Implementation of Value at Risk Measures in Java

Adrian Ng

Submitted for the Degree of Master of Science in

Data Science and Analytics

Department of Computer Science
Royal Holloway University of London
Egham, Surrey TW20 0EX, UK

October 27, 2019

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 Where other published and unpublished source materials have been used, these have been acknowledged.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: 15554

Student Name: Adrian Ng

Date of Submission: 31 August 2017

Signature:

[illegible]

Abstract

Value at Risk (VaR) is an estimate which describes, in a single figure, the risk associated with our portfolio of investments. This dissertation attempts to demonstrate the implementation of various measures of VaR using Java. We discuss the theory of the Analytical, Monte Carlo and Historical approaches and detail the differences between these approaches. Some may take probabilistic assumptions and/or sample from simulated distributions. In all cases, we take the parameters of our approaches from real-world historical market data.

We also look at the various approaches to estimating daily variance and volatility. These are the *Equal-Weighed* (EW), the *Exponentially Weighted Moving Average* (EWMA), the *Generalised Autoregressive Conditional Heteroskedastic* (GARCH(1,1)) approaches. In the case of GARCH(1,1), we demonstrate parameter estimation via maximum likelihood estimation using the Levenberg-Marquardt algorithm. For EWMA, we take the parameter used by J.P. Morgan's RiskMetrics.

We compare these measures against the real life performance in the stock market via back testing and stress testing. In stress testing, we use real-world data taken from a period of *stressed* market conditions, starting from 1st January 2007. Our experiments will show that some measures, such as EWMA, perform well. Whereas other measures, like EW, do not. We also experiment with different portfolio configurations involving a number of stocks and hedging with put options. We show that we can reduce our exposure to risk by diversifying our portfolio.

We source our data from Google Finance and demonstrate in our Java implementation the approach to acquisition of stock and options data. Additionally, we demonstrate an object-oriented approach to computing our statistics.

Contents

1 Introduction

For a given a portfolio of investments there is an associated risk. However, there are many measures of risk, such as Greek letters) that simply describe different aspects of risk in a portfolio of derivatives. The goal of Value at Risk (VaR) is to provide an estimate of risk that summarises all aspects of risk into one figure.

This one figure simply answers the question: *how bad could it get*? An answer is provided with respect to two parameters: the time horizon and confidence level. That is, we are $x\%$ sure that our portfolio will not lose more than a certain amount over the next N days. That certain amount is our VaR estimate.

This estimate is widely used in industry. Take, for instance, an investment bank. People deposit their money into this bank and in turn, the bank invests this money in the stock market and earns money on the returns. An investment with high returns is highly risky. The bank needs to keep a certain amount of cash in reserve to mitigate this risk. The size of this reserve is proportional to the bank's exposure to risk, i.e. the VaR estimate.

2 Background Research

2.1 Estimating Variance and Volatility

In this section our treatment follows Hull [8] chapter 22, page 498 onwards.

Suppose we have a vector of stock prices S_i. Each element in this vector represents a daily reading of stock prices over some fixed interval τ. This is our historical data. We assume it takes a normal distribution.

In order to calculate daily volatilities from this data, we must iterate through this vector and calculate a vector of percentage changes (=returns). Hull [8] defines this via the following equation:

$$u_i = \frac{S_i - S_{i-1}}{S_{i-1}} \tag{1}$$

This is the percentage change u_i between the end of day $i-1$ and the end of day i. In calculating the percentage change this way, we can assume the mean of u_i given by $\bar{u} = \frac{1}{m}\sum_{i=1}^{m} u_{n-i}$, is approximately zero such that we simply assume it is zero.

2.1.1 Standard Deviation

The estimate of the daily volatility is calculated as the square root of the *variance-rate* σ_n^2 on day n:

$$\sigma_n^2 = \frac{1}{m-1}\sum_{i=1}^{m}(u_{n-i}-\bar{u})^2 \tag{2}$$

Simply, daily volatility is the standard deviation of daily returns. In equation 2, we take $\frac{1}{m-1}$ so that our estimate is unbiased. However, for simplicity we take $\frac{1}{m}$ instead. Now we can simplify our equation and estimate the squared volatility as:

$$\sigma_n^2 = \frac{1}{m}\sum_{i=1}^{m}u_{n-i}^2 \tag{3}$$

In this equation, we give equal weight to each value of u_{n-i}^2. This type of model is fine if volatility is constant; this is an acceptable assumption for small periods of time. But we cannot make this assumption for long periods of time. If we compare last year's volatility estimate to today's, we can expect it would be different.

When estimating volatility, it makes sense to assume instead that more recent price changes are more relevant than those in the past. But on the other hand, we naturally want to include in our calculations as many observations as possible to produce the best estimate. This is a trade-off that needs to be reconciled.

Let us consider, as Hull suggests, the following example - a simple weighted model:

$$\sigma_n^2 = \sum_{i=1}^{m}\alpha_i u_{n-i}^2 \tag{4}$$

Here, we give observations from i days ago a weight of α_i. An observation j days ago is more recent than an observation i days ago, (i.e. $i > j$). We then assign to each observation a weight such that $a_i < a_j$. More recent observations u_j have higher weighting than older observations u_i. Note that the α's must be positive and sum to unity such that $\sum_{i=1}^{m}\alpha_i = 1$.

2.1.2 Exponentially Weighted Moving Average

We see a modification of this weighting in the Exponentially Weighted Moving Average (EWMA) model. In this case, the weights decrease exponentially for older observations. In other words, as i increases, we decrease the weight on each time step by some constant proportion λ. That is, $\alpha_{i+1} = \lambda\alpha_i$ where the inverse *rate-of-decay*, λ, is a constant between 0 and 1. If $\lambda = 0$, most of our influence comes from u_{n-1}^2. If λ is large, this means we won't discriminate as much against observations in the past. As such, λ allows us to control the influence of the most recent data.

As we iterate through our data, we maintain an estimate of σ^2 via:

$$\sigma_n^2 = \lambda\sigma_{n-1}^2 + (1-\lambda)u_{n-1}^2 \tag{5}$$

This is a recursive formula wherein the estimate for the volatility σ_n at the end of day n is calculated from the end of the previous day's estimate of volatility σ_{n-1} (which, in turn, was calculated from σ_{n-2} and so on). We can expand our formula, via substitution into:

$$\sigma_n^2 = (1-\lambda)\sum_{i=1}^{m}\lambda^{i-1}u_{n-i}^2 + \lambda^m\sigma_{n-m}^2 \tag{6}$$

Note that the observation u_{n-i}^2, which is i days old has the weight $\lambda^{i-1}u_{n-i}^2$. Consider the most recent estimate for σ_n^2:

$$\sigma_n^2 = \lambda\sigma_{n-1}^2 + (1-\lambda)u_{n-1}^2 \tag{7}$$

And consider the previous estimate:

$$\sigma_{n-1}^2 = \lambda\sigma_{n-2}^2 + (1-\lambda)u_{n-2}^2 \tag{8}$$

Substituting equation 8 into equation 7 gives us:

$$\sigma_n^2 = \lambda^2\sigma_{n-2}^2 + \lambda(1-\lambda)u_{n-2}^2 + (1-\lambda)u_{n-1}^2 \tag{9}$$

As such, σ_n^2 depends on past data. You can see in equation 9 that the coefficient of u_{n-2}^2 is $(1-\lambda)\lambda$; similar to what we had in equation 6. And because $\lambda < 1$, we can see that as i increases, the weight assigned to observations i days in the past decreases at the exponential rate, hence the name.

$$\lim_{i\to m}(1-\lambda)\lambda^{i-1} \to 0 \tag{10}$$

λ can be found via maximum likelihood estimates. In practice, J.P. Morgan uses EWMA with $\lambda = 0.94$ in their RiskMetrics platform.

2.1.3 GARCH(1,1)

The Generalised Autoregressive Conditional Heteroskedastic (GARCH(1,1)) process is an extension of the model in equation 4 wherein we assume there is a long-term average variance V_L and give it a weight γ.

$$\sigma_n^2 = \gamma V_L + \alpha u_{n-1}^2 + \beta \sigma_{n-1}^2 \tag{11}$$

We can replace γV_L with ω. The most recent squared volatility estimate is governed by the *previous* calculated observation of u^2 and the *previous* estimation for squared volatility. Hence the (1,1) suffix. The prior estimates for squared volatility are calculated recursively via the same model in equation 11. This is similar to the EWMA model, which also gives weight to its observations. In fact, EWMA model is actually a particular case of GARCH(1,1) where $\gamma = 0$, $\alpha = 1 - \lambda$, and $\beta = \lambda$.

Furthermore, GARCH(1,1) is similar to EWMA in that its parameters ω, α and, β are also estimated via maximum likelihood approaches. The likelihood function to be maximised is:

$$\chi^2(a) = \sum_{i=1}^{m} \left[-\ln(v_i) - \frac{u_i^2}{v_i} \right] \tag{12}$$

where $v_i = \sigma_i^2$ and a is the vector of parameters to be found. Hull suggests using an algorithm such as *Levenberg-Marquardt* (LM) for this. Press [12] demonstrates LM as a tool to optimize parameters for least squares, in which the merit function χ^2 to be *minimised* is:

$$\chi^2(a) = \sum_{i=1}^{N} \left[\frac{y_i - y(x_i; a)}{\sigma_i} \right]^2 \tag{13}$$

In least squares, this function is used to measure the quality of fit. According to Jansen [9], an AR(1) process is a standard regression problem; maximum likelihood estimates are interchangeable with least squares. We can therefore use equation 12 as our merit function to be maximised in LM. We take partial derivatives to find the gradient of χ^2.

$$\frac{\partial \chi^2}{\partial \omega} = \sum_{i=1}^{m} \left[-\frac{1}{\omega + \alpha u_i^2 + \beta \sigma_i^2} + \frac{u^2}{(\omega + \alpha u_i^2 + \beta \sigma_i^2)^2} \right] \tag{14}$$

$$\frac{\partial \chi^2}{\partial \alpha} = \sum_{i=1}^{m} \left[-\frac{u_i^2}{\omega + \alpha u_i^2 + \beta \sigma_i^2} + \frac{u^4}{(\omega + \alpha u_i^2 + \beta \sigma_i^2)^2} \right] \tag{15}$$

$$\frac{\partial \chi^2}{\partial \beta} = \sum_{i=1}^{m} \left[- \frac{\sigma_i^2}{\omega + \alpha u_i^2 + \beta \sigma_i^2} + \frac{u^2 \cdot \sigma_i^2}{(\omega + \alpha u_i^2 + \beta \sigma_i^2)^2} \right] \tag{16}$$

We define the vector β_k as:

$$\beta_k \equiv - \frac{\partial \chi^2}{\partial a_k} \tag{17}$$

and matrix α_{kl} as:

$$\begin{aligned} \alpha_{jj} &\equiv \frac{\partial \chi^2}{\partial a_j} \cdot \frac{\partial \chi^2}{\partial a_j} \cdot (1 + \lambda) \\ \alpha_{jk} &\equiv \frac{\partial \chi^2}{\partial a_j} \cdot \frac{\partial \chi^2}{\partial a_k} \quad (j \neq k) \end{aligned} \tag{18}$$

where λ is some non-dimensional *fudge factor* which is used to control the size of the step. Note that $(1 + \lambda)$ is multiplied across the diagonal. These terms can be written as the set of linear equations which we solve for δa_l:

$$\sum_{i=1}^{M} \alpha_{kl} \delta a_l = \beta_k \tag{19}$$

δa_l is an increment of a that, when added to the current approximation, forms the next approximation of the parameters. Altogether, the recipe for parameter optimization via the LM algorithm is as follows:

Algorithm 1 Parameter Estimation via Levenberg-Marquardt

```
1: procedure LEVENBERG-MARQUARDT
2:     Compute $\chi^2(a)$
3:     $\lambda \leftarrow 0.001$
4:     while $\delta a > 0$ do
5:         Solve $\sum_{i=1}^{M} \alpha_{kl} \delta a_l = \beta_k$ for $\delta \alpha_l$
6:         if $\chi^2(a + \delta a) > \chi^2(a)$ then
7:             $\lambda \leftarrow \lambda \times 0.1.$
8:             $a \leftarrow a + \delta a$
9:         else$\lambda \leftarrow \lambda \times 10.$
```

We stop when $\delta a << 1$. Once optimal parameters have been found, the long-term Variance can be calculated (but by then σ_n^2 will have already been estimated).

$$V_L = \frac{\omega}{1 - \alpha - \beta} \tag{20}$$

2.1.4 Estimating Covariance

Now, we have discussed how to estimate the daily volatilities σ_x and σ_y. If we calculate the vector of price changes x_i for stock X and the vector of price changes y_i for stock Y using the same method in equation 1 then we can assume $\bar{x} = \bar{y} = 0$. As such, we can then use equation 3 to estimate the *variance-rate*.

$$\sigma_{x,n}^2 = \frac{1}{m}\sum_{i=1}^{m} x_{n-i}^2, \quad \sigma_{y,n}^2 = \frac{1}{m}\sum_{i=1}^{m} y_{n-i}^2 \tag{21}$$

Our estimate for the covariance on day n between X and Y is calculated as:

$$\text{cov}_n = \frac{1}{m}\sum_{i=1}^{m} x_{n-i}y_{n-i} \tag{22}$$

As you can see this is similar to our estimate of σ^2.

