters.

Chapter 2: Theory

2.1 Concept of Value at Risk

Individual traders, along with large financial institutions, often invest their money in various assets, such as stocks and options. A collection of these assets is called a portfolio. If someone is investing a large sum of money in a particular asset, or collection of assets, it is useful to know how much risk is associated with such an investment.

Financial institutions usually calculate risks for every market variable to which they are exposed.[1] However, to provide a way of measuring the total risk the financial institution is exposed to, Value at Risk is considered a better solution.

Value at Risk (VaR), in essence, is a single number representing the total risk in a portfolio of financial assets. It is also used by treasurers of large corporations, fund managers and bank regulators to estimate the amount of risk in the portfolios they hold. In banking, VaR is used to decide how much money a bank should keep (a reserve) for the risks it is exposed to, in case there is a financial crisis and the bank needs to be able to pay off some of the money it holds from its investors.

It is basically answering the question: “What is the maximum probable amount of money that can be lost if I have X amount of a stock over the next N days?”

One advantage of using VaR as a measure of the risk in a portfolio is that it simplifies the risk as a single number which can be understood without the need for extensive knowledge about how it is calculated. However, this means that by decomposing the amount of risk a portfolio is exposed to into a single number, complicated aspects of risk which may have been important to know may be missed out.

2.2 Literature Review

The main source of information for the theory behind Value at Risk is the book titled *Options, Futures and Other Derivatives* by John Hull. Hull covers a wide variety of topics relevant to this project in his book, ranging from discussions about various methods of computing VaR to estimating volatility and covariances between sets of assets. Hull also covers methods behind pricing options using various techniques, which is also something concerning the VaR project goals. Hull discusses some aspects of verifying VaR models through Backtesting and stress testing. It is highly likely that his book may form a large basis of the new material learnt in order for this project to be completed as it is one of the recommended books in the specification of the Value at Risk project and is likely to be more reliable and accurate than sources of information on the internet.

Aside from the information relevant to the project, other aspects of the financial world such as the mechanics of the option markets and hedging against risk using futures and options should make for enjoyable reading.

Supplementary information is also available from the CS3930 Computational Finance course which covers the basics of pricing options, estimating volatilities from returns of an asset and some discussion of Value at Risk.

There are plenty of resources relevant to this project available online from websites such as Gloria-Mundi, which has a large selection of relevant papers accessible through its portal, and Investopedia, which provides helpful descriptions of financial terms and has some very informative tutorials and articles about Value at Risk and its components. These online resources should be useful if the book mentioned above does not quite provide the level of understanding desired for a certain topic or if some updated piece of information is required from the web as supplement to the knowledge from the book.

Another potential source of information is a book by Philip Best titled *Implementing Value at Risk*. This book covers a lot of the topics concerned with the goals of this project and should prove especially helpful in understanding the software implementation for calculating VaR.

From research on the internet, it seems that open-source or free to use systems which calculate Value at Risk are very rare. Many solutions, come as plug-ins for Microsoft Excel. This includes products such as PortfolioScience's RiskAPI Add-In (available at <http://www.portfolioscience.com/products/riskapi-add-in>) and Hoadley Trading & Investment Tools' VaRtools Excel Add-in (available at <http://www.hoadley.net/options/develtoolsvar.htm>) which suggests that Excel is being used to keep track of traders' portfolio performance in the market and to estimate the risk involved in the portfolios they hold.

Most of the Value at Risk (VaR) calculating products contain a similar range of features, such as the VaR measure itself, Conditional Value at Risk (CVaR), Monte Carlo simulation of VaR, VaR for options and other derivatives, Historical Simulation of VaR along with underlying features to estimate volatilities, covariance and conduct stress testing and scenario analysis.

PortfolioScience also offers a Java Enterprise Client which "offers simple, standardized access to the RiskAPI service via a jar-packaged Java object which exposes native Java language methods and parameters."

Despite there being clear evidence of Excel being using as a risk management solution, it would be much easier for a user to compute the VaR of their portfolio in a native, special purpose solution. Hence, there may be a gap in the market for such a product, especially if it is open-source and freely available to use.

2.3 Volatility

One of the most important parameters in the VaR estimation is the volatility of the assets in the portfolio. Risk is greatly reduced if the asset has a stable market price, regardless of whether the price goes up or down. Estimating the volatility of the asset, by looking at the historical stock price data, is usually one of the first steps in estimating VaR and thus it is important that reliable and accurate techniques are used to do this.

Volatility is an important aspect of estimating VaR. It is the measure of the amount of fluctuation in the price of the stock over a period of time. For a highly volatile stock, there is high risk and for a stock with low volatility there is low risk of keeping it. The closing prices over a set period of time are used to calculate the volatility of a stock in different ways.

Volatility can be measured from historical stock price data, known as historical volatility, and from the stock market itself, known as implied volatility. The following sections discuss the methods behind measuring historical volatility.

2.3.1 Standard formula

The standard formula is the simplest way of calculating the volatility of a stock. It involves comparing the closing prices from a period of time, over two consecutive days for a simple example, and simply looking at the difference between them - known as the return. The return of the stock price can be calculated by:

$$\text{return}_i = \frac{\text{closing price}_i - \text{closing price}_{i-1}}{\text{closing price}_i}$$

where i is used to keep track of the day we are working out the return for. Using this calculation, a series of returns can be generated over the days we have stock price data for. The volatility of the stock will be the standard deviation of the range of returns over the specified time period.

While this is an easy of estimating the volatility of a stock, it is not the most reliable. A slight improvement on this calculation can be made by using natural logs to work out the returns from the stock prices. We modify the calculation to be:

$$\text{return}_i = \ln \left(\frac{\text{closing price}_i}{\text{closing price}_{i-1}} \right)$$

which gives us a more convenient and slightly more reliable result. Using natural logs is considered to be the convention in finance. One of the benefits of using natural logs to work out the returns of stock prices is that logs are time-consistent i.e. we get the same result when calculating the change either over a collection of smaller time periods or over a larger time period of the same collective size as the first one. Moreover, this calculation is better at approximating the changes when they are relatively low (when changes in stock prices aren't over 15%).

The volatility of the stock in this case would be the standard deviation of the series of returns over the given time period and can be generalised in one formula:

$$\sigma_n^2 = \frac{1}{m} \sum_{i=1}^n u_{n-i}^2$$

where σ_n^2 is the variance at day n , m is the number of days of stock data we have and u_{n-i}^2 is the square of the return between days n and i . The volatility, σ_n at day n would be the square root of σ_n^2 .

The standard formula for calculating the volatility of a stock assumes equal weighting of each return. However it is accepted that more recent data about a stock's performance is generally a better indicator of its volatility than older data. Hence, there are still improvements which can be made on the estimation of the volatility of a stock price, such as using the EWMA formula.

2.3.2 EWMA - Exponentially Weighted Moving Average

While the standard formula is useful for computing the volatility of a stock, it doesn't take into account the validity and reliability of the closing prices with respect to time. The EWMA approach to calculating volatility gives more weight to the recent changes in stock prices than the older stock prices as it considers the recent data a better indicator of the performance of a stock than older data about the stock. This seems to be a valid argument and, theoretically,

results in better estimations of the volatility of a stock.

The EWMA approach requires four inputs and can calculate volatility for an infinite number of returns. The formula for the EWMA calculation for variance is:

$$\sigma_n^2 = \lambda \sigma_{n-1}^2 + (1 - \lambda) r_{n-1}^2 \quad (2.1)$$

where,

σ = volatility, σ^2 = variance

λ = exponential weight coefficient, determines how returns are weighted (between 0 and 1)

r = periodic return

n = day

This formula computes the variance, using data from all prior returns, for the current day which can then be used to compute the volatility of the stock's returns as $\sqrt{\text{variance}}$.

λ is effectively a decaying factor which specifies at which rate the weight given to a return declines as it gets further away from the most recent day. J.P. Morgan used a value of $\lambda = 0.94$ in their RiskMetrics database[1] which gave forecasts of the variance rate closest to the realised variance rate across a range of market variables[8]. A number of parameter estimation methods do exist which aim to optimise the λ value in the EWMA model for a given series of returns. These methods will be discussed in section 2.3.5.

Applying $\lambda = 0.94$ to the previous day's variance and applying $(1 - \lambda)$ to the previous day's return effectively reduces its weight by 6% ($1 - 0.94$). The overall result of this is that the most recent return has a weight of 6%, the next most recent is weighted at $6 \times 0.94 = 5.64\%$ and the next most recent return is weighted at 5.3% and so on.

If there is a large fluctuation in the the stock price on day $(n - 1)$ resulting in r_{n-1}^2 being large, the estimate of the current volatility will increase, from equation (2.1).

A higher λ (close to 1) will result in a slower decay in the weighting of previous returns whereas a lower λ will result in faster decay of weighting. By reducing the λ value, it is possible to reduce the number of days of historical price data required as the returns in the past will start to have a very low weighting much sooner than if the λ value was higher.

In terms of software implementation, the EWMA formula requires a recursive function as described in the program listing in appendix B.1. The implementation required very little data to be stored as only the current estimate of the variance and the most recent return had to be remembered by the recursive function. The old estimate of the variance and the old returns can be discarded when a new value for both is obtained. As the implementation of the EWMA function in this project was passed a series of returns anyway, it was able to move along the series without having to store any previous values of the returns.

As the EWMA approach to calculating volatility gives more weight to recent returns than older returns, it heavily distorts the long-run variance when recent stock prices are highly volatile. The long-run variance should ideally be given more weight as stock prices tend to fluctuate around a certain value from observation. Therefore, EWMA is not considered ideal for estimating the volatility of a stock.

2.3.3 GARCH(1,1)

Another model which can be used to estimate the variance of a stock's returns is the GARCH(1,1) model. It takes into account the long-run average variance rate (something

not considered in the EWMA approach) as well as the previous estimates of variance and the previous returns.

The “(1,1)” indicates that the variance estimate is based on the most recent observation of r^2 (the return) and the most recent estimate of the variance. A more general model GARCH(p, q) calculates the variance from the p most recent returns and q most recent estimates of the variance rates. However, GARCH(1,1) is the most popular of the GARCH models [1].

Generalised Auto-Regressive Conditional Heteroscedastic, its full name, further improves on the volatility calculation made by using EWMA through a technique known as *mean reversion*.

Mean reversion essentially limits the influence of larger fluctuations which can distort the mean of a range of data. The result of applying mean recursion is that the volatility estimates pull the variance closer to the long running average variance, an operation collectively known as *persistence*, so it gives more realistic estimates of volatility than EWMA and is the most popular way of calculating volatility of a stock price.

The recursive formula for GARCH is:

$$\sigma_n^2 = \gamma V_l + \alpha r_{n-1}^2 + \beta \sigma_{n-1}^2$$

where,

σ_n^2 = variance on day n ,

γ = weight given to long-run average variance

V_l = long-run average variance

α = weight given to return on day $n - 1$

r_{n-1}^2 = squared return from day $n - 1$

β = weight given to variance from day $n - 1$

σ_{n-1}^2 = variance estimated on day $n - 1$

Similar to the EWMA model of estimating variance, β acts a ‘decaying’ factor in terms of the weight assigned to previous variances. However, as variance rates tend to be mean reverting[1], the Garch(1,1) model uses γ to assign a weight to the long run average variance in order to revert the current estimation of σ^2 towards V_l .

Thus, the GARCH(1,1) model is considered to be the best method of estimating the variance of a series of returns. It also assumes that the volatility changes with the passage of time [1].

As with EWMA, the choice of parameters is important for the GARCH(1,1) model. Further discussion on the parameter values for GARCH(1,1) is present in section 2.3.5.

Daily and annual volatility All the approaches above gave a measure of volatility when calculated on a daily basis. Annual volatility is some times used when pricing stocks and options and can be easily calculated by extending it over the approximate number of trading days in a year, 252.

$$\sigma_{\text{annual}} = \sigma_{\text{daily}} \times \sqrt{252}$$

and, in general, to obtain the N -day volatility:

$$\sigma_N = \sigma_{\text{daily}} \times \sqrt{N}$$

2.3.4 Differences in variance calculation

The variance calculating approaches outlined earlier result in different estimations of variance for the same series of data. The implementations of each of the three variance estimation algorithms were tested on the stock price data contained in the following series.

Date	Close
18/10/2013	1011.41
17/10/2013	888.79
16/10/2013	898.03
15/10/2013	882.01
14/10/2013	876.11
11/10/2013	871.99
10/10/2013	868.24
09/10/2013	855.86
08/10/2013	853.67
07/10/2013	865.74
04/10/2013	872.35
03/10/2013	876.09
02/10/2013	887.99
01/10/2013	887.00
30/09/2013	875.91
27/09/2013	876.39
26/09/2013	878.17
25/09/2013	877.23
24/09/2013	886.84
23/09/2013	886.50
20/09/2013	903.11
19/09/2013	898.39

Table 2.1: Historical stock prices of Google's shares, taken from <http://finance.yahoo.com/q/hp?s=GOOG+Historical+Prices>

For this series of data, the following daily volatilities (as percentages) were estimated:

Standard formula: 0.03 (3%)

EWMA: 0.06 (6%)

The volatility estimate here is larger due to the sudden rise in the stock price between the second and first day. If that rise were not to occur the EWMA model would have estimated the volatility to be 0.05, which is closer to the standard formula but larger due to the fluctuations in the stock price towards the top end of the series.

GARCH(1,1): 0.04 (4%)

The volatility estimate from the GARCH model lies between the Standard formula estimate and EWMA. This is expected as GARCH does place some weight on recent fluctuations in the stock price but also reverts the variance of the data back towards the long term average variance. With the large fluctuation at the top taken out of the series, this model still estimated the volatility to be 0.037 which is a 0.003 reduction from the original estimate, thus proving that the GARCH model does respond to sudden changes in data but not as dramatically as the EWMA model.

If a VaR model used the standard formula to estimate the volatilities of the assets in the portfolio, it will be very likely to underestimate the VaR for that portfolio. If an EWMA model was used to estimate the volatility of the assets then recent changes in the price would contribute highly to the resulting volatility, possibly resulting in an overestimation of VaR. The GARCH(1,1) model, therefore, improves on both of these models and provides the best possible estimate of an asset's volatility and, therefore, a more reliable estimation of the VaR of a portfolio.

2.3.5 Parameter estimation for variance models

In order for volatility estimation models, such as EWMA and GARCH(1,1), to accurately estimate the volatility of a given series of data it is important to find the right values of the parameters contained within them. One of the methods used to calculate the optimal parameters for a volatility model is known as the *maximum likelihood method*. This method involves choosing values for the parameters that maximise the likelihood of the data occurring. [1][p. 475]

The maximum likelihood method can be used to estimate the parameters of the GARCH(1,1) model by first defining $v_i = \sigma_i^2$ as the variance estimated for day i . It is assumed that the probability distribution of the returns on day i , u_i conditional on the variance is normal. Combining this with the maximum likelihood method yields that the optimal parameters will maximise the value of [1][p. 476]

$$\sum_{i=1}^m \left[-\ln(v_i) - \frac{u_i^2}{v_i} \right] \quad (2.2)$$

Searching for the optimal parameters is an iterative process using trial estimates to calculate the value of function 2.2 and selecting the parameters yielding the largest value.

Alternative methods To estimate the optimal parameters for such models it is also possible to use a general purpose algorithms such as Microsoft Excel's Solver, as well as special purpose algorithms such as Levenberg-Marquardt.

2.4 VaR Measure

When computing VaR, what we are really trying to obtain is an idea of the maximum amount of loss probable over a certain time period and a confidence level in that estimate. VaR is simply a function of the time period in days and the confidence level (in percentage, with 100 being absolutely certain). Banks usually calculate VaR over a period of 10 days with the confidence level being 99%. [1]

By estimating VaR with 99% confidence, we are saying that there is only a 1% chance of the actual loss being greater than the VaR estimate.

Over a time period of N days and a confidence level of $X\%$, VaR is the loss corresponding to the $(100-X)$ th percentile of the distribution of the change in value of the portfolio. [1]

There are three main ways of computing VaR: Model Building, Historical Simulation and Monte Carlo Simulation.

2.5 Model Building

Computing VaR using the Model-Building approach is a very simple process. The function needs to know only four things in order to estimate VaR:

1. value of the portfolio

2. time period (in days)
3. confidence level (as a percentage of 100)
4. (daily) volatility of the stocks in the portfolio

In this approach, it is assumed that the change in value of the portfolio ΔP is linearly dependent on percentage changes in value of the stocks.

Calculating VaR for a single asset is straightforward. We first obtain the standard deviation of the daily changes in stock value by multiplying the volatility by the value of the portfolio (1). The one day VaR would then be the product of the standard deviation (1) and 2.33 (number of standard deviations our risk shall never exceed for 99% confidence, for example).

VaR, in the first instance, is usually measured with the time period as one day. The reason for this is that there simply is not sufficient amounts of data to estimate the behaviour of market variables over time periods of greater than a day. After computing VaR for one day, it is accepted that the VaR for N days is given by multiplying the one day VaR by \sqrt{N} .

While this seems simple, VaR is more useful when being computed for a portfolio of a large number of varying stocks. This is where, to obtain a reliable value of VaR, we must take into consideration the covariance between two or more shares, as discussed on page 17.

2.6 Historical Simulation

Historical simulation involves holding a record of data about the change in value of all stocks over a period of time. Each i^{th} simulation assumes that the changes in value of each market variable are same as those covered on day i in the database. The change in portfolio value, ΔP , is calculated for each simulation and the VaR is calculated from the appropriate percentile of the probably distribution of ΔP .

This approach places an emphasis on using past stock data as a guide to what might happen to the same stock in the future. To compute VaR using the historical simulation approach data about the stock's performance over a large enough period of time (at least a year) is desirable so that we can build an idea of how the stock's value has risen or dropped historically. Using the stock's historical performance, we can then simulate all the possible scenarios of what may happen to the stock's value between today and tomorrow. These scenarios correspond to the changes in the stock value between subsequent days over the time we have the data for.

Initially, we would calculate the daily changes in the stock price between every pair of consecutive days over the time period of the collected data. This would result in a probability distribution of the possible changes in the stock value. We can put these possible changes in the value of the stock in ascending order. This would make it easier for us to compute the Value at Risk because if we wished to obtain the VaR with 99% confidence, we would simply have to look at the 1st percentile of the ordered daily changes in data. For example, if we wished to compute the VaR with 95% confidence for a stock where we have 500 days of historical data, we would look at the 5th percentile i.e. the 25th worst loss and be able to say that the value of the portfolio will not decrease above this amount.

For each possible value of the stock in the future (e.g. tomorrow), we have the following

equation:

$$v_{\text{tomorrow}} = v_{\text{today}} \left(\frac{v_i}{v_{i-1}} \right) \quad (2.3)$$

where i denotes the i^{th} possible scenario for the change in value from all the possible scenarios in the historical stock data.

However, since the changes in value from $\left(\frac{v_i}{v_{i-1}} \right)$ can be simplified as the return over those two days, r_i , equation 2.3 can be understood as

$$v_{\text{tomorrow}} = v_{\text{today}} \times e^{r_i} \quad (2.4)$$

where, r_i is the return at the required percentile of all ordered historical returns.

The VaR can then be computed as the loss between the original investment value and the possible value from equation 2.4.

The approach outlined is useful for estimating the 1-day VaR based on historical stock price data. In order to estimate VaR over more than one day, it is necessary to make a small change. For the 1-day VaR, the series of returns is split up into a ‘windows’ covering returns over one day (individual returns). Thus, for the N -day VaR, we split up the series of returns of the asset into windows covering returns over N days.

For example, for an estimate of VaR over 5 days, a series of 100 returns (r_{1-100}) would be split up into returns over each 5 day period within the 100 day period.

e.g. $\{(r_{1-5}), (r_{2-6}), (r_{3-7}), \dots, (r_{96-100})\}$

Each of these returns will then represent a possible historical outcome from which VaR can be computed for a certain confidence. In the Value at Risk program for this project, both the final VaR and maximum VaR over the simulation are reported.

2.7 Monte Carlo Simulation

Another way of calculating the Value at Risk of a portfolio of stocks is by using the Monte Carlo simulation technique which models the changes in market variables as Gaussian distributions. A general Monte Carlo method relies on repeated random sampling of a set of data to obtain a range of results which can then be used to approximate the probability of some outcome.

In the scope of this project the Monte Carlo simulation is a method where we repeatedly simulate possible changes in the value of our portfolio using random variables in order to estimate a value by which the portfolio is at maximum $X\%$ likely to reduce in value - the Value at Risk.

The simulation is implemented by running a large number of hypothetical trials over a certain number of days using Gaussian random numbers in a formula such as:

$$S_{i+1} = S_i + \sigma S_i \times \phi(0,1)$$

where S_i is the value of the asset at day i , σ is the volatility and $\phi(0,1)$ is a Gaussian random variable.

After simulating these trials we will have a series of possible values of the asset, which can then be ordered. From the series of ordered returns of the asset, we are able to select a percentile based on the confidence level we have, from which we can deduce how much value of our asset is at risk i.e. for 95% confidence our VaR can be obtained by selecting the 5th percentile of the ordered returns of the asset.

In my implementation of Monte Carlo Simulation, I have computed both the final Value at Risk of the stock (from the values of the portfolio on the last day of each simulation) as well as the maximum Value at Risk of the stock which is the highest simulated risk experienced by the stock during the simulations.

In my program code, I simulate the prices of the stock for each of the days of the time period specified. While this is not usually necessary, the fact that I chose to compute the maximum VaR experienced by the portfolio during the simulations means it is required to simulate the price changes on a daily basis so that the lowest simulated prices can be found across the whole simulation for the maximum VaR calculation. Otherwise, it would be sufficient to simulate the price at the last day by using a larger variance for the stock (covering the time period between the current day and the last day of simulation).

2.7.1 VaR for multiple stocks

So far we have thought about computing the Value at Risk for a single asset. In reality, a portfolio consists of many different stocks which have different volatilities and hence different VaR measures.

Let us consider the situation where we have two assets in our portfolio and we compute their individual VaR. The risk of the individual assets tells us nothing about the risk of the total portfolio as sometimes the two assets' returns may balance each other out and reduce the risk. On the other hand, the returns may also vary at a high rate between the two assets and end up increasing the total risk of the portfolio. Therefore, we need to consider the correlation between the returns of the two assets, which are random variables, in our portfolio which is known as the covariance.

Covariance Covariance is a measure of how similar the changes in two random variables are. If the returns of one asset increase in the same fashion as the returns of another asset, they are said to have positive covariance. Negative covariance occurs when their behaviours are opposite to each other i.e. greater values of one asset's returns correspond to lower values of the other. In a financial context, knowing the covariance between stocks is useful for investors so that they can reduce the risk of their portfolio in a process known as diversification. The EWMA and GARCH approach to calculating variances can be extended to calculate covariances. This is done simply by replacing the square of one stock's returns with the product of the two stock's returns we are finding the covariance for. The formulas for the covariance computation using EWMA and GARCH thus become:

EWMA:

$$\sigma_n = \lambda\sigma_{n-1} + (1 - \lambda)r1_{n-1} \times r2_{n-1}$$

GARCH:

$$\sigma_n = \gamma V_l + \alpha r1_{n-1} \times r2_{n-1} + \beta\sigma_{n-1}$$

where σ_n is the covariance at day n , $r1_n$ and $r2_n$ are the respective returns of the two stocks.

Diversification Diversification is the concept of reducing the risk of a portfolio by investing in a variety of assets. A diverse portfolio consists of assets whose fluctuations in value balance each other out over time, thus reducing the total risk of the portfolio.

Once we have the series of returns (daily changes in a stock's closing prices) for a number of stocks, we can create a variance-covariance matrix which holds the values of the covariances between a set of stocks. The matrix consists of rows and columns representing each stock in the portfolio and the corresponding entries are the covariances between the stocks. We can model a sample covariance matrix for a portfolio of three stocks as follows:

$$\begin{pmatrix} cov(1,1) & cov(1,2) & cov(1,3) \\ cov(2,1) & cov(2,2) & cov(2,3) \\ cov(3,1) & cov(3,2) & cov(3,3) \end{pmatrix}$$

where $cov(i,j)$ represents the covariance between stock i and stock j and if $i == j$ then the covariance is simply the variance of stock i .

For each pair of stocks, the variance of their values is calculated by:

$$\text{variance}_{i,j} = \text{investment in stock}_i \times \text{investment in stock}_j \times cov_{i,j}$$

The sum of the variances in the matrix is the variance of our portfolio of assets. This variance can then be used to calculate the volatility of the portfolio and then the VaR of the whole portfolio, using the Model Building approach:

$$\text{VaR} = z\delta \times \sqrt{\text{variance}_{\text{portfolio}}} \times \sqrt{N}$$

where $z\delta$ is the number of standard deviations the loss should not exceed (approx. 2.33 for 99% confidence) and N is the number of days we wish to calculate VaR for.

In the Historical Simulation model, multiple stocks are relatively easy to account for when estimating VaR for a portfolio of different assets. The process simply involves building up a series of possible returns for each stock and storing the resulting portfolio value with each of those returns. These portfolio values can then be ordered and an appropriate percentile can be selected from the ordered series for VaR computation.

Computing VaR for multiple stocks using Monte Carlo Simulation requires more effort. The process can be summarised as:

1. generating a covariance matrix consisting of variances and covariances between each stock
2. decomposing the matrix into a triangular matrix using Cholesky decomposition
3. simulating a large number of uncorrelated random returns for each stock
4. multiplying the triangular matrix with the uncorrelated returns to obtain correlated returns for each stock
5. building up a probability distribution of possible values for each stock from its correlated returns and calculating the possible portfolio value as sum of these values
6. sorting the possible portfolio values and selecting an appropriate percentile for VaR computation

Source code for this method can be found in appendix B.2.

