Creating Software for Value at Risk

Final Report

BSc Final Year Project
2024

Author
Benjamin Shearlock

Supervisor
Dr. Volodya Vovk

Department of Computer Science
Royal Holloway, University of London

Declaration

I have read and understood the Universities regulations on plagiarism and I hereby declare that all work submitted in this project is my own, except where explicitly stated otherwise, such as the references that have been cited.

- Word Count:
- Name: Benjamin Charles Shearlock
- Submission Date: 12/04/2024
- Youtube Video showcasing Final Deliverable Program:
- All other programs included within my PROJECT Git repository are simple python programs that can just be ran normally, these mainly being found within the "Command Line VaR Programs" folder. The GUI program has been compiled into a .exe file within the main directory, called "Initial Design Test.exe", it will open up a terminal when ran but please ignore it, it should work perfectly otherwise. The code for it is within the "Kivy" folder, that has other kivy tests inside it as well. It is all in the branch "Master", as I was having issues with my "Main" branch.
- ADD A SECTION HERE ON HOW TO RUN THE SOFTWARE, ALSO EDIT AND CHANGE THE ABOVE ETC.

Contents

1 Abstract

In the realm of financial risk management, understanding and evaluating the level of risk associated with any investment or portfolio is of extremely high importance. Perhaps the most universally regarded metric used for this purpose is Value at Risk (VaR). VaR provides a quantitative estimate of the potential losses that a portfolio may incur over a certain period of time (specified time horizon) at a given confidence level.

Original widespread use of VaR came about in the early 1990s, the concept first being introduced by J.P. Morgan in 1994, since it helped provide an estimate of the maximum loss an investor is willing to accept for any given investment. Its historical roots can be traced back to the financial industry's increasing need for a standardized and comprehensive measure of risk following the 1987 stock market crash, so they sought a comprehensive way to assess risk in complex portfolios [3]. Mathematically, VaR is expressed as follows:

$$VaR(N, X) = -\text{Percentile}(L, 1 - X) \tag{1}$$

Where:

$$N : \text{Time horizon (in days)}$$
$$X : \text{Confidence level (in percentage)}$$
$$L : \text{Loss distribution over } N \text{ days}$$

This formula captures the loss at the $(100 - X)$th percentile of the loss distribution over the specified time horizon [6].

VaR can be mathematically computed through various methods, each with its own strengths and limitations. The most common approaches include the historical simulation method, parametric method, model building method and Monte Carlo simulation [1], to list a few. For historical simulation, past data is used to estimate future risk by examining the historical returns of an asset or portfolio, while model building employs mathematical models to predict portfolio performance. The choice of algorithm depends on data availability, computational resources, and specific requirements.

To implement VaR calculations, various pieces of software are essential. In this project, I will be utilising Visual Studio Code (VSCode) as the integrated development environment (IDE) and Python for its rich ecosystem of libraries. Python libraries like NumPy, Matplotlib, and Kivy will be invaluable for data manipulation, visualisation, and user interface design [9].

My inaugural proof of concept program will be created to visually demonstrate VaR calculations for two initial methods, these being historical simulation and model building techniques to estimate VaR for a sample portfolio. Historical simulation would involve collecting historical data and computing VaR based on past performance (can involve acquiring stock data through an API), while model building would use a predefined model to forecast future losses. Visualisation tools like Matplotlib can help in presenting the results graphically, enabling me to generate informative charts and graphs. NumPy will facilitate data manipulation and efficient mathematical operations [10]. Additionally, Kivy, a Python framework for developing multi-touch applications, will be used to create the interfaces for the visual representation of VaR, as well as giving it the option to possibly be viewed on other devices.

Later on into the project, if there is enough time, I think it may be worth exploring some more advanced topics like the variance-covariance of returns, specifically employing GARCH (Generalized Autoregressive Conditional Heteroscedasticity) models. GARCH models provide a more nuanced understanding of volatility and can enhance the accuracy of VaR estimates [6].

The objective of my project is to gain a deeper understanding of VaR, develop a functional program to calculate and visualise it, and potentially extend the research to incorporate more advanced risk management techniques if possible. This project is a stepping stone towards a deeper understanding of the risk side of finances and will contribute to enhancing the knowledge and skills necessary for effective financial decision-making, since this is not a topic that I have delved much into before, but I have aways been very interested in learning more about it. This gives me a fantastic opportunity to learn about the financial sector, as well as create something that is applicable and useful to real life.

2 Chapter 1 - Introduction

2.1 What is VaR?

Value at Risk (VaR) is a statistical value used to quantify the level of financial risk within a portfolio of stocks/derivatives over a specified time horizon. It estimates the maximum potential loss that an investment portfolio could incur with a given probability, known as the confidence level, under normal market conditions. The VaR metric is typically expressed in monetary terms and is often used to assess the risk of a portfolio of financial instruments. From a banks perspective, they would say:

> "With **X% confidence (1-X% Risk Level)**, we expect to not lose more than **V** over the next **N day(s)** on our entire **portfolio** of derivatives, worth **Z**" - [4]

Where:

X : Confidence Level

V : Value at Risk

N : Time Horizon

Z : Portfolio Value

This is incredibly useful as it allows for a simple and easy to understand metric to be used to assess the risk of a portfolio, which is extremely important in the financial sector, taking all the underlying complexities of any form of portfolio and being able to conform its financial risk into a single, universally used, number.

To include actual numbers, to easier explain how it would look, a bank would say:

> "With **95%** confidence (**5%** Risk Level), we expect to not lose more than **£3,000,000** over the next **1 day** on our entire portfolio of derivatives, worth **£100,000,000**"

This is essential to help financial institutions understand the level of risk associated with their portfolios, as well as being able to compare the risk of different portfolios, which is why it is such a widely used metric in the financial sector, limiting the level of risk that a person/organisation is exposed to.

The concept of VaR has its roots in the late 20th century, gaining prominence in the finance industry during the 1990s. It emerged from the need for more sophisticated risk management tools in the wake of financial market liberalisation and the increasing complexity of financial instruments. The widespread adoption of VaR was catalysed by the 1998 financial crisis, where it played a significant role in risk assessment and regulatory frameworks.

VaR is significant for several reasons:

- **Risk Measurement:** VaR provides a quantitative measure of potential losses in a portfolio.
- **Decision Making:** VaR aids financial managers in making informed decisions about risk tolerance and capital allocation.
- **Market Risk Management:** VaR is used to monitor and mitigate market risks, contributing to the overall stability of financial systems.

VaR has become a cornerstone in risk management for global financial markets. It is used by banks, investment firms, asset managers, and corporates to measure and control the level of risk exposure in their financial portfolios. VaR's adoption is partly driven by regulatory requirements, such as Basel Accords, which mandate financial institutions to calculate and maintain adequate capital reserves based on their risk exposure.[12]

Developing software for VaR calculation offers numerous advantages:

- **Accessibility:** It establishes a widespread access to sophisticated risk management tools.
- **Efficiency:** Automated VaR calculations save time and reduce the potential for human error.
- **Customization:** Software can be tailored to specific needs and types of portfolios.
- **Real-Time Analysis:** It enables rapid and up-to-date risk assessments.

VaR has become an essential tool in modern financial risk management. The development of software for VaR calculations aligns with the need for efficient, accurate, and accessible risk management tools in today's fast-paced financial markets. This is what I want to help contribute to within this project.

2.2 Aims and Goals

My aims and goals for this project are as follows:

1. **Comprehensive Understanding of VaR:** To touch on VaR's theoretical underpinnings and why it has such a pivotal role in modern financial risk management. This includes commenting on its applications across different financial groups, market conditions, and regulatory environments.
2. **Methodological Exploration:** To investigate various computational methods for calculating VaR, including historical simulation, model building (variance-covariance) approach, and Monte Carlo Simulation, assessing their efficacy in different market scenarios. This will first be explored through command line programs, then later on through a GUI.
3. **Technical Implementation:** Development of a comprehensive program for calculating and presenting VaR. This entails creating a user-friendly interface, ensuring accurate and efficient computation, and integrating various methods of VaR calculation to provide a comprehensive tool.
4. **Application in Diverse Financial Contexts:** To ensure the software's adaptability and applicability in different financial settings. This includes testing the software with various data sets, portfolio types, and market conditions, aiming to make it a versatile tool for different financial entities.
5. **Industry Standard Tool Development:** Creating a software application that not only performs standard VaR calculations but also offers other stock-oriented features, such as advanced data visualisation. The goal is to make the tool align with industry standards, ensuring it is suitable for professional financial risk management and has been developed whilst adhering to to industry principles in regard to software engineering.

The ambition of this report is to present a thorough understanding of VaR, culminating in the development of a robust, adaptable, and industry-relevant tool for financial risk assessment. It aims to be acceptable as an industry standard tool.

2.3 Milestone Plan

Due to unfortunate circumstances, I had rough delays to the start of this Project as well as inconsistent health concerns, but that should not effect my overall final deliverable. In the first term, I want to research and create a working program to compute Value at Risk for small portfolios, that has a serviceable GUI that can be expanded on later. I will also make sure to have amply researched about back-testing and how to incorporate it into my program in some capacity. For the second term, I will research and implement applying Value at Risk for a portfolio of derivatives, as well as looking into using the Monte Carlo simulation and allowing for the computation of all this with as many stocks as necessary. I will finalise the GUI and plan to look into completing some of the extensions provided for the project, this will depend on the overall developmental scope of the project at the time, but these are the options I would be willing to explore:

- **Computing Conditional VaR (Expected Shortfall):** Beyond the standard VaR metric, exploring Conditional VaR could be considered to provide a more comprehensive risk assessment, as it accounts for the severity of losses in the tail of the distribution.
- **Parameter Selection for EWMA and GARCH(1,1) Models:** A detailed analysis of parameter selection for the Exponentially Weighted Moving Average and GARCH(1,1) models could be explored. The choice of parameters greatly influences model performance, and overall it would enhance the robustness of the risk assessment.
- **Empirical Study of Approaches to Historical Simulation for n-day VaR:** A comparative empirical study could be undertaken to examine different approaches to extending historical simulation from 1-day to n-day VaR, which can provide deeper insights into the performance of the methods.
- **Complementing Back-Testing by Stress Testing:** The robustness of the VaR model could be assessed not only through back-testing but also by applying stress testing techniques, allowing for the evaluation of the model's performance under extreme market conditions.
- **Computing Monetary Measures of Risk Different from VaR:** Other monetary risk metrics, which could complement or provide alternatives to VaR, such as Tail Value at Risk, could be considered to present a more complete picture of the risk landscape and enhance the risk management process.

The project's milestones have been structured to ensure a systematic and efficient approach to the development of the Value at Risk software. This plan is divided into distinct phases, each with specific objectives and deliverables.

1. **Initial Research and Planning (Completed):** This phase involved a thorough investigation into the concept of Value at Risk, its calculation methods, and the requirements for the software development.
2. **Software Design and Development (Ongoing):** Currently, the focus is on designing the software and developing the core functionalities of the VaR calculation tool. This includes building the initial graphical user interface and implementing various VaR models into it.
3. **Expanded GUI, Testing and Refinement:** After the development phase, there will be a complete redesign/refinement of the Graphical Interface, followed by rigorous testing that will be conducted to ensure the accuracy and reliability of the software. This phase will also involve refining the user interface and the overall user experience.
4. **Final Evaluation and Documentation:** The final phase involves a comprehensive evaluation of the software against the project's objectives and preparing detailed documentation as well as informational videos.

2.3.1 Timescale & Timeline

The entire project is structured over two academic terms, allowing ample time for each phase while ensuring a steady progression towards the project's completion. This timescale is designed to balance the initial development and subsequent refinement and testing phases effectively, ensuring that each aspect of the software is given due attention.

Concurrent with the development, ongoing documentation is a critical aspect of this project. Documenting the process as it unfolds serves multiple purposes: it ensures a clear record of the development process and aids in identifying and resolving issues more efficiently. This approach to documentation not only enhances the quality and maintainability of the software but also ensures that the project's progress is well-documented and aligns with the overall objectives and milestones.