Likewise, if we are attempting the EWMA approach, then our covariance estimate for day n takes a similar form to our estimation of the variance-rate in equation 5:

$$\text{cov}_n = \lambda\text{cov}_{n-1} + (1-\lambda)x_{n-1}y_{n-1} \tag{23}$$

Again, we see a recursive approach for our covariance estimation. If we were to perform another substitution analysis, we would also see that the weights given to the observations decline at the exponential rate the further we look into the past.

For GARCH(1,1), covariance is estimated as:

$$\text{cov}_n = \omega + \alpha x_{n-1}y_{n-1} + \beta\text{cov}_{n-1} \tag{24}$$

where our parameters ω, α, and β can be found by maximising the following:

$$\sum_{i=1}^{m}\left[-\ln(\text{cov}_i) - \frac{x_i y_i}{\text{cov}_i}\right] \tag{25}$$

Once we have our covariance estimates, we are able to produce a *variance-covariance* matrix.

2.2 Analytical Approach

Our treatment in this section follows Hull [8] Chapter 21, page 478 onwards and Willmott [20] Chapter 22, page 461 onwards.

Now that we have discussed how to obtain estimates of daily volatilities and covariances from historical data, we now come to our first VaR measure in which these estimates can be used. We will look at how to produce a VaR estimate for a single stock and portfolio of stocks. In the former, we assume a normal distribution and in the latter we assume a multivariate normal distribution. In both cases, we specify a time horizon δt and confidence-level c.

2.2.1 Joint Positions

We have just discussed how to estimate the VaR a portfolio containing a single stock. Now we consider the VaR estimate of a joint portfolio which is worth: $\Pi = (\Pi_1 + \Pi_2)$. Our VaR estimate of this portfolio is not simply the combination of VaR for Π_1 and VaR for Π_2. This is because losses in Π_1 do not necessarily coincide with losses in Π_2.
Likewise, the change in the value of our portfolio is worth:

$$\Delta\Pi = \Big(\Delta\Pi_1 \sim \phi\big(0, \Pi_1^2\sigma_1^2\Delta t\big) + \Delta\Pi_2 \sim \phi\big(0, \Pi_2^2\sigma_2^2\Delta t\big)\Big) \tag{26}$$

We know the distributions of both parts of the portfolio. What about the sum of the distributions? Firstly, expectation of a sum is equal to the sum of expectations and both parts of $\Delta\Pi$ have a mean of zero due to the simplification we made in equation 36. So:

$$\mathbb{E}\Delta\Pi = \mathbb{E}(\Delta\Pi_1 + \Delta\Pi_2) = \mathbb{E}\Delta\Pi_1 + \mathbb{E}\Delta\Pi_2 = 0 \tag{27}$$

Secondly, any linear combination of Gaussian variables s is Gaussian and both parts of $\Delta\Pi$ are Gaussian. As such, what remains is to find the variance of $\Delta\Pi_1 + \Delta\Pi_2$.

The variance of $\Delta\Pi_1$ is defined as $\text{var}(\Delta\Pi_1) = \mathbb{E}(\Delta\Pi_1 - \mathbb{E}\Delta\Pi_1)^2$. The variance of $(\Delta\Pi_1 + \Delta\Pi_2)$ is:

$$\begin{aligned} &\text{var}(\Delta\Pi_1 + \Delta\Pi_2) = \\ &\qquad \mathbb{E}\big[(\Delta\Pi_1 + \Delta\Pi_2) - \mathbb{E}(\Delta\Pi_1 + \Delta\Pi_2)\big]^2 \\ &= \underbrace{\mathbb{E}(\Delta\Pi_1 - \mathbb{E}\Delta\Pi_1)^2}_{\text{variance}} + \underbrace{\mathbb{E}(\Delta\Pi_2 - \mathbb{E}\Delta\Pi_2)^2}_{\text{variance}} + 2\underbrace{\mathbb{E}(\Delta\Pi_1 - \mathbb{E}\Delta\Pi_1)\mathbb{E}(\Delta\Pi_2 - \mathbb{E}\Delta\Pi_2)}_{\text{covariance}} \end{aligned} \tag{28}$$

As before, means are zero so:

$$\underbrace{\mathbb{E}(\Delta\Pi_1)^2}_{\text{variance}} + \underbrace{\mathbb{E}(\Delta\Pi_2)^2}_{\text{variance}} + 2\underbrace{\mathbb{E}(\Delta\Pi_1)\mathbb{E}(\Delta\Pi_2)}_{\text{covariance}} \tag{29}$$

Since $\text{std}(\Delta\Pi_1) = \sqrt{\text{var}(\Delta\Pi_1)}$ and $\text{std}(\Delta\Pi_2) = \sqrt{\text{var}(\Delta\Pi_2)}$, we can rewrite equation 29 in terms of standard deviations (i.e. volatilities), such that:

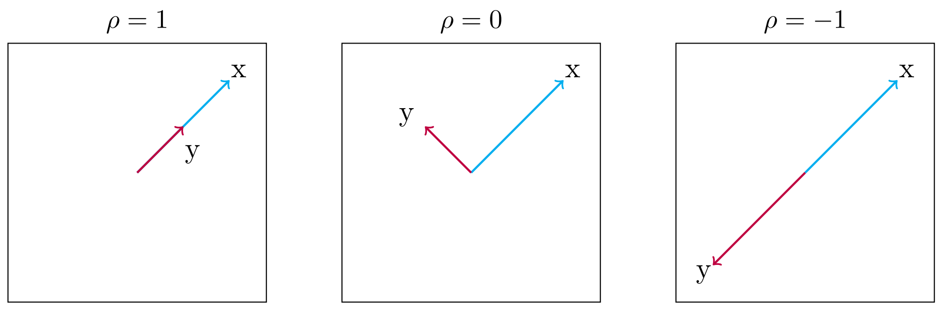
$$\text{std}(\Delta\Pi_1 + \Delta\Pi_2) = \sqrt{\text{std}(\Delta\Pi_1)^2 + \text{std}(\Delta\Pi_2)^2 + 2\rho \cdot \text{std}(\Delta\Pi_1) \cdot \text{std}(\Delta\Pi_2)} \tag{30}$$

where ρ is the correlation coefficient defined as:

$$\rho = \frac{\text{cov}(\Delta\Pi_1, \Delta\Pi_2)}{\text{std}(\Delta\Pi_1) \cdot \text{std}(\Delta\Pi_2)} \tag{31}$$

and $-1 \leq \rho \leq 1$

Figure 1: Correlation Coefficient



Correlation

In figure 1, we illustrate the correlation of two random variable X and Y. As the correlation coefficient moves through $\rho = 1, \rho = 0, \rho = -1$, we see examples of full positive correlation, independence and full negative correlation between X and Y respectively. The blue length is $\text{std}(X)$ and the purple length is $\text{std}(Y)$. We can see that ρ is analogous to the cosine of the angle between the two lengths and the covariance is the scalar product.

When X and Y have positive correlation $\rho > 0$, they have dependency such that when X is large, Y will be large too. On the other hand, when $\rho < 0$, the random variables have dependency between them but if X is large Y will be small.

Calculating VaR

In calculating the standard deviation of $\Delta\Pi_1$ and $\Delta\Pi_2$, we can now find the one day VaR. Suppose we have a confidence level $c = 99\%$, from which follows the percentile $x_{1\%}$, and a time horizon $\Delta t = N$. We calculate our Value at Risk as

$$V = x_{1\%}\text{std}(\Delta\Pi_1 + \Delta\Pi_2)\sqrt{N} \tag{32}$$

Though it may look different but this last step is, in essence, identical to equation 45 where we find the VaR of a single stock portfolio by multiplying the standard deviation of the daily changes in portfolio $\Pi\sigma$ by $x_{1\%}\sqrt{\Delta t}$.

2.2.2 Linear Model

The joint portfolio scenario, in which we have two assets, can be generalised by the following formula, in which we have M number of assets:

$$VaR = -\alpha(1-c)(\Delta t)^{1/2}\sqrt{\sum_{j=1}^{M}\sum_{i=1}^{M}\Pi_i\Pi_j\sigma_i\sigma_j\rho_{ij}} \tag{33}$$

where $a(1-c)$ is the inverse cumulative distribution function of the Gaussian distribution, which gives us the percentile $x_{(1-c)\%}$ and δt is the time horizon. Again, we multiply these parameters by the total standard deviation, which is the square root of a linear combination of the product of volatilities and correlations.

Notice that ρ_{ij} is actually a matrix in which the ith row and jth column is the correlation between the ith and jth assets. At the diagonal, where $i = j$, the correlation equals unity because a variable is always fully correlated with itself. Additionally, since $\rho_{ij} = \rho_{ji}$, the correlation matrix is symmetric.

As mentioned earlier, $\sigma_i\sigma_j\rho_{ij} = \text{cov}_{ij}$. So we could even write equation 33 in terms of a variance-covariance matrix Σ instead:

$$VaR = -\alpha(1-c)(\Delta t)^{1/2}\sqrt{\sum_{j=1}^{M}\sum_{i=1}^{M}\Pi_i\Pi_j\Sigma_{ij}} \tag{34}$$

Each element in this matrix is the product of the correlation of i and j, the daily volatility of i and the daily volatility of j. The elements at the diagonal are simply the daily volatility squared, which is the daily variance.

2.2.3 Single Asset

We can model the behaviour of the stock price over some time interval dt via the Stochastic equation:

$$\frac{dS}{S} = \mu dt + \sigma dz \tag{35}$$

where $dz = \epsilon\sqrt{dt}$. Assuming some small (but finite) interval of time Δt, we can rewrite this as:

$$\frac{dS}{S} = \mu\Delta t + \sigma\epsilon\sqrt{dt} \tag{36}$$

where $\epsilon \sim \phi 0, 1$. We can simplify this equation by ignoring $\mu\Delta t$, which is relatively small over small time intervals. This is because, in practice, the interest rate μ is much smaller than the volatility σ. Suppose $\mu = 0.1$ and $\sigma = 0.35$ and we consider a day over a yearly interval, so $\Delta t = \frac{1}{252}$, then $\mu\Delta t \approx 0.0004$ and $\sigma\sqrt{\Delta t} \approx 0.022$. We see now that the latter dominates the former and at small time intervals $\mu\Delta t$ is small enough to be ignored. So simplifying:

$$\frac{dS}{S} = \sigma\epsilon\sqrt{dt} \tag{37}$$

As such we can take the approximation of the change of our share price to be Normally distributed:

$$\Delta S \sim \phi(0, S^2 \cdot \sigma^2\Delta t) \tag{38}$$

If our portfolio consists of k shares then the value of the portfolio is $\Pi = kS$. Its change in value will therefore be $\Delta\Pi = k\Delta S$. Likewise, we multiply the variance of ΔS by k^2 such that the distribution of $\Delta\Pi$ is:

$$\Delta\Pi \sim \phi(0, k^2S^2 \cdot \sigma^2\Delta t) \tag{39}$$

Value at Risk takes two parameters: time horizon and confidence level. Suppose we take the confidence level $c = 99\%$. This means we are 99% sure that we won't lose more than V, our estimate of Value at Risk. That is, the change in the value of our portfolio is $\Delta\Pi < -V$. Since we want the probability of this event to be $100 - c = 1\%$ or less, we need to solve this equation:

$$\mathbb{P}(\Delta\Pi < -V) = 1\% \tag{40}$$

How, then, do we calculate V? Π is Gaussian with a mean of zero. The density of the change in the value of the portfolio looks like:

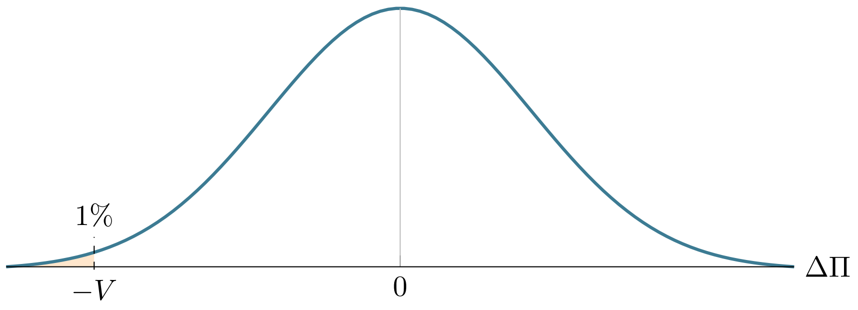
We need to find V such that the shaded area is 1%. Analytically, we look at this problem in terms of standard Gaussian. We can define $\Delta\Pi$:

$$\Delta\Pi = \epsilon\Pi\sigma\sqrt{\Delta t} \tag{41}$$

where ϵ is the standard Gaussian. If we substitute this into equation 40 we get:

$$\mathbb{P}(\epsilon\Pi\sigma\sqrt{\Delta t} < -V) = 1\% \tag{42}$$

Figure 2: Distribution of $\Delta\Pi$



Rearranging, we no longer write the question in terms of $\Delta\Pi$ but in terms of the standard Gaussian:

$$\mathbb{P}\left(\epsilon < -\frac{V}{\Pi\sigma\sqrt{\Delta t}}\right) = 1\% \tag{43}$$

We define the percentile $x_{1\%}$ so that $\mathbb{P}(\epsilon \leq x_{1\%}) = 1\%$. This number can be found by passing our significance level to the Gaussian inverse cumulative distribution function. Once we have found this number we can equate it to:

$$x_{1\%} = -\frac{V}{\Pi\sigma\sqrt{\Delta t}} \tag{44}$$

Thus we derive an exact formula for Value at Risk and find V:

$$V = -x_{1\%}\Pi\sigma\sqrt{\Delta t} \tag{45}$$

Wilmott [20] writes it differently:

$$V = -\sigma\Delta S(\delta t)^{-1/2}\alpha(1-c) \tag{46}$$

where α is the inverse cumulative distribution function for the standard Gaussian and the Greek letter Δ represents the number of stocks at price S. The two equations are equivalent since $\Delta S = \Pi$. But in this way we can explicitly see our parameters for VaR: confidence level c and time horizon δt.

2.3 Simulation

There are two simulation methods used in the estimation of VaR: the *Historical* and *Monte Carlo* method. In both these methods we attempt to generate a sizeable distribution of future price changes by sampling from the distribution of our market variables. With the Historical method, we use historical data to generate our data. If we don't have much historical data, then this approach is of not much use. With the Monte Carlo method, we generate data via a random walk.

2.3.1 Historical

Our treatment of this section follows Hull [8] Chapter 21, page 474 onwards.

The key feature here is that we don't make probabilistic assumptions (no more Gaussian distributions). Instead, we have historical data for our market variables. Our view is that what happened in the past is a guide to what will happen in the future. That is, tomorrow's price change will be sampled from the distribution of price changes in our historical data.

First we must value our portfolio using today's stock prices. Let us iterate through our historical data and, for each stock, calculate a vector of price changes ΔS_i. We do this via the same method as shown in equation 1 and once again assume the mean to be zero. If our data consists of 1001 days of history, then we get 1000 days of price changes.

For each of these price changes ΔS_i, we compute the difference between the current value of our portfolio and the future value our portfolio to build a sample of changes $\Delta\Pi$. We then sort $\Delta\Pi$ in order of largest to smallest, positive to negative. If, for example, our confidence level is $c = 99\%$. Our estimate of VaR occurs at the cut-off point in $\Delta\Pi$ at 99%. 99% of 1000 samples is 990, so we take $\Delta\Pi_{990}\sqrt{\Delta t}$, where Δt is the time horizon.

The algorithm for the Historical method is shown on page 12.

Algorithm 2 Historical method

```
1: procedure HISTORICAL
2:     Value $\Pi^{today}$ from today's $S_i$
3:     for all assets $1 \le i \le n$ do
4:         Calculate vector $\Delta S_i$ from historical data
5:         Apply all $\Delta S_i$ to $S_i$
6:     Revalue for $\Pi^{tomorrow}$
7:     $\Delta\Pi = \Pi^{tomorrow} - \Pi^{today}$
8:     Sort $\Delta\Pi$ in descending order
9:     VaR $\leftarrow \Delta\Pi_{99\%}\sqrt{\Delta t}$
```

2.3.2 Monte Carlo

Our treatment of the Monte Carlo simulation follows Hull [8] Chapter 20, page 446 onwards and Chapter 21 page 488.

The Monte Carlo method shares many similarities with the Historical method. For instance, the way our final VaR estimate is chosen is the same. The main difference is how we generate tomorrow's stock prices.

Let us suppose that our portfolio consists of a number of stocks. For each of these stocks, we once again look at historical data to calculate a vector of price changes ΔS_i. Our main assumption now is that these vectors take the multivariate Gaussian distribution, the parameters of which we will estimate from ΔS_i. Once again, we assume this vector has a mean of zero. As such, we only need find the variance-covariance matrix Σ.

$$\Delta S_i \sim \phi(0, \Sigma) \tag{47}$$

Once we know Σ, the Monte Carlo method will allow us to randomly sample from whatever distribution we have. We start with the stochastic process from equation 36, which we simplify by removing the deterministic part μdt (since $\mu = 0$):

$$S_i^{t+1} = S_i^t + S_i^t L \epsilon_i \sqrt{dt} \tag{48}$$

where L is the Cholesky decomposition of Σ, analogous to the square root of Σ. Our portfolio consists of multiple stocks so to find Σ we take the covariance between each stock and build a matrix.

Cholesky Decomposition In our treatment of the Cholesky Decomposition, we refer to Chapter 2, page 96 of Press [12]. In the univariate case, L would be equivalent to taking the square root of σ^2. But there is no direct way of taking the square root of a matrix. Our approach here is to take the Cholesky decomposition to approximate $\sqrt{\Sigma}$. This gives us a lower triangular matrix L, in which all elements above the diagonal are zero. The product of L with its transpose is Σ.

$$\Sigma = LL' \tag{49}$$

Consider the following matrix A, which is symmetric and positive definite as an example:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \tag{50}$$

We need to find L such that $A = LL^T$. Writing this out looks like:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \begin{bmatrix} l_{11} & l_{21} & l_{31} \\ 0 & l_{22} & l_{32} \\ 0 & 0 & l_{33} \end{bmatrix}$$
$$= \begin{bmatrix} l_{11}^2 & l_{21}l_{11} & l_{31}l_{11} \\ l_{21}l_{11} & l_{21}^2 + l_{22}^2 & l_{31}l_{21} + l_{32}l_{22} \\ l_{31}l_{11} & l_{31}l_{21} + l_{32}l_{22} & l_{31}^2 + l_{32}^2 + l_{33}^2 \end{bmatrix} \tag{51}$$

Then we obtain the following formulas for L: above the diagonal:

$$L_{ii} = \left(a_{ii} - \sum_{k=1}^{i-1} L_{ik}^2 \right)^{1/2} \tag{52}$$

and below the diagonal:

$$L_{ji} = \frac{1}{L_{ii}} \left(a_{ij} - \sum_{k=1}^{i-1} L_{ik} L_{jk} \right) \tag{53}$$

where $j = i+1, i+2, ..., N$

Correlating IID random variables In equation 48, ϵ is vector of independent and identically distributed (IID) random variables sampled from the standard Gaussian. The product of L and ϵ_i gives us correlated random variables. This process iterates through N number of steps of finite but small size $\sqrt{dt}$. At each step, our stock price increments by $S_i^t L \epsilon_i \sqrt{dt}$, which takes the distribution of our historical changes. For predicting tomorrow's stock price, it is conventional to choose $\sqrt{dt} = 1/N$ where $N = 24$, with each step represents an hour.

The end result is a random walk, at the end of which is a prediction for tomorrow's stock price. Repeating this process many times allows us to build a large, probabilistic distribution of tomorrow's stock prices. We subtract from today's stock price to compute ΔS_i for all prices in our new distribution. Just as in the Historical method, we revalue our portfolio at all ΔS_i to get $\Delta\Pi$. We then sort this vector in descending order and with a time horizon Δt and confidence level $c = 99\%$ cut-off, we take our value for VaR as $\Delta\Pi_{99\%}\sqrt{\Delta t}$. The algorithm for the Monte Carlo method is shown on page 15.

Algorithm 3 Monte Carlo method

```
1: procedure MONTE CARLO
2:     Value today's $\Pi$ from $S_i^0$
3:     Calculate vector $\Delta S_i$ from historical data
4:     Compute $\text{cov}_{ij}$ between $\Delta S_i$ and $\Delta S_j$
5:     Populate $\Sigma$
6:     Compute $L$ via Cholesky Decomposition
7:     for 10000 random walks do
8:         $t \leftarrow 0$
9:         while t < N do
10:            Sample $\epsilon_i$ from the standard Gaussian
11:            $S_i^{t+1} = S_i^t + S_i^t L \epsilon_i \sqrt{dt}$
12:            t++
           return $S_i^N$
13:        Value tomorrow's $\Pi$ from $S_i^N$
14:    Compute all increments $\Delta\Pi$
15:    Sort samples in descending order
16:    VaR $\leftarrow \Delta\Pi_{99\%}\sqrt{\Delta t}$
```

2.3.3 Incorporating Options

With Historical and Monte Carlo simulation, we are able to simulate a distribution of tomorrow's stock prices. We are able further simulate the incorporation of options in our portfolio if we modify our simulation. By using an option pricing method such as Black-Scholes, we use each of tomorrow's stock prices to return an estimate for tomorrow's option price, which we incorporate into our revaluation of the portfolio.

The Black Scholes formulas for pricing call and put options are as follows [8]:

$$c(S,t) = SN(d_1) - Xe^{r(T-t)}N(d_2) \tag{54}$$

$$p(S,t) = Xe^{-r(T)}N(-d_2) - SN(-d_1) \tag{55}$$

where:

S	= tomorrow's stock price
X	= strike price
r	= interest rate
T	= days to maturity
N(...)	= cumulative normal distribution function

d_1 and d_2 are defined as:

$$d_1 = \frac{lnS/X + (r + \sigma^2/2)(T)}{\sigma\sqrt{T}} \qquad d_2 = d_1 - \sigma\sqrt{T} \tag{56}$$

By further diversifying our portfolio with options, we will be able to mitigate risk. The algorithm for Historical or Monte Carlo simulation with a portfolio incorporating options is as follows:

Algorithm 4 Simulation with Options

```
1: procedure SIMULATION WITH OPTIONS
2:     Value today's $\Pi$ from $S_i^0$ and today's option price
3:
4:     Simulate distribution of tomorrow's stock prices using Historical or Monte Carlo method
5:     Estimate tomorrow's option price using Black Scholes
6:     Value tomorrow's $\Pi$ from $S_i^N$ and tomorrow's option price
7:     Compute all increments $\Delta\Pi$
8:     Sort samples in descending order
9:     VaR $\leftarrow \Delta\Pi_{99\%}\sqrt{\Delta t}$
```

2.4 Backtesting

Having discussed a number of methods of estimating VaR, we must ask ourselves whether our measure is any good. In backtesting, we test how well our VaR measures would have performed in the past. For instance, we go back 1000 days and for each of these days we compute a VaR estimate using data from one year prior to that day. Now, Value at Risk takes two parameters: time horizon Δt and confidence level c. Suppose, for instance, $\Delta t = 1$ $c = 99\%$, then we would compare each of our VaR estimates against the actually 1-day loss that occured on that day. If the real-life loss exceeds the VaR estimate, then this is a violation.

With a confidence level of 99% and with $\alpha = 1000$ moments, we would expect 50 violations to occur. If the number of violations far exceeds 50, then we underestimated VaR. Likewise, if the number of violations is far less than this, then we have overestimated VaR. But what if we have 51 or 49 violations? We need to know what number of violations falls within a certain interval around our confidence level. Moreover, we want to be able to *tighten* this interval by being able to adjust our significance level, changing the accuracy of our estimate.

We use two methods provided by Holton [6] known as coverage tests. Our implementation of both the Standard and Kupiec coverage tests follow the information on this page: https://www.value-at-risk.net/backtesting-coverage-tests/. These are hypothesis tests with the null hypothesis H_0 that states that the expected number of violations c is equal to the observed number of violations.