2.8 Comparison of VaR Estimation Approaches

The model-building approach produces results very quickly in comparison to historical simulation, however, it assumes that the market variables have a normal (Gaussian) distribution. In the real world, the distributions of the market variables are not normal. Model-building also gives relatively poor results for portfolios where the stock value changes are quite low.

The historical simulation does not require any estimation of volatilities of assets and lets historical data determine the probability distribution of the value of the portfolio [1]. However, it is still considered computationally slow due to the large amounts of historical data it must process to reliably estimate VaR.

One of the criticisms of the Monte Carlo simulation is that it can be very slow for large portfolios. In industry, a company is likely to have many portfolios containing a large number of assets, each of which have to be revalued many times [1]. This approach, therefore, is both computationally and temporally expensive.

Chapter 3: Options

The program also allows the user to compute the Value at Risk for portfolios containing options. An option is a type of contract which gives its owner a non-obligatory choice to buy or sell the asset it is associated with.

The seller of the option has an obligation to fulfil the contract (sell or buy underlying asset) if the owner of the option exercises it. Anyone wishing to hold an option pays a premium for it.

There are two types of options. A right to buy an asset at an agreed price is known as a Call option. On the other hand, a right to sell an asset at an agreed price is known as a Put option.

A number of terms are agreed between the holder of the asset and the person/entity wishing to buy the asset when an option contract is signed. These include the asset being traded, number of shares of the underlying asset, the selling price, known as the Strike of the option, type of option (Call or Put), annual interest rate and the expiration date of the option (maturity).

There are also two main styles of option contracts which differ in the way the option can be exercised. A European option can be exercised by the holder only at maturity. An American option, however, can be exercised by the holder at any time between agreeing the option and the maturity. A Bermudan option may be exercised only on specific dates on or before the maturity of the option.

Options can be traded either in a listed or unlisted manner. A listed option is one which is traded through an option Exchange, which would be a regulated financial institution. An unlisted option, also known as an 'Over-the-counter option' would comprise of a direct contract between a dealer and a buyer, with fulfilment of the option guaranteed by the Options Clearing Corporation (OCC). Unlisted options often allow for fewer restrictions on the terms included in the contract and can be tailored to meet the needs of the two participating entities.

When calculating the Value at Risk of a portfolio containing options, it is important to take into account how the value of an option changes with respect to almost every parameter it acts upon. The first step in valuing an option is finding its 'intrinsic value' at maturity i.e. the difference between the Strike price and the price of the underlying asset. The second step is to calculate the 'time value' of the option, which is dependant on a range of factors such as the type and style of option. The time value is the discounted expected value of the intrinsic value at maturity/exercise.

To value an option, it is assumed that the pricing model is risk-neutral, which means the interest rate specified in the option can be used as the dividend yield for the option and that the option price has an expected growth rate of zero. These assumptions allow for near-accurate valuations of options.

The value of an option depends on the current market price of the underlying asset, the Strike price (to deduce whether an option is profitable on exercise), the time to maturity/expiration and the estimated future volatility of the stock.

A number of different mathematical techniques are used to value different styles of options.

3.1 Black–Scholes Option Pricing Formula

One of the more known ways of valuing options is the Black-Scholes formula, which is used to find the value of European options. It was developed by Fischer Black and Myron Scholes and is derived from a set of differential equations which are satisfied by the price of the option dependant on an asset. It models a risk-neutral portfolio by replicating the returns of holding an option to obtain a theoretical price for the option. A number of hedging parameters can also be added to the formula for effective risk management.

The Black–Scholes formula consists of the following parameters:

- S - the price of the stock (starting price of the option)
- X - the strike price of the option
- r - the annual (risk-free) interest rate
- $(T - t)$ - the time to maturity of the option
- $N(x)$ - the cumulative normal distribution function, $\phi(0, 1)$
- σ - the volatility of the underlying asset

The Black–Scholes formula for pricing call and put options respectively is:

$$\begin{aligned} c(S, t) &= SN(d_1) - Xe^{-r(T-t)}N(d_2) \\ p(S, t) &= Xe^{-r(T-t)}N(-d_2) - SN(-d_1) \end{aligned}$$

where,

$$d_1 = \frac{\ln(S/X) + (r + \sigma^2/2)(T - t)}{\sigma\sqrt{T - t}},$$

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

While the initial uptake of the use of the formula was slow, traders soon realised that pricing options using the Black-Scholes formula yielded near-accurate results and so it became widely used for the valuation of European options. For their work in relation to the Black-Scholes option pricing model, Fischer Black and Myron Scholes received the Nobel prize for Achievement in Economics.

However, the Black–Scholes model assumes that the volatility of the underlying asset is constant [1], which is not accurate at all. As discussed on page 25, the volatility of an asset, along with its price, is a stochastic variable.

3.2 Binomial Tree Option Pricing Model

The binomial tree option pricing method models the changes in the underlying asset price over discrete time intervals over the life of the option. It relies on the construction of a riskless portfolio containing an option and the underlying stock.

Once the asset's initial market price and its volatility is known, the binomial tree method can be used to model the changes in the stock price, with the price on the start day, S_0 either rising to S_0u or falling to S_0d by some factor u or d determined by the volatility, σ over a time step t .

$$\begin{aligned} u &= e^{\sigma\sqrt{t}}, \\ d &= e^{-\sigma\sqrt{t}} = \frac{1}{u} \\ u &> 1, \quad d < 1 \end{aligned}$$

The binomial trees constructed are recombining, which means that following a sequence of the price going up and then down, the price at that node will be the same as if the price went down and then up, as the reduction and increasing factors complement each other, see figure 3.1 [1].

The following figures are based on examples contained in the Binomial Tree chapter in [1].

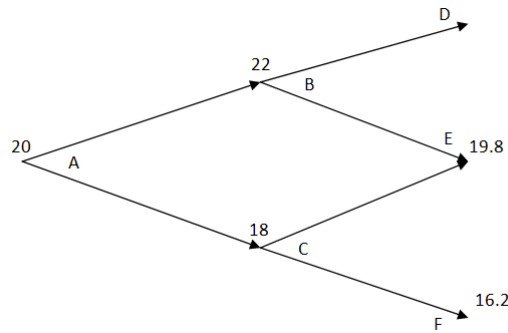


Figure 3.1: Binomial tree modelling price going up or down by 10%

The recombining property means that the value of the asset at a node, S_n can be calculated directly without building a tree first, by calculating the difference between the number of times the value goes up, N_u , and the number of times the value goes down, N_d , and then using the following equation:

$$S_n = S_0 \times u^{N_u - N_d}$$

Once the paths of the underlying stock price have been modelled, the option prices at maturity/expiration of the option are found. This is the 'intrinsic' value of the option and can be found by the following calculations:

For a call option: $\text{Max} [(S_n - X), 0]$

For a put option: $\text{Max} [(X - S_n), 0]$

where, S_n is the price of the asset at the n^{th} time step and X is the strike price of the option.

Once the option prices at maturity have been found, it is possible to work backwards from the maturity to the initial time step to find the binomial value, f , of the option at each node based on the future prices from that node. The binomial value can be thought of as the fair price of the option at a point in time or the value of option if it was held (not exercised).

$$f = e^{-rT}(pf_u + (1 - p)f_d)$$

where, r is the interest rate, T is the time to maturity, f_u is the binomial value of option going up from this node, f_d is the binomial value of option going down from this node, and,

$$p = \frac{e^{rT} - d}{u - d}$$

where, p is the probability of the value of the option going up and, therefore, $(1 - p)$ is probability of the value of the option going down.

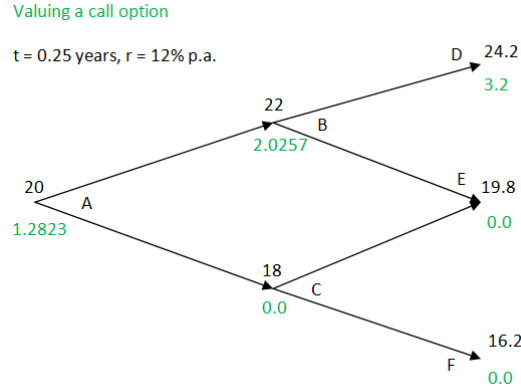


Figure 3.2: Valuing a call option

In figure 3.2, the value at node B can be obtained by calculating:

$$p = \frac{e^{0.12 \times 0.25} - 0.9}{1.1 - 0.9} = 0.6523$$

$$e^{-0.12 \times 0.25} (0.6523 \times 3.2 + (1 - 0.6523) \times 0) = 2.0257$$

And at node A:

$$e^{-0.12 \times 0.25} (0.6523 \times 2.0257 + (1 - 0.6523) \times 0) = 1.2823$$

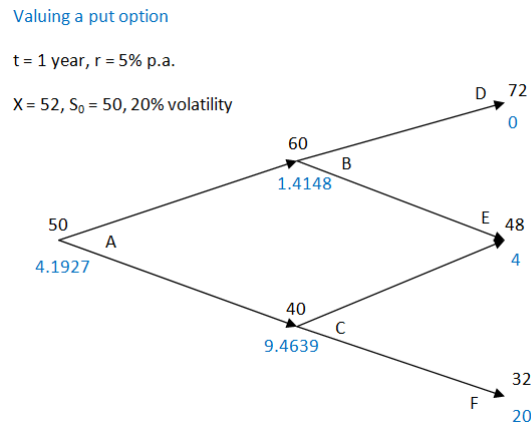


Figure 3.3: Valuing a put option

In figure 3.3, the value at node B is given by calculating:

$$p = \frac{e^{0.05 \times 1} - 0.8}{1.2 - 0.8} = 0.6282$$

$$e^{-0.05 \times 1}(0.6282 \times 0 + (1 - 0.6282) \times 4) = 1.4148$$

And at node C:

$$e^{-0.05 \times 1}(0.6282 \times 4 + (1 - 0.6282) \times 20) = 9.4639$$

And at node A:

$$e^{-0.05 \times 1}(0.6282 \times 1.4148 + (1 - 0.6282) \times 9.4639) = 4.1927$$

Once all the nodes have been assigned the calculated binomial value, it is possible to work out the value for different styles of options. If the option is European there is no possibility of it being exercised early so the binomial value is the value of the option at all of the nodes of the binomial tree.

If the option is American, it can be exercised before expiry. The value of an American option at a node before maturity is given by the greater of the binomial value and the intrinsic value. An example of valuing an American option can be seen in figure 3.4.

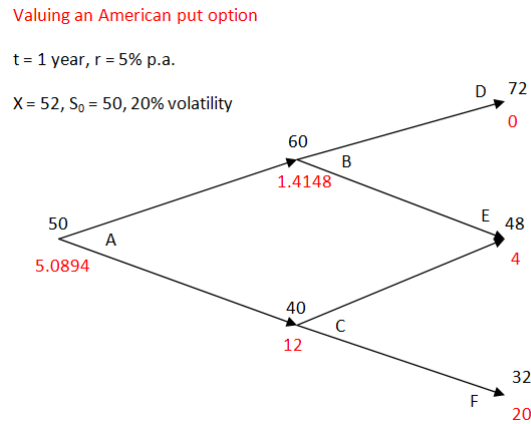


Figure 3.4: Valuing an American put option

For a Bermudan option the value is the greater of the binomial value and the intrinsic value where early exercise is allowed, and the binomial value where early exercise is not allowed.

The binomial tree method is generally considered to be more accurate than the Black-Scholes formula as it is more flexible and can be used to value American and European options. It is also possible to divide the tree into smaller time steps to yield a more accurate calculation of the option price.

There is also a notion of a ‘Trinomial’ tree which plots the path of the stock price at each stage in terms of it rising, falling and also staying the same. For fewer time steps than the Binomial tree, the Trinomial tree is considered more accurate but is not used widely due to its greater complexity.

3.3 Monte Carlo Option Pricing

Monte-Carlo methods also have an application when valuing options. The general idea is to simulate random price paths for the underlying asset which can be used to calculate the exercise values (payoffs) from the option at maturity. It is then possible to take the average of these payoffs to find an expected value for the option.

3.4 Historical Simulation

Pricing options follows a very similar approach to computing the VaR for an asset using Historical Simulation. Just as it is assumed that the next value of an asset would be one of its historical ordered values, the start price of an option can be estimated from the underlying stock's return on a given day and a possible value of the option can then be computed using one of the Black-Scholes, Binomial Tree or Monte Carlo Simulation techniques.

The price of an option for each return is stored in a list of possible values. This list is then ordered and the estimated price of the option can be taken from the percentile corresponding to the confidence level.

Source code for this approach is listed in appendix B.3.

3.5 Stochastic Volatility Models

The implied volatility of an option is the value of volatility of the underlying asset which will return a value equal to the current market price of the option when use in a pricing model such as Black-Scholes. It has been shown that the implied volatility for options with low strike prices is greater than the implied volatility for those with higher strike prices. This shows that volatility of an asset is Stochastic. A Stochastic property is one where the variance of a process is itself randomly distributed.[3] Volatility can be modelled as a random process which is directed by some variables such as the price of the underlying asset. Volatility also tends to eventually revert to a long-run average volatility.

Modelling the volatility of an asset's price as Stochastic helps to resolve the shortcomings of the Black-Scholes formula, which assumes that the volatility of the asset is constant and is unaffected by changes in the price of the asset over the time the option is active. Some models can be shown to indicate that the implied volatility changes with respect to the option's Strike price and time to maturity. By modelling the volatility as a Stochastic process it is possible to obtain more accurate option valuations.

Chapter 4: Criticisms of Value at Risk

While the VaR measure has been used widely in industry to help ensure financial security for large institutions, it is not without its criticisms.

The main concern seems to be that VaR is often understood to be the maximum possible or worst case loss for a portfolio [6]. This is not true as for a VaR estimate with 99% confidence there is still a 1% chance of the value of the portfolio decreasing beyond the VaR estimate, and that it is expected that the estimated VaR will be exceeded by an actual loss at least two or three times over a year.

It has also been claimed that VaR is not a reliable risk estimation measure as it ignores centuries of experience in favour of models developed by people who are not traders. [4]

Moreover, some have said that the VaR measure gave false confidence [4] to its users, especially when its use leads to a false sense of security among senior executives at financial institutions. [5]

However, it has been noted through interviews with risk experts that VaR is very useful to them, especially during the global financial crisis of 2007–2008.[7]

Even supporters of VaR, such as Aaron Brown [6], have suggested that VaR should not be the central concern of risk management. He claims that it is far more important to worry about what happens when losses exceed the VaR estimate, which could involve holding sufficient amount of capital in reserve to account for large losses. This reserve could be estimated by conducting stress tests on the portfolios held by an institution.

Also, it is considered important to report VaR estimations which have passed a backtest. This could involve using models which have performed consistently well in backtesting.

Chapter 5: Model Testing

As with most models that are implemented to develop a solution, it is necessary to analyse the validity of its solutions and outputs.

It is important here to note that model testing and software testing are different techniques concerning different entities. Software testing of the Value at Risk program is talked about in section 7.6.

For a model that estimates Value at Risk, it is important to know how well it performs in terms of how good its estimations of Value at Risk are. One of the approaches to test the validity of a VaR model is to test its estimations of VaR against real world data. This is known as Backtesting.

5.1 Backtesting

Back testing is conducted by measuring the number of times the actual loss in the value of a portfolio exceeded the Value at Risk estimate from data over a specified time period.

Using historical stock price data for a portfolio, a number of VaR estimations are made for a number of subsequent days. For some models, it is necessary to use some of the historical data to infer the volatility of the individual stocks so that some future possible losses can be estimated according to the stocks' volatility.

In order to Backtest a VaR model efficiently it is necessary to be able to accurately estimate the volatility of the stocks in the portfolio and to have a good amount of historical data.

These estimations of VaR are then compared with the actual losses in value experienced by the portfolio using historical stock data for the days we have estimated VaR for. A count of the number of times the VaR estimation was exceeded by the actual loss (an exception) is kept to track the performance of the VaR model under test.

As VaR is usually estimated with a confidence level from 99% downwards, it is expected that there will be a certain number of times where the estimation of VaR will be exceeded by actual losses. In order to gauge the reliability of the model, it is possible to compare the number of exceptions that occurred with the number of exceptions that are expected. For example, if VaR was estimated with 99% confidence, it is acceptable for the model to underestimate VaR in 1% of cases. This is because, by the definition of VaR 2.4, for a 99% confidence estimation there should be a 1% chance of the losses exceeding the estimation of VaR. If VaR was estimated daily for 100 days, the model is reliable if the actual losses exceeded the estimation once. For 95% confidence, the number of acceptable exceptions would be 5% of the cases, and so on.

If the number of exceptions is greater than the expected amount, it is necessary to analyse the VaR estimation model and find out why it is underestimating the VaR and whether the excess exceptions may have happened by chance. This can be done by computing the 'p-value' of such an event, as covered in section 5.3.

Backtesting VaR models for a large number of days and over a variety of confidence levels can lead to a good assessment of how good those models are at estimating VaR.

Industrial backtesting techniques seem to agree on the following aspects for a good backtesting strategy for VaR models:

- Data sets are updated at least every 3 months - to keep stock data fresh and updated with recent market developments.
- VaR is calculated on a daily basis.
- 99% confidence used when estimating VaR.
- Time period of 10 days to estimate VaR over - to smooth out sudden large changes in stock prices.
- One year of historical stock price data held - so that volatilities of the stocks can be reliably estimated.

In the Value at Risk estimation program associated with this report all VaR models can be backtested using similar settings.

Some backtesting was done as part of verifying the reliability of the VaR models implemented in the program resulting in the output in appendix C.

The results from the backtests show that while the VaR models implemented in the program are not perfect, they are also not very far from estimating VaR reasonably reliably. The models seem to be underestimating VaR as the level of confidence required decreases. It is also easy to infer that the Historical Simulation VaR model performs generally better than the other models. Note that the Historical Simulation model does not take into account any volatilities from the sets of data it is using, thus, one possible deduction from these backtesting results is that the the volatility calculating models in the program must be improved.

Through experiments and program testing, it was shown that use of the EWMA model to estimate volatility resulted in the worst backtesting results for every VaR estimation model. GARCH, in theory, should be the best estimating model for volatility, however, due to a lack of optimal parameter estimation in the program, the GARCH model could not return optimal estimates of the volatility for a series of returns. This could have led to the underestimation of the volatility of a set of assets and resulted in the underestimation of the VaR of a portfolio containing these assets. The results for backtesting conducted when using EWMA and GARCH models can be seen in appendix C.3 and C.1 respectively.

On the other hand, the number of actual exceptions (where the actual loss was greater than the estimated VaR) greater than the number of expected exceptions could also have happened by chance. In light of this possibility, p-Values can be calculated from the results of the backtests to determine whether the number of actual exceptions was a random error. This is discussed further in section 5.3.

5.2 Stress Testing

Stress testing for software involves analysing its robustness when operated beyond the limits of its usual operation. Stress testing of VaR models follows a similar approach.

The stock market does not satisfy the assumptions made by some algorithms for estimating VaR. For example, the Monte-Carlo Simulation algorithm assumes that stock prices follow a normal distribution, and while Monte-Carlo methods are used to estimate VaR for stocks and options, sudden changes in the market are not in the scope of such methods. This means

that VaR estimation models cannot predict sudden large losses in the value of a portfolio which can be disastrous for an organisation if they did not have enough capital in reserve to account for such a situation.

To account for such situations, Stress testing is often used to complement Backtesting of VaR models. Stress testing is conducted by measuring what changes our portfolio would have experienced under some of the extreme market moves, which are unlikely but plausible, over the past few decades.

Such extreme moves in the market can either be simulated, by generating a series of stock prices with a sudden drop in value, or taken from historical stock data from events such as stock market declines after the September 11, 2001 attacks on the World Trade centre and the Russian Default in 1998.

In the program associated with this report, the user is able to conduct stress tests on a specified portfolio and see what affect a large market decline would have had on the value of their portfolio. In combination with Backtesting, it can be seen that many of the extreme declines in value would not have been estimated by the VaR models, which makes stress testing a useful technique for institutions to ensure their financial security.

5.3 p-Values

When testing the validity of a claim about a set of data, p-values can be used to determine the significance of results from such data against the claim.

The claim on trial is known as the *null hypothesis*, H_0 . If the null hypothesis is concluded to be false through strong evidence against it from the data set, then an alternative hypothesis, H_a could be held as valid.

A p-value is used, in the above scenario, to weigh the strength of the evidence supporting or against the null hypothesis. A p-value is a number between 0 and 1.

The p-value of a set of results can be calculated from a graph charting the probability distribution of a set of data, as seen in 5.1.

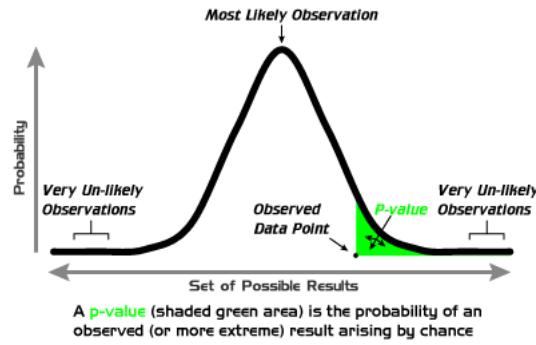


Figure 5.1: Example of p-value computation [9]

A small p-value, (≤ 0.05), indicates that there is a (≤ 0.05) chance of H_0 being incorrectly rejected i.e. strong evidence against H_0 , which can lead to H_0 being rejected. On the contrary, a large p-value, (> 0.05), indicates weak evidence against the H_0 , so H_0 can be accepted to be a valid claim. A p-value close to 0.05 is marginal which means that H_0 can be either rejected or accepted. In such cases, the decision is left up to the reader and their interpretation from the data set.

Results from the backtesting of the VaR models in appendix ?? can be analysed using p-values. In this case:

H_0 = The VaR model in my program will underestimate VaR in $(100-X)\%$ of cases when computed with $X\%$ confidence.

H_a = The VaR model in my program will underestimate VaR in more than $(100-X)\%$ of cases when computed with $X\%$ confidence.

The p-value of the results can also be computed using the R package for statistical computing [10], which provides the 'pbinom' function to calculate p-values from results of data.

Usage:

$$1 - \text{pbinom}(n, N_t, p)$$

where, n is the number of exceptions over the acceptable number of exceptions, N_t is the number of trials i.e. number of days VaR was estimated and compared with actual losses and p is the probability of an exception happening e.g. 0.01 (1%) for 99% confidence.

For the data in appendix C, the following p-values and results can be inferred:

For 99% confidence:

Model Building: $1-\text{pbinom}(1,100,0.01) = 0.264238$ (weak evidence against H_0)

Historical Sim: $1-\text{pbinom}(-1,100,0.01) = 1$ (no evidence against H_0 at all)

Monte Carlo: $1-\text{pbinom}(1,100,0.01) = 0.264238$ (weak evidence against H_0)

For 98% confidence:

Model Building: $1-\text{pbinom}(3,100,0.02) = 0.1410384$ (weak evidence against H_0)

Historical Sim: $1-\text{pbinom}(0,100,0.02) = 0.8673804$ (very weak evidence against H_0)

Monte Carlo: $1-\text{pbinom}(1,100,0.02) = 0.5967283$ (very weak evidence against H_0)

For 97% confidence:

Model Building: $1-\text{pbinom}(4,100,0.03) = 0.1821452$ (weak evidence against H_0)

Historical Sim: $1-\text{pbinom}(1,100,0.03) = 0.8053779$ (very weak evidence against H_0)

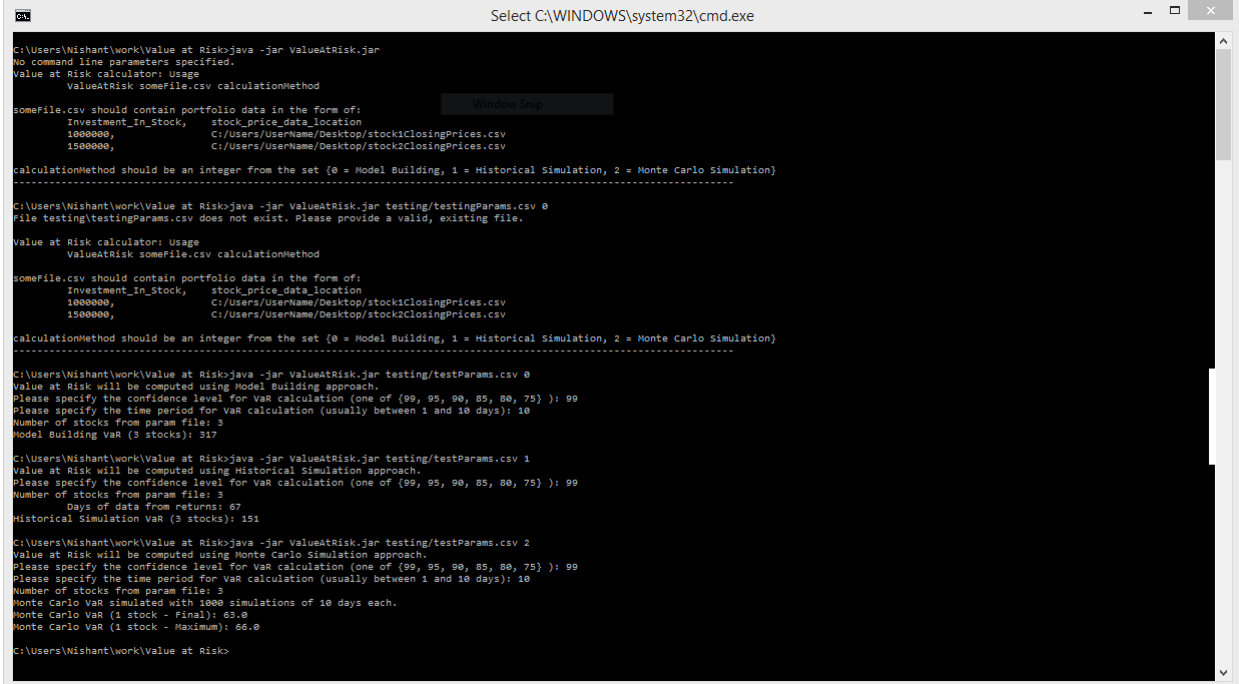
Monte Carlo: $1-\text{pbinom}(5,100,0.03) = 0.08083713$ (slight evidence against H_0) ... and so on.

