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 Where other published and unpublished source materials have been used, these have been acknowledged.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

# 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 [6] 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 $\tau$. 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 [6] 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* $\sigma_n^2$ 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 $\alpha_i$. 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 $\alpha$'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 $\lambda$. That is, $\alpha_{i+1} = \lambda\alpha_i$ where the inverse *rate-of-decay*, $\lambda$, is a constant between 0 and 1. If $\lambda = 0$, most of our influence comes from $u_{n-1}^2$. If $\lambda$ is large, this means we won't discriminate as much against observations in the past. As such, $\lambda$ allows us to control the influence of the most recent data.

As we iterate through our data, we maintain an estimate of $\sigma^2$ 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 $\sigma_n$ at the end of day $n$ is calculated from the end of the previous day's estimate of volatility $\sigma_{n-1}$ (which, in turn, was calculated from $\sigma_{n-2}$ 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 $\sigma_n^2$:

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

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, $\sigma_n^2$ 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.

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

$\lambda$ 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 $\gamma$.

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

We can replace $\gamma V_L$ with $\omega$. 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 $\omega, \alpha$ and, $\beta$ 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 [9] demonstrates LM as a tool to optimize parameters for least squares, in which the merit function $\chi^2$ 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 [7], 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 $\chi^2$.

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

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

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

We define the vector $\beta_k$ as:

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

and matrix $\alpha_{kl}$ 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 $\lambda$ 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 $\delta a_l$:

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

$\delta a_l$ 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

**procedure** LEVENBERG-MARQUARDT
    Compute $\chi^2(a)$
    $\lambda \leftarrow 0.001$
    **while** $\delta a > 0$ **do**
        Solve $\sum_{i=1}^{M} \alpha_{kl} \delta a_l = \beta_k$ for $\delta \alpha_l$
        **if** $\chi^2(a + \delta a) > \chi^2(a)$ **then**
            $\lambda \leftarrow \lambda \times 0.1.$
            $a \leftarrow a + \delta a$
        **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 $\sigma_n^2$ 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 $\sigma_x$ and $\sigma_y$. 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 $\sigma^2$.

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 $\omega, \alpha$, and $\beta$ 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 [6] Chapter 21, page 478 onwards and Willmott [15] 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 $\delta t$ 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 $\Pi_1$ and VaR for $\Pi_2$. This is because losses in $\Pi_1$ do not necessarily coincide with losses in $\Pi_2$.
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 $\mathrm{var}(\Delta\Pi_1) = \mathbb{E}(\Delta\Pi_1 - \mathbb{E}\Delta\Pi_1)^2$. The variance of $(\Delta\Pi_1 + \Delta\Pi_2)$ is:

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

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:

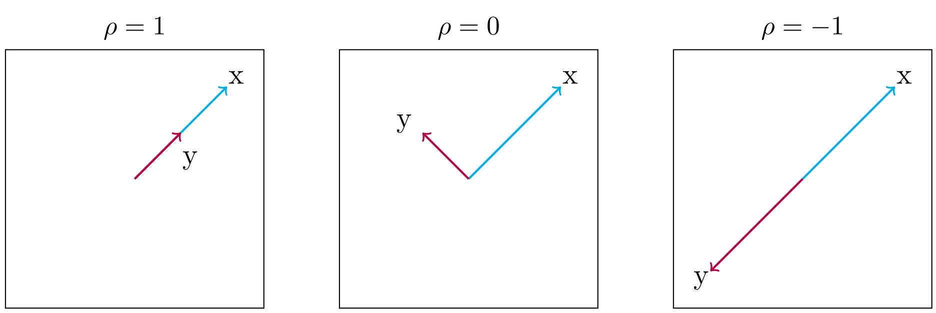
$$\text{std}(\Delta\Pi_1 + \Delta\Pi_2) = \sqrt{\text{std}(\Delta\Pi_1)^2 + \text{std}(\Delta\Pi_2)^2 + 2\rho \cdot \text{std}(\Delta\Pi_1) \cdot \text{std}(\Delta\Pi_2)} \tag{30}$$

where $\rho$ 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 $\rho$ 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 $\delta t$ 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 $\rho_{ij}$ is actually a matrix in which the $i$th row and $j$th column is the correlation between the $i$th and $j$th 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 $\Sigma$ 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 $\Delta t$, 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 $\mu$ is much smaller than the volatility $\sigma$. 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 $\Delta S$ by $k^2$ such that the distribution of $\Delta\Pi$ is:

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

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 $V$? $\Pi$ is Gaussian with a mean of zero. The density of the change in the value of the portfolio looks like:

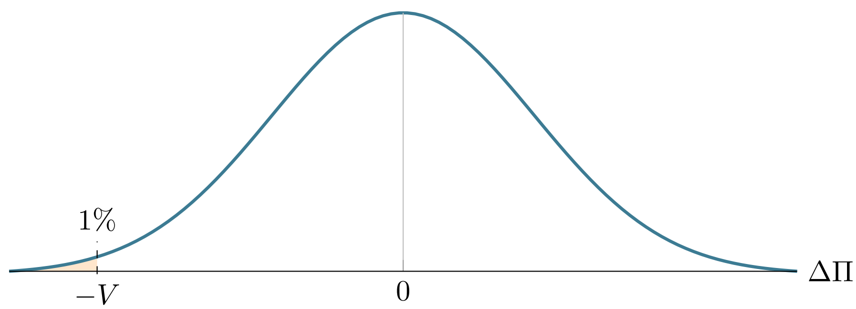
We need to find $V$ such that the shaded area is 1%. Analytically, we look at this problem in terms of standard Gaussian. We can define $\Delta\Pi$:

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

where $\epsilon$ 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 [15] writes it differently:

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

where $\alpha$ is the inverse cumulative distribution function for the standard Gaussian and the Greek letter $\Delta$ 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 $\delta t$.