2.4.1 Standard Coverage Test

Let us formally define what constitutes a violation via the excedence process I:

$$I^t = \begin{cases} 0 \text{ if loss } \Pi^{t-1} - \Pi^t \leq VaR \\ 1 \text{ if loss } \Pi^{t-1} - \Pi^t > VaR \end{cases} \tag{57}$$

The number of violations observed in the data is then:

$$v = \sum_{t=0}^{\alpha} I^t \tag{58}$$

we take v as the realization of the random variable V which takes the binomial distribution:

$$V \sim B(\alpha + 1, 1 - c) \tag{59}$$

where α is the number of moments of VaR we take in our backtesting.

Suppose we have some significance level ϵ, which controls the precision of our estimation. We are able to test H_0 at any ϵ. To do this, we must determine the interval $[v_1, v_2]$ such that:

$$\mathbb{P}(V \notin [v_1, v_2]) \leq \epsilon \tag{60}$$

is maximized. We also want an interval that is *generally* symmetric, i.e.:

$$\mathbb{P}(V < v_1) \approx \mathbb{P}(v_2 < V) \approx \epsilon/2 \tag{61}$$

Subject to these constraints, we optimize the parameter n that maximises equation 60 having defined our interval as either:

$$[a + n, b] \text{ or } [a, b - n] \tag{62}$$

where:

n = non-negative integer
a = maximum integer such that $\mathbb{P}(V < a) \leq \epsilon/2$
b = minimum integer such that $\mathbb{P}(b < V) \leq \epsilon/2$

Initially $n \leftarrow 0$, so a and b are starting parameters for our interval. As we optimize, we increment n until our conditions are met.

The result of this is a non-rejection interval in which v is permissible. If, however, v is outside this interval, we must reject our VaR measure at our significance level ϵ.

The confidence level is a measure of how confident we are in our portfoilio. The significance level is a measure of how precise we want our estimate to be. It dictates the width of our interval. Experimentally, we can begin with a significance level that governs a wide non-rejection level and decrease until we start to see rejections.

2.4.2 Kupiec's PF Coverage Test

Kupiec's PF (Point of Failure) coverage test offers no advantage over the Standard coverage test. But nonetheless, gives us an alternative for comparison.

Once again we find a non-rejection interval, the width of which is governed by significance level ϵ. Now we find the interval $[v_1, v_2]$ such that:

$$\mathbb{P}(V < v_1) \leq \epsilon/2 \text{ and } \mathbb{P}(v_2 < V) \leq \epsilon/2 \tag{63}$$

We can test H_0 at any ϵ.

First we calculate the ϵ quantile of the χ^2 distribution. Then we compute the following log-likelihood ratio:

$$\log L(n) = 2\log\left(\left(\frac{\alpha+1-n}{c(\alpha+1)}\right)^{\alpha+1-n}\left(\frac{n}{(1-c)(\alpha+1)}\right)^{n}\right) \tag{64}$$

where:

n = non-negative integer
α = number of VaR moments
c = confidence level

In theory, we set $\log L(n)$ equal to our quantile and solve for n, which has two solutions. In practice, we need n to be pair of integers that define our non-rejection interval.

Instead, we find n programmatically. We initialize $n \leftarrow 0$ and increment until $\log L(n) >=$ percentile. We compare $m \leftarrow n$ and $m \leftarrow n-1$ and take whichever value of m that minimizes the distance from $\log L(n)$ to our percentile. We take this value of m as our lower interval. We keep incrementing n until $\log L(n) <=$ percentile and do the same as before to find our upper interval.

2.4.3 Stress Testing

In stress testing, we look at extreme market movements as seen in the past. For instance, we might want to look at the so-called *Black Monday*, October 19, 1987, when the S&P 500 moved by 22.3 standard deviations. To test this, we set our market variables to equal those of that day and see how our VaR measures hold up.

We can also look at an entire period of stressed market conditions, such as during the 2007-2008 financial crisis. Our VaR estimate during this period is called *stressed* VaR.

3 Implementation

3.1 Data acquisition

Our first task was to get some historical financial data, preferably in CSV format. An immediate goal was to be able to use an API to access data for a given stock symbol and time-frame from within a Java program. We initially looked at *Yahoo Finance* and *Google Finance*, which were understood to each have their own API. We found a Java library [7] that pulls data using the *Yahoo* API. After some testing, it was deemed unsatisfactory for our uses. We decided that it would be necessary to write our own program instead.

After discovering that the *Yahoo Finance* API had been recently shut down [18], we found success in accessing historical stock data in csv format and option chains in JSON format via the *Google Finance* API.

3.1.1 Historical Stock Data

Historical stock data comes in csv format. To download from the Google Finance API, we must append some parameters to the URL http://www.google.com/finance/historical.

Description	Parameter	Example Argument
Stock Symbol	q	=GOOG
Date from which to start collecting data	startdate	=Aug+23%2C+2016
Download csv format	output	=csv

To do so, we concatenate a parameter and its argument and separate each pair with an ampersand. The resulting string is known as a *query string* Then we prefix this string with a question mark, which dermarcates the query string section in a URL [13].

The full URL would look something like:

http://*www.google.com/finance/historical?q=GOOG&startdate=Aug+23%2C+2016&output=csv*

The csv file takes the following schema:

Date	Open	High	Low	Close	Volume

Since we are only concerned with the price change between the stock price at the end of the day and the end of the next day (see equation 1), we need only the Close field.

subsubsectiongetStocks.java

We have a class called getStocks.java. It takes two inputs: String [] symbols and **int** intYears. The former input is an array that contains our desired stock symbols, of which there could be any number. The latter input dictates the number of years in the past from which we access our data.

We first need to convert **int** intYears into something that the API can understand. The aptly-named method CalculateYearDiffReturnDAteAsString(**int** intYears) subtracts the desired number of years from today's date and outputs it in the API's required format MMM+dd%2Cyyyy , where %2C is a comma in Unicode.

Because our VaR estimates could involve portfolios that contain multiple stocks, we need to be able to download multiple csv files. To do so, we must loop through each element in String [] symbols. In each loop we construct our URL, then download the csv from the URL and lastly grab all the stock prices in the Close field.

getCSVfromURL() We use this method to download the csv data. Its only input is our URL string. This method follows steps outlined by Oracle [14]a submission by user BalusC on Stack Overflow [17]. Here, we use the constructor URL(urlstr) and its method openStream() to initialise an InputStream.

```
InputStream is = new URL(urlstr).openStream();
```

We parse the InputStream using BufferedReader csv, which gets returned to the main class.

```
BufferedReader csv = new BufferedReader(new InputStreamReader(is
    , "UTF-8"));
```

getStocksFromCSV() This method iterates through each line of BufferedReader csv. Each line is simply a comma separated string. So on each iteration, we use the split() method to populate an array where each element contains a value from the corresponding attribute in the csv.

```
            String[] strTuple = inputLine.split(",");
```

Then we take the value in the 4th index, which is the Close attribute, and add it to ArrayList<Double> alData.

```
            alData.add(Double.parseDouble(strTuple[4]));
```

We use an ArrayList because we don't know how much data we will have. If we request one year's worth of data, then, depending today's day of the week, we could return either 252 or 253 rows of data. But we won't know how many rows we have until we have iterated through the entire set. As such, we take advantage of the flexibility of ArrayList which, unlike an array, does not need to know its dimensions at point of construction.

We return the ArrayList to the main method where we have declared HashMap<String, ArrayList<Double>> mapStocks. Using the stock symbol as the key, the ArrayList is added to the HashMap, where it sits while we move onto the next stock symbol to download. A HashMap might seem like a complication but it does in fact simplify things.

Firstly, we have multiple ArrayLists - one for each stock Symbol. If we were dealing with having multiple primitive arrays, we would simply store them all in a single two-dimensional array. Unfortunately there is nothing analogous to a two-dimensional ArrayList. Secondly, the HashMap allows us to store ArrayLists of varying size. That is because even though we may use the same date in the past to download our data, the number of stock prices in this time interval for AAPL may still differ from TSLA.

Once everything is downloaded, we declare a two-dimensional array:

```
double[][] stockPrices = new double[numSym][numTuples];
```

where numSym is simply the number of symbols we are dealing with and numTuples is size of the smallest ArrayList (we need our historical data to be of equal lengths). Then, using a nested for-loop, we populate the array with the contents of the HashMap:

```
ArrayList<Double> arrayListStockPrices = mapStocks.get(symbols[i
    ]);
                stockPrices[i][j] = arrayListStockPrices.get(j);
```

Ultimately, this two-dimensional array stores all our stock prices and gets passed to all the other classes for VaR estimation. We use a two-dimensional array because accessing each element is very easy. This makes it perform calculations that require looking at interactions between stocks - for instance when computing covariance.

So the general procedure is as follows:

Algorithm 5 Class: getStocks.java

```
1: function GETSTOCKS.JAVA(String [] symbols,int intYears)
2:     Convert int intYears into MMM+dd%2Cyyyy
3:     for all String [] symbols do
4:         Concatenate URL string
5:         Download csv
6:         Populate ArrayList from csv
7:         Add Arraylist to HashMap
8:     double [][] stockPrices ← HashMap
9:     return double [][] stockPrices
```

Originally, we wanted to be able to use days instead of years because as Hull [8] points out on page 474, it leads to 500 price changes. Unfortunately it became apparent that trying to query a given number of days of data using the API is problematic. Financial data is not published everyday which means that if we were to instruct Java to subtract 501 days from today's date, we would not return a date range than covers 501 days worth of data. Compensating for weekends is not a complete solution; bank holidays and public holidays are the issue. These vary from year to year and country to country. Tracking their occurrences was simply too complicated and too much of a low priority so this idea was abandoned.

3.1.2 Options Data

Google Finance provides the option chains in a complicated JSON format. These are downloaded from a slightly different URL: http://www.google.com/finance/option_chain. And this time, we only need two parameters in our query string.

Description	Parameter	Example Argument
Stock Symbol	q	=GOOG
Download JSON format	output	=json

With the same treatment as before, the final URL will look something like:

http://www.google.com/finance/option_chain?q=GOOG&output=json

3.1.3 getOptions.java

The getOptions.java class takes just one input: String[] symbols. Here, we mainly use two methods: getJSONfromURL() and getOptionsfromJSON().

In our main method we instantiate an array: optionsData[] options. This data type will store our options data. The array is as long as the number of stock symbols in symbols.

We then iterate through each element of symbols in a for-loop. On each iteration, we concatenate the URL for our JSON data. Then we invoke getJSONfromURL() using our URL as an input and afterwards invoke getOptionsfromJSON() using our JSON data as an input.

getJSONfromURL() Here, we use Google's Gson library [5] and follow instructions submitted by user2654569 [19] on Stack Overflow. This method downloads the JSON file as an InputStream. Using JsonParser, we parse this into a JsonElement, which we convert and return as a JsonObject.

getOptionsfromJSON() The JsonObject retains the hierarchical structure of a JSON file and allows us direct access to objects, arrays and their elements via a key-value-pair mapping. However, it is difficult to navigate this data unless you know the keys in advance. This is our reasoning for storing these data in optionsData[]: ease of access.

The first object we are interested in is expiry:

```
"expiry": {
  "y": 2018,
  "m": 1,
  "d": 19
},
```

To access the values it contains we must first construct a new JsonObject using expiry as the key.

```
JsonObject expiry = json.get("expiry").getAsJsonObject();
```

Now we must use further keys on this object to access the values for year, month and day-of-month. At this point there are no more underlying objects, so we can store these values in Java strings.

```
String expiryYear = expiry.get("y").toString();
String expiryMonth = expiry.get("m").toString();
String expiryDayofMonth = expiry.get("d").toString();
```

These strings give us a date with which we are able to compute the number of days until the option expires. We use the method getNumDaystoExpiry() for this.

The rest of the data are contained in JSON arrays. These arrays contains a number of elements, each of which have differing strike prices and are as such priced accordingly, depending on how far in the money the strike price gets them. The structure of a put or call element in the JSON array looks like:

```
    {
      "p": "0.02",
      "strike": "20.00"
    }
```

where p represents the price of the option.

As before, to get at the values, we must use the necessary keys. But doing so gives us the option price and strike price. Since there are so many puts in the array, simply iterate through each element and extract everything. In Historic.java and MonteCarlo.java, however, we only use the parameters of the first put.