From these p-values, it is possible to infer that the VaR models implemented in the Value at Risk program are underestimating VaR by a small margin. The backtesting results for the Historical Simulation model produced very weak or no evidence against the null hypothesis, meaning it is the best model for computing VaR in this program. However, as mentioned before, other models of computing VaR can be improved if the volatility estimation using the GARCH model is improved through parameter optimisation.

Chapter 6: Program Design

6.1 Initial Command Line Interface

The user interface for the interim submission was a command line interface which tried to be as informative as possible to the user and took in user input for some parameters.



```

C:\Users\Nishant\work\Value at Risk>java -jar ValueAtRisk.jar
No command line parameters specified.
Value at Risk calculator: Usage
    ValueAtRisk someFile.csv calculationMethod

someFile.csv should contain portfolio data in the form of:
    Investment_In_Stock,    stock_price_data_location
    1000000,                C:/Users/UserName/Desktop/stock1ClosingPrices.csv
    1500000,                C:/Users/UserName/Desktop/stock2ClosingPrices.csv

calculationMethod should be an integer from the set {0 = Model Building, 1 = Historical Simulation, 2 = Monte Carlo Simulation}

C:\Users\Nishant\work\Value at Risk>java -jar ValueAtRisk.jar testing/testingParams.csv 0
File testing/testingParams.csv does not exist. Please provide a valid, existing file.

Value at Risk calculator: Usage
    ValueAtRisk someFile.csv calculationMethod

someFile.csv should contain portfolio data in the form of:
    Investment_In_Stock,    stock_price_data_location
    1000000,                C:/Users/UserName/Desktop/stock1ClosingPrices.csv
    1500000,                C:/Users/UserName/Desktop/stock2ClosingPrices.csv

calculationMethod should be an integer from the set {0 = Model Building, 1 = Historical Simulation, 2 = Monte Carlo Simulation}

C:\Users\Nishant\work\Value at Risk>java -jar ValueAtRisk.jar testing/testParams.csv 0
Value at Risk will be computed using Model Building approach.
Please specify the confidence level for VaR calculation (one of {99, 95, 90, 85, 80, 75}): 99
Please specify the time period for VaR calculation (usually between 1 and 10 days): 10
Number of stocks from param file: 3
Model Building VaR (3 stocks): 317

C:\Users\Nishant\work\Value at Risk>java -jar ValueAtRisk.jar testing/testParams.csv 1
Value at Risk will be computed using Historical Simulation approach.
Please specify the confidence level for VaR calculation (one of {99, 95, 90, 85, 80, 75}): 99
Number of stocks from param file: 3
Days of data from returns: 67
Historical Simulation VaR (3 stocks): 131

C:\Users\Nishant\work\Value at Risk>java -jar ValueAtRisk.jar testing/testParams.csv 2
Value at Risk will be computed using Monte Carlo Simulation approach.
Please specify the confidence level for VaR calculation (one of {99, 95, 90, 85, 80, 75}): 99
Please specify the time period for VaR calculation (usually between 1 and 10 days): 10
Number of stocks from param file: 3
Monte Carlo VaR simulated with 1000 simulations of 10 days each.
Monte Carlo VaR (1 stock - Final): 65.0
Monte Carlo VaR (1 stock - Maximum): 66.0

C:\Users\Nishant\work\Value at Risk>
  
```

Figure 6.1: Example of program usage

For the final submission, I have implemented a graphical user interface (GUI) which makes it easier for the user to specify parameters and files to be used in computing Value at Risk.

6.2 Graphical User Interface

Before implementing any user interface, it is important to have a list of requirements which dictate what it should look like and its ease of use. The Value at Risk program needs to take as input a portfolio containing data about assets and options, such as historical stock prices and the corresponding investments. The program also takes as input some parameters for VaR computation, such as the confidence level, time horizon and, in this case, the type of model to be used to calculate the VaR itself.

Therefore, it was important that the user interface made it easy for the user to enter all that data into the program. Moreover, it was also important for the user interface to follow an aesthetically pleasing theme and to be readable on all screens.

The GUI for the program was constructed using WindowBuilder, an Eclipse plug-in, which allows drag-and-drop style user interface creation with real-time rendering. By using WindowBuilder it is possible to build large, complex user interfaces for Java programs faster than

doing it exclusively by writing the code by hand.

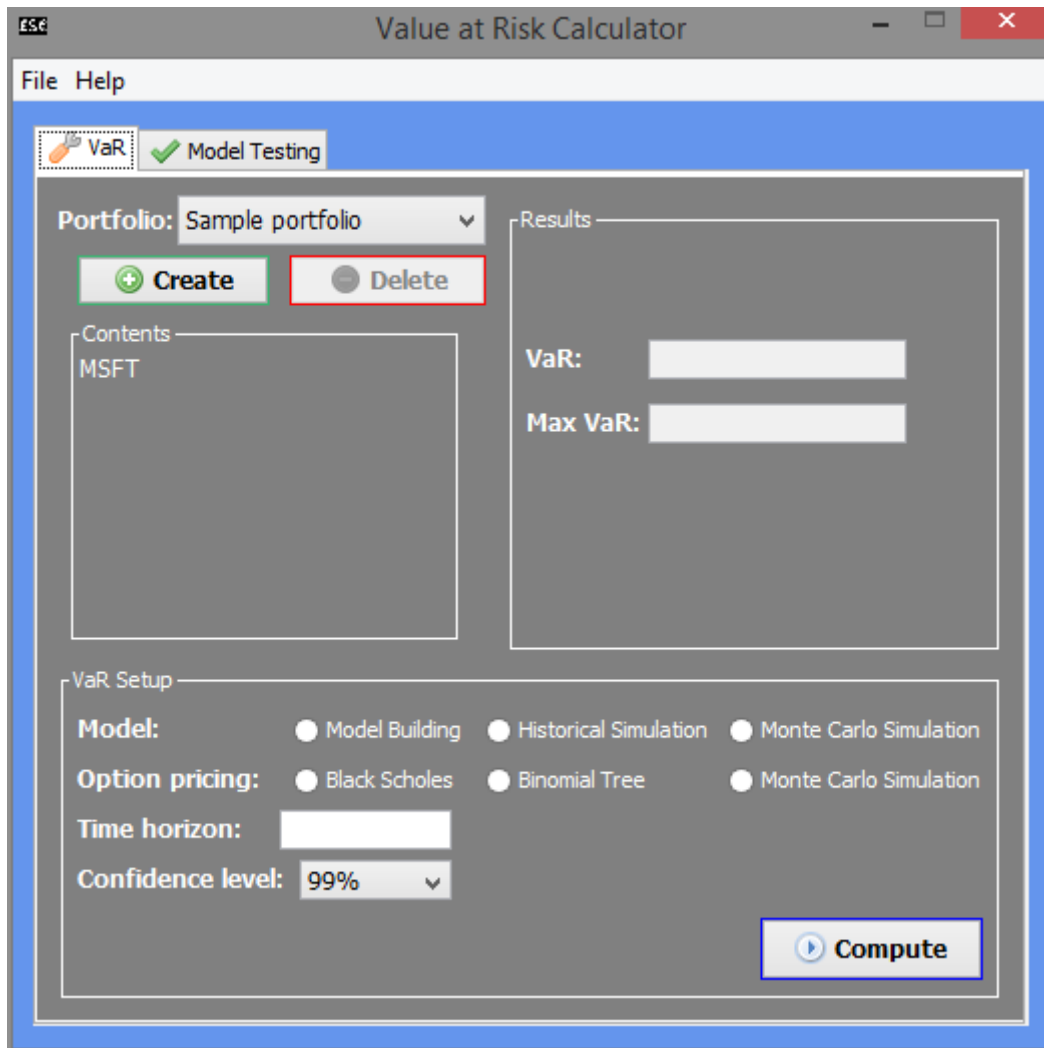


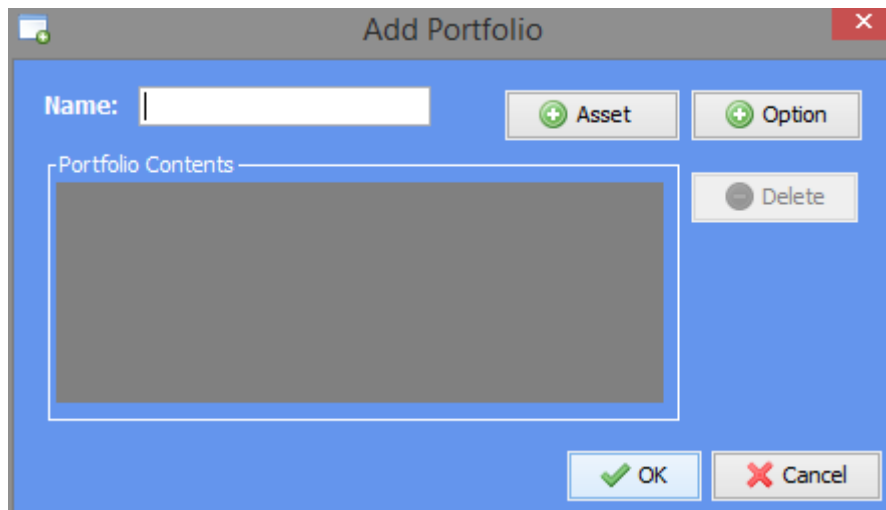
Figure 6.2: Start screen of the Value at Risk program.

The start screen of the Value at Risk program can be seen in figure 6.2. The user interface of the program groups the data that needs to be input into bordered sections with informative titles. The contrast between the background canvas and foreground text is high to make the text stand out and readable. Tooltips and helpful messages are also shown to the user wherever it is deemed necessary.

The colour scheme is constant throughout the program, which helps to make every visual element a strong part of the whole program.

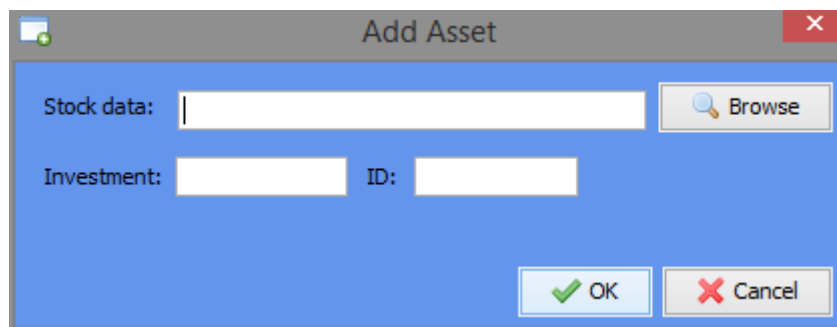
The GUI also makes use of colour coding the buttons with familiar colours for positive and negative actions as well as well-known, sensible, icons so that the user does not always have to read the text on the button to understand what it does. After using the program for a few minutes, this pattern becomes familiar to the user and they will often find that they are using the correct buttons without reading the text on them.

The GUI allows the user to specify a portfolio to use in VaR computation using a drop-down list. Since at the start of the program there were no portfolios created, the program informs the user that no portfolios are available for selection. When the user clicks on the ‘Create’ button, a new dialog is shown which handles the creation of a new portfolio for the program as seen in 6.3. Similar dialogs have been implemented in this modular way to handle the addition of an asset 6.4 or option 6.5 to the portfolio.



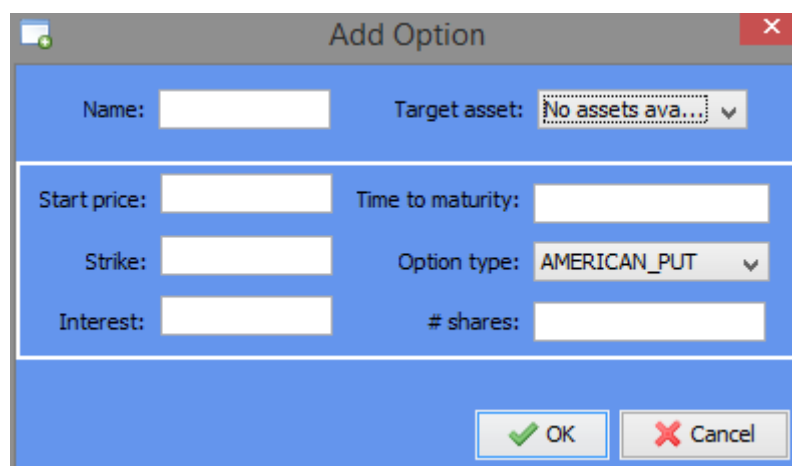
The 'Add Portfolio' dialog box has a title bar with a close button. It contains a 'Name:' text input field. To its right are two buttons: '+ Asset' and '+ Option'. Below the name field is a large rectangular area labeled 'Portfolio Contents'. To the right of this area is a 'Delete' button with a minus icon. At the bottom right are 'OK' and 'Cancel' buttons.

Figure 6.3: A dialog to handle the creation of a new portfolio.



The 'Add Asset' dialog box has a title bar with a close button. It contains a 'Stock data:' text input field with a 'Browse' button (magnifying glass icon) to its right. Below this are two text input fields: 'Investment:' and 'ID:'. At the bottom right are 'OK' and 'Cancel' buttons.

Figure 6.4: A dialog to handle the addition the addition of an asset to a new portfolio.



The 'Add Option' dialog box has a title bar with a close button. It contains a 'Name:' text input field and a 'Target asset:' dropdown menu showing 'No assets ava...'. Below these is a group box containing several fields: 'Start price:', 'Time to maturity:', 'Strike:', 'Option type:' (dropdown showing 'AMERICAN_PUT'), 'Interest:', and '# shares:'. At the bottom right are 'OK' and 'Cancel' buttons.

Figure 6.5: A dialog to handle the addition the addition of an option to a new portfolio.

There is a certain degree of input validation and sensible logic implemented in the user interface. For example, in figure 6.2, where there are no portfolios available for use, the ‘Delete’ button is disabled so that the user cannot cause exceptions in the running code by trying to delete a portfolio which does not exist. Also the ‘Compute’ button is also disabled until all required inputs have been correctly specified.

These features are easy to implement but add to the improved usability and good impression of the program’s user interface. Further discussion about the usability of the program is present in section 6.3 of this chapter.

6.3 Usability

The user interface for the Value at Risk program contains helpful features to make computing VaR easier.

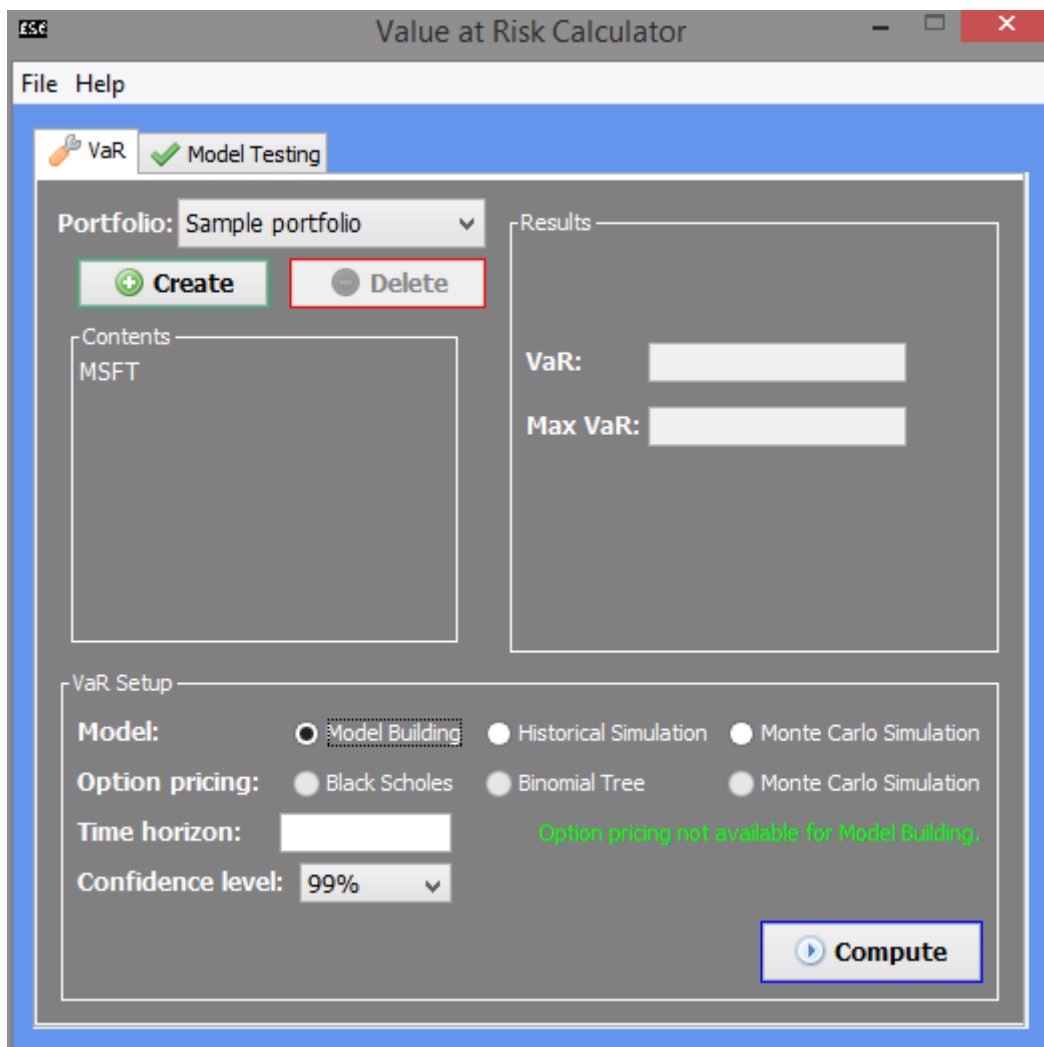


Figure 6.6: Option pricing model selection disabled with informative message.

Firstly, due to the option pricing models being implemented for the Historical Simulation and Monte Carlo Simulation VaR models only, the model selection for option pricing is disabled when the Model Building VaR model is selected for computing VaR. A message is shown to the user in such a case, as shown in figure 6.6.

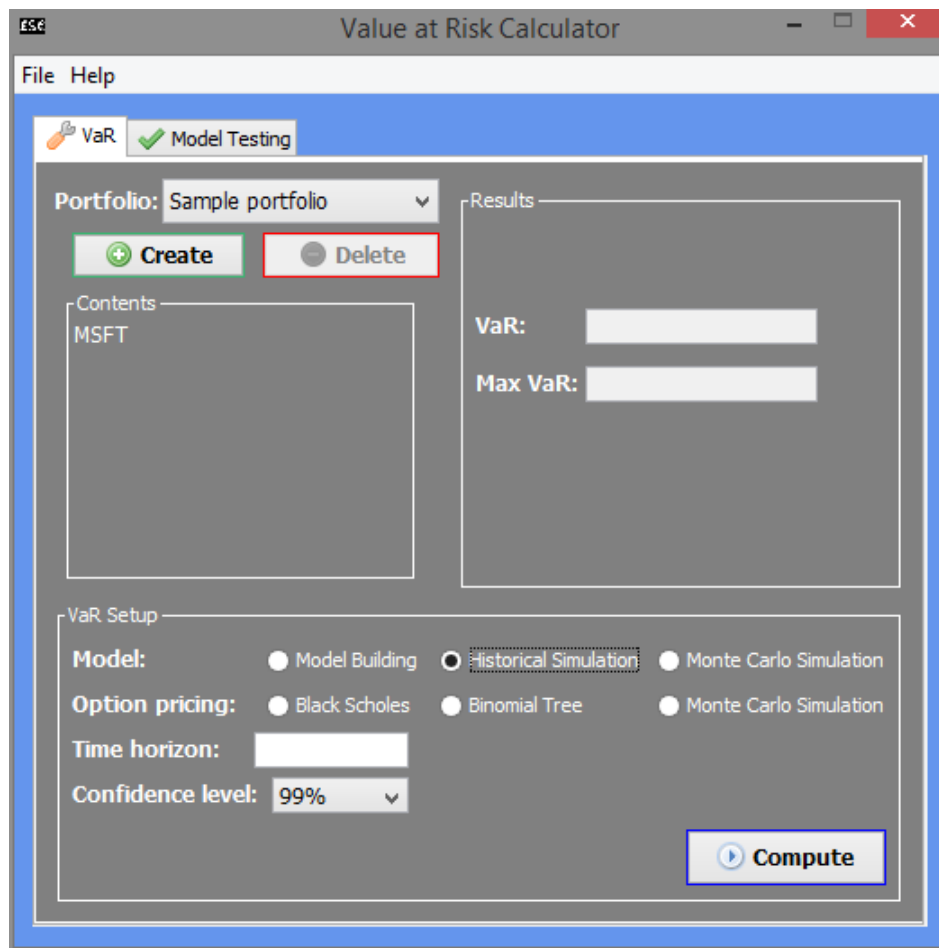


Figure 6.7: Option pricing available for other models of VaR computation

When one of the models other than Model Building is selected, option pricing model selection is made available to the user once again, as shown in figure 6.7. This feature acts as first-hand validation against the user causing errors in the program.

When creating a new portfolio, the user must specify a portfolio name so that it can be identified later in the list of portfolios selectable in the ‘Setup’ tab. If a user does not enter a portfolio name, a message dialog is shown to user upon clicking ‘OK’, as shown in figure 6.8.

Furthermore, if a user tries to create a portfolio with no assets or options, a message is also presented as shown in figure 6.9.

When creating a new asset, the user must specify valid data for each field, otherwise a message is shown to the user informing them about it, see figure 6.10.

When an asset is successfully added to the portfolio, the asset name is shown in the contents of that portfolio, see figure 6.11.

The added asset is also available for selection as the underlying asset for a new option, as demonstrated in figure 6.12.

When an option is added to the portfolio, it is also shown in the contents of that portfolio, as shown in figure 6.13.

When a user tries to remove an asset or an option from a portfolio, a confirmation message is shown to the user before deletion of the selected item, as shown in figure 6.14.

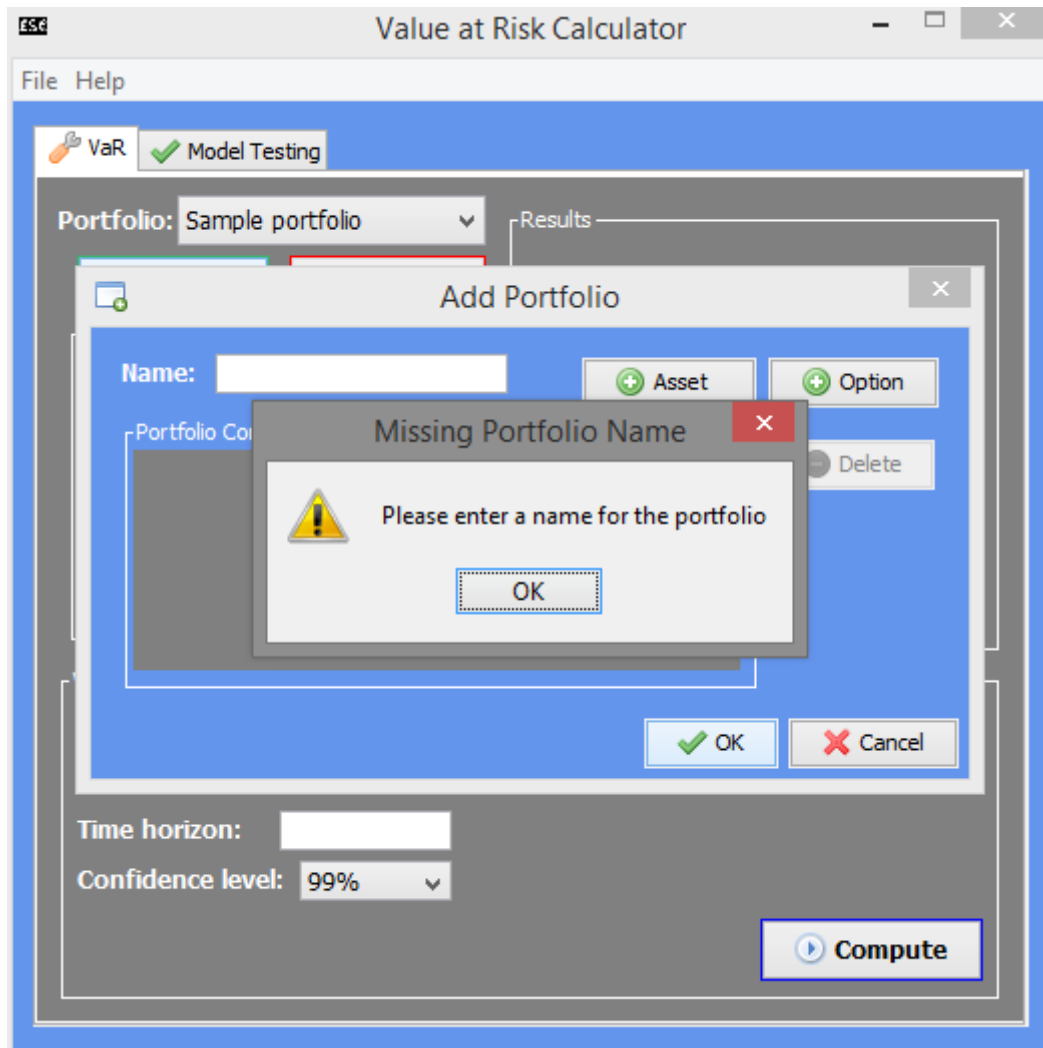


Figure 6.8: Message dialog informing the user of missing portfolio name.