Weeks 1–3

Project Research

- Research the fundamentals of Value at Risk (VaR)
- Research best coding language to use (Python)
- Familiarize myself with LaTeX and prepare IDE & Git for Project specified use

Week 4

Finalize Plan and Start Coding

- Complete Project Plan
- Continue researching VaR and Python
- Begin project coding

Week 5–7

Coding and Data Preparation

- Continue to work on the VaR program (No GUI)
- Start collecting and organizing sample data for small portfolios so it can be used by the program
- Finalizing understanding of the two computational methods needed, this being model-building and historical simulation

Week 8

Back-Testing Research & Implementation

- Investigate methods and techniques for VaR back-testing
- Start integrating back-testing into the project

Week 9–10

GUI Development

- Initiate the development of the GUI
- Ensure the GUI is robust for its current task as well as expandable for future enhancements

Week 11

Interim Report and Presentation Preparation

- Fine-tune programs and report so they are at a satisfactory level, will also allow for easier preparation for the interim presentation
- Prepare for the interim presentation

Weeks 1–2

Reflection and Research

- Spend time to reflect on the progress of the project so far, make any changes that I think are warranted after having the winter break time to think about
- Research the Monte Carlo simulation method for VaR, as well as how I could start implementing derivatives as portfolios into the project

Week 3–4

Start Implementing New Features

- Start implementing the Monte Carlo simulation method and continue derivative implementation

Week 5–7

GUI Finalisation

- Decide on the final visual product I want to represent with the GUI and start implementing it (if progress on this needs to continue into the next period, then it will be done so)
- Set the program up to work portably/allowing it to work on mobile OS's as well as different desktop OS's

Week 8–9

Extend Project Scope (if time permits)

- Explore additional features or enhancements for the project, possibly decided upon at the start of Term 2
- Implement as many as can be appropriately managed, with all additional time spent within this period being used to ensure the project is at its most refined state

Week 10–11

Perfect Final Report

- Make sure the program has been achieved to the best of its ability
- Finalise and perfect the final report

3 Chapter 2 — Understanding VaR

3.1 Historical Simulation

Historical Simulation is a method of estimating Value at Risk (VaR). It relies on historical market data to predict future risks, making it a straightforward yet powerful method for calculating VaR. The process of computing VaR through Historical Simulation involves several steps, as outlined below:

1. **Data Collection:** Historical price data of the asset is collected for a specified time period. This can be done using an API or through a CSV file.
2. **Calculate Percentage Differences:** The daily returns are calculated, this being done by comparing the closing price of the current day to the closing price of the previous day by dividing one by the other, the taking away 1. This is done for each day in the data set.
3. **Sort Returns:** The calculated returns are sorted in ascending order.
4. **Determine the VaR Threshold:** A percentile is chosen based on the confidence level (e.g. 95%). This is used to work out 100% - X percentile (e.g. 5th), this being the risk level of the portfolio and X being the confidence level. This is the VaR threshold.
5. **VaR Estimation:** The Value at Risk is estimated as the value at the chosen percentile. For example, if you have 500 days of data, the 5th percentile would be the 25th value in the sorted list of returns, then multiplied by the portfolio value, simulating the worst percentage decrease that your portfolio could encounter based on the last Z days of data.
6. **Correct Negative Number:** Since it takes the 5th percentile, it will be a negative percentage difference multiplied by the portfolio, since we need to represent VaR as a positive value of potential loss, it is multiplied by -1 to make it positive.

As you can see, it takes a very literal approach, assuming that, since you've got all this historical data, then it is likely that the future will be similar to the data of the past. It is an empirical method, meaning it is based on historical data, and as such is a simple and easy to understand, but it does have its limitations, such as the fact that it does not take into account any changes in the market, such as a financial crisis, which could have a huge impact on the market.

Refer to Figure 1, it expresses that:

- The bell-shaped curve represents the probability distribution of gains and losses for the asset or portfolio over the specified period.
- The left tail of the distribution marks the area of losses, where we can observe the VaR loss at a specific confidence level, say $(100 - X)\%$. This tail area signifies the worst losses incurred in the historical period.
- The point where the left tail cuts the horizontal axis (gain/loss over N days) corresponds to the VaR at the chosen confidence level. For example, if X is 95, then the VaR would represent the maximum expected loss over the N days period that only 5% of the time is expected to be exceeded.

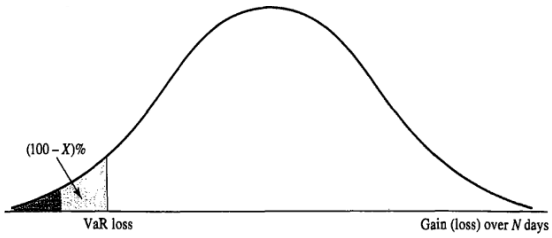

Figure 1: The Value at Risk (VaR) is calculated from the probability distribution of the variations in the portfolio's value, where gains are represented as positive values and losses as negative. The confidence level for this calculation is set at X%.[4]

- The graph assumes that the past can be a predictor of future risk, thus the frequency and magnitude of historical losses are directly used to estimate the VaR.

In the future, I could also use extreme value theory (EVT) to smooth our the distribution, which would allow for a more accurate estimation of the VaR, as it would allow for a better estimation of the tail of the distribution, which is where the worst losses are.[6]

3.2 Model Building (Variance-Covariance)

The Variance-Covariance method, used in this analysis for Value at Risk (VaR) calculation, is based on the assumption of normally distributed market returns. The key steps in the model are as follows:

1. **Data Collection:** Historical price data of the asset is collected for a specified time period once again. This can be done using an API or through a CSV file.
2. **Calculate Daily Returns:** Daily returns are computed as the percentage change in the adjusted closing prices of the stock.
3. **Statistical Analysis:** The mean and standard deviation of the daily returns are calculated to represent the average return and the volatility of the stock, respectively.
4. **VaR Calculation:** The Value at Risk is calculated using the formula:

$$VaR = -P \times (\text{norm.ppf}(rl, \text{mean}, \text{sD}) + 1) \tag{2}$$

   where:

   - P represents the total value of the portfolio.
   - norm.ppf is a function from the `scipy.stats` library that calculates the inverse of the cumulative distribution function (CDF) of the normal distribution, also known as the Percent Point Function (PPF).
   - rl is the risk level, which corresponds to the confidence level for VaR. For example, a 95% confidence level would use an rl of 0.05 (5% risk level).
   - mean and sD are the mean and standard deviation of the historical returns, respectively.

The Value at Risk (VaR) calculation in the Variance-Covariance method is executed using a specific equation that combines statistical measures with the portfolio value to estimate the potential loss over a specified time horizon. The equation is as follows:

1. The norm.ppf function takes the risk level rl, mean, and standard deviation of returns as inputs and outputs a Z-score. This score corresponds to the point on the normal distribution curve where the cumulative probability is equal to the risk level.
2. The equation then multiplies this score with the portfolio value P and subtracts from P. This represents the potential loss in value of the portfolio at the given confidence level.
3. The addition of 1 in the formula adjusts for the fact that the norm.ppf function returns a negative value for typical VaR confidence levels. This is because the function calculates the Z-score for the left tail of the distribution, while VaR is the loss at the right tail. The addition of 1 converts the negative value to a positive one.

Thus, the equation calculates the VaR as the maximum expected loss over a certain period, given normal market conditions and a specified confidence level. The Variance-Covariance method is a simple yet effective approach to VaR calculation, but it does have its limitations, such as the fact that it assumes that the returns are normally distributed, which is not always the case.

3.3 Coding VaR Methods

To validate that I could apply the knowledge I had gained from my research, I decided to code the two methods of VaR calculation that I had researched, this being the historical simulation and model building methods. I have decided to use Python (3.11.1) as my coding language, as it is a language that I am familiar with and has a rich ecosystem of libraries that I can use to help me with the project. I also decided to use command line to display the results, as it utilises pythons inbuilt print() command to easily display variables, allowing for a quick and easy way to display the results of the VaR calculations, as well as any tests run as well.

3.3.1 Command Line — Historical Simulation

I decided to get the stock data I wanted to use for this directly from Yahoo Finance's online website, since it would allow me to specify the days I wanted to use, as well as the stock I wanted to use. I decided to use Nike (NKE) as my stock, as it is a company that I am familiar with and I know has been around for a long time, so it would have a lot of historical data to use. I decided to use the last 500 days of data, as it would allow for a good amount of historical stock data coverage, but not too much that it would take a long time to run.

To start, I imported the libraries I would need, this being `numpy` and `csv`. I then created an empty array called `closes`, which would be used to store the closing prices of the stock. I then opened the CSV file that I had downloaded from Yahoo Finance, and used a `for` loop to iterate through each row of the CSV file, adding the closing price of the stock to the `closes` array.`diffs` array, this being stored in the 6th column of the CSV file.

```
closes = np.array([])
with open('NKE.csv', 'r') as file:
    reader = csv.reader(file)
    for row in reader:
        closes = np.append(closes, float(row[5]))
```

I then created an empty array called `diffs`, which would be used to store the percentage differences of the closing prices of the stock, followed by a `for` loop to iterate through each element of the `closes` array, calculating said percentage difference between the current element and the previous element, then adding it to the `diffs` array. Finally, I used the `np.percentile()` function to calculate the 5th percentile of the `diffs` array, which would be the VaR for the portfolio (this meaning I didn't need to manually order it to find the 25th worst case result, I can just use this useful function). I multiplied this by the portfolio value, which in this case was 100,000,000, to get the VaR for the portfolio and then multiplied this by -1, as VaR is always represented as a positive value, to get the final VaR value.

```
diffs = np.array([])
for i in range(1, len(closes)):
    diffs = np.append(diffs, (closes[i]/closes[i-1] - 1))
print("500 Day stock history, for 1 day time horizon, 95% certainty,
       Portfolio of 100,000,000, VaR for the 23/10/2023: £" +
       str(round(np.percentile(diffs, 5)*100000000*-1, 3)))
```

This results in this output, which appears to be a reasonable VaR result.

```
500 Day stock history, for 1 day time horizon, 95% certainty, Portfolio of 100,000,000, VaR for the 23/10/2023: £3330243.68
```

Figure 2: Historical Simulation VaR Result

For this project, I did not look up any online VaR results for this stock or any other further stock used, since I believe the methods were robust enough to be reliable. Since I also create multiple methods for calculating VaR, I was able to cross-reference them with each other, with them mostly showing similar results for the same stock, so I am confident that these are accurate VaR predictions.

3.3.2 Command Line – Model Building (Variance-Covariance)

To begin with this method, I discovered that I could import the historical stock price data conveniently by using the yfinance package, which provides a convenient way to download financial market data from Yahoo Finance directly into Python. I selected Nike (NKE) as the target stock so I could compare it to the previous result given by Historical Simulation. A time span of the last 500 trading days was once again chosen to balance between a sufficient data sample size and computational efficiency, as well as for comparative reasons.