We now initialize optionsData:

```
optionsData options  = new optionsData();
options.setCallPrices(callPrices);
options.setPutPrices(putPrices);
options.setStrikePrices(strikePrices);
options.setDaystoMaturity(NumDaystoExpiry);
```

then return the result to main().

getNumDaystoExpiry We now parse our the expiry values from some strings into a Java Date object. We follow instructions submitted by BalusC [16] on Stack Overflow. First we specify the format of our string: yyyy MM d. Then we parse it.

```
expiryDate = format.parse(expiryYear + " " + expiryMonth + " " +
    expiryDayofMonth);
```

Afterwards, we follow instructions submitted by jens108 [1] on Stack Overflow that detail how to find the number of days between two dates. It is simply a two step process:

```
long diff = expiryDate.getTime() - currentDate.getTime();
        int NumDaystoExpiry = (int) TimeUnit.DAYS.convert(diff,
    TimeUnit.MILLISECONDS);
```

We then return the result to `getOptionsfromJSON`.

To summarise, the general procedure of `getOptions.java` is as follows:

Algorithm 6 Class: getOptions.java

```
1: function GETOPTIONS.JAVA(String [] symbols)
2:     Declare optionsData [] options
3:     for all String [] symbols do
4:         Declare optionsData options
5:         Concatenate URL string
6:         Download JSON string
7:         Parse into JsonObject
8:         Extract data
9:         Set optionsData options
10:        return optionsData options
11:    return optionsData [] options
```

Black Scholes Implementation `optionsData.java` gives us an object that facilitates the access of our options data (of which we have lots). The second benefit is that we can define instance methods that use these data to perform certain calculations.

Specifically, we have implemented a private method called `getBlackScholesOptionPrices()`. This class makes use of the `distribution` package from the `org.apache.commons.math3` library [2]. Its purpose is to implement the Black Scholes formulas described in equations 54, 55, and 56. To invoke this method, we use the public methods `getBlackScholesPut()` or `getBlackScholesCall()`.

3.2 Stats.java

The `Stats.java` class contains a number of methods that perform the necessary statistical calculations for all our VaR measures. This class is not intended to be used as an instance variable like `Parameteres.java`, `Results.java` or `optionsData.java`. Whereas these classes have *Setters* for writing data, the methods in

Stats.java are essentially just functions - give it some data and it will return a result.

It does, however, contain some instance variables:

```
public class Stats {
    //instance variable
    private double[] singleVector;
    private double[] xVector;
    private double[] yVector;
    private double[][] multiVector;
```

Accompanying these are a number of constructors in which these instance variables get initialized. The constructor that we use dictates the type of method we can call. Now, the methods in Stats.java are not consistent on what data types they require. Some require just a **double**[]. Others require a **double**[][] or a pair of **double**[]s. Either way, these methods are reliant upon the instance variables for their input variables. As such, we need a variety of constructors to ensure that the right instance variables for our method are used.

3.2.1 Constructors

double[] singleVector For instance, we have a method called getMean(). It takes a single variable - a **double**[] - and returns a **double**. That is, suppose we have an array representing a single vector of price changes: **double**[] priceChanges. To calculate the mean, we invoke the desired constructor and call the method as follows:

```
double mean = new Stats(priceChanges).getMean();
```

Because we only used a single **double**[], this means the following constructor was used:

```
    private int numCol;
    //constructor
    public Stats(double[] singleVector){
        this.singleVector = singleVector;
```

double[] xVector, double[] yVector On the other hand, we might want to compute the covariance between a pair of variables, both **double**[]. For this, we use a method called getVariance() which, this time takes, two variables of the type **double**[] and returns a **double**. We compute the variance in the following fashion:

```
double covariance = new Stats(priceChanges1, priceChanges2).getVariance();
```

This time the constructor used looks like

```
        this.numRow = singleVector.length;
    }
```

double [][] multiVector Let us also consider, for instance, the case when we need to compute the covariance matrix of some multivariate distribution. In the case of our multivariate price changes, we would represent such a distribution with a **double** [][]. We use a method called getCovarianceMatrix() which is called in the following way:

```
double [][] covarianceMatrix = new Stats(priceChanges).getCovarianceMatrix();
```

where priceChanges is a **double** [][]. The constructor we use for this looks like:

```
        this.numRow = xVector.length;
    }
    public Stats(double [][] multiVector){
        this.multiVector = multiVector;
        this.numCol = multiVector.length;
```

3.2.2 List of Methods

The following is a list of public methods in Stats.java.

Name	Constructor	Returns
getMean()	**double**[]	**double**
getVariance(**int** measure)	**double**[], **double**[]	**double**
getEWVariance()	**double**[], **double**[]	**double**
getEWMAVariance()	**double**[], **double**[]	**double**
getGARCH11Variance()	**double**[], **double**[]	**double**
getVolatility (**int** measure)	**double**[]	**double**
getEWVolatility ()	**double**[]	**double**
getEWMAVolatility()	**double**[], **double**[]	**double**
getGARCH11Volatility()	**double**[], **double**[]	**double**
getCorrelationMatrix (**int** measure)	**double**[]	**double** [][]
getCovarianceMatrix()	**double** [][]	**double** [][]
getCholeskyDecomposition()	**double** [][]	**double** [][]
getPercentageChanges()	**double** [][]	**double** [][]
getAbsoluteChanges()	**double** [][]	**double** [][]
printMatrixToCSV()*	**double** [][]	**void**
printVectorToCSV()*	**double**[]	**void**

(*) While these methods do not perform statistical calculations, they do provide us with an easy way of printing data to csv. This is desirable if we want to analyse the distribution of our stock price changes produced from Monte Carlo simulation or look at the 1000 VaR estimates produced in Backtesting. To implement these methods, we used information submitted by Tataje [15] and Pasini [10] on Stack Overflow on how to output data to csv using BufferedWriter and StringBuilder.

3.2.3 getMean()

One of our main statistical assumptions is that, for a given vector of price changes u_i, its mean $\bar{u} = 0$. Despite this, it is still useful to include a method that will return the mean of our array of price changes as a sanity check. Indeed, this is how getMean() is used in the class PortfolioInfo.java, where we print a break-down of our portfolio.

```
    //INSTANCE METHODS
    public double getMean(){
        double sum = 0.0;
        for(int i = 0; i < numRow; i++)
```

In our implementation of this method, we simply iterate through each element in double[] singleVector and sum their values. Then we return the average of this sum.

3.2.4 getEWVariance()

This method returns an equal-weighted estimation of daily variance. It is our Java implementation of equation 22, which describes the calculation of the equal-weighted covariance between two market variables. We implement this equation (and not the one that describes variance) out of convenience. We are able to use it for estimating *both* variance and covariance. This is because this method takes two double[] variables as inputs. In computing the variance, both inputs must represent the same market variable. That is, the covariance of a market variable is simply the variance. When computing covariance the two inputs should represent different market variables.

```
    }

    public double getEWVariance(){
        double sum = 0;
```

We simply iterate through each element in double[] xVector and double[] yVector simultaneously and sum their product of their values. Then we return the average of this sum.

3.2.5 getEWMAVariance()

This method follows equation 23 which is the EWMA estimate of daily-covariance. The reasoning here for implementing covariance and not simply variance is the same as in `getVariance()`.

```
double lambda = 0.94;
double EWMA = xVector[numRow -1] * yVector[numRow -1];
```

Here, we use J.P. Morgan's RiskMetrics estimation of lambda $\lambda = 0.94$. The computation of EWMA is recursive and iterative so we must define some initial estimate for variance to begin with. We use the product of the two earliest data points. We then iterate through the rest of the data, going forwards through time. At each iteration, we update our EWMA estimate.

```
for (int i = 1; i < numRow; i++)
        EWMA = lambda * EWMA
    + (1-lambda) * xVector[numRow -1 - i] * yVector[numRow -1 - i];
```

At the end of the loop, we return `EWMA`.

3.2.6 getEWMAVariance()

This method follows equation 23 which is the EWMA estimate of daily-covariance. The reasoning here for implementing covariance and not simply variance is the same as in `getVariance()`.

```
double lambda = 0.94;
double EWMA = xVector[numRow -1] * yVector[numRow -1];
```

Here, we use J.P. Morgan's RiskMetrics estimation of lambda $\lambda = 0.94$. The computation of EWMA is recursive and iterative so we must define some initial estimate for variance to begin with. We use the product of the two earliest data points. We then iterate through the rest of the data, going forwards through time. At each iteration, we update our EWMA estimate.

```
for (int i = 1; i < numRow; i++)
        EWMA = lambda * EWMA
    + (1-lambda) * xVector[numRow -1 - i] * yVector[numRow -1 - i];
```

At the end of the loop, we return `EWMA`.

3.2.7 getGARCH11Variance()

This is our implementation of the GARCH(1,1) estimate of daily-variance. Just as in the previous two methods for estimating daily-variance, we take the covariance approach and follow equation 24. GARCH(1,1) requires the estimation of three parameters, `double omega, alpha, beta`. These are found using the private method LevenbergMarquardt(**double**[] uSquaredArray), where uSquaredArray is simply an array containing the product of the instance variables **double**[] xVector and yVector.

```
for (int i = 1; i < uSquared.length;i++)
            sigmaSquared = omega
    + (alpha*uSquared[i]) + (beta*sigmaSquared);
```

Once these parameters are known, then the process of estimating the variance is similar to steps taken in getEWMAVariance(). We simply iterate through each observation, going forwards through time. At each step, we update our estimation of the variance. At the end of the loop, we return this estimate.

LevenbergMarquardt() The Levenberg-Marquardt algorithm is a hybrid of gradient descent and Newtonian method. We use it to maximise the log-likelihood function from equation 12. This log-likelihood is computed using the private method likelihood ().

In order to estimate the three parameters for GARCH(1,1), we must define some initial estimates for omega, alpha and beta.

```
parameters[0] = 0.000001346;     //omega
parameters[1] = 0.08339          //alpha
parameters[2] = 0.9101;          //beta
```

These initial estimates were lifted out of Chapter 22, page 506 in Hull [8]. Choosing the right initial parameters is important: The algorithm is capable of making rapid descents, but it can get stuck easily in local valleys, instead of the global valley.

We also need to specify some small fudge factor parameter. To begin with, we specify **private double** lambda = 0.001; This parameter governs the size of our steps. The algorithm adjusts this parameter accordingly depending on whether each step managed to increase the maximum likelihood estimate. Because a number of methods need to be able to write to lambda, it is an instance variable.

First we calculate return an estimation of the log-likelihood using the likelihood (uSquaredArray,parameters); method. Then we being a while loop which, on each iteration, maximises the log-likelihood until it can only produce insignificant increases and exits the loop.

In this loop, we produce some trial parameters using the method getTrialParameters(perameters,uSquaredArray). Using these trial parameters, we compute a trial estimate of the maximum likelihood estimate. If we have increased the likelihood, the trial parameters are accepted and we decrease the fudge factor by 10. Otherwise, we ignore the trial parameters and revert to the previous parameters and increase the fudge factor by 10.

likelihood() This `likelihood()` method is an implementation of the log-likelihood seen in equation 12. But first, we must generate an entire history of variance estimates by iterating through our data and computing the variance using a process similar to that found in the `getGARCH11Variance()` method.

```
for(int i = 1; i < variance.length; i++)
            variance[i] = omega + (alpha * uSquaredArray[i])
    + (variance[i-1]* beta);
```

That is, we implement the formula from equation 24, this time populating each result into an array `double[] variance` as we iterate through the data.

```
for(int i = 0; i < variance.length; i++)
            likelihood += -Math.log(variance[i])
    - (uSquaredArray[i+1]/variance[i]);
```

Lastly, to compute the maximum likelihood estimate, we iterate through each of our variance estimates, summing the log-likelihood as we go. At the end of the loop, we return `likelihood`.

getTrialParameters() `getTrialParameters()` takes `double[] parameters` and `double[] uSquaredArray` as inputs. These being our current parameter estimates and the squared price changes respectively. First we take a history of our variance estimates `double[] variance` just as line `likelihood()`. Then, iterating through `variance`, we compute the variables `double dOmega, dAlpha, dBeta`. The computation of these variables are implementations of the partial differential equations seen in equations 14, 15, and 16.