Once all parameters have been set using valid values, the compute button is enabled so that the computation of VaR for the selected portfolio may begin. See figure 6.15.

From the figures in this section, it is also possible to see that all the dialogs are displayed in a position relevant to their parent dialogs and that there is some modality applied to the dialogs so that the user has to deal with the foremost dialog before returning to its parent dialog. A combination of these features makes the dialogs easy to find for the user, as the user expects the dialogs to show up close to their mouse click. Also, modality ensures that data to the program is input in a logical, manageable order and that multiple dialogs arising from the same parent cannot be created, thus avoiding confusion when using the program.

Moreover, to keep the program responsive to user input, all code that manipulates the user interface is executed on the 'Event Dispatch Thread' while the back-end computations are run on the main thread. If the GUI code and back-end code was run on the same thread, the user will be blocked from performing any actions on the GUI while back-end computations are taking place, which is not a good behaviour for programs with a user interface to follow.

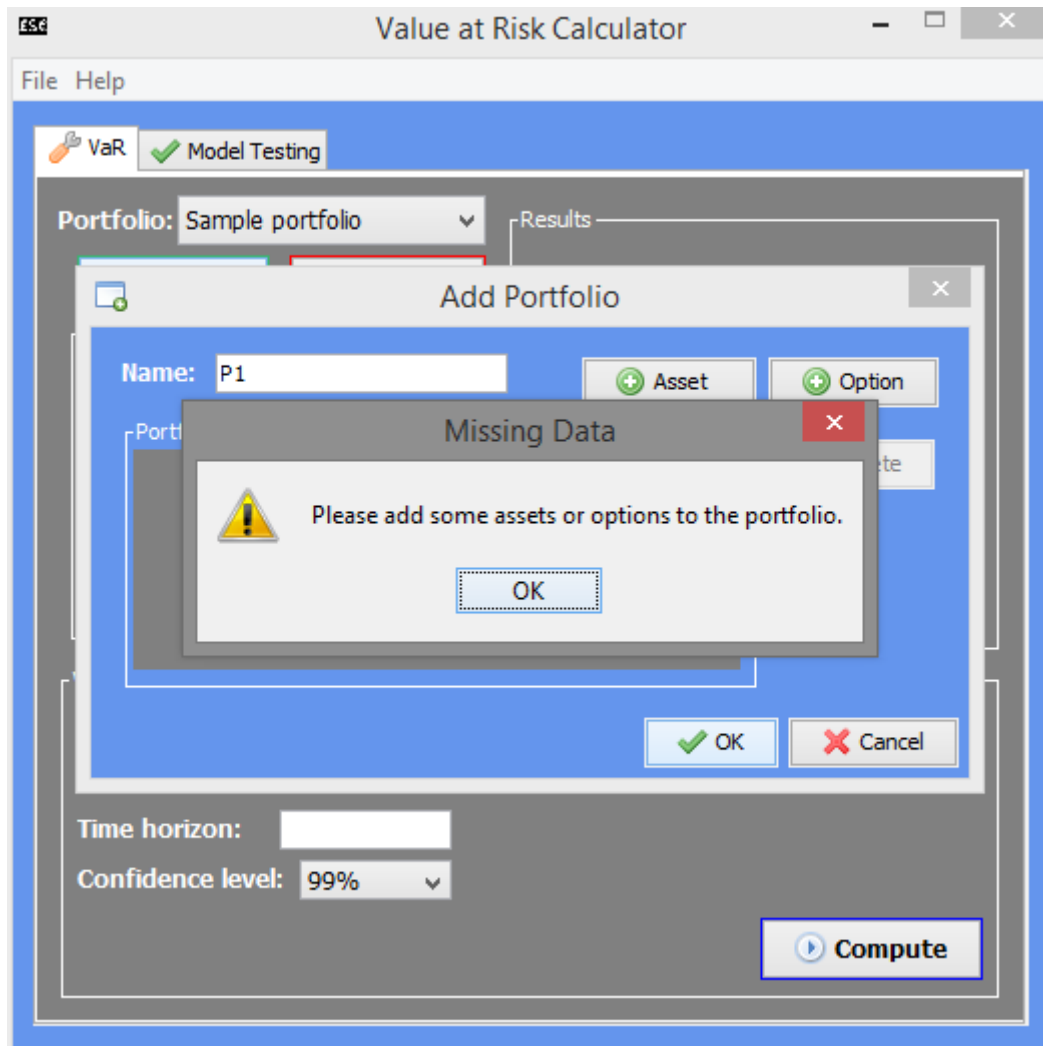


Figure 6.9: Message dialog informing user of missing portfolio contents.

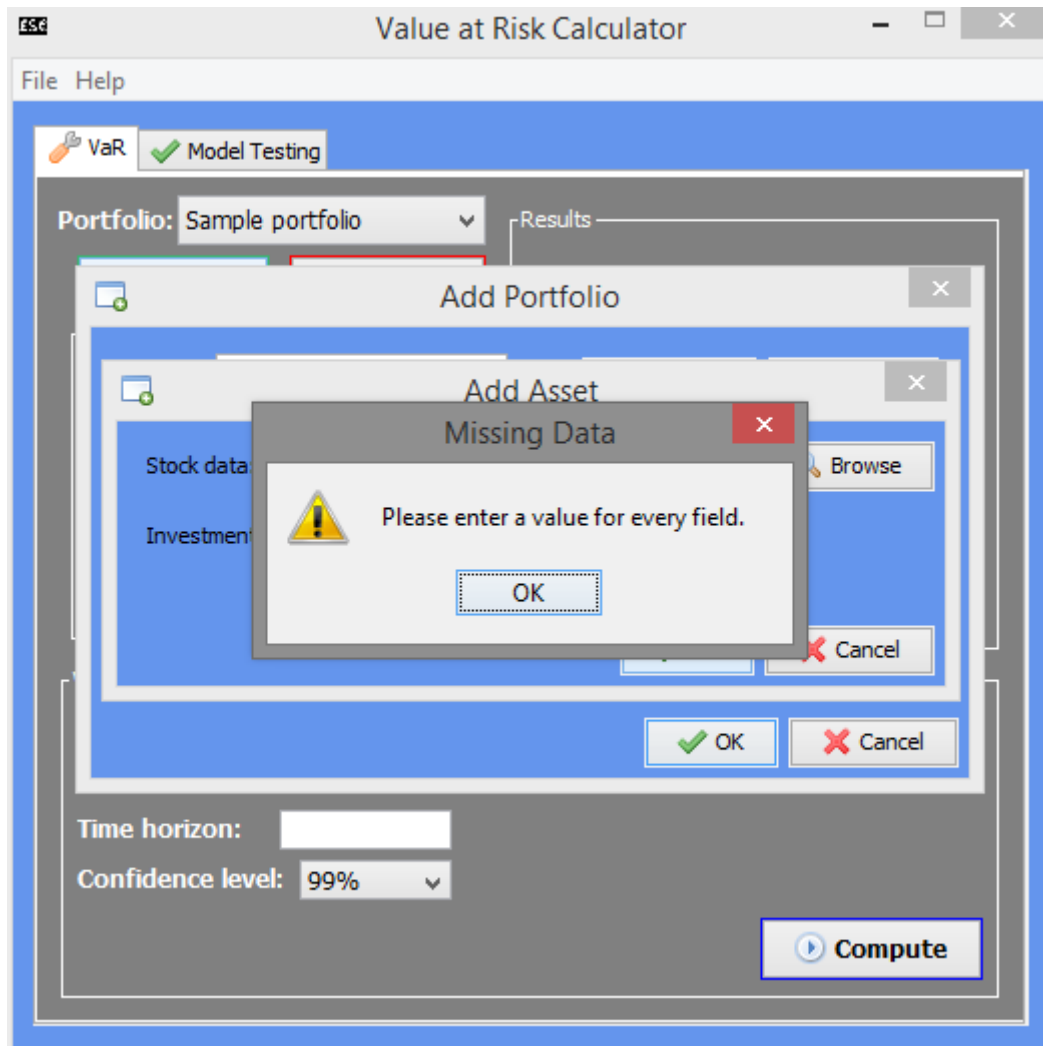


Figure 6.10: Message dialog informing user of missing data for asset.

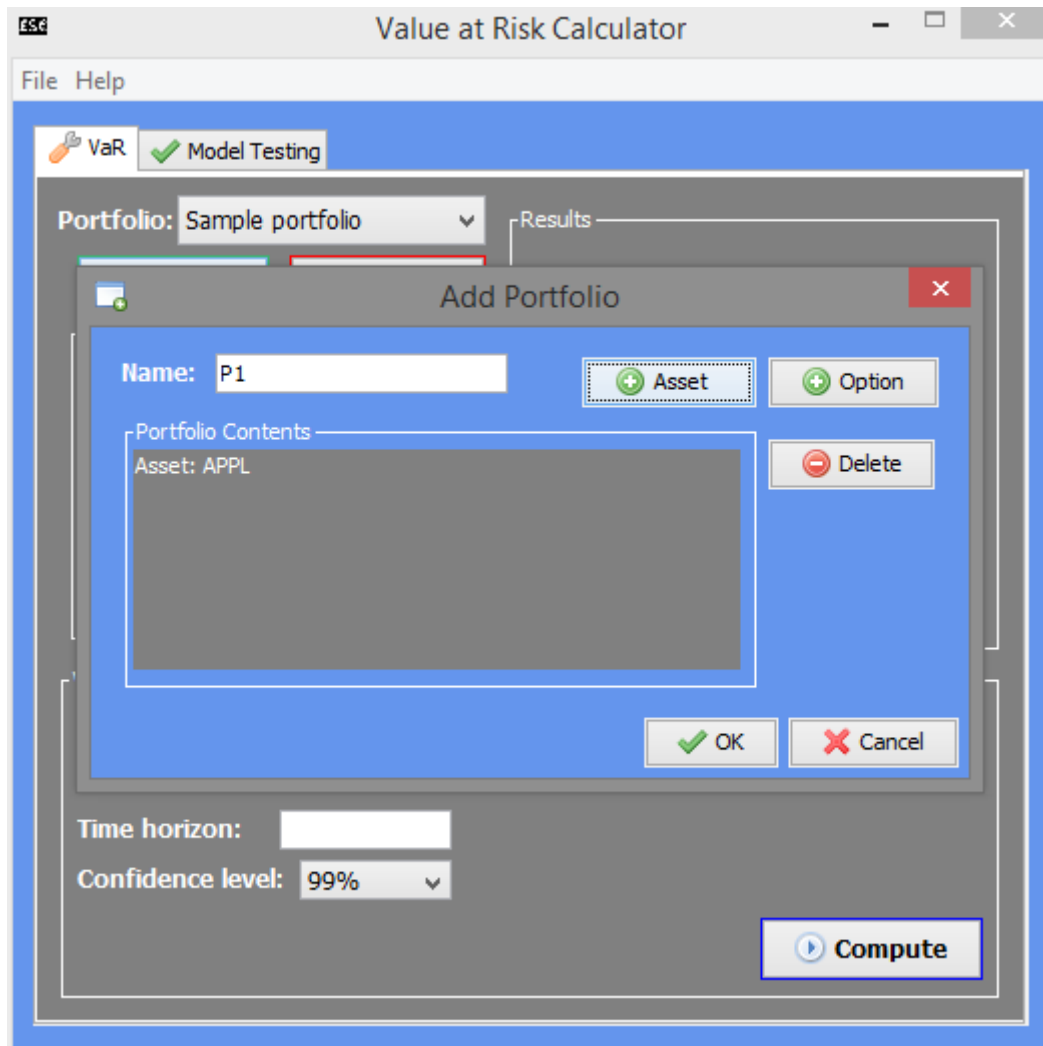


Figure 6.11: Add Portfolio dialog showing asset added to portfolio.

The screenshot displays the 'Value at Risk Calculator' application. The main window features a menu bar with 'File' and 'Help'. Below the menu bar, there are two tabs: 'VaR' (active) and 'Model Testing'. The 'VaR' tab contains a 'Portfolio:' dropdown menu set to 'Sample portfolio' and a 'Results' section. A modal dialog box titled 'Add Option' is open in the foreground. This dialog has a 'Name:' text field and a 'Target asset:' dropdown menu set to 'Asset: APPL'. Below these are two rows of input fields: 'Start price:' and 'Time to maturity:', and 'Strike:' and 'Option type:' (set to 'AMERICAN_PUT'). The bottom row has 'Interest:' and '# shares:'. At the bottom right of the dialog are 'OK' and 'Cancel' buttons. In the background, below the dialog, the main window shows 'Time horizon:' and 'Confidence level:' (set to '99%') fields, and a 'Compute' button at the bottom right.

Figure 6.12: Added asset shown as possible underlying asset for new option.

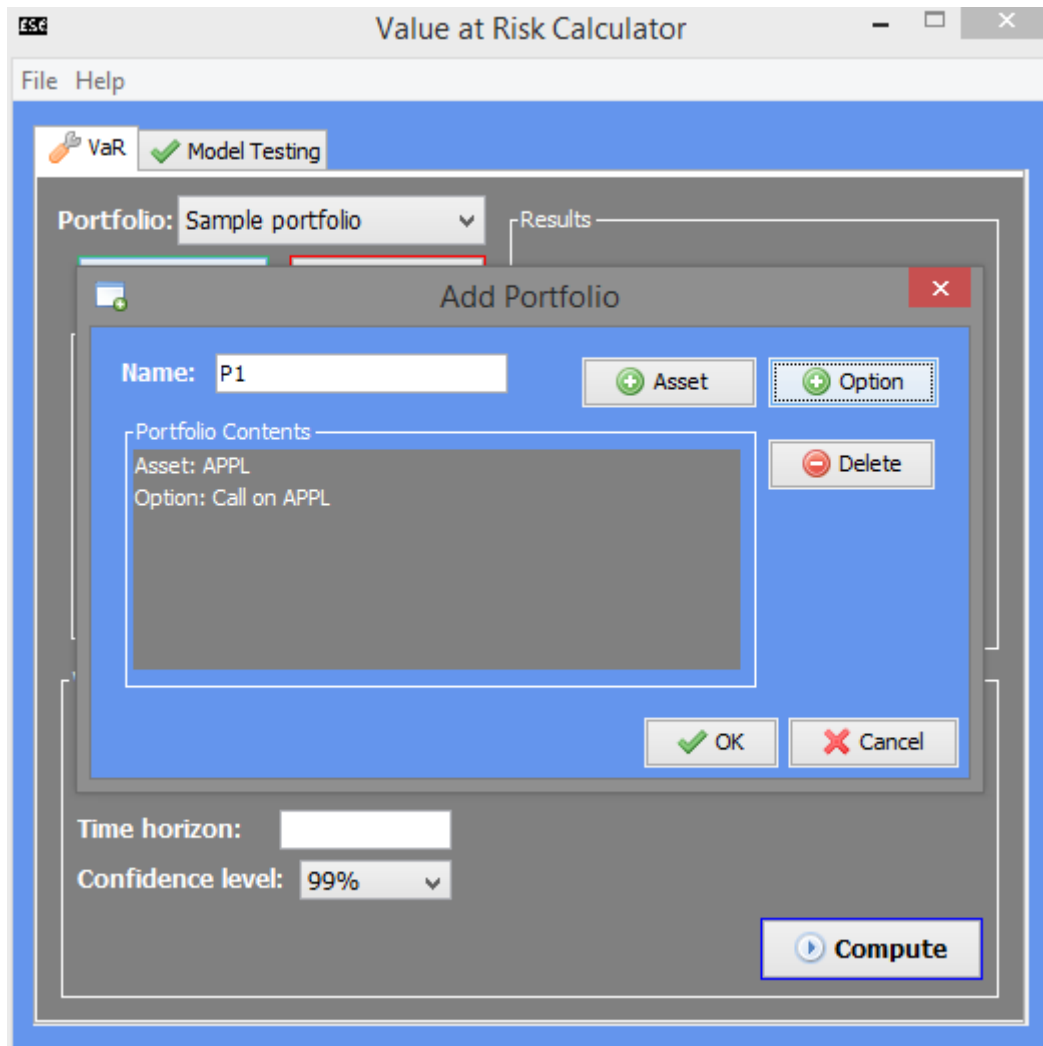


Figure 6.13: Added option shown in portfolio.

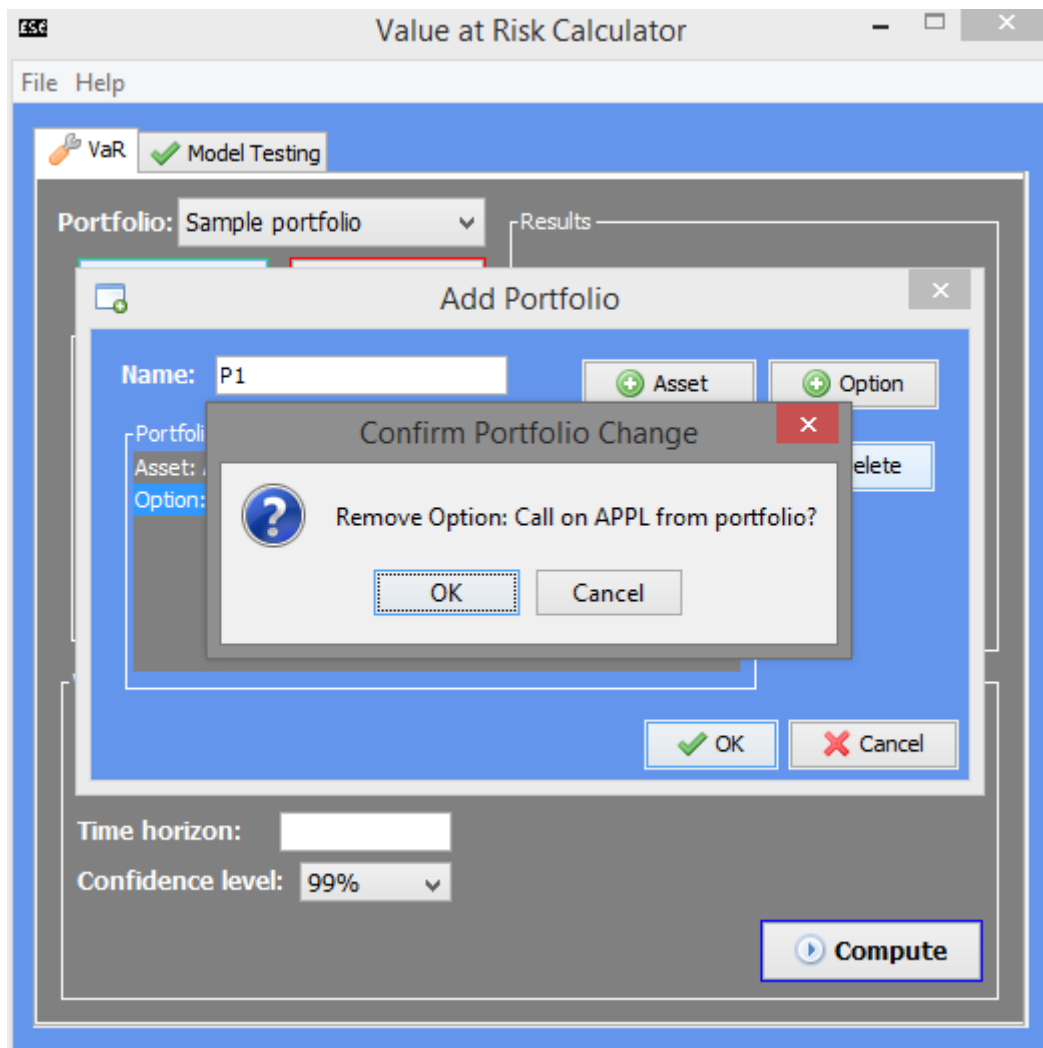


Figure 6.14: Confirmation dialog when removing option from portfolio.

The screenshot shows a software application titled "Value at Risk Calculator". It features a menu bar with "File" and "Help". Below the menu bar, there are two tabs: "VaR" (active) and "Model Testing". The main interface is divided into several sections:

- Portfolio:** A dropdown menu showing "P1".
- Buttons:** A green "Create" button and a red "Delete" button.
- Contents:** A list box containing "Asset: APPL" and "Option: Call on APPL", with the latter selected.
- Results:** Two input fields labeled "VaR:" and "Max VaR:".
- VaR Setup:** A section containing:
 - Model:** Three radio buttons: "Model Building" (selected), "Historical Simulation", and "Monte Carlo Simulation".
 - Option pricing:** Three radio buttons: "Black Scholes" (selected), "Binomial Tree", and "Monte Carlo Simulation".
 - Time horizon:** A text input field containing the value "1".
 - Confidence level:** A dropdown menu showing "99%".
- Compute:** A blue button with a play icon and the text "Compute".

Figure 6.15: VaR program ready for estimation when all parameters set.

Chapter 7: Software Engineering

This chapter details the software engineering aspect of the Value at Risk project. It contains the class diagram of the Value at Risk program with descriptions of individual classes and their roles in the computation of VaR. Moreover, a discussion about the implementation of the software is present later in this chapter.

7.1 UML Class Diagram and Class Descriptions

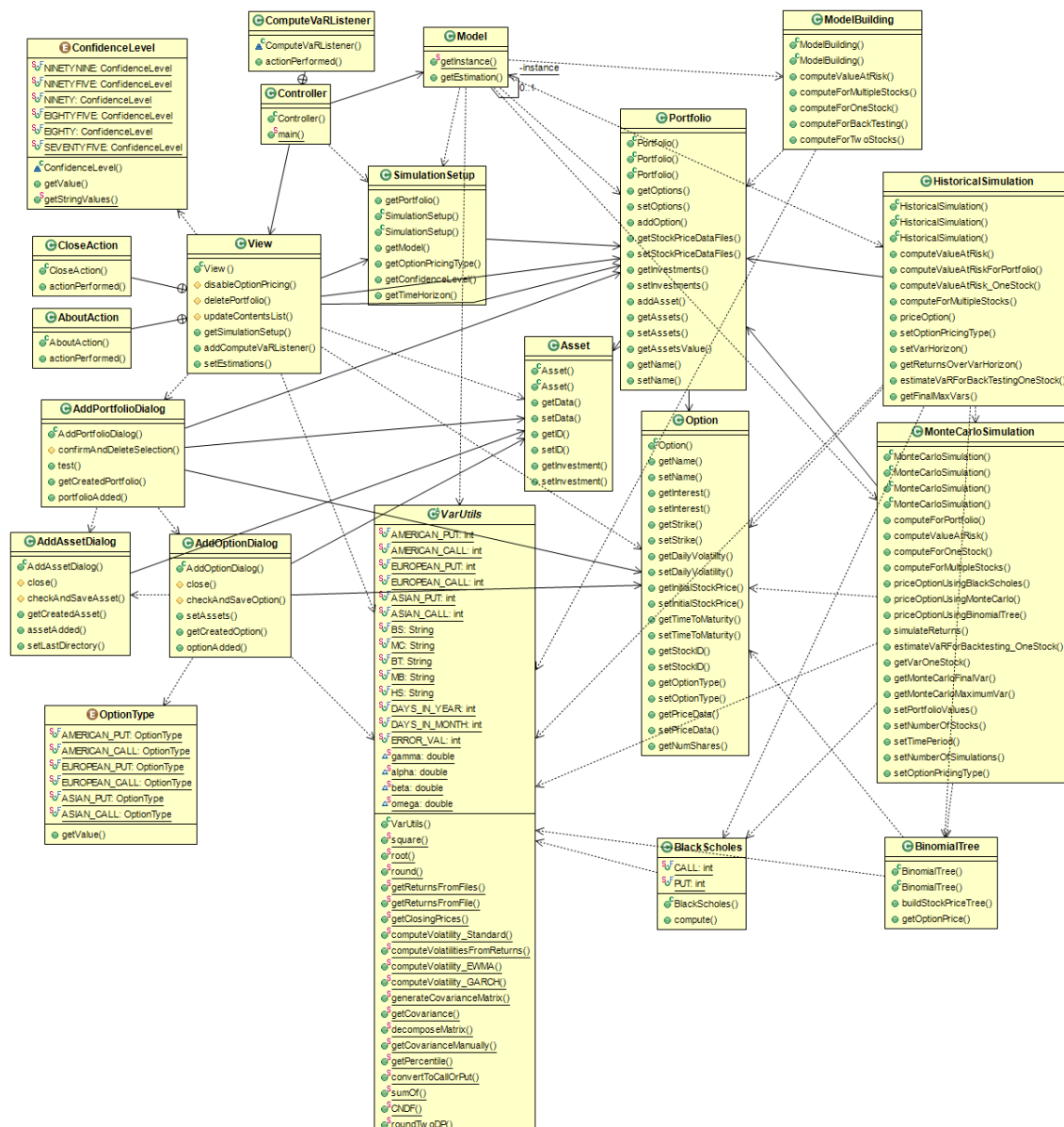


Figure 7.1: Simplified class diagram for Value at Risk program

From the class diagram it is clear that the three classes in the middle, `Portfolio`, `Asset` and `Option` are the most referenced entities in the system. This is expected as every VaR model utilises these entities in its VaR computation as well as all the other program logic such as adding and removing portfolios in the user interface. The `VarUtils` class is also used

frequently for computations common to the VaR models and for passing constants specifying types of options and types of VaR computation models between the user interface and the back-end.