The first step was to import the necessary libraries and download the stock data using the `yf.download()` function, one of these additional libraries being scipy, which is a used for scientific and technical computing, due to it allowing the user to manipulate and visualize data with a wide range of high-level commands, one of such being the `norm.ppf()` function, which is used to calculate the inverse of the cumulative distribution function (CDF) of the normal distribution, also known as the Percent Point Function (PPF) that I will need to utilise in a future step. I then discovered that you can use the inbuilt `pct_change()` function to calculate the daily returns/percentage changes for the whole data set, which is a much more efficient way of doing it than the method I used for Historical Simulation. I also used the `dropna()` function to remove any missing values from the data set, in case using the API rather then the CSV could cause this issue.

```
from scipy.stats import norm
import yfinance as yf

stock = yf.download('NKE', dt.datetime(2021, 10, 26), dt.datetime(2023, 10, 24))
closeDiffs = stock['Adj Close'].pct_change()
```

I then proceeded to define the necessary variables to calculate the VaR, following the formula outlined in Equation 2. The portfolio and risk level were set to 100,000,000 and 0.05 (5%) respectively, since these were the value's used in Historical Simulation, then the mean was calculated using the `np.mean` function, followed by the standard deviation using the `np.std` function.

```
portfolio = 100000000
rlPercent = 0.05
mean = np.mean(closeDiffs)
sD = np.std(closeDiffs)
```

Finally, calculating the VaR result takes place with the negative portfolio value multiplied by the `norm.ppf()` function, which takes the risk level, mean, and standard deviation of returns as inputs and outputs a Z-score, which corresponds to the point on the normal distribution curve where the cumulative probability is equal to the risk level, giving us the VaR estimation.

```
print("500 Day stock history, for 1 day time horizon, 95% certainty,
       Portfolio of 100,000,000, VaR for the 23/10/2023: £" +
       str(round(-portfolio*(norm.ppf(rlPercent, mean, sD)), 3)))
```

The output of this computation provides a VaR estimate under the assumption of normally distributed returns and a linear correlation between the assets. This assumption is a of course a simplification and might not hold during periods of financial turmoil, which I acknowledge as a limitation of the model, but for this data it is acceptably accurate.

```
500 Day stock history, for 1 day time horizon, 95% certainty, Portfolio of 100,000,000, VaR for the 23/10/2023: £3640086.568
```

Figure 3: Variance-Covariance VaR Result

As you can see, the VaR result is very similar to the one calculated using Historical Simulation, which is a good sign that both methods are accurate. But how do I check the validity of both methods/models?

3.4 Back-Testing

So for the methods that I had already coded, I had achieved VaR estimations, but I had no way of validating how accurate they were, which is where back-testing comes in. Back-testing is a way of testing the accuracy of a model by using historical data to see how well it would have predicted the actual results. For example, if I had a model that predicted the weather, I could use back-testing to see how accurate it was by using historical weather data to see how well it has predicted the weather correctly in the past

To achieve this, you need to see how many times you model has previously predicted the correct VaR, or more so, how many times they have predicted the VaR wrong. To do this, lets say you have 500 days of historical data, and you want to see how many times your model has predicted the VaR wrong for a 1 day time horizon, 95% certainty, and a portfolio of 100,000,000. You would first need to take a portion of the data, lets say the first 50 days, and calculate the VaR for it using the model. You would then compare this VaR result to the actual value that the portfolio lost/gained from the 50th day to the 51st day, thus seeing if the model predicted the VaR correctly. This would then continue, keeping the 50 day data span, but moving it along by 1 day each time, so the next data span would be from the 2nd day to the 51st day, then the 3rd day to the 52nd day, and so on, until you reach the end of the data set. This can be seen in Figure 4 below.

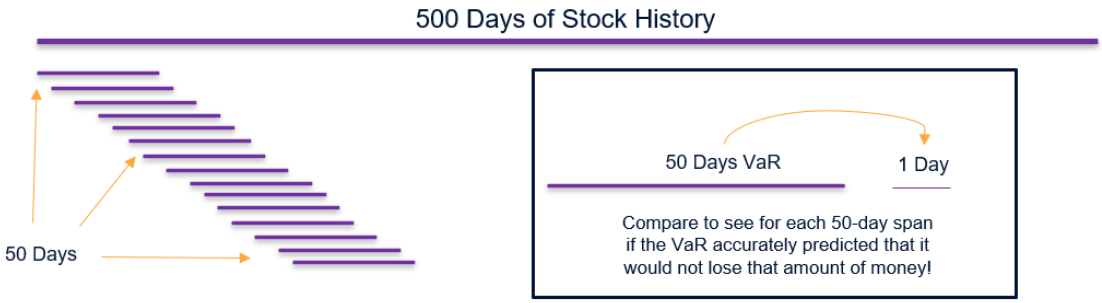

Figure 4: Back-Testing Diagram (Source: My Presentation Powerpoint)

Every time one of these spans creates a VaR that fails to accurately predict the next days loss, you would make note of it and add it to a counting total. This total can then be utilised with p-values. A p-value is a statistical measure that helps to determine the significance of the results obtained from a hypothesis test. In the context of back-testing, the null hypothesis typically states that the model's predictions are correct, and the alternative hypothesis suggests that the model's predictions are not accurate. A low p-value indicates that the observed data is unlikely under the null hypothesis, leading to its rejection in favour of the alternative hypothesis. So for our VaR back-testing, a low p-value would suggest that the model's performance is not as good as predicted, potentially failing to capture the risk adequately.

To do this within Python whilst leading on from our previous two coded methods, I added the following codee to the end on both models. It creates a variable called `count`, which is used to count the number of times the model has predicted the VaR incorrectly, and a variable called `adjust`, which is used to adjust the data span to be used for the VaR calculation, this being 10% of the data set. It then creates a `for` loop, which iterates through each day of the data set, calculating the VaR for the data span, then comparing it to the actual value that the portfolio lost/gained from the last day of the data span to the next day, thus seeing if the model predicted the VaR correctly. This will repeat for the length of the entire historical data history, minus the span adjust and minus 1 as well, since you would need to compare 449–499 day VaR to day 449–500's return. The next day return is also multiplied by -1, so it accounts so that the losses are positive like the VaR estimations, making it easy to compare if one is incorrectly larger then the other. If the VaR was predicted incorrectly, the `count` variable is increased by 1.

```
count = 0
adjust = int(len(stock)/10)
for i in range(1, len(stock) - adjust - 1):
    backTest = stock['Adj Close'].pct_change()[i:i+adjust]
    if Historical Simulation:
        VaR = np.percentile(backTest, rlPercent)*portfolio*-1
    else: #Model Building
        VaR = -portfolio*(norm.ppf(rlPercent, mean, sD))
    nextDay = stock['Adj Close'].pct_change()[i+adjust:i+adjust+1].values[0]
    nextDay = nextDay*portfolio*-1
    if nextDay > VaR:
        count += 1
```

Next, I compute the p-value using the cumulative distribution function (CDF) of the binomial distribution, which in turn calculates the probability of observing a certain number of VaR exceedances (or fewer) given the expected number of exceedances under the model's assumptions.

```
pValue = binom.cdf((len(stock)-adjust)-count,len(stock)-adjust,1-rlPercent/100)
```

The parameters for the `binom.cdf` function are as follows:

- The number of successes, which is the total number of days minus the adjustment for the data window and the count of exceedances.
- The number of trials, which is the length of the stock data minus the adjustment for the data window.
- The probability of success on each trial, which is 1 minus the risk level percentage divided by 100, reflecting the confidence level of the VaR estimate.

The calculated p-value is then compared to the risk level percentage to determine the statistical significance. If the p-value is greater than the risk level percentage, the model passes the back-test, indicating that the number of VaR exceedances is within acceptable limits of the model's predictions. Conversely, a p-value lower than the risk level percentage suggests that the model's poor predictive performance is statistical significance, and it fails the back-test.[11]

```
Back Test: PASSED with 10.0% statistical significance level (p-value)
Back Test: PASSED with 19.0% statistical significance level (p-value)
```

Figure 5: Results using Tesco stock data, for Historical Simulation followed by Model Building.

```
if pValue > rlPercent/100:
    print("Back Test: PASSED with " + str(round(pValue*100, 0)) +
    "% statistical significance level (p-value)")
else:
    print("Back Test: FAILED with " + str(round(pValue*100, 0)) +
    "% statistical significance level (p-value)")
```

By utilizing p-values, we can assign a level of confidence to our back-testing results, supporting the validation process of our VaR model. It enables us to make informed decisions about the model's reliability and whether it can be trusted for making future risk assessments. It help provide a quantitative method to validate the accuracy of VaR models, ensuring that risk managers and financial analysts can rely on the model's predictions to make critical decisions. In the future, along a set of portfolio's with multiple stocks within, I could perform multiple back test on multiple stocks using both VaR, and form a statistical conclusion on how many time each model fails the p-value back-testing, thus seeing which model is more reliable and should possibly be used in further graphical builds, although I have not touched on Monte Carlo simulation yet, which could be far more accurate then either of these two methods, so I will have to see how my research into it goes.[11]

3.5 Combining Implementations

Before I decided to embark on the creation of any of my graphical elements, I thought it would be good to have some way to test my VaR models on different single stocks, so I could see a range of results, and better gauge the consistency levels of my back-tests, as well as also being able to simple see more VaR results then just what I had seen so far. I created a fully functioning command line program that allows the user to select a stock from the FTSE100 list (I chose this as I do not have a wide knowledge of stock lists, but since I knew of this one already and all the stocks on it are significant, it would be good to implement), enter a given portfolio value that they would have invested within this one particular stock, and then select a time horizon and confidence level for the VaR calculation. It would then give them 3 options for how much historical data they would want to use, either 100, 500 or they could input their own dates, and it would finally ask what model they would like to compute the VaR estimation with, either Historical Simulation or Model Building. It would compute the VaR and then display the result, as well as display the p-value back-testing result.

Since the code for this is just a large and robust amount of verification and input statements, I will not include it in this report, but I will display a full interaction in figure 6. I used pandas to retrieve the FTSE100 from Wikipedia, pandas being a powerful data manipulation library in Python, allowing for web scraping, reading and writing data, which I utilised it for in this case. Everything else included within the code has been covered within previous pages within this chapter. Full interaction found on the next page:

```
WELCOME TO THE VAR CALCULATOR
-----------------------------
Which companies stock would you like to calculate the VaR for?
1 - 3i
2 - Admiral Group
3 - Airtel Africa
...
40 - Haleon
41 - Halma plc
42 - Hargreaves Lansdown
43 - Hikma Pharmaceuticals
44 - Howdens Joinery
...
97 - Vodafone Group
98 - Weir Group
99 - Whitbread
100 - WPP plc
Enter the number of the company: 42
Enter the portfolio value (£): 100000000
Enter the risk level percentage (1%, 5%, etc.): 5
Enter the time horizon (Max: 100 days): 1
How many days of historical data would you like to use,
or would you like to choose your own date boundaries?
1. 100 days
2. 500 days
3. Choose your own
Enter the number of your choice: 2
[**********************100%%**********************]  1 of 1 completed
Would you like to calculate VaR using Historical Simulation,
or using Model Building/Variance-Covariance (H/M): m

VaR is: £5,818,845.68
Back Test: PASSED with 32.0% statistical significance level (p-value)
```

Figure 6: Console output of the Single Stock VaR.py program.

Additionally, I wanted to mention how Time Horizon in handled within this program. The best way to show how it is handled would be to quote from reference [6] — *Options, Futures, and Other Derivatives.* by John C. Hull, which states:

> VaR has two parameters: the time horizon N, measured in days, and the confidence level X. In practice, analysts almost invariably set $N = 1$ in the first instance. This is because there is not enough data to estimate directly the behavior of market variables over periods of time longer than 1 day. The usual assumption is:
>
> $$N\text{-day VaR} = \text{1-day VaR} \times \sqrt{N} \tag{3}$$
>
> This formula is exactly true when the changes in the value of the portfolio on successive days have independent identical normal distributions with mean zero. In other cases, it is an approximation.

Since VaR is scaled with time due to volatility. Volatility tends to be proportional to the square root of time.

$$\text{Volatility(Time)} = \text{Volatility(One Period)} \times \sqrt{\text{Time Horizon}} \tag{4}$$

Since we are dealing with the other cases frequently throughout the use of the VaR Models used within the programs, I utilise the formula 3 above to accommodate for the VaR value over a given time period, by multiplying the VaR by the square root of the given time horizon, which is why the user is asked to input the time horizon in days, and not in years or months, this providing sufficient estimations so far.

4 Chapter 3 — Graphical User Interface

4.1 What is Kivy?

Kivy (https://kivy.org/) is an open-source Python library for developing multitouch application software with natural user interface (NUI) integration. It is fantastically versatile and can run on various operating systems, including Windows, macOS, Linux, Android, and iOS. This cross-platform compatibility is due to Kivy's use dynamic use of OpenGL ES 2, allowing it to render consistent graphics across all supported platforms.[9] Kivy is also free to use, even for commercial purposes, as it is distributed under the MIT license.