We then initialize the following array from equation 17:

```
double[] vectorBeta = {-0.5*dOmega, 0.5*dAlpha, -0.5*dBeta};
```

Then we enter a while-loop. Firstly, we initialize the curvature matrix from equation 18:

```
double[][] curvatureMatrix = {
{0.5*dOmega*dOmega * (1 + lambda), 0.5*dOmega*dAlpha, 0.5*dOmega*dBeta
    },

{0.5*dAlpha*dOmega, 0.5*dAlpha*dAlpha * (1 + lambda), 0.5*dAlpha*dBeta
    },

{0.5*dBeta*dOmega, 0.5*dBeta*dAlpha, 0.5*dBeta*dBeta * (1 + lambda)}};
```

Mathematically, we have a matrix and a vector, which is represented just as in equation 33. In order to solve the simultaneous equations, we use the `linear` package from the library `org.apache.commons.math3` [2]. The solution to the simultaneous equations is a an increment to the trial parameters. If this increment is too small, then we break out of the while loop. Otherwise, we increment the trial parameters. The trial parameters are subject to a few conditions:

$$a_j = \begin{cases} 0 \le \omega \le \infty \\ 0 \le \alpha \le 1 \\ 0 \le \beta \le 1 - \alpha \\ \alpha + \beta < 1 \end{cases} \tag{65}$$

where a_j is the parameter vector. If the trial parameters break these conditions, we increase the fudge factor by a factor of ten and move on to the next iteration of the loop. Otherwise, we will have found new trial parameters. So we break the loop and return `trialParameters`.

3.2.8 Volatility methods

In the univariate case, each volatility estimate is the square root of the estimate of daily variance. We have three methods that return volatility estimates: `getEWVolatility()`, `getEWMAVolatility()`, and `getGARCH11Volatility()`. Implementing each of these methods is simple.

```
public double getEWVolatility(){return Math.sqrt(getEWVariance());}
public double getEWMAVolatility(){return Math.sqrt(getEWMAVariance())
    ;}
public double getGARCH11Volatility(){return Math.sqrt(
    getGARCH11Variance());}
```

We call a method for the corresponding estimate of daily-variance and take the square root.

3.2.9 Encapsulating Variance and Volatility

Because we have so many methods for computing variance and volatility, it can be difficult to write code that isn't cumbersome. For instance, suppose we have a single array of price changes. We want to use a loop to compute all three variance estimates, but we cannot. We have to instead avoid a loop and write repetitive code that calls each method one by one.

To avoid this, we have implemented some helper methods. These take an input variable `int measures`, in which certain values are encoded to correspond to another method. The integers 1, 2, 3 encode to *Equal-Weighted*, *EWMA*, and *GARCH(1,1)* respectively.

We have two methods in this case: `getVariance(int measure)` and `getVolatility(int measure)`. For example, the following code will return an Equal-Weighted variance estimate of our price changes.

```
double variance = new Stats(priceChanges, priceChanges).getVariance(1);
```

The next example will return the GARCH(1,1) volatility estimate:

```
double volatility = new Stats(priceChanges, pricechanges). getVolatility (3);
```

This is a particularly useful feature when it comes to our implementation of `getCorrelationMatrix ()`.

3.2.10 getCorrelationMatrix()

We are able to construct correlation matrices using any of our variance and volatility methods. Thanks to the encapsulation discussed above, we are able write quite general code that still allows us to switch between the types of estimates very easily. First we declare a two-dimension square matrix.

```
double [][] matrix = new double[numCol][numCol];
```

It has a length equal to the number of stock variables in our portfolio. Using a nested loop, we iterate through each element and compute the correlation as in equation 31.

That is, at each element we compute the covariance between the ith and jth variable using the method `getVariance(measure)`, where `measure` is an integer that simply specifies what sort of estimate we want (EW, EMWA or GARCH(1,1). At the same time, with the `getVolatility (measure)` method, we compute the volatility estimate for the ith variable and the same for the jth variable.

```
matrix[i][j] = covXY / (sigmaX * sigmaY);
```

Then we take the covariance estimate and divide it by the product of our two volatility estimates to compute the correlation estimate for that element in the matrix. Once we populate the entire matrix, we return `matrix`.

3.2.11 getCovarianceMatrix()

Building a variance-covariance matrix is similar to building the correlation matrix - especially since covariance and correlation are so mathematically intertwined. As before, we declare a two dimensional matrix

```
double[][] covarianceMatrix = new double[numCol][numCol];
```

We iterate through each element in the same way using a nested loop and populate the value of each element by calling the `getVariance(measure)` method.

```
covarianceMatrix[i][j]
    = new Stats(multiVector[i], multiVector[j]).getVariance(measure);
```

When the entire matrix is populated, we return `covarianceMatrix`.

3.2.12 getCholeskyDecomposition()

This method is an implementation of the Cholesky Decomposition, following equations 52 and 53. In our Monte Carlo simulation, we are trying to find L such that $\Sigma = LL^T$ where Σ is our variance-covariance matrix.

This method does not take a covariance matrix as an input. That is because in MonteCarlo.java, we do not need one and as such never create one. Therefore, we first build a covariance matrix using the method getCovarianceMatrix(measure). Then we initialize a matrix for L.

```
double[][] covarianceMatrix = getCovarianceMatrix(measure);
double[][] cholesky
     = new double[covarianceMatrix.length][covarianceMatrix.length];
```

Then we enter a nested loop and iterate through each of the remaining elements. At every element, we do the following:

```
double sum = 0;
for (int k = 0; k < j; k++)
    sum += cholesky[i][k] * cholesky[j][k];
if(i==j)
    cholesky[i][j] = Math.sqrt(covarianceMatrix[i][j] - sum);
else
    cholesky[i][j] = (covarianceMatrix[i][j] - sum) / cholesky[j][j];
```

That is, we compute sum as $\sum_{k=1}^{i-1} L_{ik}L_{jk}$. Then, if we are on a diagonal, we subtract sum from the value on the same element on the covariance matrix and take the square root. If we are *under* the diagonal, we don't take the square root but divide by the diagonal element at i. When we exit the loop, we return the matrix cholesky.

3.2.13 getPercentageChanges()

Underpinning all our calculations when estimating VaR is our historical data. We need to compute the daily percentage changes, as seen in equation 1 when using any VaR measure. The method getPercentageChanges() will take a two-dimensional array of stock prices and iterate first by asset and then by day. Or, in other words, we use a nested loop in which we iterate through every element in a column and then move onto the next column. At each element we compute:

```
priceDiff[i][j] = ((multiVector[i][j]- multiVector[i][j+1])
    / multiVector[i][j+1]);
```

The end result is the two-dimensional array **double**[][] priceDiff that will be one row shorter than our array of stock data. If we call getMean() on priceDiff, we will return a result very close to zero.

3.3 VaR Measures

The following section aims to discuss how our VaR measures were implemented in Java. We have separated each measure into its own Java classes. Below is a list of the VaR measures implemented as Java classes. Included are their inputs and the data types they return.

Class	Input	Returns
Analytical . java	Parameters p **double** [][] stockPrices	**double**[]
Historic . java	Parameters p **double** [][] stockPrices optionsData [] options **int** printFlag	**double**
Montecarlo.java	Parameters p **double** [][] stockPrices optionsData [] options **int** printFlag	**double**[]
BackTest.java	Parameters p optionsData [] options	ArrayList <BackTestData>

where p is an instance of class Parameters.java and options is an instance of class optionsData. Similarly, BackTestData is an instantiation of class BackTestData.java.

All of these classes rely on methods that were described in Section 3.2, which in turn are implementations of mathematical theory discussed in Section 2 As such, we can now move on to describe the implementation of our VaR classes in a broader scope.

3.3.1 Analytical.java

This class makes use of the distribution library from the apache.commons.math3 package [2]. This is our implementation of the Analytical approach described in Section 2.2. More specifically, this is an implementation of the Linear Model shown in equation 33. The output of this class is a vector of three VaR estimates: one for each of our variance and volatility measures.

Preparation Before we can perform the necessary calculations, we must first prepare our variables, of which there are a few. Notably, we first initialize a standard Gaussian variable:

```java
NormalDistribution distribution = new NormalDistribution(0,1);
```

We use the inverseCumulativeProbability () method to find our percentile as described in equation 43:

```
double riskPercentile = - distribution.
    inverseCumulativeProbability(1-p.getConfidenceLevel());
```

where p.getConfidenceLevel() is our confidence level c.

Additionally, we initialize the array stockDelta and populate it from Parameters p using the method getStockDelta(). And for convenience, we initialize a one-dimensional array **double**[] currentStockPrices for our current stock prices and populate it from stockPrices, which is a two-dimensional array.

The process for Analytical.java continues as follows:

Algorithm 7 Class: Analytical.java

```
1: function ANALYTICAL.JAVA(Parameters p, double[][] stockPrices)
2:     String[] volatilityMeasures ← {EW, EWMA, GARCH}
3:     double[][] priceChanges ← stockPrices.getPercentageChanges()
4:     for all i in volatilityMeasures do
5:         correlationMatrix ← priceChanges.getCorrelationMatrix(i+1)
6:         volatility ← priceChanges.getVolatility(i+1)
7:         Compute Sum of Linear Components → sum
8:         VaR ← Math.sqrt(p.getTimeHorizon()) × riskPercentile × Math.sqrt(sum)
9:     return VaR
```

3.3.2 Historic.java

This is our implementation of the Historical VaR measure as described in Section 2.3.1. Unlike some other measures we implement, it returns only a single estimate of VaR.

Preparation We must first get some data from Parameters p. In doing so, we initialize **int**[] stockDelta and **int**[] optionDelta.

Then we declare some variables.

```
double[] currentStockPrices = new double[numSym];
double[][] strikePrices = new double[numSym][];
double[][] currentPutPrices = new double[numSym][];
int[] daystoMaturity = new int[numSym];
double todayPi = 0;
```

These will hold current stock prices, the parameters of our put options and today's value of our portfolio. We iterate through each of our stock symbols in a for-loop and populate these in the following way:

```
currentStockPrices[i] = stockPrices[i][0];
strikePrices[i] = options[i].getStrikePrices();
daystoMaturity[i] = options[i].getDaystoMaturity();
currentPutPrices[i] = options[i].getPutPrices();
int numPuts = currentPutPrices[i].length;
todayPi += stockDelta[i] * currentStockPrices[i]
    + optionDelta[i] * currentPutPrices[i][numPuts-1];
```

The process for Historic.java continues as follows:

Algorithm 8 Class: Historic.java

```
1: function HISTORIC.JAVA(Parameters p, double[][] stockPrices, optionsData[] options)
2:     double[][] priceChanges ← stockPrices.getPercentageChanges()
3:     tomorrowStockPrices ← priceChanges × currentStockPrices
4:     tomorrowPutPrices ← options.getBlackScholesPut(tomorrowStockPrices)
5:     Revalue Portfolio → tomorrowPi
6:     Sort tomorrowPi Ascending
7:     index = (1−p.getConfidenceLevel())×tomorrowPi.length
8:     VaR =(todayPi − tomorrowPi[index])×Math.sqrt(p.getTimeHorizon())
9:     return VaR
```

3.3.3 MonteCarlo.java

This is our implementation of the Monte Carlo VaR measure as described in Section 2.3.2. In MonteCarlo.java, we produce three estimates for VaR. This is because we have three variance and volatility measures. In addition to the main() method, we will define here two private methods randomWalk and weinerProcess. Both are integral parts to simulating a so-called random walk.

Preparation We begin in identical fashion to Historic.java in which we get some data from Parameters p, initialize some arrays and calculate the current value of the portfolio. Please refer to 3.3.2 for more detail as we will skip the parts that are identical.

We first initialize some variables that are key to the Monte Carlo simulation. We initialize **int** N = 24; and **int** paths = 10000;. These are the number of steps in each random walk and the number of random walks respectively. We also initialize **double** dt = 1/N; which gives us time steps 1 hour in size.

The procedure for MonteCarlo.java continues as follows:

Algorithm 9 Class: MonteCarlo.java

```
 1: function MonteCarlo.java(Parameters p, double [][] stockPrices, optionsData [] options)
 2:     int N ← 24
 3:     int paths ← 10000
 4:     double dt ← $1/N$
 5:     String [] volatilityMeasures ← {EW, EWMA, GARCH}
 6:     double [][] priceChanges ← stockPrices.getPercentageChanges()
 7:     for all i in volatilityMeasures do
 8:         choleskyDecomposition[i] ← priceChanges.getCholeskyDecomposition(i+1)
 9:         for all j in paths do
10:             tomorrowStockPrices[j] ← randomWalk(choleskyDecomposition)
11:         tomorrowPutPrices ← options.getBlackScholesPut(tomorrowStockPrices)
12:         Revalue Portfolio → tomorrowPi
13:         Sort tomorrowPi Ascending
14:         index = (1−p.getConfidenceLevel())×tomorrowPi.length
15:         VaR =(todayPi − tomorrowPi[index])
16:     return VaR
```

randomWalk() This method is our implementation of equation 48, the stochastic process describing the Monte Carlo simulation. It takes four parameters: `int N`, the number of steps; `double dt`, he magnitude of each step; `double[] currentStockPrices` and `double [][] choleskyDecomposition`.