Roles of the classes forming the Value at Risk program are now explained in the following sections.

7.1.1 Model-View-Controller

A description of this part of the program is contained in section 7.8.

7.2 Role of Classes in Program

An explanation of each class, or collection of similar classes, along with a diagram showing their layout, is present in this section.

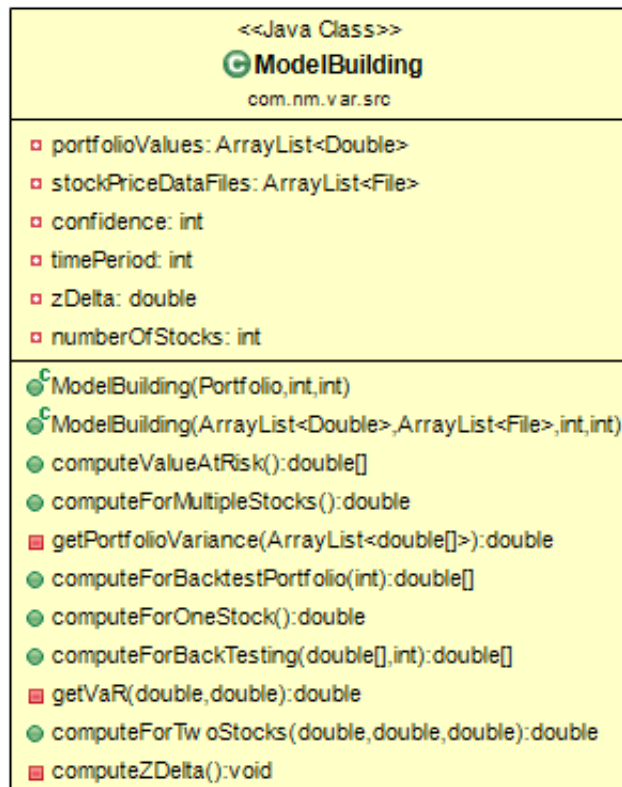


Figure 7.2: ModelBuilding.java

The ModelBuilding class illustrated in figure 7.2 contains methods to compute the VaR of a given portfolio using the Model-Building approach outlined in section 2.5. It can be instantiated using either a portfolio or its underlying data (second constructor) along with a time horizon and confidence level for computing VaR. The function `computeZDelta()` computes the number of standard deviations the loss should not exceed.

The HistoricalSimulation class shown in figure 7.3 implements the Historical Simulation model, as described in section 2.6. It can be instantiated using a portfolio and a confi-

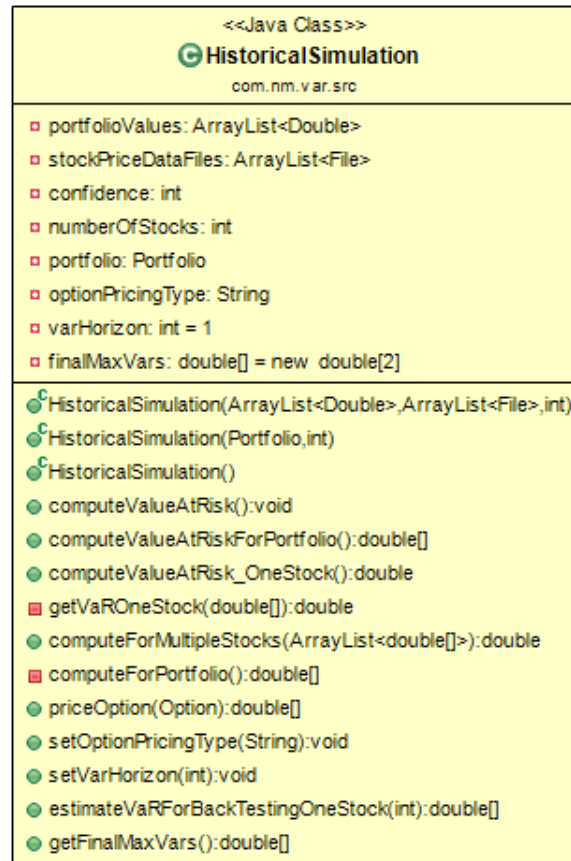


Figure 7.3: HistoricalSimulation.java

dence level to compute VaR. This class also uses one of the three available option pricing techniques, as specified by the user, for any options that are present in a portfolio in order to compute VaR for the portfolio.

The MonteCarloSimulation class in figure 7.4 implements the namesake model for VaR computation, as described in section 2.7. The class contains methods to compute the VaR of a given portfolio as well as the Monte Carlo option pricing algorithm. In order to compute VaR, the class has functions to simulate random stock prices or returns over a specified time period.

The classes in figure 7.5 implement the Binomial Tree option pricing model, section 3.2 and the Black–Scholes option pricing formula, section 3.1. The BinomialTree class requires instantiation with a series of option parameters or an Option and returns the option price as calculated by this model. The Black–Scholes class requires direct usage of the `compute()` function with a series of parameters from the option which needs to be priced in order to obtain the price of the option.

The classes in figure 7.6 implement the Backtesting, section 5.1, and Stress testing, section 5.2, techniques used to verify the performance of a VaR model. The BackTesting class allows the use of one or many models to test. The StressTester class simulates crashes in values of portfolios and keeps track of the simulated values and corresponding losses incurred from such simulations.

The Portfolio class shown in figure 7.7 acts as a customised datatype to store a collection of objects of datatypes Asset and Option. The goal of the Portfolio class is to replicate a financial portfolio by combining assets (groupings of some historical stock data, investment and an identifier) and options into a single entity. A Portfolio object can be populated

<<Java Class>>	
MonteCarloSimulation	
com.nm.var.src	
<ul style="list-style-type: none"> monteCarloFinalVar: double monteCarloMaximumVar: double rng: Random = new Random() portfolioValues: ArrayList<Double> stockPriceDataFiles: ArrayList<File> numberOfStocks: int confidence: int timePeriod: int = 10 numberOfSimulations: int = 1000 portfolio: Portfolio optionPricingType: String 	
<ul style="list-style-type: none"> MonteCarloSimulation(int,int) MonteCarloSimulation(ArrayList<Double>,ArrayList<File>,int,int) MonteCarloSimulation(Portfolio,int,int) MonteCarloSimulation() computeForPortfolio():double[] computeValueAtRisk():void computeForOneStock(double,double):double[] computeForMultipleStocks(ArrayList<Double>):double[] priceOptionUsingBlackScholes(Option):double[] priceOptionUsingMonteCarlo(Option):double[] getDiscountedValue(double,double,double):double priceOptionUsingBinomialTree(Option):double[] simulateReturns():ArrayList<double[]> simulatePrices(double,double):double[][] estimateVaRForBacktesting_OneStock(int):double[] getVarOneStock(double[]):double getMonteCarloFinalVar():double getMonteCarloMaximumVar():double setPortfolioValues(ArrayList<Double>):void setNumberOfStocks(int):void setTimePeriod(int):void setNumberOfSimulations(int):void setOptionPricingType(String):void 	

Figure 7.4: MonteCarloSimulation.java

<<Java Class>>	
BinomialTree	
com.nm.var.src	
<ul style="list-style-type: none"> S: double X: double T: double volatility: double interest: double optionType: int numberOfSteps: int dt: double p: double optionPrice: double 	
<ul style="list-style-type: none"> BinomialTree(double,double,double,double,double,int) BinomialTree(Option) buildStockPriceTree():void calculateOptionPrices(double[][]):void printTree(double[][]):void getOptionPrice():double 	

<<Java Class>>	
BlackScholes	
com.nm.var.src	
<ul style="list-style-type: none"> CALL: int = 0 PUT: int = 1 	
<ul style="list-style-type: none"> BlackScholes() compute(int,double,double,double,double,double):double 	

Figure 7.5: Classes for option pricing algorithms - BinomialTree.java & BlackScholes.java

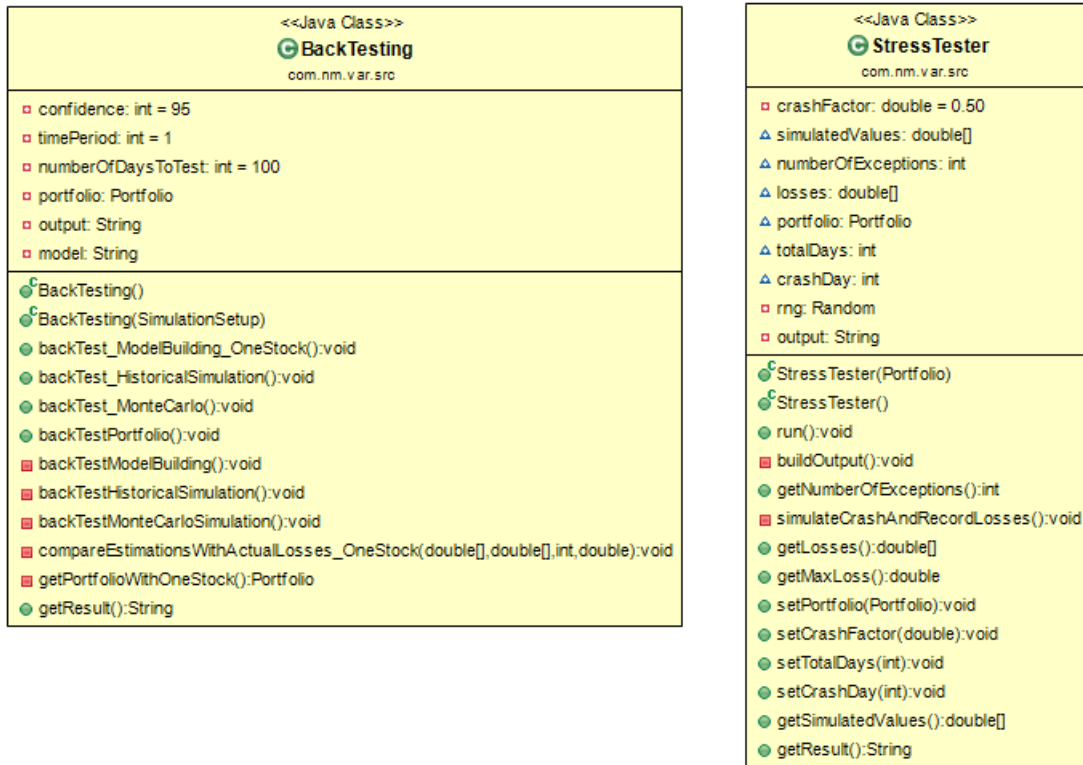


Figure 7.6: Classes implementing Backtesting and Stress testing

with many Asset and Option objects and passed between the classes in the program as a single entity rather than a large number of individual variables. This makes the process of programming the computation of VaR easier as data about the contents of a portfolio can be extracted from a Portfolio object and subsequently the Asset and Option objects. Essentially, these custom datatypes make the software implementation of a portfolio less repetitive and easier to understand and use.

VarUtils, figure 7.8, is a static utility class with methods accessible to all classes in the program. It combines some common functions and helper methods into one reusable class to help reduce code repetition. Some of the responsibilities of VarUtils.java include obtaining returns from historical stock data, computing volatilities from returns, generating covariance matrices, obtaining values at percentiles for a given series of data and providing some constant variables used across the program for identifying VaR models and option pricing models selected by the user.

To make adding a portfolio to the program easier, a selection of hierarchical modular dialogs have been created. The AddPortfolioDialog.java class allows the user to subsequently configure the portfolio by adding assets through the AddAssetDialog.java class and options through the AddOptionDialog.java class, see figure 7.9. All dialogs have their own `checkAndSaveXXXX()` method to verify the data input for each entity and present the user with an appropriate message if something is missing. Portfolios created within the AddPortfolioDialog object are passed back to Value at Risk program and made available to the user for computing VaR.

In order to simplify the software implementation around passing the selected VaR and option pricing models, time horizon, confidence level and portfolio for VaR computation, a SimulationSetup datatype to hold all of that data has been created, as shown in figure 7.10. The SimulationSetup object can also be used to hold data about a Backtesting setup a user has created by simply omitting the option pricing setting.

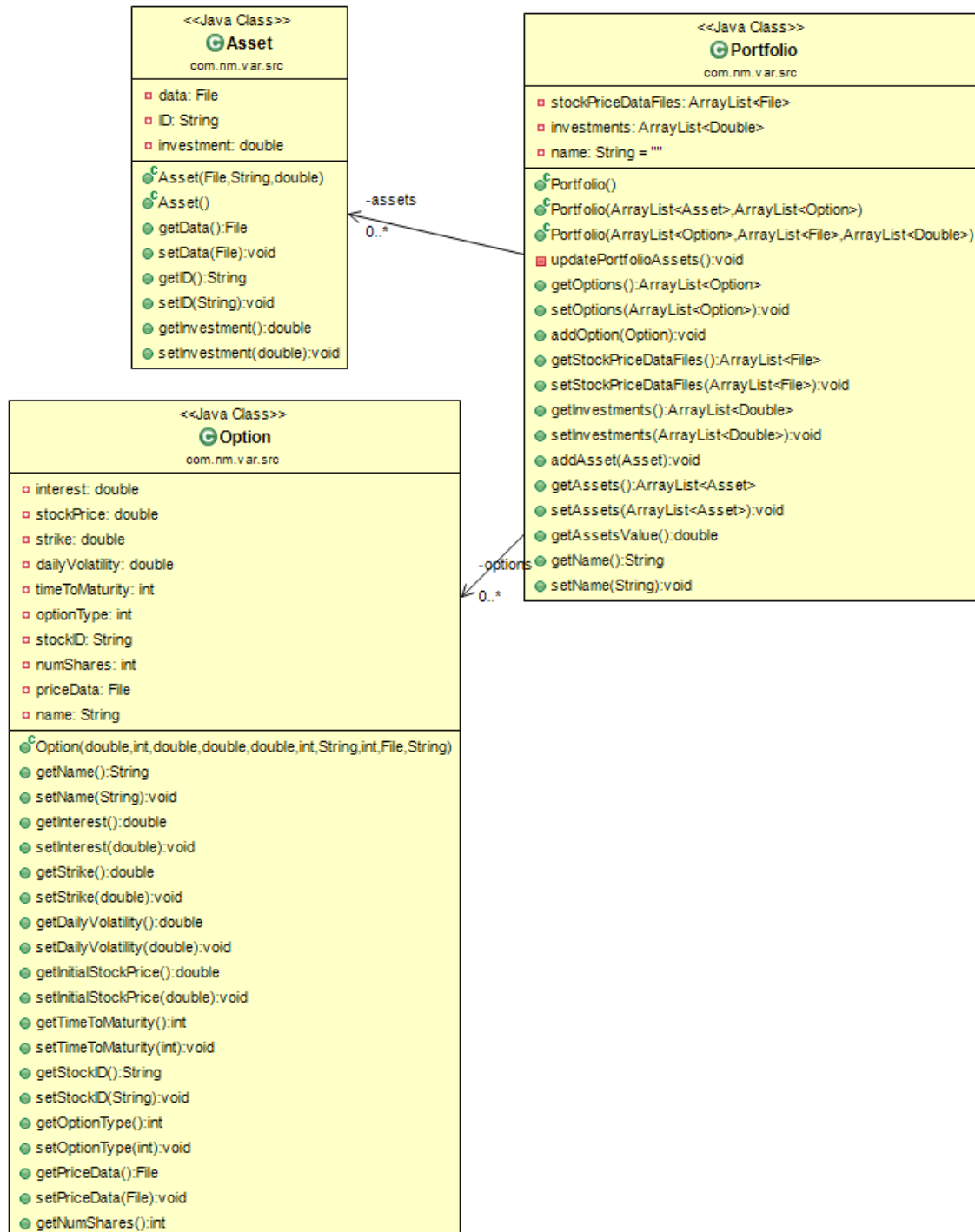
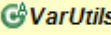


Figure 7.7: Collection of classes representing a portfolio in the program

```

<<Java Class>>

com.nm.var.src

S F AMERICAN_PUT: int = 0
S F AMERICAN_CALL: int = 1
S F EUROPEAN_PUT: int = 2
S F EUROPEAN_CALL: int = 3
S F ASIAN_PUT: int = 4
S F ASIAN_CALL: int = 5
S F BS: String = "BS"
S F MC: String = "MC"
S F BT: String = "BT"
S F MB: String = "MB"
S F HS: String = "HS"
S F DAYS_IN_YEAR: int = 252
S F DAYS_IN_MONTH: int = 21
S F ERROR_VAL: int = -1
S F COMMA: String = ","
S lambda: double = 0.94
S firstDayVariance: double = 0.01
S firstDayReturn: double = 0.02
S gamma: double = 0.05
S alpha: double = 0.13
S beta: double = 0.90
S omega: double = 0.000002

C VarUtils()
S square(double):double
S root(double):double
S round(double):double
S getReturnsFromFiles(ArrayList<File>):ArrayList<double[]>
S getReturnsFromFile(File):double[]
S getClosingPrices(File):ArrayList<Double>
S computeDailyReturns(ArrayList<Double>):double[]
S computeVolatility_Standard(double[]):double
S computeVolatilitiesFromReturns(ArrayList<double[]>):ArrayList<Double>
S computeVolatility_EWMA(double[]):double
S getVariance_EWMA(int,double[]):double
S getCovariance_EWMA(int,ArrayList<Double>,ArrayList<Double>):double
S computeVolatility_GARCH(double[]):double
S getVariance_GARCH2(int,double[]):double
S getVariance_GARCH(int,double,double[]):double
S getCovariance_GARCH(int,double,ArrayList<Double>,ArrayList<Double>):double
S generateCovarianceMatrix(ArrayList<double[]>,int):double[][]
S getCovariance(double[],double[]):double
S decomposeMatrix(double[][]):double[][]
S getCovarianceManually(double[],double[]):double
S getPercentile(double[],int):double
S convertToCallOrPut(int):int
S sumOf(ArrayList<Double>):double
S getReturnsOverVarHorizon(double[],int):double[]
S CNDF(double):double
S roundTwoDP(double):double

```

Figure 7.8: Utility class VarUtils.java

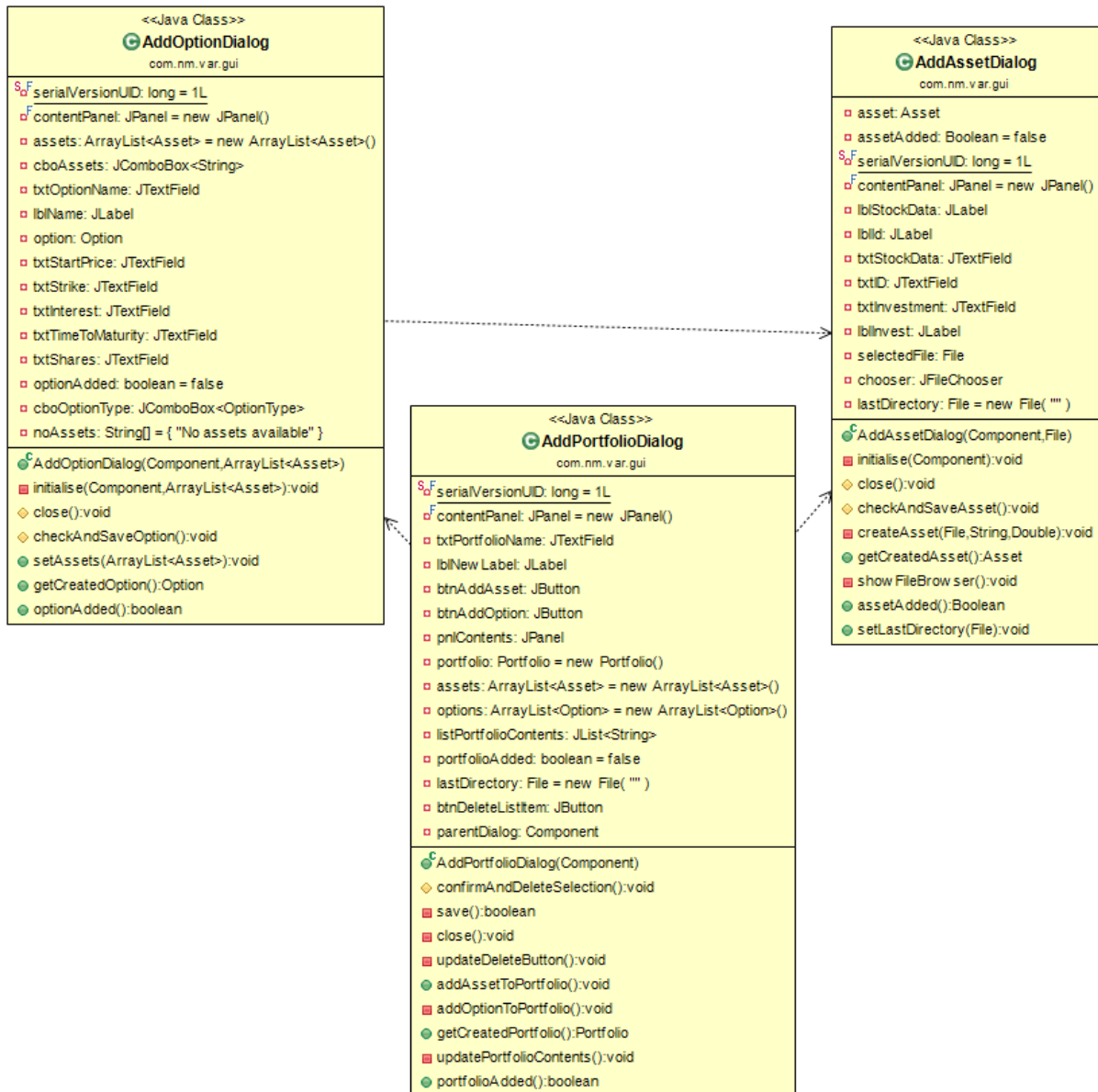


Figure 7.9: Collection of classes implementing dialogs to create portfolios

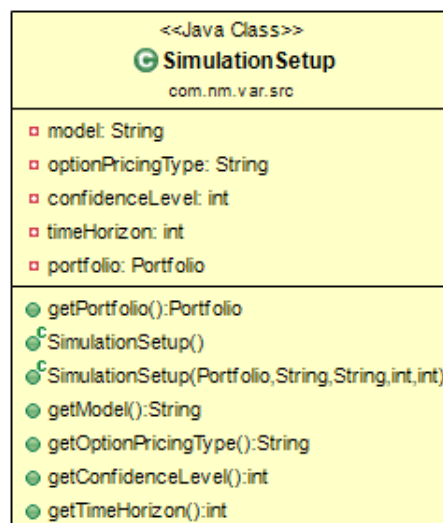


Figure 7.10: SimulationSetup.java class to hold the user-set parameters for VaR computation

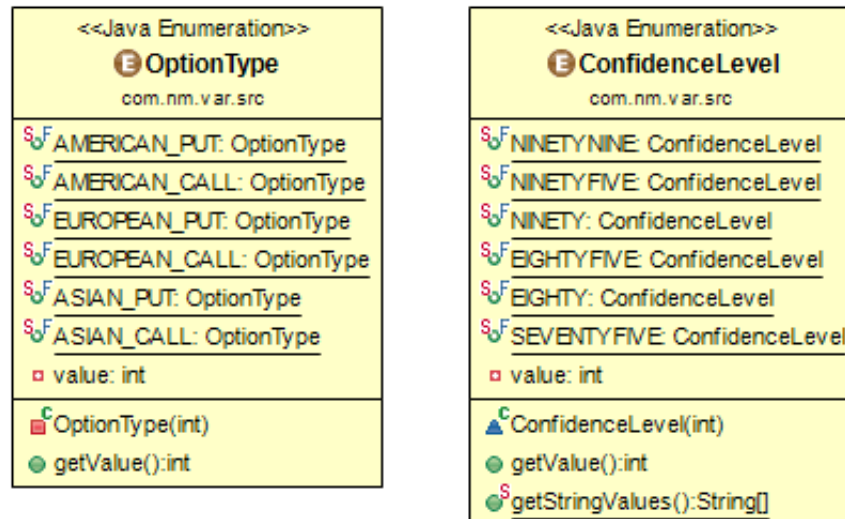


Figure 7.11: Enumerations for confidence level and option types

Finally, a couple of enumerations have been used to define the sets of usable confidence levels and option types within the program. These enumerations are used to populate the drop-down selection boxes for confidence levels for VaR computation and to define the type of an option upon its creation. The enumerations are also useful for identifying the values of user-provided settings for the above mentioned variables when it comes to computing the VaR of a portfolio.

7.3 Implementation of Algorithms

The algorithms implemented for the interim submission have been implemented to be as readable as possible. However, there are some aspects of the computation of Value at Risk for which there were no improvement substitutions. These included:

- High number of copy & compare operations - most of the algorithms in the program need to work with arrays or lists of numbers in order to compute returns, variance, covariance and percentiles. These operations can slow down the program given a large enough input.
- Array copying and sorting - algorithms such as the Historical Simulation and Monte Carlo Simulation rely heavily on sorted arrays to work. Sorting an array is an expensive operation in terms of running time.
- Recursive computations - the EWMA and GARCH variance estimations are recursively computed, which can be slow for a large number of returns.