Because of this, Kivy's compatibility with multiple operating systems makes it an excellent choice for financial software development. The ability to write code once and deploy it on various platforms without modification is particularly advantageous in a financial context, where users may access software from different devices.

- **Cross-Platform:** Kivy apps can run on desktop and mobile devices, enhancing accessibility for users who need to monitor financial markets or run VaR calculations on the go.
- **Pythonic Nature:** Given that Python is a leading language in financial modelling due to its simplicity and the powerful data analysis libraries available, Kivy integrates well within the Python ecosystem, as well as it being the chosen language for this project.
- **Graphics Engine:** Kivy's graphics engine is built over OpenGL ES 2, providing the capability to handle intensive graphical representations, which is essential for visualizing complex financial data that I will want to do in the future.

Developing financial software with Kivy brings several benefits:

- **Rapid Development:** Kivy's straightforward syntax and powerful widgets allow for rapid development of prototypes and software, which is crucial in the fast-paced environment of financial markets.
- **Multitouch Support:** The library's inherent support for multitouch can be leveraged to create interactive financial dashboards, enhancing data exploration and manipulation for important tasks such as risk analysis which we will want to use it for.[9]
- **Customizable UI:** Kivy provides the tools to create a highly customizable user interface (UI), enabling the development of aesthetic financial applications tailored to the specific needs of financial analysts.
- **Community and Support:** Kivy has a growing community and good documentation, which will be highly useful for me as a beginner to the library.

Kivy presents itself as a robust framework for financial software development. Its compatibility with different operating systems, ease of operation within Python, and the capability to create sophisticated UIs make it an appealing choice for developers such as myself to use when creating financial applications, especially when catered for VaR analysis and other risk management tools, which is why I've chosen to use it for this project for the GUI.

4.2 Conceptualisation of Initial Design Ideas

The initial design phase of the Value at Risk (VaR) calculation tool will focus on creating a tool that will allow the user to interact with it in an efficient and effective manor, with less of an emphasis on a visually pleasing design, more on practicality for what has been researched and commented on so far. I had an idea in mind for what I wanted to initially be able to create, so I decided to create a design sketch (Figure 7), which reflects an early conceptualization of the interface, emphasizing simplicity and functionality.

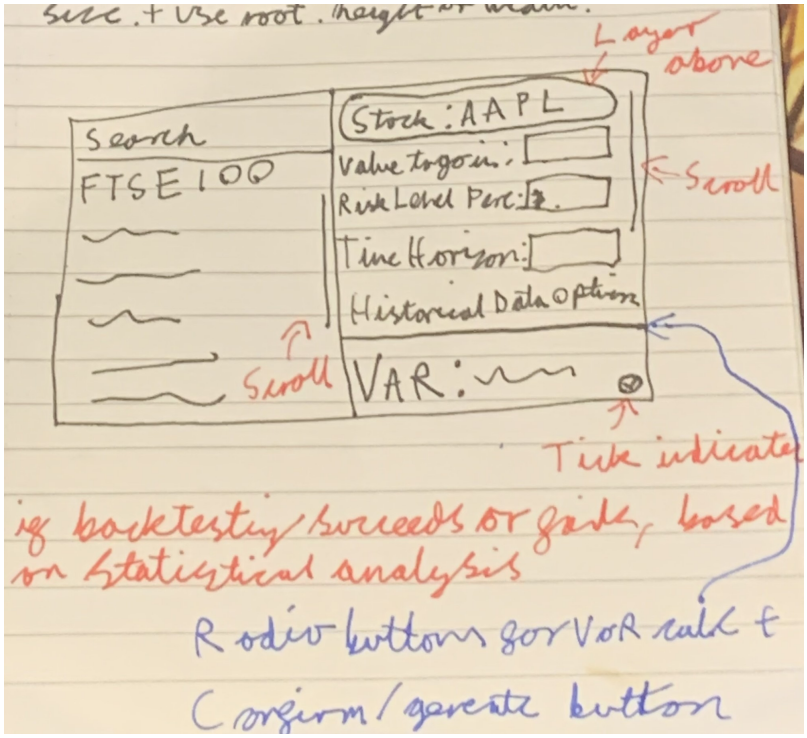

Figure 7: Initial GUI Design Sketch

The layout was envisioned to comprise of two main sections: a search and selection panel on the left and an input/detail panel on the right. The search panel allows users to filter and select stocks from the FTSE100 to then be used on the right, as well as search for any stock that they want from an autocomplete database. The detail panel on the right is structured to enable users to specify parameters for the selected stock, such as portfolio value, risk level percentile, time horizon and what VaR model you want to use. A scroll feature could also be implemented to accommodate future desired adjustable parameters, such as historical data options. Below the scroll is a fixed section that will display the final VaR after the rest of the informational parameters have been interacted with/data has been submitted into them. There will also be a tick to indicate if back-testing's hypothesis through statistical analysis had been rejected, this influencing if this model with this stock passes.

A key feature highlighted in the sketch is the inclusion of the layered "current stock" display in the top right, as it lets the users easily see what they have selected, whilst also letting the scroll move underneath it, helping the program look and act dynamically. Additionally, a tick indicator was conceptualized to provide immediate visual feedback when back-testing succeeds or fails, based on statistical analysis (p-values), enhancing the user's understanding of the model's performance, showing evidence of system status. The design also incorporates radio buttons for VaR calculation modes for historical and model-based approaches, allowing for quick toggling between different methods whilst still maintaining the other pre-selected parameters.

This initial design was guided by principles of user centered design (UCD) best practices, aiming to streamline the complex process of risk analysis into a manageable and approachable workflow for end-users. Each element was chosen for its potential to make the software accessible to both novice and experienced users, allowing for ease of user regardless of skill level.

The conceptualisation phase was pivotal in laying the groundwork for the subsequent development of this GUI. It helped me visualise how it needed to look and how the interactions would have to take place, so even if the final result wasn't visually the same, it still provided an excellent user story for how it needed to perform.

4.3 Initial GUI Creation

Before my initial creation, I briefly followed the book [9], *Kivy - Interactive Applications and Games in Python - Second Edition* by Roberto Ulloa, which I have referenced in my bibliography, as it's a very useful book for learning the basics of Kivy, and I would highly recommend it to anyone who wants to learn the library.

4.3.1 First Iteration

Since the book taught me to separate the GUI into two files, I have done so for my baseline of the application, these files being the .py file, which contains the code for the GUI, and the other being the .kv file, which contains the layout and styling of the GUI, I will be explaining the code for both of these files, as well as the code for the main.py file, which is the file that runs the GUI.

The Python script 'Initial Design Test.py' serves as the backbone of the application. Utilizing Kivy's standard libraries, the script defines my ApplicationView class, which inherits from Kivy's BoxLayout to establish the fundamental structure of the GUI:

```
class ApplicationView(BoxLayout):
    stockList = ObjectProperty(None)
    userInputs = ObjectProperty(None)
    ...
    def populateList(self):
        ...
    def populateInputs(self):
        ...
```

The 'ApplicationView' class is integral to the GUI, as it initializes the user interface and binds the graphical components to the backend logic. The ObjectProperty instances 'stockList' and

'userInputs' are object placeholders for dynamic python elements within the GUI. The methods 'populateList' and 'populateInputs' are responsible for filling these placeholders with actual content — in this case, labels representing stock data and text input fields for user parameters, respectively.

The layout and styling of the GUI are defined in the Kivy language file 'IDT.kv', which allows for a clear separation of the interface design from the logic of the application:

```
<ApplicationView>:
    orientation: 'horizontal'
    BoxLayout:
        orientation: 'vertical'
        ...
        ScrollView:
            ...
```

This .kv file outlines two main sections in a horizontal arrangement — a searchable stock list and a detailed input area for configuring VaR parameters. The use of ScrollView elements ensures that the content is accessible even when it exceeds the screen space. This design decision was made to enhance the application's scalability and to provide a seamless user experience.

The 'populateList' and 'populateInputs' functions demonstrate the dynamic nature of the interface. They are called upon initialization to populate the GUI with interactive elements:

```
def populateList(self):
    for i in range(100):
        self.stockList.add_widget(Label(...))

def populateInputs(self):
    for i in range(20):
        self.userInputs.add_widget(Label(...))
        self.userInputs.add_widget(TextInput(...))
```

These functions are responsible for creating the labels, along with the text input fields that are displayed in the GUI. These labels are populated with stock data, and the text input fields will be used to configure VaR parameters. This approach allows for the creation of a scalable interface that can be easily adapted to accommodate additional stocks and parameters in the future, once they are implemented.

Upon executing the 'Initial Design Test.py' script, the Kivy application builds the GUI based on the defined classes and .kv file, initiated by the following code:

```
if __name__ == '__main__':
    IDTApp().run()
```

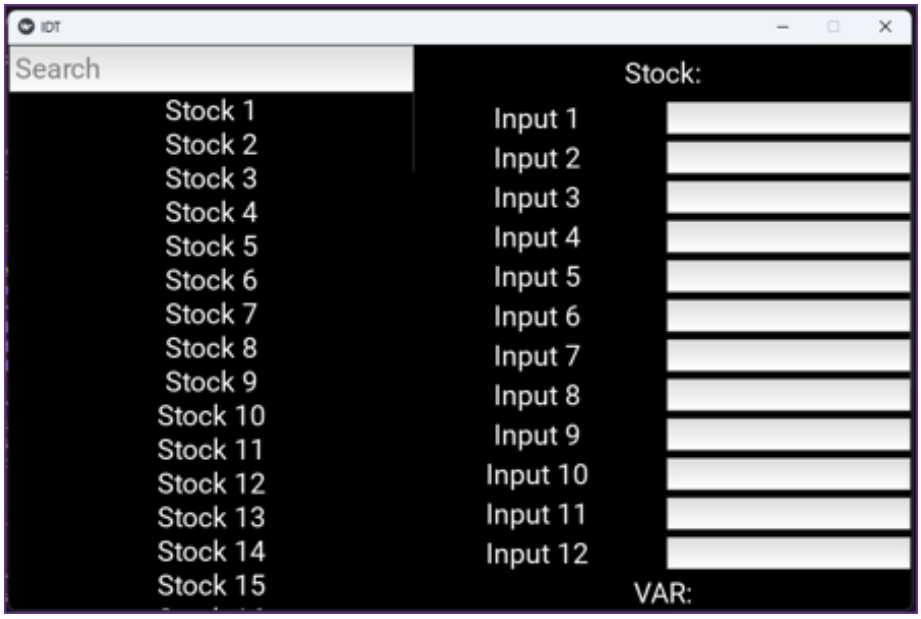

Figure 8: Initial Design Test - First Iteration

4.4 Completed Initial Design

For my final iteration of my first design, I took all of the elements that were explored within the Combining Implementations chapter, and implemented them into the GUI, as well as adapting some of the old features so they could work in the boundary of what I was able to achieve.

4.4.1 Enhanced Application Structure

The `ApplicationView` class was expanded expanded to include more properties and functionalities that can be used throughout the program to enhance its functional data analysis capabilities:

```
class ApplicationView(BoxLayout):
    portfolio = 100000000
    rlPercent = 5
    timeHori = 1
    simMethod = "Historical"
    currentTicker = ""
    ...
```

Above I initialise the portfolio value, risk level percentage, time horizon, simulation method, and current stock ticker as class attributes, providing default values for these parameters. This allows for easy access and modification of these values throughout the application, ensuring consistency and flexibility as I adapt them to meet the needs of whatever section they are needed within.

4.4.2 Dynamic Stock List Generation

The method `populateList` dynamically generates a list of stocks from the FTSE 100 index, allowing users to select a company for analysis:

```
def populateList(self):
    for i in range(len(ftse100)):
        button = Button(text=ftse100['Company'][i], ...)
        button.bind(on_release=lambda btn, i=i: ...)
        self.stockList.add_widget(button)
```

This function iterates over the FTSE 100 stock list, originally stored within my application by parsing the FTSE100 wikipedia link in through pandas, and taking the results from the 4th index, a table containing the company name and subsequent ticker information for all stocks. The code will then create a button for each company that, when clicked, updates the current stock ticker and displays it in the GUI. The lambda function is used to pass the index of the selected stock to the button's event handler, ensuring that the correct stock is selected when the button is clicked.