Monte Carlo is a discrete process, so we declare a grid in which we simulate our random walks:

```java
double [][] grid = new double[numSym][N];
```

The starting values at the beginning of each walk will be our current stock prices. So for `N` steps, we build some random walks, one for each stock symbol. We generate a vector of correlated random variables that take the multivariate Gaussian distribution $N \sim \phi(0, L)$, where L is represented by the variable `choleskyDecomposition`. We do this in the following way:

```java
grid[j][i] = (correlatedRandomVariables[j]
    * grid[j][i-1] * Math.sqrt(dt)) + grid[j][i-1];
```

where `correlatedRandomVariable[j]` represents our vector of correlated random variables at step `j`. This is initialized by the method `weinerProcess()`. When we have reached the end of the grid, we return the terminal stock price for each stock symbol via: `terminalStockPrice[i] = grid[i][N−1];`.

weinerProcess() The Weiner Process is the $dz = \epsilon\sqrt{dt}$ part of equation 48, where ϵ is the standard Gaussian variable. We intialize the following to simulate our Gaussian variable.

```
Random epsilon = new Random();
```

For each of our stock symbols, we sample from this variable and populate an array that is IID.

```
dz[i] = epsilon.nextGaussian();
```

Then we perform the matrix multiplication of `double[] dz` and `double[][] choleskyDecomposition`, which gives us `double[] correlatedRandomVariables`.

```
for(int i = 0; i < numSym; i++) {
            double sum = 0;
            for (int j = 0; j < numSym; j++)
                sum += choleskyDecomposition[i][j] * dz[j];
            correlatedRandomVariables[i] = sum;
```

Then we reach the end of `weinerProcess()` and return `correlatedRandomVariables`.

3.3.4 BackTest.java

This is our implementation of the Back Testing as discussed in Section 2.4. `BackTest.java` uses five years of historical stock data to compute one thousand moments of VaR, which have each been computed using data from the 252 days prior to that moment. It is capable of handling portfolios with options, and does so without having any historical options data - it simulates them from the stock data. In addition to the `main()` method, we also have `testCoverage()` and `testKupiecPF()` which use the `distribution` package from the `org.apache.commons.math3` library [2]. These are implementations of the the two coverage tests: Standard and Kupiec's PF. Both define non-rejection intervals that tell us what number of violations is acceptable.

Preparation Part of the whole implementation of the `VaR` package involves printing information to a log file, which contains helpful information with regards to the status of the program. During back testing, we invoke our VaR measure many times, resulting in an information overload - the log is meant to legible for human eyes and more information is not necessarily better. As such, we simply turn it off at certain points in `BackTest.java`. To do this, this implementation uses code submitted by Baydoan [3] on Stack Overflow. This means redefining standard output from `System.out` to the folowing:

```
PrintStream dummyStream
    = new PrintStream(new OutputStream() public void write(int b) //NO-OP);
```

Next, we initialize two copies of `optionsData[] options`: `optionsBackTest1` and `optionsBackTest2`. We cannot simply write, for example, `optionsData[] options2 = options`; they both reference the same underlying object and are not unique objects. To do this, we defined a copy constructor in `optionsData.java`. Therefore, we simply initialize the variable as follows:

```
optionsBackTest1[i] = new optionsData(options[i]);
```

These copies serve two purposes. Firstly, we need to iterate through our five years of stock data and predict all option prices. As we go back in time, we must increment the maturity date in the instance of `optionsData`. We use a modulo of 252 to ensure the maturity data resets to zero on the same date each year, as it is not realistic that we hold on to the same put option for so long.

Secondly, each time we invoke `Historic.java` or `MonteCarlo.java`, we will need to update the maturity date and current put price accordingly.

All in all, we will return seven VaR estimates at each moment: Historical and (EW, EWMA and GARCH(1,1)) for both Analytical and Monte Carlo. We want to declare a two-dimensional array that can accommodate all these VaR estimates:

```
double[][] momentsVaR = new double[numMeasures][numMoments];
```

Procedure The procedure for `BackTest.java` is as follows:

doCoverageTests() Once we have a count of the number of times the real losses in the portfolio exceeded the VaR estimate (i.e., the number of violations), we invoke `doCoverageTests()`. Now, we perform our coverage tests at a number of significance levels. This method iterates through each significance level and runs `testCoverage()` `testKupiecPF()`. At each iteration, the results from these methods are stored in instances of `BackTestData` called `standardBT` and `kupiecBT` respectively. These results are then added to an ArrayList and returned to `main()`.

testCoverage() This method is the implementation of the Standard Coverage test from Section 2.4.1. First we initialize `int alpha = numMoments + 1` and construct a binomial distribution in the following way:

```
BinomialDistribution distribution
    = new BinomialDistribution(alpha, 1-confidenceX);
```

Algorithm 10 Class: BackTest.java

```
 1: function BackTest.java(Parameters p,optionsData[] options)
 2:     int numYears←5
 3:     int numMoments←1000
 4:     double[][] stockPrices ← getStocks(numYears)
 5:     for all i in stockPrices do
 6:         optionsBackTest1.setDaystoMaturity((daystoMaturity+i)%252)
 7:         optionPrices[i] = optionsBackTest1.getBlackScholesPut(stockPrices[i])
 8:         Value Portfolio → valuePi
 9:         Calculate daily returns() → deltaPi
10:     for all i in numMoments do
11:         Take 252 days prior to moment $i$: stockPrices → momentStockPrices
12:         Take options at moment $i$: optionsPrices → momentOptionPrices
13:         Estimate VaR at moment $i$ →momentsVaR[][i]
14:         if deltaPi > momentsVaR then violations ++
15:     ArrayListBT←doCoverageTests(p.getConfidenceLevel, numMoments,violations)
16:     return ArrayListBT
```

This class gives us the cumulativeProbability method. We use this a series of while-loops in which we increment **int** a until pr <= epsilon/2 where:

```
pr = distribution.cumulativeProbability(a);
```

and likewise we minimize **int** b until pr >= epsilon/2 where:

```
pr = 1 - distribution.cumulativeProbability(a);
```

This process ultimately arrives us at some most likely lower and upper values of the the non-rejection interval which could be either $a + n, b$ or $a, b + n$. Deciding between the two is another similar step involving maximising likelihoods. Once decided, we return the one-dimensional array **int** [] nonRejectionInterval .

testKupiecPF() This method is the implementation of the Standard Coverage test from Section 2.4.2. First we construct a χ^2 distribution and use the inverseCumulativeProbability method to compute the quantile.

```
double quantile
    = distribution.inverseCumulativeProbability(1-epsilon);
```

where epsilon is a given significance level.

Then we enter a while-loop in which we increment from **int** i = 0. On every iteration, we call loglikelihood (), which is the log-likelihood ratio given in equation 64. If `loglikelihood ()` decreases below `quantile`, then the lower interval is either the current or previous increment, depending on whichever value gets us a likelihood closest to `quantile`. We do the same for the upper interval, except we look for an increase of `loglikelihood` above `quantile`.

Once we have found our intervals, we return the one-dimensional array `int [] nonRejectionInterval`.

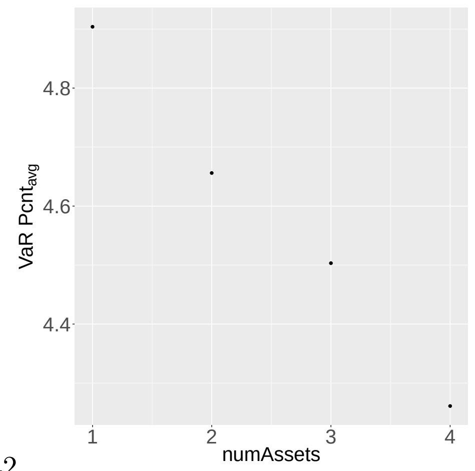
4 Experimentation

We test four types of portfolio: single-stock, two-stock, three-stock, and three-stock with options on the stocks. The stock symbols we use are: GOOG, AAPL, and MSFT. We will look at each of these portfolios with 1-day and 10-day time horizons and 95% and 99% confidence levels. Each experiment will use 5 years of data and perform back testing and stress testing, where our stress test is a five year back test beginning on 1st January 2007. It's important to note that options were excluded from out portfolio during the stress test, due to a complete lack of data. The full battery of inputs for each experiment can be found in `runTests.java`.

4.1 Results

Diversification First, let us consider what impact increasing the number of assets in our portfolio had on our VaR estimates. In figure 3 on page 42, we can see the average VaR as a percentage against the number of assets. Increasing the number of assets in our portfolio decreases our exposure to risk. Thus we demonstrate the benefits of diversification.

Figure 3: Diversification



Non-Rejection Intervals In our back test, we defined non-rejection intervals using coverage tests. The size of these intervals were varied depending on the chosen significance level. In figure 4 on page 43, we can see two sets of intervals: the set with the mean at 50 represents our intervals at 95% confidence. The other set has a mean at 10 and represents our intervals at 99% confidence. The range between the intervals is larger at 95%. We would expect that the number of violations that land within the non-rejection interval to be much smaller at 99% than at 95%.

In figure 5 on page 5 we see the distribution of the violations with respect to confidence during Back Testing and Stress Testing. It is apparent that our during our Stress Test, we an increase in violations. This suggests that our VaR measures are not particularly affective during crises. But nonetheless, even during Back Testing, the mean violations fall below our expected number of violations. It is even below the lower bound of our widest non-rejection interval at 95%. If we must widen this interval, then this might suggest that our VaR estimates are not particularly accurate.

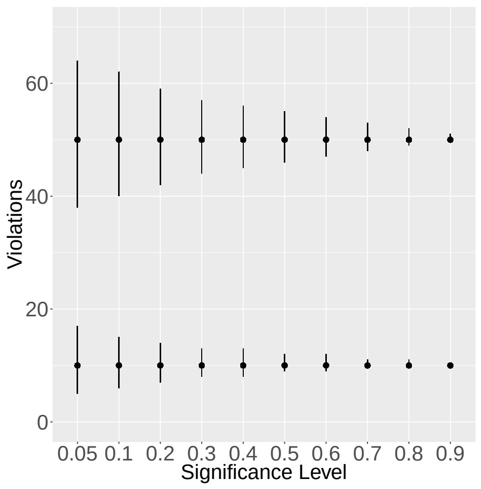


Figure 4: Non-Rejection Intervals at 95% and 99%

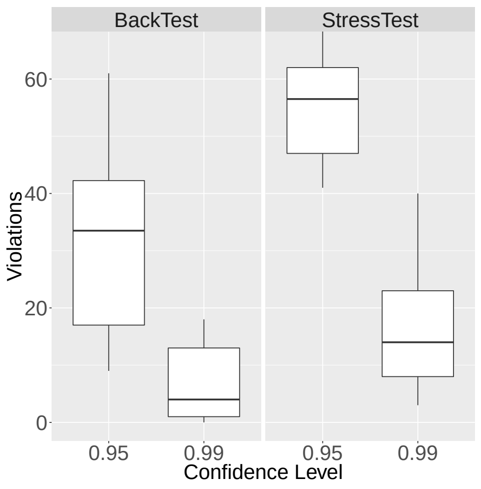


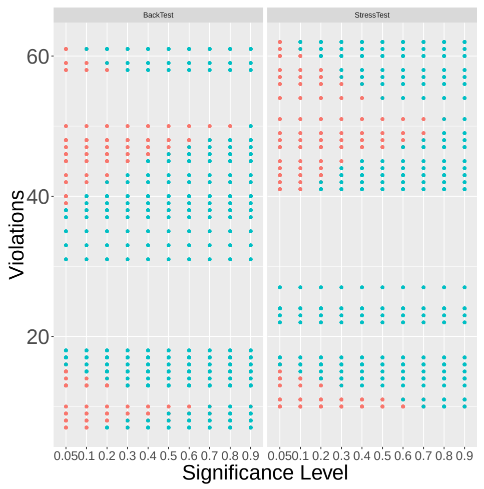
Figure 5: Distribution of Violations at 95% and 99%

In figure 6 on page 44, we see the distribution of our violations in a dot plot at a time horizon of 1 day. In red are the the violations that fall within the non-rejection interval. The blue violations fall outside this interval. Interestingly, the maximum number of violations that occurred during the Stress Test is not all that much higher than during the Back Test. Comparing this distribution against the box plot in figure 5, we see that the increase in violations is largely due to the exclusion of data from the experiments that used a 10 day time horizon. We can even see that at the

95% confidence level we have no rejected violations during the Stress Test at a significance level of 5%, which gives us an indication of how confident and accurate we are being.