In order to keep the running time of the algorithms low for a large input, I looked to optimise the code at regular intervals, which included discarding unused variables, removing unnecessary computations, minimising use of `if-else` ladders in favour of `switch` statements and combining iterative loops where possible.

When programming a software based project I agree that it is important to follow some good Software Engineering principles so that the project not only functions well, but is also designed well to allow for improvements and extensions to be developed easily. In the following sections I talk about a few principles I have followed when writing my Value at Risk program which have helped improve the standard of my code.

7.4 Agile Development

The software for this project was implemented under the Agile methodology. This methodology was chosen because the time frame for the implementation and delivery of the software did not allow time for a sequential cycle of the software development life-cycle. Thus, the analysis of the requirements, design, implementation and testing of the software solution was done for every feature in quick succession. An agile methodology places emphasis on fast planning and implementation, thus delivering an increasingly featureful product over regular time periods. The Agile manifesto states that it values:[2]

Individuals and interactions over Processes and tools
Working software over Comprehensive documentation
Customer collaboration over Contract negotiation
Responding to change over Following a plan

Which means that it was important to be motivated and organised to complete the tasks agreed with the client, having a functioning program was more useful than showing the client documents relating to it, client involvement was important in understanding the requirements which should be addressed by the software and that changes in requirements were easier to respond to when the agile process was followed during the development of the Value at Risk program.

It made sense to utilise the agile development process because meeting with my supervisor (the client) were on a weekly basis, so it was appropriate to update him with the progress made over the past week, highlight any issues with the development and talk about new tasks, just as the agile methodology outlines when it mentions keeping the customer involved frequently and gaining their feedback so their requirements can be met.

The usual process for feature development could be generalised as follows:

1. agree task with supervisor, note down requirements
2. research task, if needed
3. plan how to integrate new feature into current implementation
4. add new unit tests and verify previous functionality not broken
5. implement feature
6. test using known inputs and expected results
7. speak to supervisor if clarification needed

7.5 Good code design

During the implementation of the Value At Risk program I have tried to follow some good software engineering principles, some of which are explained below:

- DRY (Don't Repeat Yourself) - Where possible, I have tried to avoid writing repeated code by extracting common code to its own method and call it wherever needed. The best example of this principle in my program can be seen in the VarUtils.java class which contains the most commonly used methods in the whole program.
- KISS (Keep It Simple, Silly) - I have tried to avoid using complex code in order to make the program easier to debug and fix.
- Code readability - I have made use of descriptive variable names and assigned complicated calculations to such variables in order to improve the readability of my code.
- Code formatting - I have used a consistent and clear code formatter by modifying the default formatter in Eclipse which makes code easier to read.
- Custom datatypes - Entities such as an asset, option or portfolio have been represented as a collection of underlying data in order to make the program structure simpler and easier to understand. This also made passing this entities between methods and classes easier than passing a variety of variables.
- Javadoc and inline comments - In line comments are present in the code to help explain complex parts of an algorithm. Javadoc comments for methods and classes are also present to help explain the role of a method or class, its input parameters and its output. Javadoc can be exported as a HTML document and used for understanding the program through the descriptions of its classes and methods.

I have decided against using code style checking tools, such as CheckStyle. This is because, from my own experience in professional software engineering, such tools are not considered productive as a team of software developers working on the same product will produce a slightly different style of programming from each team member. Professional development

teams are under high pressure to deliver features which have been well tested and document and tools such as CheckStyle would take up a lot of time away from implementing those features. While CheckStyle can be configured to be more lenient in terms of its code style governance, I believe it is not useful to use a tool if compromises have to be made in order to accommodate it.

7.6 Unit Testing

JUnit is a unit testing framework used in Java. It can be used for test driven development, where a feature is developed by writing tests first and then making them pass by implementing the feature incrementally. This development approach ensures all the code written has been tested extensively.

Unit testing is the process by which individual units of source code (e.g. a function/method, class or collection of methods) are tested to verify that they are successful in achieving their individual goals. The idea of unit testing is to verify the correct functionality of the modules of the program to assert that a larger module is functioning correctly.

I made use of JUnit to unit test the Value at Risk project code where possible in order to test the functionality of individual methods and classes by preparing some input parameters and comparing the data returned from the module being tested with some expected output. Unit testing comes with benefits such as the testers being able to find additional problems in the code they are testing early on, allowing more time for the bugs to be fixed and to check if some refactoring/modification in the code has caused an issue by running the existing unit tests and checking for failures.

Through this practice, I have been able to build up a collection of unit tests which can be run after every modification to ensure no previous functionality is broken and that new functionality meets its requirements.

Unit testing also encourages modular code design, a practice where algorithms and functionality is developed in individual modules rather than a big bulk of code. The advantage in developing code in modules is that it makes the code easier to understand, fix and refactor.

7.7 Software versioning and revision control

I have used Subversion (SVN) for maintaining my project code safely in a secure repository. Subversion is a commercial version control system designed to provide secure repositories for software projects. It provides all the features expected from a version control system e.g. managing a list of modified files, submitting the files to the repository, checking out entire projects, merging and integrating files from one branch to another etc.

A version control system provides complete file revision history, a centralised repository which is access-controlled and management of modifications made to files (source code, in this case).

It is necessary to use a version control system in software engineering because there are always multiple users working with and updating the same source code. This code needs to be available for modification, fixing and updating to every user in a way which does not compromise its integrity.

Initially, I did not realise that I had not created a proper project structure in my SVN repository as I had not used SVN for a whole year prior to that. However, later on, I created a new project structure on my repository complete with the trunk, branches and tags which I was able to make better use of.

I regularly committed work to my repository and wrote informative comments for each update I committed to the repository.

After setting up the proper project structure, I also branched off the trunk to start the refactoring work on my program. I made regular commits to the refactoring branch until I was ready to merge the refactored program back into the trunk.

Finally, I created tags whenever I had a candidate, fully working, release of my program, one of which is the InterimSubmission tag.

In the second term, I refactored the program structure in a branch just for refactoring work so that classes represented an appropriate entity in the system which also made extending the program a lot easier as the system was now more modular than before. After merging the changes from the branch to the trunk successfully I had a version of the trunk which was fully functioning and passed all tests, as the trunk should always have working code.

I also created a branch for working on the user interface, however, some of the back-end work had to be done on this branch due to urgent tasks and bug fixing. Nevertheless, working in branches meant I kept the trunk safe from dangerous changes to my project and kept on adding functionality until it was fully working.

7.8 MVC Design Pattern

Design patterns can be utilised when developing software to lay out the structure of a program in a logical and efficient manner. Many programs suffer from classes which have far too many responsibilities. It is considered a good software engineering practice to write classes with high *cohesion* i.e. classes whose methods (responsibilities) work together to achieve the goal of the class they are contained in. In the context of this project, this explicitly means that the front-end of the program should focus entirely on being just the front-end and not try to do complicated computations, which the user may want the program to do, by itself.

Also, when implementing a graphical user interface (GUI) for a program, it is considered a good software engineering practice to reduce the reliance of the front-end classes and the back-end classes on each other, as well as other classes in the program, otherwise known as reducing the *coupling* between the classes.

I have implemented the Model-View-Controller (MVC) architectural design pattern in order to reduce the coupling between classes which implement the user interface and the back-end of the program and to increase the cohesion of these classes. The MVC pattern is the ideal solution to help decouple the intensive computations and program logic from the GUI. It relies on the following structure concerning the GUI and rest of the program:

- Model - entity representing the data produced from requested back-end computations, and the updates using this data.
- View - (GUI) displays the contents of the Model, only concerned with how data is presented to the user and is updated whenever the Model's data is changed.
- Controller - looks out for user interactions on the View, translates these interactions

into tasks for the model to do.

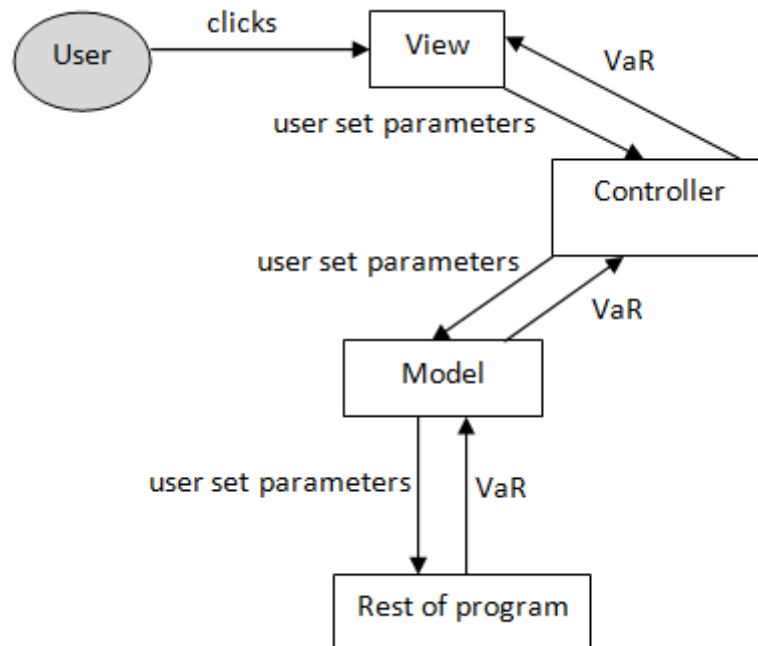


Figure 7.12: Illustration of the Model-View-Controller design pattern for VaR

In the context of the Value at Risk program, a sample interaction between the components of the MVC design pattern can be shown:

- User clicks the ‘Compute’ button on the View to calculate VaR using some selected parameters.
- Controller sees this event on the View, gathers the parameters set by the user and asks the Model to obtain the VaR using these parameters.
- Model uses the parameters it has been passed to create objects of the required type in the rest of the program and returns the computed VaR back to the Controller.
- Controller receives the VaR and updates the results on the View with the data it received from the Model.

It can be seen in figures 7.12 and 7.13 that the Model is completely decoupled from the View using this design pattern as the Controller mediates the data flow from the View to the Model and vice versa.

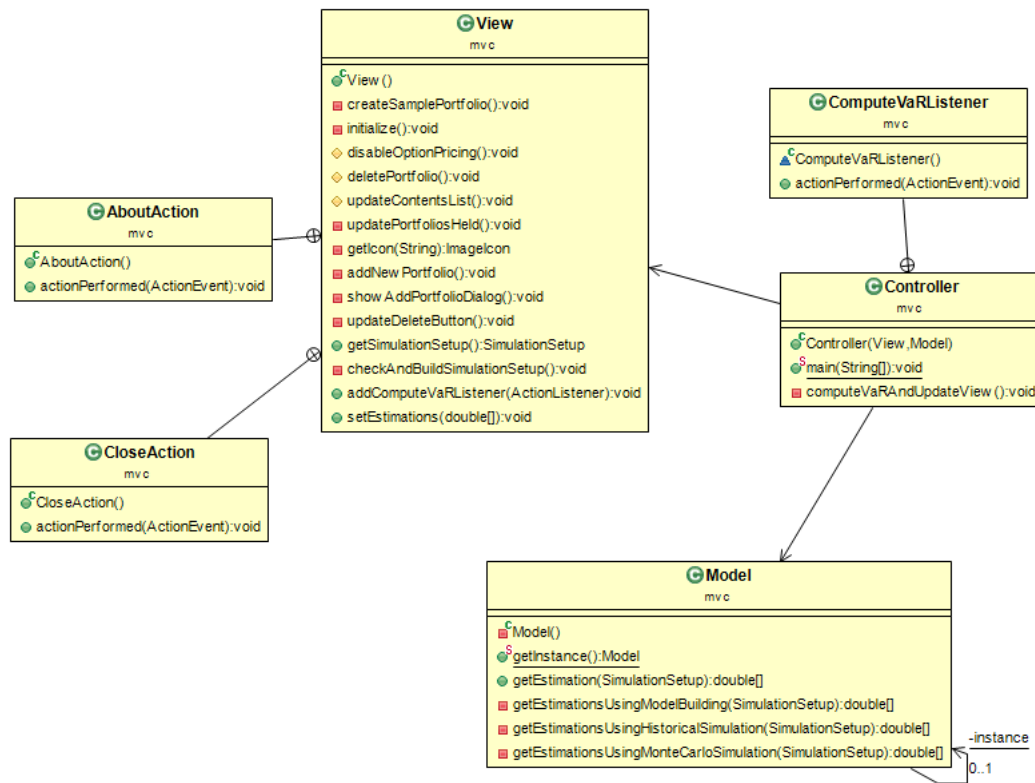


Figure 7.13: Class diagram of the MVC pattern in the software implementation.

7.9 Facade and Singleton Design Pattern

As the Value at Risk program has only one model that calculates the VaR using the rest of the program, I deemed it useful to make the Model in the MVC design pattern a *Singleton Facade*. The Singleton creational design pattern allows only one instance of an object to be active at any time and the Facade design pattern provides a simple interface for the Controller to get the VaR calculation from the objects implemented behind the Model.

The singleton pattern is implemented in the Model by having a `private` constructor in the Model class, so that it can only be instantiated once (inside the Model itself) and a `public` method to get the created instance of the Model from other classes. An illustration of this technique can be seen in figure 7.13 where the `Model()` constructor has a red dot to indicate that it is a `private` constructor and the `getInstance()` method has a green dot to indicate that it is a `public` method.

Implementing the Model as a facade hides the complexity of the underlying program from the Controller. The Controller does not need to know how the VaR calculation models are implemented as it is only interested in obtaining the VaR estimates for some user-defined setup from the View. This pattern allows flexibility in the development of a system as it makes the code and structure of the program easier to understand, test and refactor by reducing the dependency of the front-end code on the implementation of the VaR models hidden behind the Model. An illustration of this design pattern is shown in figure 7.14.

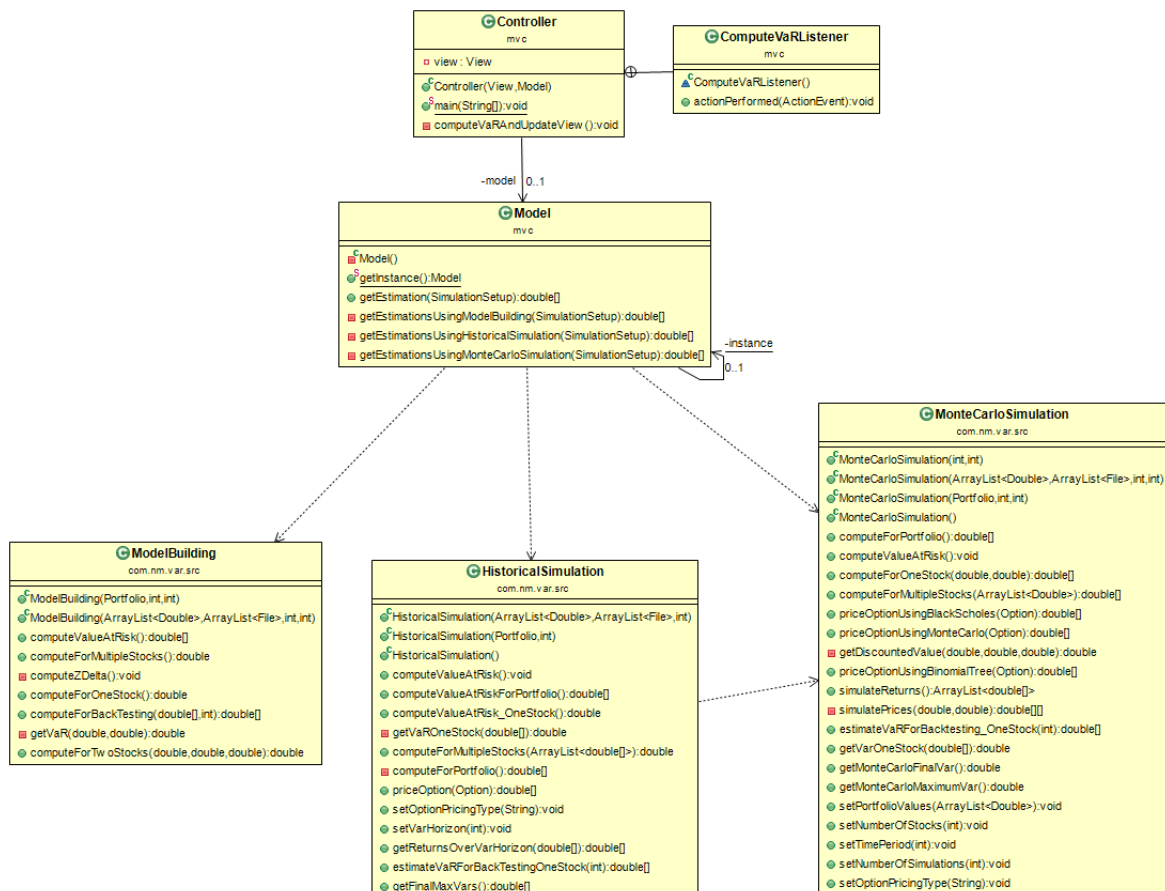


Figure 7.14: Class diagram showing the Facade design pattern for the Model.

Chapter 8: Professional Issues

Computing and IT professionals are represented internationally through many organisations, one of which is the British Computer Society (BCS). Like any professional organisation, the BCS lays out a code of conduct for professionals in its field. As a soon-to-be member of the computing profession, I am obliged to follow such codes of conduct and guidelines when engineering and documenting software when faced with certain professional issues during the process.

During the Value at Risk project, I have come across professional issues concerning plagiarism in the development and documentation, and usability of the software produced. The BCS Code of Conduct mentions that a computing professional should ensure that the technology they produce is accessible to all and that they should always act in a honest and professional manner about their work. It is important that ethics such as the ones laid out in the BCS Code of Conduct are followed by all computing professionals in order to give a good impression of the profession to others.

In the following sections, I discuss the impact of these issues on the computing profession and how I dealt with them.

8.1 Plagiarism

I had very little previous knowledge about risks concerned with investing in the stock markets. Hence, in order to understand, document and develop the solution for calculating Value at Risk and the techniques underlying its computation, I had to read thoroughly about every one of its aspects.

While reading through material in relevant books and from the web, I began documenting the background theory for this report. It was important here that I did not claim the work of others as my own and that references to knowledge and ideas which were not inferred by myself were made clearly to the reader.

Claiming the work of others as your own is known as plagiarism and it is a very serious matter which professionals must be concerned with. The penalties of plagiarism and the legal implications of related offences such as copyright violations are substantial and no computing professional should ever put themselves in such a position.

In this project, I have tried my best to cite components which were not my work. Which citing other works in a report is relatively simple, citing code and resources from other developers is also something computing professionals must be aware of. I openly talk about the external Java libraries which were used for developing the program for this project which can be seen in section 9.2.1 of this report. In developing the user interface for the Value at Risk program, I also made use of some icons which were made by a developer known as mjames at <http://www.famfamfam.com/lab/icons/silk/>. I ensured that the icons were legally usable for my work by reading through the licensing requirements and that I referred to the original developer when using these icons in the javadoc for my code, as below.

On another occasion, I had to use an implementation of the Cumulative Normal Distribution Function when pricing options using the Black-Scholes formula. I obtained an implementation of this function from one of the resources within the CS3930 Computational Finance course. The original location of this code is clearly referenced in the Javadoc comments for the

```

public class View
{
    /**
     * The main frame for the user interface.
     *
     * All icons courtesy of <a href="http://www.famfamfam.com/lab/icons/silk">mjames at
     * famfamfam.com</a>
     */

    private JFrame          frmApp;

```

Figure 8.1: Reference to icons developed by mjames in my code.

CNDF(double x) function in the VarUtils.java class, as shown in figure 8.2.

```

/**
 * Cumulative Normal Distribution Function
 * Taken from CS3930 Moodle Website
 *
 * @see <a href="http://moodle.rhul.ac.uk/mod/resource/view.php?id=123643">CNDF Function</a>
 * @param x
 *      The number for which to compute the value
 * @return
 */
public static double CNDF( double x )
{
    int neg = ( x < 0d ) ? 1 : 0;
    if( neg == 1 )
        x *= -1d;

    double k = ( 1d / ( 1d + 0.2316419 * x ) );
    double y = ( ( ( ( 1.330274429 * k - 1.821255978 ) * k + 1.781477937 ) * k - 0.356563782 )
                * k + 0.319381530 )
                * k;
    y = 1.0 - 0.398942280401 * Math.exp( -0.5 * x * x ) * y;

    return ( 1d - neg ) * y + neg * ( 1d - y );
}

```

Figure 8.2: Reference to source of CNDF function in my code.

8.2 Usability

When developing a software product, it is important to take into account the distribution of its potential users. A good computing professional would think about the different types of users which may use their program and take pre-emptive action to ensure the software is easily accessible to such users. For example, some users may struggle to read text if it is not in high contrast to its background and more generally, some users will need the text in the program to be large enough to read.

To this end, I have ensured that the user interface of the Value at Risk program contains a mix of colours in high contrast to each other. Moreover, the font has been emboldened in the main components of the program and the font size has been increased wherever possible. These two small improvements should make the text in the program more readable for its users.

A broader discussion, with examples, on the usability of the Value at Risk program is available in section 6.3 of this report.

Chapter 9: Technologies Used

9.1 Java Programming Language

I chose to write the code my Value at Risk project using the Java programming language. This is because almost all of my programming experience comes from working with Java, especially from my Year in Industry. Java provides ample resources, libraries and extensions for the scope of this project and I felt confident I would be able to implement all of the algorithms I need for this project using Java.

Java is considered to be one of the safer languages in terms of the likelihood of memory leaks, arrays being overwritten and the use of strong typing. While it may be that languages such as C++ are more efficient in terms of memory usage, efficiency was not a major factor for this project at this time. If the project was being developed for high frequency industrial use with very large sets of data then some consideration would have been given to the potential performance and efficiency constraints of the programming language used to implement such as system.

9.2 Development of Code

In order to develop the code for the Value at Risk project, I made use of the Eclipse IDE (Integrated Development Environment). Eclipse supports multiple languages and numerous plug-ins for added functionality and interaction with third-party products. Eclipse is used widely for development using the Java programming language as it provides intelligent code-completion, spell-checking, checks for basic coding and compilation errors and a highly customizable and feature-packed interface. Moreover, I was able to consistently make use of a variety of keyboard shortcuts to aid fast navigation and code refactoring across my Java project.

9.2.1 External libraries

I have made use of the following external Java libraries in my project:

Apache Commons Math 3.2 - provides useful statistics functions which are used to obtain values at percentiles and triangular covariance matrices through Cholesky decomposition for VaR calculations

Apache Commons CLI 1.2 - library providing enhanced command line parsing features, may be useful if the program grows to accept more complex parameters from user

Apache Commons IO 2.4 - library providing enhanced file handling utilities for Java, useful when implementing the GUI where the user is able to select files from their directories, which need to be verified for use within the program.

9.3 Knowledge From Previous Courses

During my Computer Science degree at Royal Holloway, I have gained a lot of knowledge from past courses which were useful in the implementation of the software for the Value at Risk project.

- good coding standards - CS2800 Software Engineering
- design patterns - CS2800 Software Engineering
- algorithms and improvements - CS2860 Algorithms & Complexity I
- multithreading - CS3750 Concurrent & Parallel Programming and CS2850 Network Operating Systems
- planning work and client meetings - CS2810 Team Project in Software Engineering

Chapter 10: Evaluation

10.1 Experience of Project

At the beginning of the project, I have to admit I did not have a very clear idea of exactly how I would be implementing the Value at Risk program. This was because I did not get a chance to review the project over the summer as I was on my Year in Industry, working full time as a Software Engineer. After reading around the concept behind Value at Risk I slowly began to understand its relevance in the financial world.

After discussing the initial expectations with my supervisor, I was able to plan my project work until December in terms of reports written and programs developed. Mathematics is not a strength of mine but I was confident that I would be able to implement the computations using my programming skills after I'd taken time to understand the individual algorithms by hand using John Hull's book: Options, Futures and Other Derivatives.

I had originally struggled to implement what I now consider to be relatively simple algorithms and I feel that was because I had not yet grasped their roles in the computation of Value at Risk. However, as the term went by, I began to take a great interest in the program and as the Computational Finance course covered Value at Risk everything seemed to come together and make sense for me.

As I had been writing the program in one big class, from the start of the project to mid-November, I decided to improve its design by refactoring the program into logical sub modules and implemented command line parameter handling for the command line interface which made the program relatively easy to use for the December review.

There was still a lot of back-end work left to do at the start of the second term and the volume of project work, including report-writing and developing code, increased by a considerable amount. However, as the term went by, increasing amounts of effort was also being put into the project as more complex pieces of code were implemented together with a graphical user interface, where each task presented interesting, and sometimes time consuming, challenges.

I have learnt the value of researching a topic well while doing this project. Having an understanding of what I needed to implement or write about in my report was key to getting it right as soon as possible. I had a chance to revisit and use the knowledge from the second year Software Engineering module, when implementing design patterns for the program for example. Doing the project also helped refresh much of the key technical knowledge I needed in my job interviews and assessments and its experience assisted in securing a job after I graduate.

Overall, I felt at the end that the project went well and gave me a chance to utilise the skills I gained from my Year in Industry on a project I had full ownership of. While at times the amount of effort I could have put into the project may have been higher, I am pleased to have got this far in the implementation of the Value at Risk program and especially the final report for the project.

10.2 Achievements

One of the main achievements for this project is that all the final deliverables were implemented in the program including portfolios containing many stocks and options, Monte Carlo simulation and a graphical user interface.

Algorithms for option pricing using Black–Scholes formula, Binomial Trees and Monte Carlo simulation were also implemented for the Historical Simulation and Monte Carlo VaR models so that VaR could be computed for portfolios containing options.