4.4.3 Value at Risk (VaR) Calculation

The `generateVaR` method calculates the Value at Risk for the selected stock, using either historical data or a model simulation approach, both of which have been described within previous sections:

```
def generateVaR(self):
    ...
    stock = yf.download(self.currentTicker, startDate, endDate).tail(500)
    ...
    if self.simMethod == "Historical":
        VaR = str('{:,}'.format(int(round(np.percentile(closeDiffs, self.rlPercent)
                                    *self.portfolio*-1*np.sqrt(self.timeHori), 0))))
    else:
        VaR = str('{:,}'.format(int(round((-self.portfolio*norm.ppf(self.rlPercent/100,
                np.mean(closeDiffs), np.std(closeDiffs)))*np.sqrt(self.timeHori), 0))))
    ...
    setattr(self.valAtRisk, 'text', " VaR: £" + VaR)
```

This function retrieves historical stock price data for the selected stock(start and end date around 1000 days apart, `.tail` only retrieving the last 500 non-nan days), calculates the VaR using the chosen method above that match their command line implementations, and updates the GUI with the calculated VaR value. The VaR value is displayed in the `valAtRisk` label, providing users with immediate feedback on the risk level (as soon as the data for the simulation is downloaded from yfinance) associated with their hypothetical investment.

4.4.4 User Input Handling and Validation

The application includes mechanisms to handle and validate user inputs, ensuring that the analysis is based on accurate and realistic parameters:

```
def validateInput(self, current, varName, maxVal):
    try:
        if current.text == "" or int(current.text) < 0:
            raise Exception
        if int(current.text) > maxVal:
            setattr(self, varName, maxVal)
        else:
            setattr(self, varName, int(current.text))
    except:
        if varName == 'portfolio':
            setattr(self, varName, 100000000)
        elif varName == 'rlPercent':
            setattr(self, varName, 5)
        else:
            setattr(self, varName, 1)
    self.populateInputs()
```

The `validateInput` function takes in the current text input field, the variable name, and the maximum value allowed for that parameter. It validates the input by checking if the text field is empty or contains a negative value, also accounting for if the input exceeds the maximum value. If the input is invalid, the function sets the parameter to a default value, and then repopulates the input fields to reflect the updated values. This ensures that no invalid data is used in the VaR calculation, maintaining the integrity of the analysis.

4.4.5 Radio Buttons

The application includes radio buttons for selecting the simulation method, allowing users to switch between historical and model-based approaches:

```
hButton = ToggleButton(text='Historical', group='simMethod', size_hint_x=None, width="100sp")
hButton.bind(on_press=self.simMethodPressed)
if self.simMethod == 'Historical':
    hButton.state = 'down'
else:
    hButton.state = 'normal'
radioButtons.add_widget(hButton)

mButton = ToggleButton(text='Model', group='simMethod', size_hint_x=None, width="100sp")
mButton.bind(on_press=self.simMethodPressed)
if self.simMethod == 'Model':
    mButton.state = 'down'
```

```
else:
    mButton.state = 'normal'
radioButtons.add_widget(mButton)
```

The radio buttons are created using Kivy's ToggleButton widget, allowing users to select the simulation method by toggling between the historical and model-based options. The buttons are bound to the `simMethodPressed` function, which updates the simulation method based on the user's selection. The state of the buttons is set based on the current simulation method, ensuring that the correct button is highlighted when the application is launched, and ensuring that only one button will displayed that they are selected at any given time.

4.5 Initial Deliverable with Pictures

The final iteration of the initial design successfully integrates the core functionalities of the VaR calculation tool into a user-friendly interface. The application allows users to select an available stock, configure VaR parameters, and calculate the VaR using either historical data or a model-based approach. The dynamic stock list generation, user input handling, and radio buttons enhance the application's usability and flexibility, providing any users with a sufficient tool for risk analysis. The accompanying `.kv` file provides the structural backbone behind the visual appearance, but it is less complex than the `.py` file, more on the visual alteration of kivy can be found in later section. For what it was created to accomplish, I believe this final initial design is aesthetically pleasing, computationally sound and accomplishes exactly what is required of it in a robust and efficient manner. Pictures of this deliverable can be found on the subsequent page in figure 9 and figure 10.
Youtube Video: Presentation Demonstration For Final Initial Design - FYP

4.6 Reflectional Adjustments

When completing the development for the rest of my program, there were times where I would have to refer back to this initial design. When doing so, I realised a few mistakes that I had overlooked. I will briefly detail them as to affirm that I have acknowledged and improved upon some of them within this time-frame.

To begin with, I discover that whilst my code would attempt to download a stock, and upon this failing, it would try again with a ".L" denoting London Stock Exchange, which worked for the vast majority of the stocks, with the few exceptions displaying an error. This was weird, since when I went to the Yahoo Finance website, I could see that the stock was readily available and updating, but I assumed it was to do with the API having a specific issue. Then came a time where I was retrieving stock names, and realised that the names I was generating did not match with many of the FTSE100 names I had displayed within this section. After doing some more experimentation, I discovered that in fact every single one of these stocks were on the London Stock Exchange (which in hindsight is obvious due to the FTSE100 being a London based and ran list), so they would all need to be appended with ".L" to be able to be downloaded. Upon running this through the program, even the stocks that had previously caused errors now retrieved information perfectly.

I realised that the few times that stock errors did occur, it was due to the stocks ticker being an

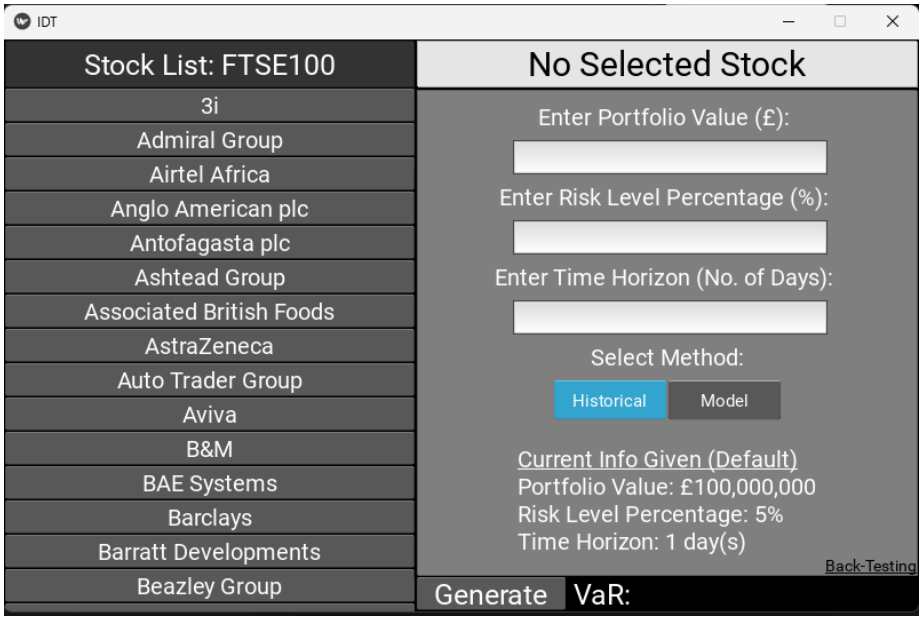

Figure 9: Initial Design Test - Final Iteration 1

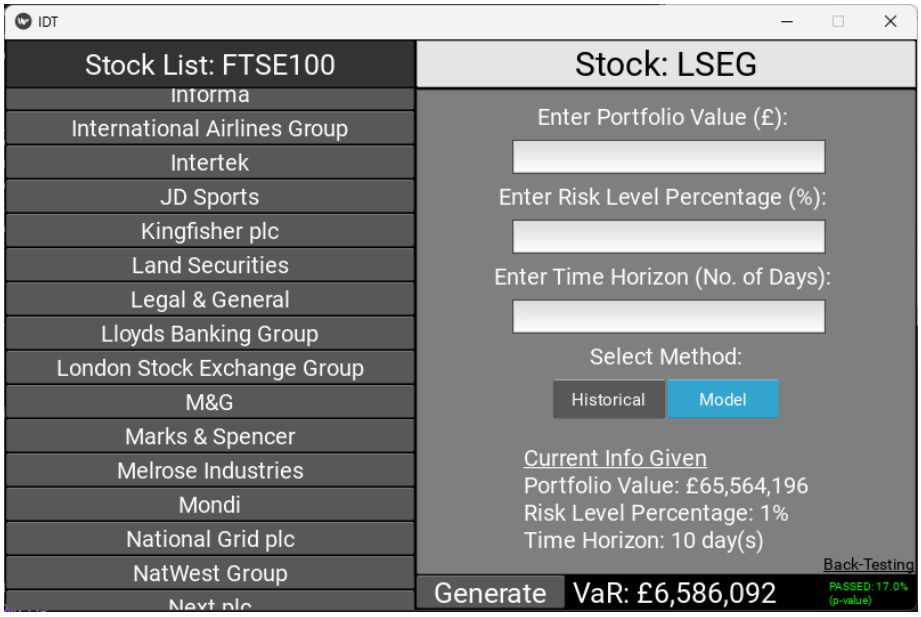

Figure 10: Initial Design Test - Final Iteration 2

applicable non-denoted ticker pointing to a company that had no tracked financial information, resulting in a null stock being downloaded, rather then the stock download failing and retying again with the .L denotation. Because of this, a lot of previous VaR results I was getting were from American companies with direct non-denoted tickers. To fix this, I removed the need for verifying the stock downloads, since all of the stocks would successfully work, as I just needed to concatenate the previous list of tickers with ".L" to get the correct stock information every time.

A different huge problem was that I had laid out and specifically positioned the app to work with its current size, thus me restricting the scaleability of the window to not be possible. However, when you moved the application from one screen size (that I had been doing the majority of the visual creation work with) to another of different dimensions, it would completely mess up the layouts and make parts of the visuals illegible, even with both monitors being the same resolution. This was extremely problematic, since it would entirely restrict who could use my program effectively, since they would have to be lucky enough to have the same size screen as my monitor just to be able to use it properly. Once again, when searching through the Kivy documentation [13], not only did I figure out that `size_hint`, a property I had tried to avoid due to not being able to specify specific sizes, could be used to scale for whatever sized window you may have (based on percentages), but there was also a built-in unit of measurement called `sp` (Scale-independent Pixels) that could be used to specify sizes in such a way that they would automatically scale depending on screen type, which I had not known about before. Going back through and changing all my specifically defined values, I was able to ensure that the application would be accessible on all types of windows hardware with this change, keeping the view consistent and optimised, and allowing my future development to adhere to the same practice as well.

One other final thing to note, I have observed in hindsight a weird reliance on me changing and writing a vast majority of Kivy information inside of my python code, and I'm not referring to the general python integrated code, I was regularly defining layout aspects within all parts of my python code, something that is not common practice for this type of development. I believe this to be due to my lack of understanding of the .kv file at the time as a whole, since this was the first time I had programmed in Kivy. COntinuing on from this point, I adapted the framework and the workstation that my project undertook, allowing me to better make use of the .kv files for its fantastic layout and styling capabilities, and its ability to seamlessly communicate with its respective python file. I did not go back and alter the contents of my python and kivy files in this regard, since its functionality remained working optimally, although I completely acknowledge that it was a suboptimal way to program within the kivy and python combined framework. Evidence on my growth and aptitude in this regard can be seen through the exploration of my future GUI developmental sections.

5 Chapter 4 — Portfolio Creation

5.1 Monte Carlo Simulation

Monte Carlo Simulation is arguably the most important computational technique I will need for calculating Value at Risk, since it can be utilised to perform repeated random sampling to estimate complex mathematical or physical models across a series of values. It's extremely useful in finance for the valuation of instruments, portfolios, and investments under uncertainty, since it simulates a

range of possible outcomes for these random processes and proceeds to average the results to find a probabilistic estimation of the what the actual outcome would be.