## 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 [6] 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 $\Delta S_i$. 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 $\Delta S_i$, 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 $\Delta t$ is the time horizon.

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

**Algorithm 2** Historical method

**procedure** HISTORICAL
- Value $\Pi^{today}$ from today's $S_i$
- **for all** assets $1 \leq i \leq n$ **do**
  - Calculate vector $\Delta S_i$ from historical data
  - Apply all $\Delta S_i$ to $S_i$
- Revalue for $\Pi^{tomorrow}$
- $\Delta\Pi = \Pi^{tomorrow} - \Pi^{today}$
- Sort $\Delta\Pi$ in descending order
- VaR $\leftarrow \Delta\Pi_{99\%}\sqrt{\Delta t}$

### 2.3.2 Monte Carlo

Our treatment of the Monte Carlo simulation follows Hull [6] 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 $\Delta S_i$. Our main assumption now is that these vectors take the multivariate Gaussian distribution, the parameters of which we will estimate from $\Delta S_i$. Once again, we assume this vector has a mean of zero. As such, we only need find the variance-covariance matrix $\Sigma$.

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

Once we know $\Sigma$, 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 $\mu dt$ (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 $\Sigma$, analogous to the square root of $\Sigma$. Our portfolio consists of multiple stocks so to find $\Sigma$ 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 [9]. In the univariate case, $L$ would be equivalent to taking the square root of $\sigma^2$. 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 $\Sigma$.

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

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:

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

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

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

and below the diagonal:

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

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

**Correlating IID random variables** In equation 48, $\epsilon$ is vector of independent and identically distributed (IID) random variables sampled from the standard Gaussian. The product of $L$ and $\epsilon_i$ 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 $\Delta S_i$ for all prices in our new distribution. Just as in the Historical method, we revalue our portfolio at all $\Delta S_i$ to get $\Delta\Pi$. We then sort this vector in descending order and with a time horizon $\Delta t$ 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

**procedure** MONTE CARLO
    Value today's $\Pi$ from $S_i^0$
    Calculate vector $\Delta S_i$ from historical data
    Compute $\text{cov}_{ij}$ between $\Delta S_i$ and $\Delta S_j$
    Populate $\Sigma$
    Compute $L$ via Cholesky Decomposition
    **for** 10000 random walks **do**
        $t \leftarrow 0$
        **while** t < N **do**
            Sample $\epsilon_i$ from the standard Gaussian
            $S_i^{t+1} = S_i^t + S_i^t L \epsilon_i \sqrt{dt}$
            t++
        **return** $S_i^N$
        Value tomorrow's $\Pi$ from $S_i^N$
    Compute all increments $\Delta\Pi$
    Sort samples in descending order
    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 [6]:

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

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

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:

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

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

```
procedure SIMULATION WITH OPTIONS
    Value today's Π from S_i^0 and today's option price

    Simulate distribution of tomorrow's stock prices using Historical or Monte Carlo method
    Estimate tomorrow's option price using Black Scholes
    Value tomorrow's Π from S_i^N and tomorrow's option price
    Compute all increments ΔΠ
    Sort samples in descending order
    VaR ← ΔΠ_99% √Δt
```

## 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 $\Delta t$ 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 [4] 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 $\alpha$ is the number of moments of VaR we take in our backtesting.

Suppose we have some significance level $\epsilon$, which controls the precision of our estimation. We are able to test $H_0$ at any $\epsilon$. 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 $\epsilon$.

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 $\epsilon$. 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 $\epsilon$.

First we calculate the $\epsilon$ quantile of the $\chi^2$ 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
$\alpha$ = 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 [5] 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 [13], 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 [10].

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.

### 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 [3] and follow instructions submitted by user2654569 [14] 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 [12] 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 5** Class: getOptions.java

```
function getOptions.java(String [] symbols)
    Declare optionsData [] options
    for all String [] symbols do
        Declare optionsData options
        Concatenate URL string
        Download JSON string
        Parse into JsonObject
        Extract data
        Set optionsData options
        return optionsData options
    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:

```java
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 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 [11] and Pasini [8] on Stack Overflow on how to output data to csv using BufferedWriter and StringBuilder.

### 3.2.2 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.

### 3.2.3 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.

### 3.2.4 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`.

### 3.2.5 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 [6]. 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:

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

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`.

### 3.2.6 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.

### 3.2.7 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 ()`.

### 3.2.8 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 $i$th and $j$th 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 $i$th variable and the same for the $j$th 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.

### 3.2.9 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.

### 3.2.10 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 $\Sigma$ 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.

### 3.2.11 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.

### 3.2.12 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;
```

## References


[1]

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

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

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

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

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

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

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

[9] 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.

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

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

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

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

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

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