Stress testing has been implemented as a complement to Backtesting. Despite being naive, the stress testing model in the program allows the user to see what losses their portfolio could suffer in the event of a crash in the markets.

Not being highly mathematically skilled, I felt that I manage to overcome my fears about the mathematical aspects of this project through practice and research.

Lastly, I felt that computing the maximal VaR during simulations was a useful feature alongside the computation of the more usual, percentile-based, VaR as this gives the user an idea of the actual worst possible loss for a certain set of parameters.

10.3 Enhancements

Almost all software solutions can be enhanced and extended in many ways. The Value at Risk program covers many aspects of the concept of Value at Risk but could also be extended by computing Conditional VaR. Another enhancement to the current interface could be the addition of visual representations of the data held in the program, for example graphs showing the historical stocks prices for a portfolio and the simulations of the portfolio prices using Monte Carlo simulation over time could be easily implemented using open-source Java libraries such as JFreeChart.

The program should also be improved so that it can be resized and allow for high contrast or inverted colour schemes to be more accessible to its users. On top of that, it would be useful to save the portfolios into a local file and have the program reload them for use next time it is launched. Features to edit currently configured portfolios and to compute VaR for multiple portfolios at once would make a nice addition and be easy to implement.

The VaR models implemented could be improved by implementing a maximum likelihood estimator for optimising the parameters for the GARCH and EWMA variance models, which would improve the computations of VaR. Backtesting should also be extended to take into account large portfolios and greater flexibility in its input parameters.

The program could also provide a feature for obtaining stock price data automatically for a given symbol so that users do not have to download the data themselves. An option to simulate a series of stock prices would also make a good enhancement to the current product.

It is likely that I will utilise some of time after this term to implement these enhancements into the project with the aim of taking this project in the direction of being a publicly useful resource.

10.4 Conclusion

In conclusion, I felt that doing a final year project has been an enjoyable and rewarding experience where I was able to challenge myself by learning about a field in which I had an interest but no real knowledge about and implementing a program to compute one of the important risk measures in the financial world.

It is my belief that my software engineering credentials have been boosted through the experience of the project and that the project has helped me realise the importance of being an all-round good computing professional.

Appendices

Appendix A: Program Usage Instructions

A.1 Requirements

The Value at Risk program has been successfully tested for use on a machine running Java version 1.7.0_40. Any Java version above 1.7 should be able to run this program. No other specialist software is required to run the program.

A copy of the Java package is available at <https://www.java.com/en/download/>.

A.2 Launching the program

Please obtain the runnable jar file from the project repository at <https://svn.cs.rhul.ac.uk/personal/zvac052/Value%20at%20Risk/main/> and download the ValueAtRisk.jar file. Double-click the jar file to launch the program.

In order to use the ‘Sample Portfolio’ contained in the program upon start-up, please also download a copy of the testing folder to the same directory as the jar file.

If you have not yet obtained historical stock price data you can download it for some stocks from <http://uk.finance.yahoo.com/> and searching for a stock next to the ‘Look Up’ button. Once a stock has been found, click on ‘Historical Prices’ under the left-hand section where it is possible to define a date range to download data for. Select the date range and click ‘Get Prices’ and then scroll to the bottom of the table to find the link ‘Download to Spreadsheet’. Upon clicking this link, a Comma Separated Values (csv) file should be available to save in a location of your choice, which you can then use in the program.

Otherwise, some recent historical stock price data is available in the testing folder of the repository mentioned above.

A.3 User Manual

Upon launching the program, you will be presented with the main screen of the program, see figure A.1. Here you are able to see the ‘VaR’ and ‘Model Testing’ tabs.

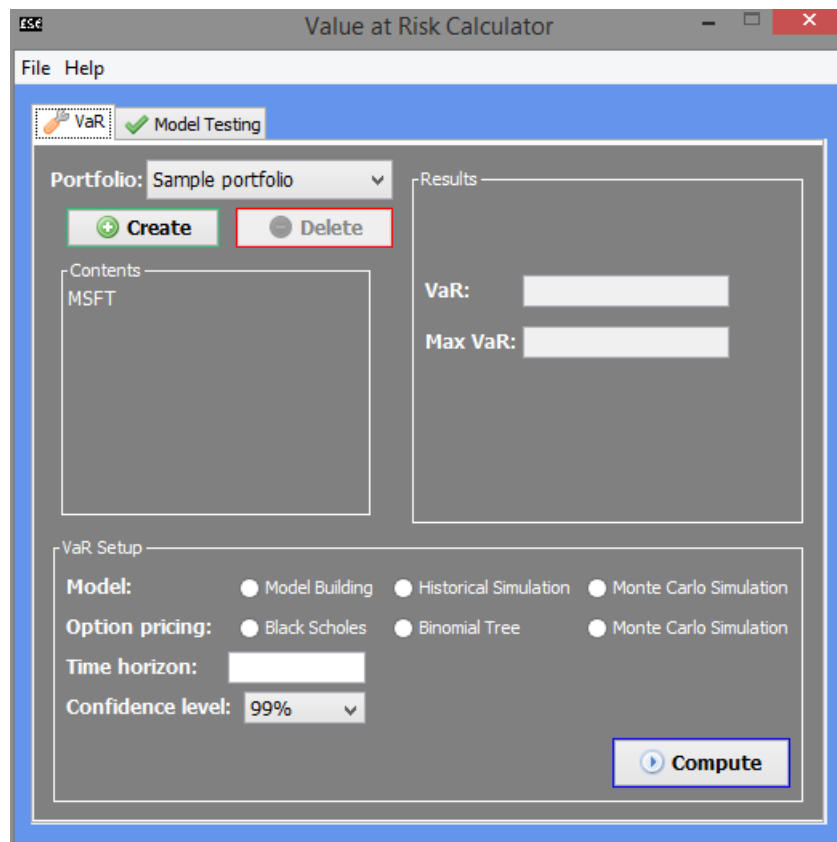


Figure A.1: Program when launched

A.3.1 VaR

The ‘VaR’ tab has all the tools necessary for a user to compute the Value at Risk of a portfolio. It contains (anti-clockwise from top-left):

- portfolio selection drop-down menu
- button to create a new portfolio, labelled ‘Create’
- button to delete the selected portfolio, labelled ‘Delete’
- panel labelled ‘Contents’ which displays the contents of the currently selected portfolio
- panel labelled ‘VaR Setup’ to allow specification of the VaR computation for the selected portfolio
- panel labelled ‘Results’ to show the computed Value at Risk and the maximum VaR (if applicable).

A.3.2 VaR Setup panel

The VaR setup panel allows the user to specify exactly the parameters for VaR computation for the portfolio selected from the drop-down menu.

The ‘Model’ buttons represent the VaR model to be used for VaR computation.

The ‘Option pricing’ buttons represent the type of algorithm used when pricing options in the computation of VaR for the selected portfolio.

The ‘Time horizon’ text box allows the user to specify the number of days to compute VaR over, e.g. 1, 2, 5, 10 etc.

The ‘Confidence Level’ drop-down menu allows the user to specify the confidence at which to compute VaR.

The ‘Compute’ button will initiate the computation of VaR using the defined parameters for the selected portfolio and display the VaR computations ‘Results’ panel.

PLEASE NOTE: Option pricing is not available within the Model Building VaR model. The program will disable the ‘Option pricing’ buttons and inform you of this when ‘Model Building’ is selected.

A.3.3 Results panel

The ‘Results’ panel displays two types of VaR computed for the selected portfolio.

‘VaR’ is the standard VaR measure and is computed for every selected VaR model.

‘Max VaR’ is the maximum VaR experienced during the Historical Simulation and Monte Carlo Simulation calculations and, thus, it will only be available when those models are selected for VaR computation.

A.3.4 Creating a portfolio

The ‘Create’ button under the portfolio selection drop-down menu opens a new dialog to allow the user to create a new portfolio, see figure A.2. To create a portfolio it is necessary to provide a portfolio name and at least one asset. Omission of a name or asset will result in a message being displayed to inform about missing data, see figures A.3 and A.4.

An asset can be added to the portfolio by clicking the ‘Asset’ button in the dialog. This opens new dialog to allow for an asset to be configured. All fields are mandatory in the ‘Add Asset’ dialog, see figure A.5.

To add historical stock price data, click on the ‘Browse’ button. This opens up a file browser which shows only the files of the ‘Comma Separated Value’ (csv) type as these are the types of the files which contain historical stock price data, see figure A.6.

Once the file containing historical stock price data is found, double-click on the file or click ‘Select’ to add it to the asset. Then, provide the investment made and an identifier (name) for the asset, see figure A.7.. Clicking ‘OK’ will add the asset to the portfolio and the asset will be shown in the portfolio contents, see figure A.8.

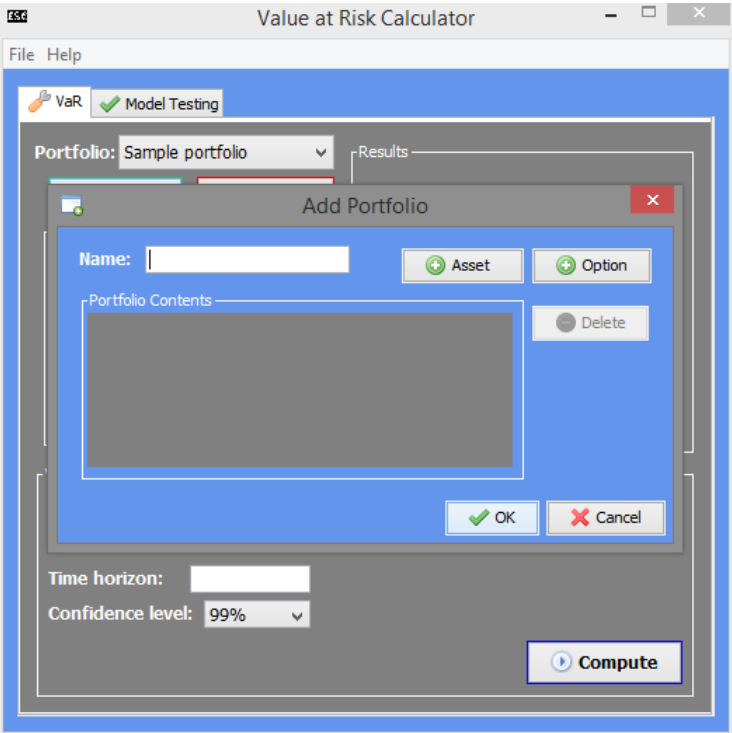


Figure A.2: Creating a new portfolio

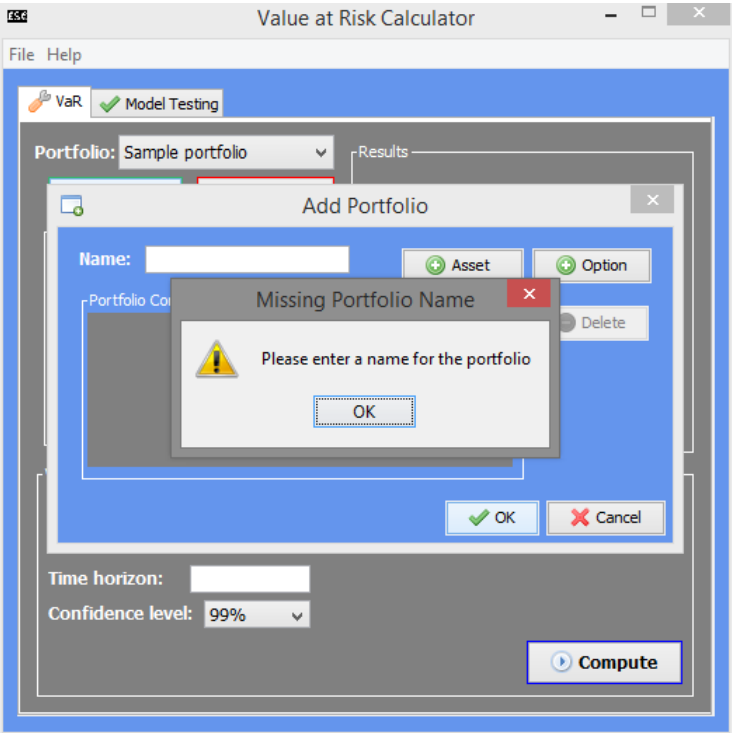


Figure A.3: Missing portfolio name

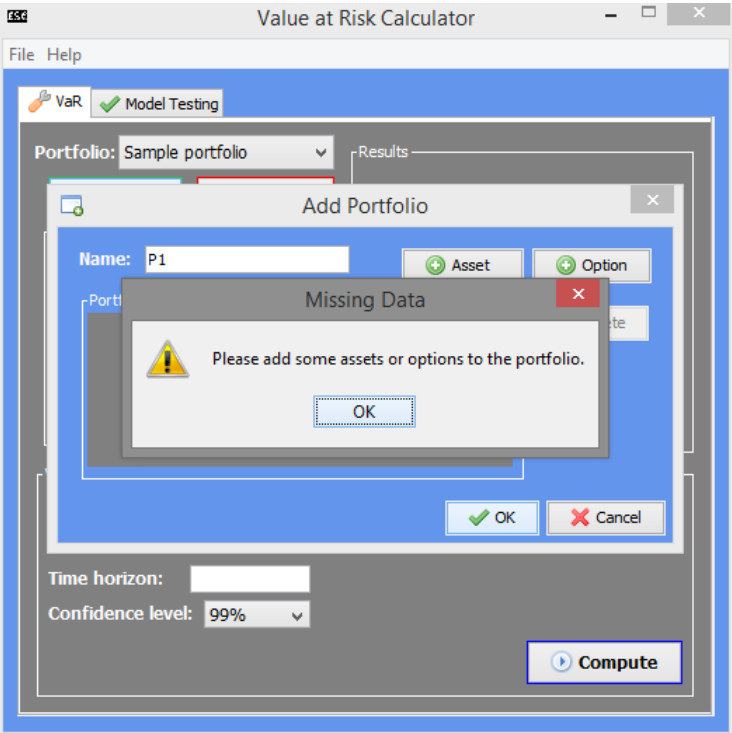


Figure A.4: Empty portfolio

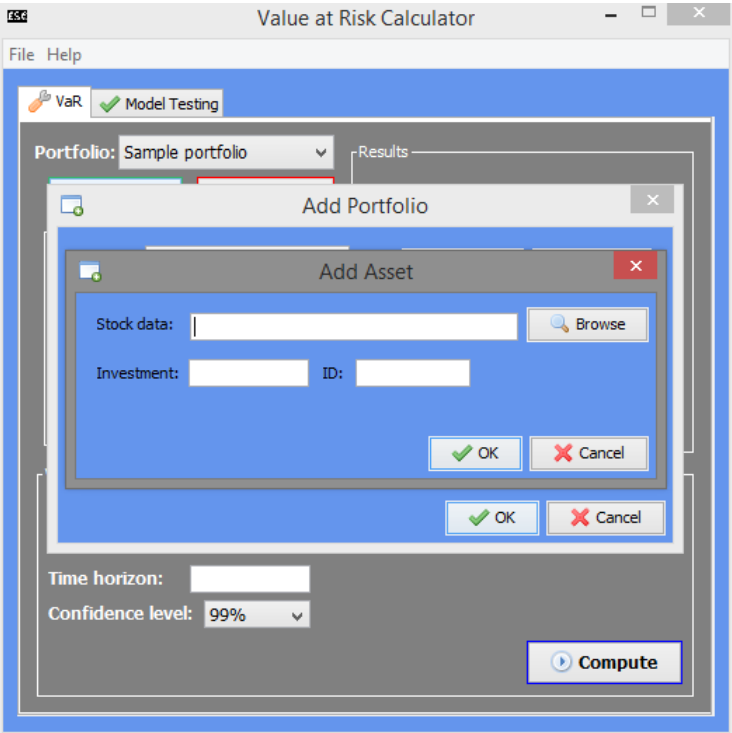


Figure A.5: Adding asset to portfolio

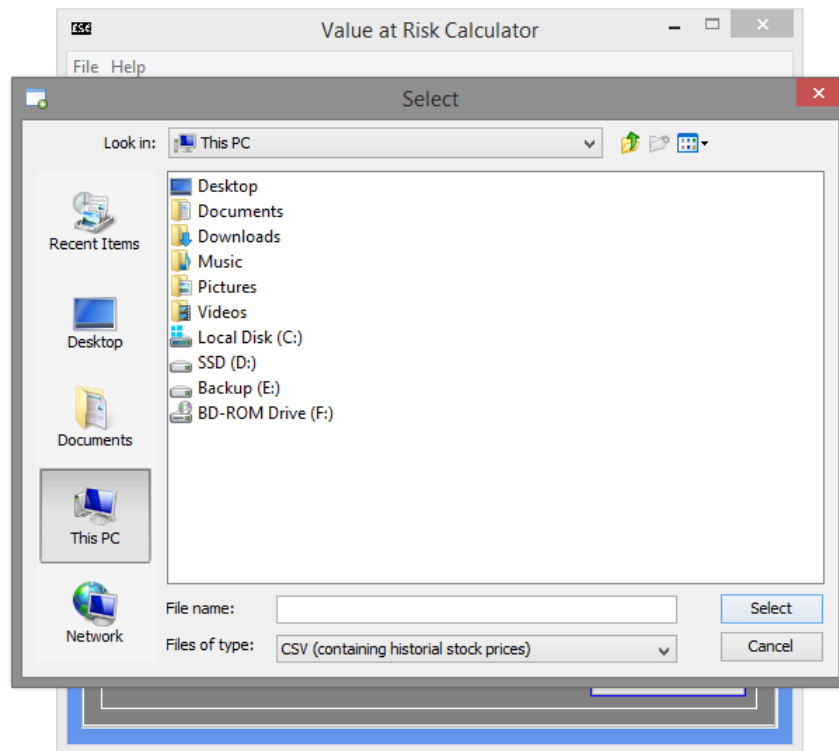


Figure A.6: Adding historical stock price data for asset

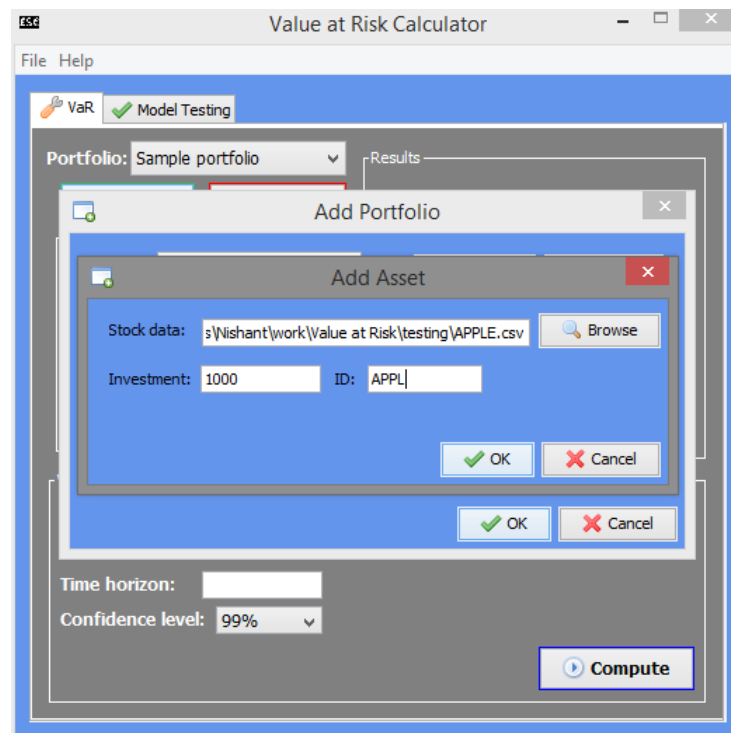


Figure A.7: Adding asset to portfolio

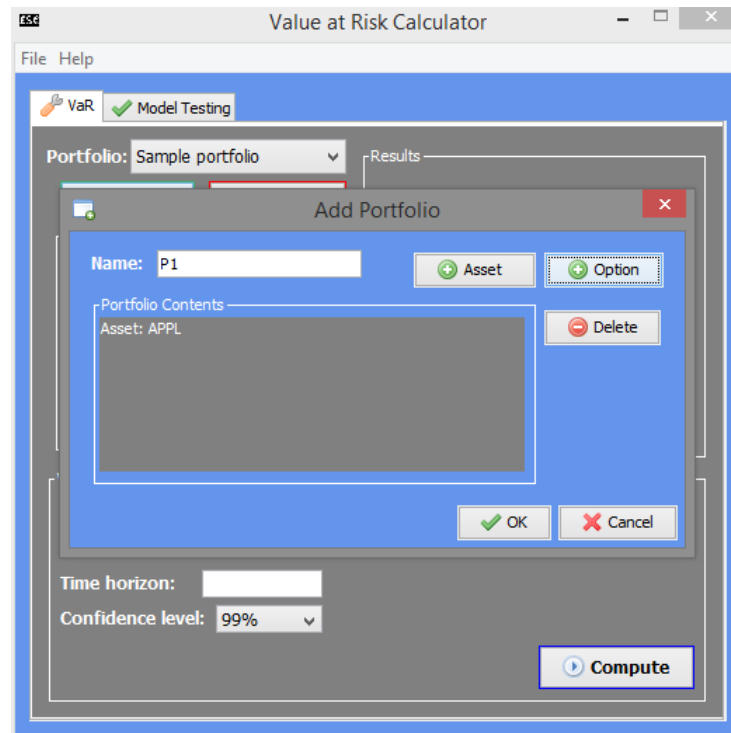


Figure A.8: Added asset shown in portfolio contents

Adding options to the portfolio follows a similar pattern. Clicking 'Option' in the 'Add Portfolio' dialog will bring up a dialog asking for the parameters of the option, see figure A.9. All fields are mandatory here.

The added option will also be shown in the portfolio contents, see figure A.10.

It is also possible to remove an item from the portfolio during its creation. Selecting an item from the 'Portfolio Contents' list and clicking 'Delete' will remove the item from the portfolio after a confirmation, as shown in figure A.11.

Once all the changes have been made to the portfolio, clicking 'OK' in the 'Add Portfolio' dialog will add the created portfolio to the list of portfolios in the 'Setup' tab, see figure A.12.

A.3.5 Computing VaR

After selecting a created portfolio and specifying all the required parameters, VaR can be computed by clicking the 'Compute' button in the 'VaR Setup' panel. The calculated VaR will be displayed in the 'Results' panel, as shown in figure A.13.

A.3.6 Backtesting using portfolio

The program allows the user to conduct backtesting of the VaR models through the 'Model Testing' tab.