For these Value at Risk (VaR) calculations, Monte Carlo Simulation allows for an estimation of the potential loss in value of a whole portfolio (different to the single stocks I've been dealing with using previous methods) with any given confidence level over the specified period. It does this by simulating the returns of the portfolio across numerous simulated scenarios to create a distribution of possible outcomes, and since VaR can then be derived as a percentile of this randomly estimated distribution, such as the 5th percentile, it helps indicate that there is a 95% confidence level that losses will not exceed this value, or an equivalently based 5% risk level that the losses will exceed.

It helps model the probability of different outcomes in a process that cannot easily be predicted due to the intervention of random variables, such as with multiple stocks within a stock market. The method involves generating a large number of random portfolio return paths based on historical return distributions and then determining the VaR from these simulated paths. The general understandings of the explanations below are taken respect from reference [16], although I have adapted the equations to better understand and increase comprehensibility, due to how they are needed within the context I utilise them for, me personally preferring my proposed format.

Given a portfolio consisting of n assets, let $r = (r_1, r_2, \ldots, r_n)$ denote the vector of daily returns of these assets. The return of the portfolio on any given day can be expressed as a weighted sum of the individual asset returns:

$$R = w \cdot r^T \tag{5}$$

where:

- R is the portfolio return.
- $w = (w_1, w_2, \ldots, w_n)$ is the vector of portfolio weights corresponding to each asset, stocks in our case.
- r^T is the transpose of the return vector.

When using Monte Carlo Simulation for VaR calculation, we simulate this return R for a given time horizon using the historical mean (μ) and covariance (Σ) of the asset returns. For a single-day horizon, the simulated return for the portfolio can be generated from a multivariate normal distribution:

$$\tilde{r} \sim \mathcal{N}(\mu, \Sigma) \tag{6}$$

where:

- $\tilde{r}$ represents a vector of simulated daily returns for the portfolio assets.
- The process is repeated N times to generate a distribution of simulated portfolio returns.

This corresponds, based on the amount of repeated simulations N and subsequent returns $\tilde{R}$, to the following:

$$\{\tilde{R}_1, \tilde{R}_2, \ldots, \tilde{R}_N\} = \{w \cdot \tilde{r}_1^T, w \cdot \tilde{r}_2^T, \ldots, w \cdot \tilde{r}_N^T\} \tag{7}$$

The VaR at a confidence level (e.g. 95%) or equivalent risk level α (e.g. 5%) is then determined by finding the (α)th-percentile of this simulated portfolio returns distribution:

$$VaR_\alpha = -\min\{\tilde{R} : P(R \leq \tilde{R}) \geq \alpha\} \tag{8}$$

where:

- VaR_α is the Value at Risk at the risk level α. This represents the loss that is not exceeded with probability α, indicating a worst-case scenario within the bounds of the specified risk level.
- $\tilde{R}$ represents a possible return.
- R is the random variable representing portfolio returns.
- $P(R \leq \tilde{R})$ is the probability that the portfolio return is less than or equal to $\tilde{R}$.
- α is the risk level, indicating the portion of the distribution under consideration for VaR. For example, a confidence level of 95%, would result in a risk level α of 0.05 (5%), reflecting the lower 5% of the return distribution where the losses lie.

In theory, when the simulated returns are sorted in ascending order, the (α)-quantile is directly computed as the value at the $N(\alpha)$-th position in the sorted list, where N is the total number of simulations. This quantile represents the maximum expected loss at the risk level α, representing the worst-case scenario within the specified risk bounds over the given time horizon. For a given portfolio value P, the monetary value of the VaR can be calculated as:

$$VaR_{\text{Monetary}} = P \times VaR_\alpha \tag{9}$$

This given mathematical framework helps underpin the Monte Carlo Simulation approach to VaR calculation, providing a probabilistic estimate of potential portfolio losses over a specified time horizon based on historical asset performance. This is what I will need to adapt to python in my subsequent program to be able to estimate VaR calculations within a portfolio of stocks.

5.1.1 Command Line — Monte Carlo Simulation

Continuing on from my proof of concept programs for Historical and Method simulation, I used Yahoo Finance package again to obtain stocks from Nike (NKE), Adidas (ADS.DE), and Under Armour (UAA) over a period of 100 days. I will leverage historical price data to help model my portfolios future returns distribution. I will also be using the numpy package, since it lets me perform the random multivariate normal distribution, as well as the dot product sum between this result and my weightings.

Firstly, I download the historical price data for the specified stocks using the `yfinance` package and subsequently calculate the daily returns, dropping any missing values. I assign theoretical weights to each stock in the portfolio and calculate the mean and covariance of these daily returns, key inputs for the simulation.

```
stock = yf.download(['NKE', 'ADS.DE', 'UAA'], period='100d')
closeDiffs = stock['Close'].pct_change().dropna()

weighting = np.array([0.333, 0.333, 0.334])
mean = closeDiffs.mean()
cov = closeDiffs.cov()
timeHori = 1
```

The Monte Carlo simulation is conducted by generating random samples from a multivariate normal distribution, utilising the mean and covariance of the daily returns, simulating the portfolio's returns over the specified time horizon. The process is then repeated a large number of times to form a distribution of the portfolio's returns, in this case I opted for 10,000 simulations for the program. I initialise an empty list to store the simulated portfolio returns and iterate over the number of simulations, calculating the weighted sum of the simulated returns to represent the portfolio return for each simulation, appending these results to the list [17].

```
portfoReturns = []
for x in range(10000):
    simReturns = np.random.multivariate_normal(mean, cov, timeHori)
    singleReturn = np.sum(simReturns * weighting)
    portfoReturns.append(singleReturn)
```

Finally, I sort the simulated returns and calculate the VaR by determining the value at the specified percentile of the distribution. In this case, using the 5th percentile, a risk level of 5%, which I then multiply by my theoretical portfolio value (100,000,000) to obtain the monetary VaR, rounded to two decimal places for logical pounds and pence articulation.

```
portfoReturns = sorted(portfoReturns)
rlPercent = 0.05
print("VaR: \pounds" + str("{:,}".format(round(-np.percentile(portfoReturns,
                                        100 * rlPercent)*100000000, 2))))
```

Using this methodology, I can provide a robust framework for estimating the VaR for any portfolio I chose to create in the future, adapting to various market conditions, and through the use of this simulation, capturing the inherent uncertainties and correlations between these assets.

5.2 Kivy Screen Manager

I knew that next I would want to implement what I had learnt about Monte Carlo Simulation into my GUI, but my current deliverable couldn't handle multiple stocks being selected at the same time, let alone the calculation of the VaR for them all, as it was only designed for single stock VaR calculations. I figured the best way to implement this would be to add more sections to my program that can actually handle these multiple stocks needed, in essence, I needed to have the ability to

create a portfolio. And since this was building towards my final deliverable, I knew I wanted to produce something that would adhere to the industry standards that I was creating my software for.

Subsequently, upon researching more into the capabilities of kivy and its framework using the online Kivy documentation [13], I discovered the Kivy Screen Manager, a powerful tool that allows for the creation of multiple screens within a single application, each screen able to be utilised as a different section and form of functionality within the program. This was perfect for what I wanted to achieve, the ability to have multiple screens represented within one application, bouncing between these different screen options depending on what you need to do, possibly allowing for communication between screen adding to a more in-depth functionality provided by my program. It also allowed for impressive transitional animations between defined screens, an impressive gif showcasing this being found on the main page for it within the documentation. With this, I knew I had to plan out and draft what I initially needed for my subsequent final deliverable, so just as I had done with my previous design, I drew what I envisioned in the figure below:

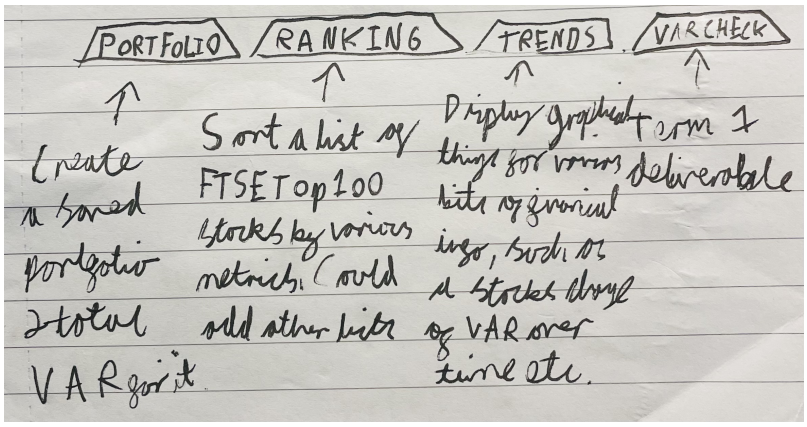

Figure 11: Tabbing Conceptualisation for Switching Between Screens

With this, I came up with the 4 screens that I wanted to implement into my program, those being:

- **Portfolio Screen:** The main screen when you open up the program, where you can create a portfolio, select the stocks and amount of shares that you want for it and generate the Value at Risk for the total value.
- **Rankings Screen:** Have a list of stocks, such as the FTSE100, and be able to rank them based on their VaR values, as well as other metrics, possibly allowing for being able to select different lists.
- **Trends Screen:** Using financial information to display graphical trends for stocks, such as the price of a stock over a period of time, or the VaR of a stock over a period of time.
- **VaRCheck Screen:** A home for my previous initial design deliverable, still practical for what its used for, but not versatile enough to work as an independent program, so it can be used to check theoretical values at risk for the individual FTSE100 stocks.

I was inspired by the design of tabs in most modern browsers, which are used to switch back and forth between websites. I also liked the trapezium shape I used, but I knew I would fit the constrains of whatever my program needed to function optimally. With this, I knew that I had to implement it into my program, so I could begin the development of these screens, allowing for the functionality I was aiming for within my final deliverable.

5.2.1 Tabbing System and Workspace Structure

Up until now, I had my whole program within one file, `Initial Design Test.py`, and the subsequent `IDT.kv` file alongside it, but for this, I knew that I wanted to be connecting a bunch of screens together, which would be far too much code to have it all handled within one file. So, since I'm developing this program adhering to Object Oriented principles, I decided to use a hub program, and split up all my other screens into separate files, and then put into specific folders within the same directory as the main program.

For this I had to separate my original file into one called `Final Design.py`, which would be the hub program, and my renaming my old file to be the screen, `VaRChecker.py`. This involved removing the kivy app definition and running section and putting it in the new program, initialising the app, certain imports and the window size (which had to be increased to be able to fit in the new tabs). The old program was then placed into a folder called `Screens`, and the new program remained at the base of the previous directory. Following on from this, I created the other 3 screens, `PortfolioScreen.py`, `RankingsScreen.py`, and `TrendsScreen.py`, and placed them in the same folder as the `VaRChecker.py` file.