In Back Testing, there are noticeable gaps in the distribution where we should have some non-rejected violations. It is possible that there is a gap in the data. More worryingly, it could suggest that the past five years of market conditions are much worse than the period in which the Stress Test takes place.

Figure 6: Accepted/Rejected Violations



Which Measure? Now, having tested seven different VaR measures, we must ascertain whether their performance differs and even whether one measure is definitively better than the others. We would expect the Analytical approach to perform similarly to the Monte Carlo approach as they both make the same underlying probabilistic assumptions. The only difference being is how they go about producing their estimates.

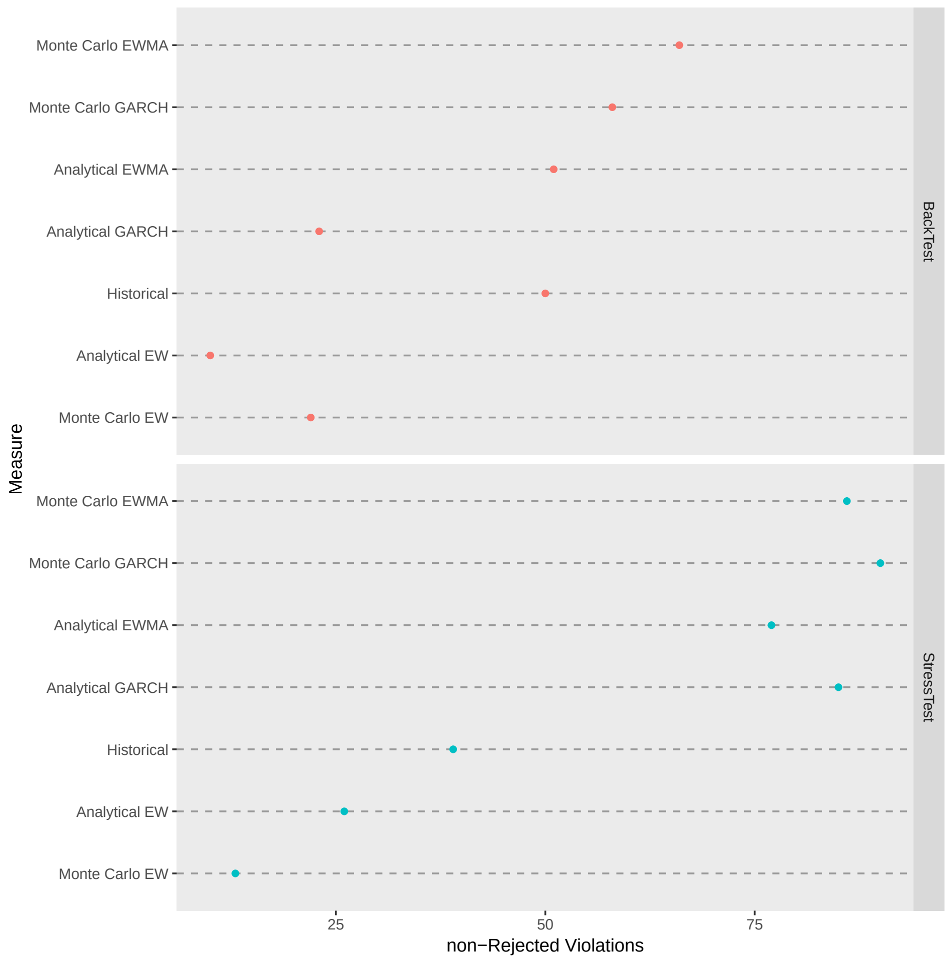
In figure 7 on page 45, we can see the number of non-rejected violations for each measure. We see more non-rejections during the Stress Test, which is consistent with our other observations. In either case, the two equal-weighted measures have performed the worst. Confusingly, Analytical GARCH is in the bottom three during Back Test but is in the top three during Stress Test. Both EWMA measures had among the highest levels of non-rejections; in particular for the Monte Carlo EWMA measure.

4.2 Conclusions

In our experiments, we have had some successes and on the other hand, some questions remain unanswered.

Firstly, we were able to demonstrate the benefits of diversification. In Section 2.2.1 on page 7 onwards, we talk about how uncorrelated market assets can reduce the VaR in the portfolio. As such we saw that the proportion

Figure 7: Measures/Non-Rejected Violations



of VaR decreases as we increase the number of assets in the portfolio.

With regards to the time horizon, we saw more violations on the whole with a 10-day time horizon. This is in line with our expectations.

Some of results, on the other hand defy expectations. For instance, we saw more violations in our non-rejection intervals during the Stress Test. Now, is our implementation at fault or is the market at the current point in time *stressed*? One factor to take into consideration is that we did not have any options in our portfolios during the Stress Test.

The Monte Carlo and Analytical approaches generally performed well, but their results were expected to be similar. Perhaps the number of random walks plotted was insufficient. However, the Monte Carlo simulation is very slow. At 10,000 random walks, it took us seven hours on a *Core i5* processor to run the entire battery of experiments. Unfortunately, increasing the number of random walks was not viable, given the time constraints.

Despite this, the Monte Carlo and Analytical EWMA performed well. It is interesting to note that the lambda parameter that EWMA relies upon was taken from J.P. Morgan's RiskMetrics. Whereas parameters for GARCH(1,1) were estimated by our own implementation of Levenberg-Marquardt and we saw strong inconsistencies in its performance.

The equal-weighted measures performed the worst It could be argued that the equal-weighted assumption does not work. And as for the Historical approach, which makes no probabilistic assumptions, but assumes that future market performance will be like the past, there is not much to say about it. It is consistently in between the other measures.

Some questions which remain unanswered pertain to confidence and significance level. We saw, in once instance, that we had zero rejections with a 1-day time horizon, 95% confidence and 5% significance levels during the Stress Test. But ultimately, we still are not able to say how confident and how accurate we are willing to be with regard to significance and confidence levels.

5 Self-Assessment

Planning The Gantt chart I produced was an overly optimistic plan for how everything would happen on a consistent, week-by-week basis. It was of course, far from reality. Instead of implementing each feature in piecemeal blocks of time, it was a process of continual improvement and refinement in all areas. For instance, I expected to have data acquisition methods over and done with early on in the project. Instead, I found myself revisiting different aspects of it again and again. Options data was not fully implemented until mid-July, give or take.

I suffered unforeseen set-backs as well; I fell ill *twice*, during which time no work was done. Ultimately, while I didn't manage to stick to the plan in any chronological sense, I did manage to stick with the *spirit* of the plan. That is, nearly all the features written on there are implemented - except for the χ^2 distributions.

Theory and Application Mathematics is not my expertise and has not been present in my life since my undergraduate studies some time ago. The main challenge of this project was understanding the theory and applying it to the real-world, which is not very straight forward. Just as there are many ways of estimating variance, I have learnt that there are many ways of approaching a problem.

So when it came to writing Java, I found that it was expected that I could potentially spend a lot of time trying one approach only to find that it was a dead end. For instance, when I spent a few days looking into Yahoo Finance only to discover that the API had been shut down a month prior. Or, for example, when I tried following Hull's [8] approach to taking 501 days of data and found that bank holidays are a mystery unto themselves.

The Future Ultimately, I have enjoyed this project. I hope it will lead me into a career in finance as a Data Scientist.

6 Professional Issues

Conflicts of Interest With full disclosure, it must be stated that my project supervisor, Dr Kalnishkan, has also been my lecturer in two modules: CS5200 and CS5930. As Dr Kalnishkan is the second marker of this dissertation, there presents itself a potential conflict of interest: does prior information regarding my performance in these modules impact the marking of this dissertation in any way? It does not and cannot.

In both modules, coursework and exams results were anonymised. Additionally, it must be taken at face-value that, throughout the length of this project, whether in meetings or in email correspondence, Dr Kalnishkan and I never discussed my results ever.

Inequality Economists, such as Thomas Piketty [11], have voiced strong concerns regarding the problem of the widening inequality gap between the rich and poor. As such, one question that must be asked with regards to the technology discussed in this dissertation is: *who does this benefit*? Perhaps it is more likely to benefit a select few: wealthy individuals who have the means to invest such as or banks, for instance.

Additionally, this technology is depends on access to reliable data. While we were able to source data for free, there were some restrictions placed on it. For instance, real-time price data are delayed by 15 minutes in many cases. But more problematic for us was that we did not have any historical options data. Ultimately, good financial data is costly and out of reach for most.

On the other hand, let us consider pension funds, for example. The beneficiaries of pension funds are not just the rich. Since pension funds typically invest their funds, those managing these type of funds certainly have a fiduciary responsibility to manage risk.

Commercial Use In sourcing data from Google Finance, we agree to not use it in a commercial enterprise without prior written consent [4].

7 How To Use My Project

Using the .jar Our program is VaR.jar.

Suppose we want to run an experiment where our portfolio consists of a single stock symbol: GOOG at 100 shares and zero options. We wish to take 5 years of data, with a 1-day time horizon and confidence interval of 95%.

We execute the .jar with the command: `java -jar VaR.jar GOOG 100 0 5 1 0.95`. Now, suppose want three stocks in our portfolio, but no options. We must use a pipe separated string to separate the stock symbols and stock deltas. This time our command is

```
java -jar VaR.jar GOOG|AAPL|MSFT 100|200|100 0 5 1 0.95
```

That is, we assume a portfolio consisting of the symbols GOOG, AAPL and MSFT with 100,200, and 100 shares respectively. Now we include options on the shares. The command is now:

```
java -jar VaR.jar GOOG|AAPL|MSFT 100|200|100 40|30|40 5 1 0.95
```

That is we now have 40, 30, 40 put options for GOOG, AAPL, and MSFT.

Compilation The `.jar` file was compiled using IntelliJ IDEA 2016.3.7 on a machine running Windows 10. The source files and their dependencies are included in a folder called `src` and `lib` respectively.

Output Running the program will take some time. On a Core i5, a single experiment can take between 5 and 20 minutes to run, depending on the portfolio. On `linux.cim.rhul.ac.uk`, this will take considerably longer. Program outputs will be saved to a folder in the directory of the `.jar` file. These will include: `.csv` files containing the simulated stock prices and put prices from Monte Carlo and Historical simulations; a `.csv` containing the 1000 moments of estimated VaR; a `output.txt` file containing logging; and lastly `rawData.csv` which contains the experimental results.

References

[1]

[2] Apache. Apache commons math 3.6 api. Accessed: 2017-08-25.

[3] Kerem Baydoan. Hiding system.out.print calls of a class, 2011. Accessed: 2017-08-25.

[4] Google. Accessed - 2017-08-30.

[5] Google. Github — google.gson: A java serialization/deserialization library to convert java objects into json and back, 2017. Accessed: 2017-08-24.

[6] Glyn A. Holton. *Value-at-Risk: Theory and Practice.* Published by the author, 66 Winslow Rd. Belmont, MA 02478, United States, second edition, 2014.

[7] http://financequotes api.com. Java finance quotes api for yahoo finance, 2017. Accessed: 2017-08-13.

[8] John C. Hull. *Options, Futures, And Other Derivatives.* Prentice Hall, New Jersey, United States, eighth edition, 2012.

[9] Bob Jansen. How to apply levenberg marquardt to max likelihood estimation, 2014. Accessed: 2017-08-14.

[10] Tommaso Pasini. How to save a 2d array into a text file with bufferedwriter?, 2016. Accessed: 2017-08-25.

[11] Thomas Piketty. *Capital*. Harvard University Press, Cambridge, Massachusetts, United States, first edition, 2014.

[12] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C*. Cambridge University Press, Cambridge, United Kingdom, second edition, 1992.

[13] Query String. Query string — Wikipedia, the free encyclopedia, 2017. Accessed: 2017-08-23.

[14] Reading Directly from a URL. Reading directly from a url, 2015. Accessed: 2017-08-24.

[15] Marcelo Tataje. Export array values to csv file java, 2013. Accessed: 2017-08-25.

[16] Username: BalusC. Java string to date conversion, 2010. Accessed: 2017-08-24.

[17] Username: BalusC. Programatically downloading csv files with java, 2010. Accessed: 2017-08-24.

[18] Username: Nixon. Has yahoo stopped history data api service?, 2017. Accessed: 2017-08-13.

[19] username: user2654569. simplest way to read json from a url in java, 2014. Accessed: 2017-08-24.

[20] Paul Wilmott. *Paul Wilmott Introduces Quantitative Finance*. John Wiley & Sons, Ltd, Hoboken, New Jersey, United States, second edition, 2007.