By selecting a portfolio and specifying the 'Time horizon', 'Confidence level' and a 'Model' to conduct the backtest and clicking 'Run', the program will run the backtest and show the results in the text area at the bottom of the 'Backtesting' panel. See figure A.14

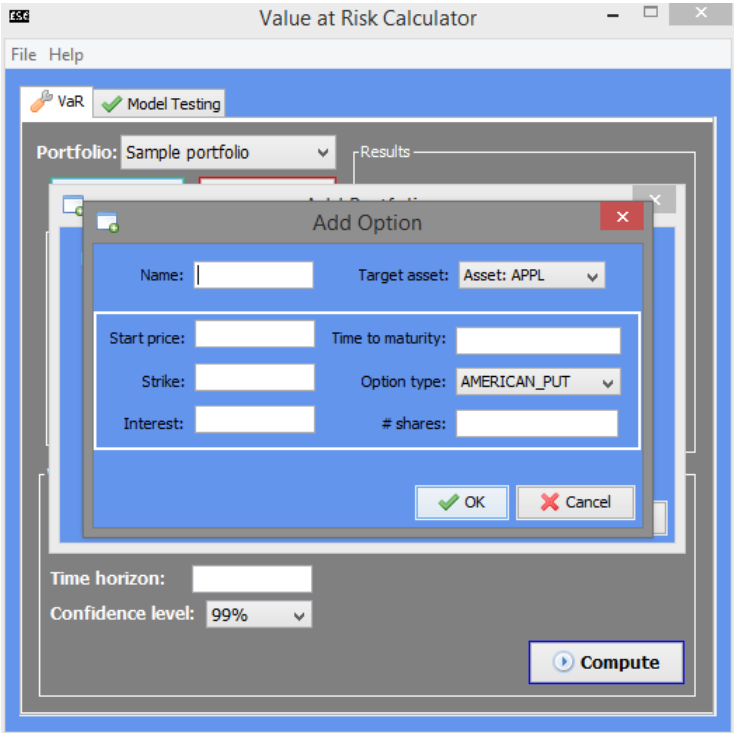


Figure A.9: Adding option to portfolio

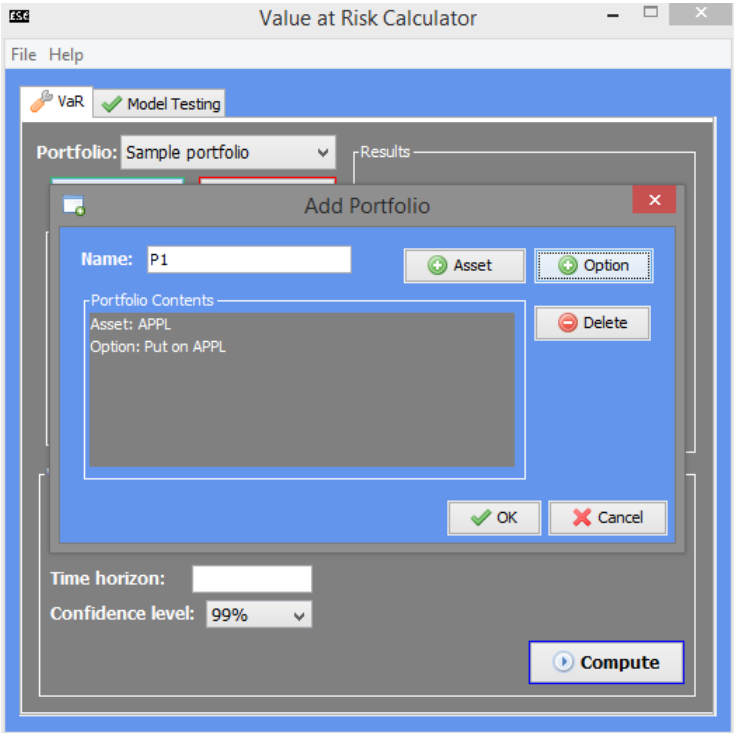


Figure A.10: Added option shown in portfolio contents

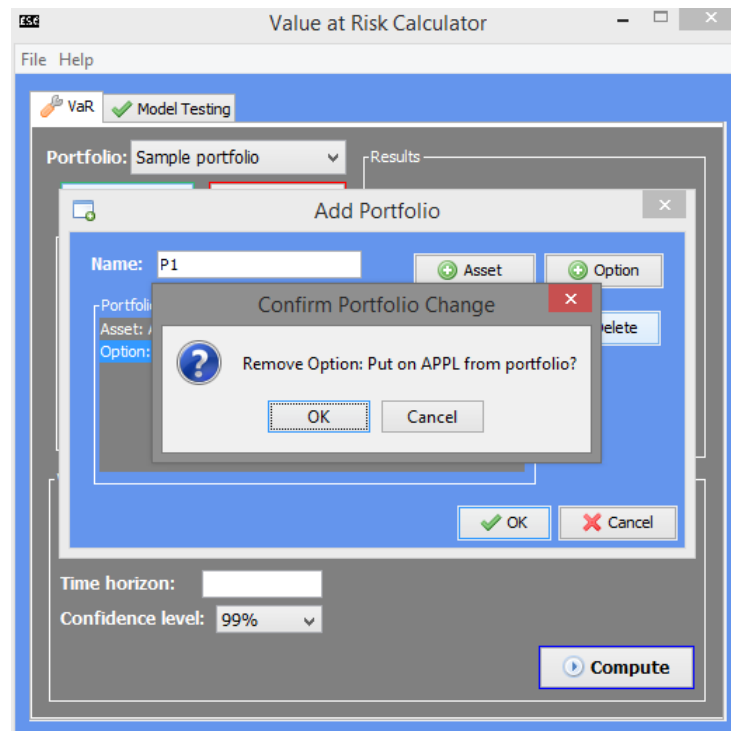


Figure A.11: Removing item from portfolio

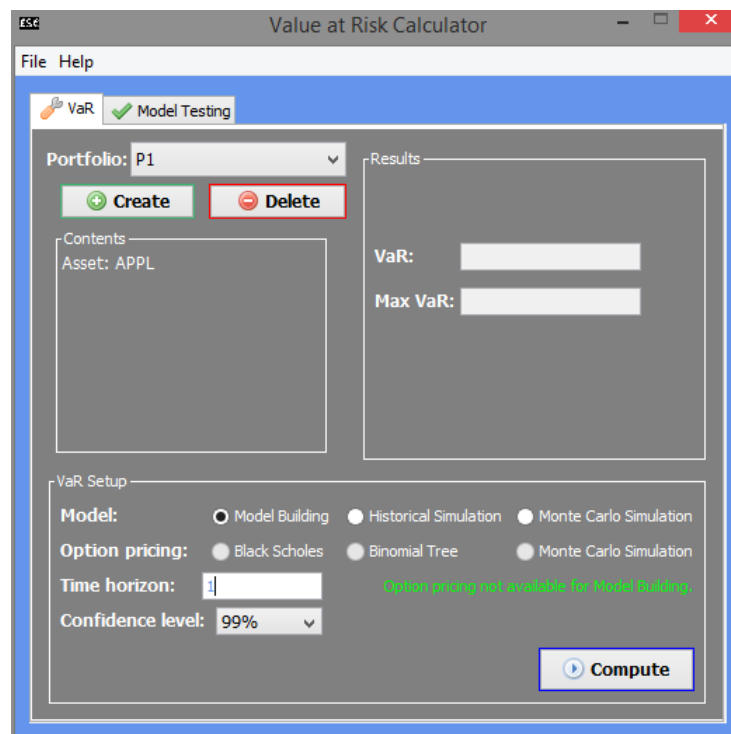


Figure A.12: Added portfolio shown in portfolio drop-down menu

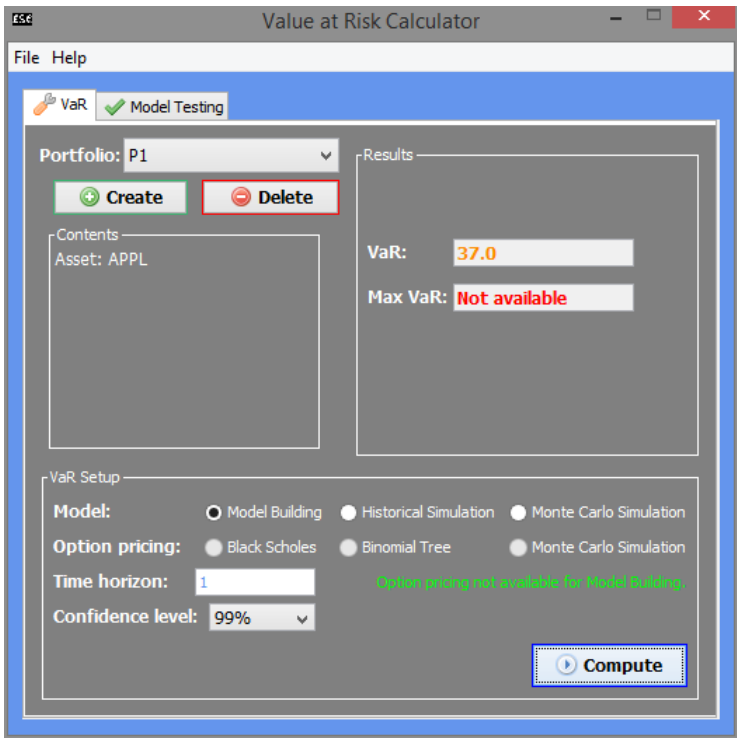


Figure A.13: Computing VaR

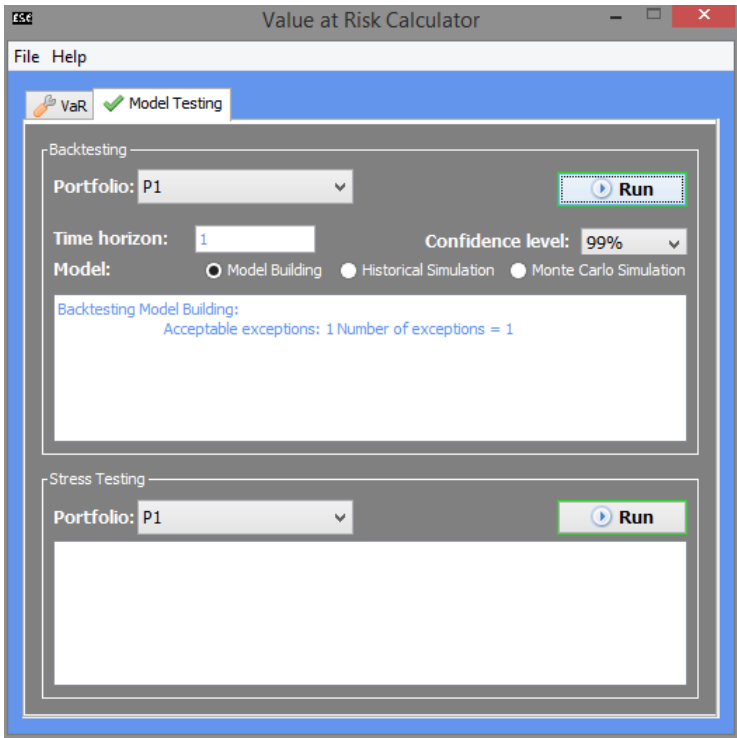


Figure A.14: Backtesting VaR models

A.3.7 Stress testing using portfolio

The program allows the user to carry out stress tests on a portfolio to see what affect a crash in the prices of its assets would have on the value of the portfolio.

Conducting stress testing only requires the user to select a portfolio in the ‘Stress Testing’ panel and clicking ‘Run’. The program will run the stress test on the selected portfolio and show the results in the text area at the bottom of the ‘Stress Testing’ panel. See figure A.15.

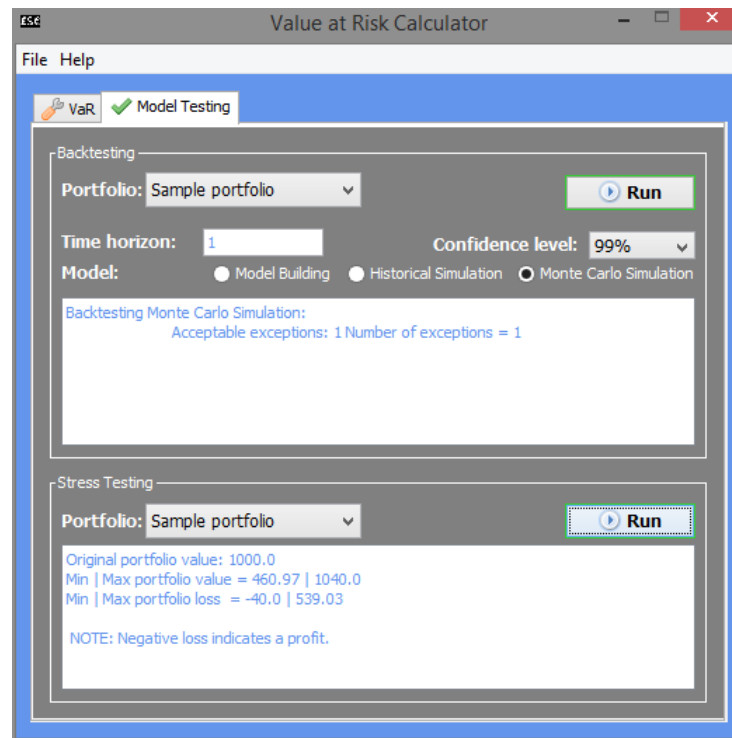


Figure A.15: Stress testing portfolio

A.3.8 Other information

Information about the development of the program is available through the ‘Help’ menu option at the top of the program.

The program can be closed by clicking the ‘File’ menu option and selecting ‘Exit’ or simply by clicking the red cross in the top-right corner of the program window.

Appendix B: Program Listing

B.1 EWMA

```

1  /**
   * Estimates the volatility of the stock using the closing prices provided.
   * EWMA works by placing higher weight on more recent data and maintaining a
   * running average of the variance of the data which can be used to
   * calculate the volatility of the stock.
   *
   * @return estimated volatility of the stock as per the EWMA recursive
   *         calculation.
   */
9  public static double computeVolatility_EWMA( double[] returns )
11 {
    double varianceCurrentDay = getVariance_EWMA( 0, returns );
13    double ewma = ( lambda * varianceCurrentDay )
                   + ( ( 1 - lambda ) * Math.pow( returns[0], 2 ) );
15    double volatility = Math.sqrt( ewma );
    return volatility;
17 }

19 /**
   * Recursively computes the running variance using the EWMA formula.<br>
   * today's variance = yesterdaysVariance^2 * lambda + yesterdaysReturn^2 * (
   * 1 - lambda )
   *
   * @param currentDay
   *         the day we are estimating the variance for.
   * @param returns
   * @return variance for the current day, based on yesterday's variance and
   *         yesterday's return
   */
29 private static double getVariance_EWMA( int currentDay, double[] returns )
31 {
    // day 0 is most recent day
33    double variance = 0.0;
    // first day variance = lambda * firstDayVariance^2 + yesterdayReturn^2
35    // * ( 1 - lambda )
    if( currentDay == returns.length - 1 )
37    {
        double firstDayReturnSquared = Math.pow( firstDayReturn, 2 );
39        double weight = ( 1 - lambda ) * Math.pow( lambda, currentDay );
        variance = ( lambda * firstDayVariance )
41                + ( weight * firstDayReturnSquared );
    }
43    // today's variance = yesterdaysVariance^2 * lambda + yesterdaysReturn^2
    // * ( 1 - lambda )
45    else if( currentDay >= 0 )
    {
47        // recursively work out variance for prior days

```

```

        double yesterdayVariance = getVariance_EWMA( currentDay + 1, returns
    );
49     double yesterdayReturnSquared = Math
                                   .pow( returns[currentDay + 1], 2
    );
51     double weight = ( 1 - lambda ) * Math.pow( lambda, currentDay );
        variance = ( lambda * yesterdayVariance )
53             + ( weight * yesterdayReturnSquared );
    }
55
    return variance;
57 }

```

B.2 VaR for multiple assets using Monte Carlo Simulation

```

1 public void computeForMultipleStocks( ArrayList<Double> stockValues )
    {
3         ArrayList<double[]> returnList = new ArrayList<double[]>();

5         for( File stockFile : stockPriceDataFiles )
            {
7                 double[] returnsFromFile = VarUtils.getReturnsFromFile( stockFile );
                    returnList.add( returnsFromFile );
9             }
            double[][] covarianceMatrix = VarUtils.generateCovarianceMatrix(
11 returnList, numberOfStocks );
            double[][] decomposedMatrix = VarUtils
13                                     .decomposeMatrix( covarianceMatrix
            );

15            double[][] finalDayPrices = new double[numberOfStocks][
numberOfSimulations];
            double[][] maximumLosses = new double[numberOfStocks][numberOfSimulations
            ];
17            int iteration = 0;
            while( iteration < numberOfSimulations )
19            {
                // need to do this 1000 times, and then record the final prices and
21                // lowest prices (highest VaR)
                ArrayList<double[]> simulatedReturns = simulateReturns();
23
                double[] finalDayReturns = simulatedReturns.get( 0 );
25                double[] minReturns = simulatedReturns.get( 1 );

27                RealMatrix L = MatrixUtils.createRealMatrix( decomposedMatrix );

29                double[] correlatedFinalReturns = L.operate( finalDayReturns );

```

```

double[] correlatedMinReturns = L.operate( minReturns );
31
for( int stock = 0 ; stock < numberOfStocks ; stock++ )
33
{
    // price = e^(return) * stockValue
35    finalDayPrices[stock][iteration] = Math
        .exp(
correlatedFinalReturns[stock] )
        * stockValues.get( stock );
37    maximumLosses[stock][iteration] = Math
        .exp( correlatedMinReturns[
39    stock] )
        * stockValues.get( stock );
41    }
    iteration++;
43    }

double[] portfolioFinalSimulatedValues = new double[numberOfSimulations];
double[] portfolioMinSimulatedValues = new double[numberOfSimulations];
45
47
for( int sim = 0 ; sim < numberOfSimulations ; sim++ )
49
{
    double sumOfFinalStockValues = 0.0, sumOfMinStockValues = 0.0;
51    for( int stock = 0 ; stock < numberOfStocks ; stock++ )
    {
53        sumOfFinalStockValues += finalDayPrices[stock][sim];
        sumOfMinStockValues += maximumLosses[stock][sim];
55    }
    portfolioFinalSimulatedValues[sim] = sumOfFinalStockValues;
57    portfolioMinSimulatedValues[sim] = sumOfMinStockValues;
    }

59    Arrays.sort( portfolioFinalSimulatedValues );

61    double valueAtPercentile = VarUtils.getPercentile(
63    portfolioFinalSimulatedValues,
        confidence );

65

double portfolioValue = 0.0;
67

for( double stockValue : stockValues )
69
{
    portfolioValue += stockValue;
71
}

73
double finalVaR = portfolioValue - valueAtPercentile;

75
System.out.println( "Monte Carlo VaR simulated with "
    + numberOfSimulations + " simulations of " +
timePeriod
    + " days each." );
77
System.out.println( "Monte Carlo VaR (Portfolio - Final): "
79    + VarUtils.round( finalVaR ) );

```

```

    this.monteCarloFinalVar = finalVaR;
81
    Arrays.sort( portfolioMinSimulatedValues );
83
    double maximumVaR = portfolioValue - portfolioMinSimulatedValues[0];
    System.out.println( "Monte Carlo VaR (Portfolio - Maximum): "
85
                        + VarUtils.round( maximumVaR ) );
    this.monteCarloMaximumVar = maximumVaR;
87
}

```

B.3 Option pricing using Historical Simulation

```

public double priceOption( Option option )
2 {
    double[] returns = VarUtils.getReturnsFromFile( option.getPriceData() );
    int numberOfReturns = returns.length;
    ArrayList<Double> possibleOptionValues = new ArrayList<Double>();
    int type = VarUtils.convertToCallOrPut( option.getOptionType() );
    double historicalValue = 0.0;
    double stockPrice = 0.0;
    int initialTimeToMaturity = option.getTimeToMaturity();
10 for( int i = 0 ; i < numberOfReturns ; i++ )
    {
12     int currentTimeToMaturity = initialTimeToMaturity - i;
        stockPrice = Math.exp( returns[i] ) * option.getInitialStockPrice();
14     if( currentTimeToMaturity >= 0 )
        {
16         switch( optionPricingType )
            {
18             case VarUtils.BS:
                BlackScholes bs = new BlackScholes();
                historicalValue = bs.compute( type, stockPrice, option.getStrike(),
20
                                            currentTimeToMaturity,
22
                                            option.getInterest(),
                                            option.getDailyVolatility() );
24
                break;
            case VarUtils.BT:
                option.setInitialStockPrice( stockPrice );
                option.setTimeToMaturity( currentTimeToMaturity );
                BinomialTree bt = new BinomialTree( option );
                historicalValue = bt.getOptionPrice();
30
                break;
            case VarUtils.MC:
                option.setInitialStockPrice( stockPrice );
                option.setTimeToMaturity( currentTimeToMaturity );
                MonteCarloSimulation mc = new MonteCarloSimulation();
                historicalValue = mc.priceOptionUsingMonteCarlo( option );
36
                break;
        }
    }
38
}

```

```
        possibleOptionValues.add( historicalValue );
40    }
    else
42    {
        break;
44    }
    }
46    double[] values = new double[possibleOptionValues.size()];
    for( int i = 0 ; i < possibleOptionValues.size() ; i++ )
48    {
        values[i] = possibleOptionValues.get( i );
50    }
    // sort possible values
52    Arrays.sort( values );

54    // select value from percentile
    double valueAtPercentile = VarUtils.getPercentile( values, confidence );
56
    return valueAtPercentile;
58 }
```

Appendix C: Backtesting Results

C.1 Using GARCH(1,1) to estimate volatility of returns

```

Backtesting VaR Estimation over 1 day at 99%
2 Backtesting Model Building:
  Acceptable exceptions: 1 Number of exceptions = 2
4 Backtesting Historical Simulation:
  Acceptable exceptions: 1 Number of exceptions = 0
6 Backtesting Monte Carlo Simulation:
  Acceptable exceptions: 1 Number of exceptions = 2
8
Backtesting VaR Estimation over 1 day at 98%
10 Backtesting Model Building:
  Acceptable exceptions: 2 Number of exceptions = 5
12 Backtesting Historical Simulation:
  Acceptable exceptions: 2 Number of exceptions = 2
14 Backtesting Monte Carlo Simulation:
  Acceptable exceptions: 2 Number of exceptions = 3
16
Backtesting VaR Estimation over 1 day at 97%
18 Backtesting Model Building:
  Acceptable exceptions: 3 Number of exceptions = 7
20 Backtesting Historical Simulation:
  Acceptable exceptions: 3 Number of exceptions = 4
22 Backtesting Monte Carlo Simulation:
  Acceptable exceptions: 3 Number of exceptions = 8
24
Backtesting VaR Estimation over 1 day at 96%
26 Backtesting Model Building:
  Acceptable exceptions: 4 Number of exceptions = 9
28 Backtesting Historical Simulation:
  Acceptable exceptions: 4 Number of exceptions = 7
30 Backtesting Monte Carlo Simulation:
  Acceptable exceptions: 4 Number of exceptions = 9
32
Backtesting VaR Estimation over 1 day at 95%
34 Backtesting Model Building:
  Acceptable exceptions: 5 Number of exceptions = 9
36 Backtesting Historical Simulation:
  Acceptable exceptions: 5 Number of exceptions = 10
38 Backtesting Monte Carlo Simulation:
  Acceptable exceptions: 5 Number of exceptions = 9

```

C.2 Using Standard Formula to estimate volatility of returns

```

1 Backtesting VaR Estimation over 1 day at 99%
  Backtesting Model Building:
3   Acceptable exceptions: 1 Number of exceptions = 2
  Backtesting Historical Simulation:
5   Acceptable exceptions: 1 Number of exceptions = 0
  Backtesting Monte Carlo Simulation:
7   Acceptable exceptions: 1 Number of exceptions = 1

9 Backtesting VaR Estimation over 1 day at 98%
  Backtesting Model Building:
11  Acceptable exceptions: 2 Number of exceptions = 5
  Backtesting Historical Simulation:
13  Acceptable exceptions: 2 Number of exceptions = 2
  Backtesting Monte Carlo Simulation:
15  Acceptable exceptions: 2 Number of exceptions = 4

17 Backtesting VaR Estimation over 1 day at 97%
  Backtesting Model Building:
19  Acceptable exceptions: 3 Number of exceptions = 7
  Backtesting Historical Simulation:
21  Acceptable exceptions: 3 Number of exceptions = 4
  Backtesting Monte Carlo Simulation:
23  Acceptable exceptions: 3 Number of exceptions = 7

25 Backtesting VaR Estimation over 1 day at 96%
  Backtesting Model Building:
27  Acceptable exceptions: 4 Number of exceptions = 9
  Backtesting Historical Simulation:
29  Acceptable exceptions: 4 Number of exceptions = 7
  Backtesting Monte Carlo Simulation:
31  Acceptable exceptions: 4 Number of exceptions = 8

33 Backtesting VaR Estimation over 1 day at 95%
  Backtesting Model Building:
35  Acceptable exceptions: 5 Number of exceptions = 9
  Backtesting Historical Simulation:
37  Acceptable exceptions: 5 Number of exceptions = 10
  Backtesting Monte Carlo Simulation:
39  Acceptable exceptions: 5 Number of exceptions = 9

```

C.3 Using EWMA to estimate volatility of returns

```

1 Backtesting VaR Estimation over 1 day at 99%
  Backtesting Model Building:
3   Acceptable exceptions: 1 Number of exceptions = 20
  Backtesting Historical Simulation:

```

5	Acceptable exceptions: 1 Number of exceptions = 0
	Backtesting Monte Carlo Simulation:
7	Acceptable exceptions: 1 Number of exceptions = 21
9	Backtesting VaR Estimation over 1 day at 98%
	Backtesting Model Building:
11	Acceptable exceptions: 2 Number of exceptions = 25
	Backtesting Historical Simulation:
13	Acceptable exceptions: 2 Number of exceptions = 2
	Backtesting Monte Carlo Simulation:
15	Acceptable exceptions: 2 Number of exceptions = 23
17	Backtesting VaR Estimation over 1 day at 97%
	Backtesting Model Building:
19	Acceptable exceptions: 3 Number of exceptions = 27
	Backtesting Historical Simulation:
21	Acceptable exceptions: 3 Number of exceptions = 4
	Backtesting Monte Carlo Simulation:
23	Acceptable exceptions: 3 Number of exceptions = 27
25	Backtesting VaR Estimation over 1 day at 96%
	Backtesting Model Building:
27	Acceptable exceptions: 4 Number of exceptions = 29
	Backtesting Historical Simulation:
29	Acceptable exceptions: 4 Number of exceptions = 7
	Backtesting Monte Carlo Simulation:
31	Acceptable exceptions: 4 Number of exceptions = 28
33	Backtesting VaR Estimation over 1 day at 95%
	Backtesting Model Building:
35	Acceptable exceptions: 5 Number of exceptions = 29
	Backtesting Historical Simulation:
37	Acceptable exceptions: 5 Number of exceptions = 10
	Backtesting Monte Carlo Simulation:
39	Acceptable exceptions: 5 Number of exceptions = 29

Bibliography

- [1] Hull, John. (2009). Options, Futures and Other Derivatives. New Jersey: Pearson Prentice Hall.
- [2] Beck, Kent; et al. (2001). Manifesto For Agile Software Development. <http://www.agilemanifesto.org>.
Last accessed 22 March 2014.
- [3] Gatheral, Jim (2006). The Volatility Surface: A Practitioner's Guide. New Jersey: Wiley.
- [4] The Jorion-Taleb Debate (1997).
<http://www.derivativesstrategy.com/magazine/archive/1997/0497fea2.asp>
Last accessed 22 March 2014.
- [5] VaR debate between David Einhorn and Aaron Brown.
<http://www.garpdigitallibrary.org/download/GRR/2012.pdf>
Last accessed 22 March 2014.
- [6] Aaron Brown (March 2004). The Unbearable Lightness of Cross-Market Risk. Wilmott Magazine
- [7] Joe Nocera (January 2009).
http://www.nytimes.com/2009/01/04/magazine/04risk-t.html?pagewanted=1&_r=2&
Last accessed 22 March 2014.
- [8] J.P. Morgan (1995). RiskMetrics Monitor, Fourth Quarter.
- [9] <http://en.wikipedia.org/wiki/P-value>
Last accessed 24 March 2014.
- [10] <http://www.r-project.org/>
Last accessed 24 March 2014.