For each of these however, I needed to create the specific .kv file to match, to define the kivy visuals. To do this, I created a folder called `kvFiles`, and implemented the files `PortfolioScreen.kv`, `RankingsScreen.kv`, `TrendsScreen.kv`, and `VaRChecker.kv` (which had the same code as `IDT.kv`). But even with these individual files, my new Final Design file could not see them. To fix this, I created the final new file, `FD.kv`, within the same directory as my `Final Design.py`, that I would use to link all the others together, using:

```
#:include kvFiles\VaRChecker.kv
#:include kvFiles\Trends.kv
#:include kvFiles\Portfolio.kv
#:include kvFiles\Rankings.kv
```

This would allow all my .kv files to work together, but the same could not be said for the python files, since the main app being run within the final design did not have access to the other screens yet. To implement the communication between the screens to the app, I needed to use:

```
from kivy.uix.screenmanager import ScreenManager
from Screens.Portfolio import Portfolio
from Screens.Rankings import Rankings
from Screens.Trends import Trends
from Screens.VaRChecker import VaRChecker
```

This would give my main program access to the other screens, and subsequently link them all together within the screen manager. To do this, within the initial build method immediately defined within the app, I defined my screen manager, and added all the screens to it as widgets. Now that they were inside my screen manager within the building app, I needed to create the tabs for them to be displayed, this being done by creating a BoxLayout taking up the top 10% of the screen, that for each screen within the screen manager, would create a named button 25% of the apps size wide, allowing for the creation of 4 equal sized buttons spanning the top of the screen, with a bound on_release function set to a method I created, taking the text instance found on the selected button and switching the screen managers "current screen" to be the screen with the clicked name. This ended up functioning perfectly, I had to create blank kv files that were gray for all the screens that had not been populated yet, but the code and tabs worked great, with it implementing a swipe animation clicking between the screens already implemented (although it would only swipe from left to right, no matter what direction you were swapping between screens). The tabs and code can be seen below:

```
sm = ScreenManager()
sm.add_widget(Portfolio(name='Portfolio'))
sm.add_widget(Rankings(name='Rankings'))
sm.add_widget(Trends(name='Trends'))
sm.add_widget(VaRChecker(name='VaRChecker'))

def screenSwitch(instance):
    sm.current = instance.text

tabs = BoxLayout(size_hint=(1, 0.1), pos_hint={'top': 1})
for screen in sm.screens:
    print(screen.name)
    tabButton = Button(text=screen.name, size_hint=(None, 1), width=200)
    tabButton.bind(on_release=screenSwitch)
    abs.add_widget(tabButton)

layout = BoxLayout(orientation='vertical')
layout.add_widget(tabs)
layout.add_widget(sm)
return layout
```

Figure 12: Tabs Implemented into the Program

5.3 Developing a Portfolio Screen

When initially creating portfolio management software, you need to take into account many things, such as:

- **Data Collection**: The software needs to be able to collect all relevant data for a stock and update it's pricing in effective real-time.
- **Data Storage**: You need to consider how and where the data will be stored. This is usually done within a database or in the cloud, but it depends on the software you're utilising.
- **User Interface**: The software should have a user-friendly interface that allows users to easily manage their portfolio and understand the interactions between the options that they are presented with.
- **Performance**: The software should be able to handle large amounts of data being stored and processed, to perform calculations quickly enough for the user experience to not be affected.
- **Scalability**: The software should be able to handle an increasing number of stock data, and it should be able to handle multiple users if being user in different environments.

These in tandem will all help when creating this form of software, and help when visualising how you want the software to look and act. For this, I once again drafted an initial design, which I will explain in parts below:

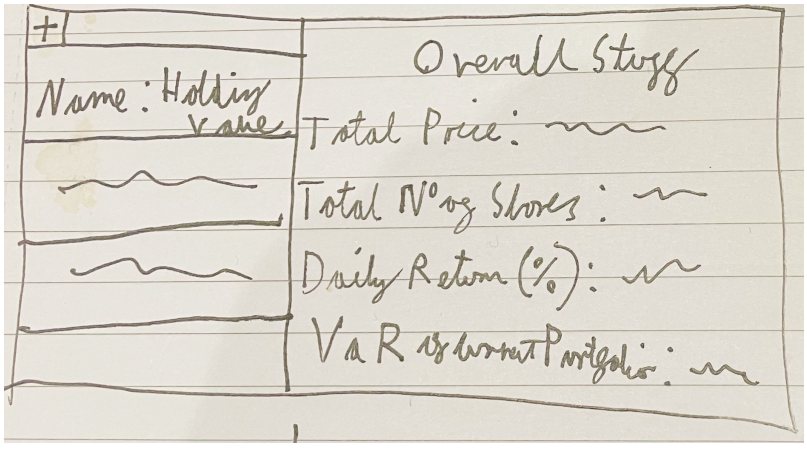

Figure 13: Draft Portfolio Screen - Top

In figure 13, my plan is to create a portfolio system where you can store your stock within a scrolling list on the left hand side. This is the same concept I used for my initial design, within my VaRChecker screen now, but you can add and remove stocks from this dynamic list. To add the stocks, you would click on the button in the top left, which would create a pop-up window, a feature that I did not utilise within my initial design, but upon further use of the kivy documentation [13] I discovered its existence and apparent usefulness for a situation like this. On the right hand side, it would display the overall total price/value of the portfolio, being calculated by getting the live

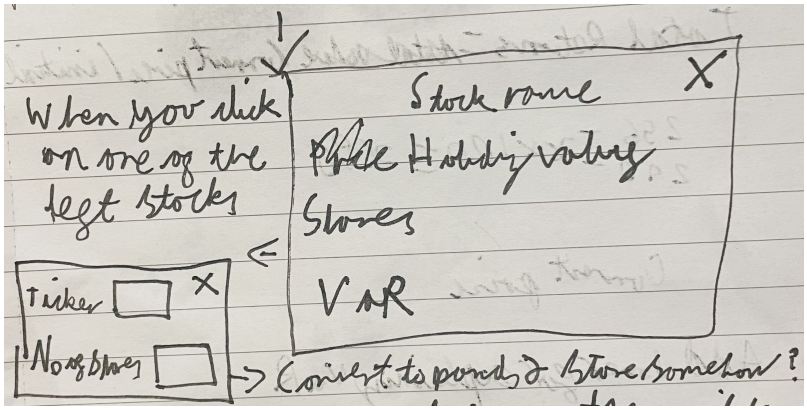

Figure 14: Draft Portfolio Screen - Bottom

prices of stocks, multiplied by the number of stored shares for each stock. It would also display the number of stocks (a rather useless metric from a financial standpoint but a nice feature to have), the daily return of the portfolio, and the Value at Risk of the portfolio, which would be calculated using the Monte Carlo Simulation method I had previously implemented. It is also noted that I want the stocks on the left to have their name displayed, as well as their holding value, since it can be used to compare to the overall value on the right, since both would be shown at the same time.

For figure 14, I wanted to have it so when you select one of the stocks on the right, it replaces the overall portfolio stats on the right with specific ones for the currently selected stock, such as its name, holding value, No. of shares (much more important for this context), and individual VaR. This would allow for a more in-depth analysis of individual stocks within the portfolio, and allow for a more detailed understanding of the risks and rewards of each stock whilst still seeing how diversification affects the stability of Value at Risk. It would have a cross in the top right, allowing you return to your overall view of the portfolio. I also created a small sketch for how I would want the pop-up to look, having two inputs for ticker (since this is how yfinance would download it) and no. of shares, as well as a button to exit the pop-up, resulting in the end of the interaction.

With these initial ideas in place, I set to work on implementing them. Since this was a large program, with many different computational elements and visual designs within both the files used, I will attempt to showcase key points of its development with accompanying visual aid when possible, but since it cannot cover everything, the full code can be found in the appendix.

5.3.1 Initial Structure and Pop-Up

My original structure for the file consisted of 3 classes, these being the screen class `Portfolio`, the one being referenced within my main hub, the button class `Stocks`, where I can display the current stocks that my system deals with and can be pressed, and a pop-up class `InputStock`, a way for me to add those stocks to the system. You would use a button in the top left to access the pop-up, which would create and store stock information. Since I wanted the stocks to be retained after you had closed and reopened the application, I had different options available to me to consider, due to

the vast amount of imports and libraries that Python can utilise to store data, such as database frameworks, even using a notepad file, etc.

But once again, within the kivy documentation [13], there's a built in way to easily store data within a JSON file in the directory your program is running in, using the module `kivy.storage.jsonstore`. It uses an easy dictionary format with a key, and can store all the relevant stock information I would need. When the pop-up is successfully dismissed, it will call a function within `Portfolio` to run an update on the stocks list, checking the JSON file and adding all of them as widgets into a scrollable list, with each stock element containing the current stock being displayed's data, which can be referenced when it's selected later to have its information displayed specifically on the right hand side. Below are the respective sections generally described above:

```
def openPopup(self):
  popup = InputStock()
  popup.bind(on_dismiss=self.loadStocks)
  popup.open()
```

This creates an instance of my pop-up class, binds the `on_dismiss` function to the loadStocks method, and then opens the pop-up.

```
  from kivy.storage.jsonstore import JsonStore
  ...
  class InputStock(Popup):
      def __init__(self, **kwargs):
          super().__init__(**kwargs)
          self.size_hint = (0.5, 0.5)
          self.title = 'New Holding'

      def saveStock(self):
          stockData = {
              'ticker': self.inputTicker.text,
              'overallShares': self.inputShares.text,
          }
          JsonStore('holdings.json').put(stockData['ticker'], **stockData)
          self.dismiss()
```

At the top of the file I import the JsonStore module, and then create my pop-up class, setting its size and title. The saveStock method is called when the save button is pressed, defined within the `.kv` file, creating a dictionary of the inputted stock data and storing it in the JSON file, then finally dismissing itself. The resulting stored data, for one of my initial uses when using my "holdings", looks like this - {"test ": {"ticker": "test ", "overallShares": "test"}}

```
  def loadStocks(self, *args):
```

```
store = JsonStore('holdings.json')
self.ids.stockCards.clear_widgets()

for stockKeys in store:
    stockData = store.get(stockKeys)
    self.addStock(stockData)
```

Upon dismiss, the `loadStocks` method is called, which opens the JSON file, clears the current stock list to make sure duplicates are not created, and iterates over all the keys in the JSON file, adding each stock to the list of stocks using my addStock function, that passes the relevant data into the stocks class.

```
class Stocks(Button):
      self.stockInfo = {
          'ticker': ticker,
          'overallShares': overallShares,
      }
```

Within the class, I create a dictionary to store the stock information, so it can be later referenced when the stock is selected. This allowed me to store and accumulate stocks, that I could visually see added to my screen, but I did not have any deleting process implemented, so I would have to delete them by manually deleting them within the JSON for now. I also set it so that I had default values displayed on the stock information section, and utilised the neutral colour scheme for what I could with the presentation, sticking to colours the I felt were inoffensive from my VaRChecker's style. This resulted in figure 15, very basic but apparent how the screen will be developed from now on.

5.3.2 Storing Stocks and Calculating Portfolio Value

At this point in my project's development cycle, I realised that the nuance between my Rankings Tab and my Trends tab were irrespective for what I needed to create and what I could deliver. Because of this, I realised I could integrate both into each other, graphically display whatever ranking I may compute within its own section, that also would show the trends that my stocks were taking. This resulted in me removing the Rankings tab and planning to just create the Trends tab, which can be seen at the top of the program in figures going forward.

Since I had not implemented verification, I had to be certain that the tickers I entered into my pop-up were correct, and if so, yfinance would be able to correctly download the data. This is crucial, since I would need to retrieve and store the initial stock price of the data as soon as I added it to my stocks list, since it would be needed to refer upon in the future to compare to whatever the continued current price would be going forward, needed when calculating return. To do this I used:

```
initialPrice = yf.download([self.inputTicker.text], period='1d')
                                          .tail(1)['Close'].iloc[0]
```

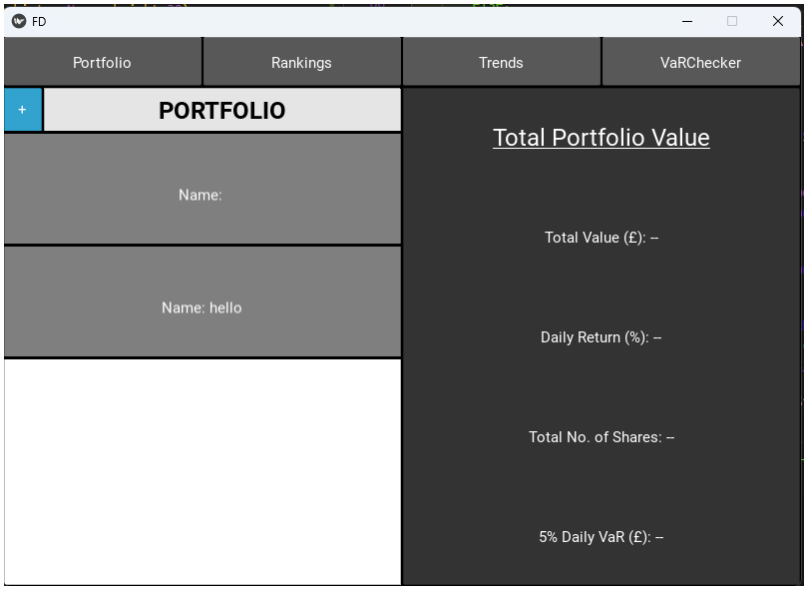

Figure 15: Early Portfolio

Which would download the stock price for the last day, and then take the last value in the list, which would be the current price of the stock. Previously, using yfinance's `.info['regularMarketPrice']` command would have been a viable way for me to retrieve the current price easily, but due to certain limitations that Yahoo's financial API had put into place recently, I was unable to use this method, and had to find this work around, which I continue to use going forward since it presents the same results. This initial price would be stored within the JSON to be referenced later.

With this initial price now stored, I could now calculate the majority of the necessary information that my program would need to consistently display within my totals section. This would be done (once checking that the length of the JSON is not 0) by iterating over all the stocks in the JSON file, and for each stock, downloading its current price, calculating its total value by multiplying it with the stored shares quantity, summing these all together, resulting in the Portfolio's current total value. The total shares would be added up within the iteration, and the daily return I adapted into just being the total return of the portfolio, a percentage based on all the current prices together being compared to all of the initial prices summed together.

```
def initialStockTotals(self, *args):
    store = JsonStore('holdings.json')
    if len(store) != 0:
        totalValue = 0
        ...
        totalShares = 0
```

```
for stockKeys in store:
    stockData = store.get(stockKeys)
    currentPrice = yf.download([stockData['ticker']], period='1d')
                                            .tail(1)['Close'].iloc[0]

    totalValue = totalValue + (currentPrice * float(stockData['sharesOwned']))
    totalCurrentPrices = totalCurrentPrices + currentPrice
    totalInitialPrices = totalInitialPrices + float(stockData['initialPrice'])
    totalShares = totalShares + int(stockData['sharesOwned'])
totalReturn = ((totalCurrentPrices / totalInitialPrices) - 1) * 100

self.stockName.text = "[u]Total Portfolio Value[/u]"
self.totalValue.text = "Total Value: £{:,.2f}".format(totalValue)
self.totalReturn.text = "Total Return: {:.2f}%".format(totalReturn)
self.totalShares.text = f"Total No. of Shares: {totalShares}"
```

At the end, it changes the references to the id's in the `.kv` file's text to be the computed value, displaying them on the right hand side. This method would be called upon when the program is being initialised, and upon the dismissal of the pop-up, to ensure that the totals are always up to date.

5.3.3 Background App Refresh and Race Conditions

But I want to have it so that the program can update the stock prices in real-time, so that the user can see the changes in their portfolio as they happen. Whilst realtime is a bit of an impossibility using python, since the download isn't instant and gets worse with more stocks being stored, having it re-download all stocks every 60 seconds seems a lot more reasonable, especially since stocks don't change in value that quickly. Within my time to decide to do this, I had implemented a `specificStockTotals` method that would be called upon when a stock was selected, and would display the specific stocks information on the right hand side, with generally the same logic as the previous section. Since I would still want this data to be updated every minute when a stock is selected, as well as when the portfolio totals are being displayed, I decided to utilise Kivy's Clock functionality.

By using `Clock.schedule_interval()`, a function found within the `kivy.clock` module, I was able to call a function every 60 seconds, which would be my `initialStockTotals` method, which would update the stock prices and the portfolio totals. I also had a subsequent one within my `specificStockTotals` method that would do the same thing when a specific stock was selected. But this could result in the program trying to update and download the stock prices at the same time, since only downloading one stock when all are necessary would crash the program. To ensure that one clock was being run at one time (preventing any race conditions) and that the clock was not calling an instance inside itself recursively (resulting in it never being able to be stopped), I utilised the code below:

```
def specificStockTotals(self, *args):
```

```
    if isinstance(self.iSTCheck, ClockEvent):
        Clock.unschedule(self.iSTCheck)
        self.iSTCheck = None

    if not isinstance(self.sSTCheck, ClockEvent):
        self.sSTCheck = Clock.schedule_interval(self.specificStockTotals, 60)
```

Using stored checker methods that were initialised during the classes startup, it would check if the other clock was already running, and if so, unschedule it, and then schedule this new clock. This would ensure that only one clock was running at one time, and that the clock was not calling itself recursively, allowing for the program to update the stock prices every 60 seconds completely dependant on what they had selected at the time.

5.3.4 Value at Risk Calculators

You may have noticed that when creating the displays for the total and specific stocks, that I had not displayed or generated the Value at Risk to be displayed for either. This was because I had not yet implemented the Monte Carlo Simulation method into the program, and I would need to do so to be able to calculate the VaR for the portfolio. I had tested single stock calculations for VaR with this method as well, but it did not work, so I decided that I would just use my Model Simulation (Variance-Covariance) method for the single stock VaR calculations, and the Monte Carlo Simulation for the portfolio VaR calculations, as I have had both working in other programs/screens. I chose to use model over historical simulation as when back-testing, the resultant p-value were much higher/more consistent when using the model method, so I believed it to be more reliable. To implement these methods, I decided to create another class (all subsequent new classes mention all being within this same `Portfolio.py` file.) called `VaRCalculators`, that would contain all the methods needed to calculate the VaR for the portfolio and the individual stocks. An instance of this would then be created when the screen was initialised (`self.varCalc = VaRCalculators()`), and the methods within could be referenced whenever needed in the stock total clock cycles.

```
class VaRCalculators:
  def __init__(self, *args):
      self.rlPercent = 0.05
      self.timeHori = 1

  def modelSim(self, totalValue, stocks):
      closeDiffs = stocks['Close'].pct_change().dropna()
      return "{:,.2f}".format((-totalValue*norm.ppf(self.rlPercent/100,
      np.mean(closeDiffs), np.std(closeDiffs)))*np.sqrt(self.timeHori))
```

The `VaRCalculators` class is initialised with the risk level percentage and the time horizon, and the `modelSim` method is used to calculate the VaR for the single selected stock. It takes in the current value of the stock (being the current price times the shares owned) and the downloaded stock data, necessary for calculating the daily returns, and then uses the Model Simulation seen in

section 3.3.2, which is then returned. This method would be called inside the specific stock cycle, and return the VaR to be displayed, rounded to two decimal places and with comma formatting.

For the total portfolio calculations however, I needed to implement the Monte Carlo method, as well as calculate the weightings of all the stocks. But since I had already implemented these kind of calculations in a previous section and had passed in all the stock information, I would use the same logic to calculate weightings and run my subsequent simulation.

```
def monteCarloSim(self, totalValue, stocks):
    weightings = np.zeros(len(stocks['Close'].columns))
    for x, stockKey in enumerate(store):
        stockData = store.get(stockKey)
        currentPrice = stocks['Close'][stockData['ticker']]
        .loc[stocks['Close'][stockData['ticker']].last_valid_index()]

        currentValue = currentPrice * float(stockData['sharesOwned'])
        weightings[x] = currentValue / totalValue

    closeDiffs = stocks['Close'].pct_change(fill_method=None).dropna()
    simNum = 100000
    portfoReturns = np.zeros(simNum)

    optimisedSim = np.random.multivariate_normal(closeDiffs.mean()
                        ,closeDiffs.cov(), (self.timeHori, simNum))

    for x in range(simNum):
        portfoReturns[x] = np.sum(np.sum(optimisedSim[:, x, :]
                                        * weightings, axis=1))

    return "{:,.2f}".format(-np.percentile(sorted(portfoReturns), 100
                                        * self.rlPercent)*totalValue)
```

This new version of my Monte Carlo Simulation would calculate the weightings of all the stocks by iterating over all the stocks in the JSON file, using this to access each index within the downloaded stock data, calculating the current value of the stock, and then dividing it by the total value of the portfolio. It would then calculate their daily returns, and run the simulation 100,000 times. However, I realised that numpy had a built in way to perform the simulation by utilising how `np.random.multivariate_normal()` works. By defining the final section within it with both `(self.timeHori, simNum)`, it would allow me to create a 3D array of the simulations, providing me with the data I needed in one statement. I still needed to run all the simulations, so I created a for loop that would multiply the optimised simulation by the weightings, summing them all together along the outer axis, and then summing them all together again, we result in finally obtaining the portfolio returns. This method ends up being a lot more efficient than my previous method, saving a lot of loading time for the application, which is crucial for the user experience when navigating a the main portfolio screen. Finally, the VaR is calculated by taking the 5th percentile of the sorted returns, and then multiplying it by the total portfolio value, returned correctly formatted.

5.3.5 Finding Comparny Names and Back Button

As I was developing the program, there was a lot of visual iterations and changes to formatting and presentation that I can't cover in this report, but one of the features I knew I would need to have would be the actual display of the full company name, since trying to remember what each company was based on tickers in a stock list felt terrible, this being seen in figure 16 (you can also see the reduced tabs at the top). Usually, this could have been done once again using the slightly altered aforementioned yfinance `.info['name']` method, but this was still broken, so I had to come up with a more creative solution.

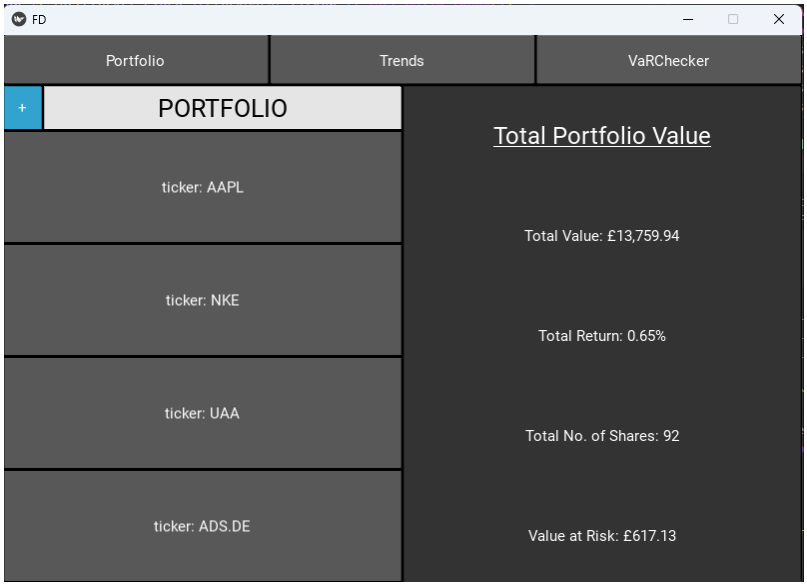

Figure 16: Ticker as Stock Names

Yahoo finance has a fantastic website system for displaying their stocks, where every single stock page you find is located at: `https://uk.finance.yahoo.com/quote/XXXX`, where `XXXX` is the ticker of the stock you are querying. And on each one of their pages, towards the top you will always find the full company name displayed next to its ticker. Upon using inspect element on this section (a feature of most modern browsers allowing the user to see the underlying HTML of a webpage), I could see this `h1` element was always created using the same class, `D(ib) Fz(18px)`.

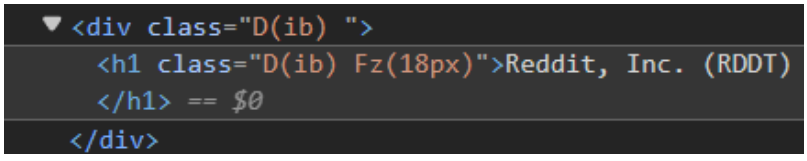

Figure 17: The HTML Element for the Company Name

With this knowledge, I knew that I had to find a way to obtain the html of these pages I can query, with just the information given by a stock ticker. For retrieving the page, I could use the python `requests` module to download the html of a webpage, but for the parsing of the html, I would need to use `BeautifulSoup`. This module allows for the parsing of html and xml documents, and can be used to extract information from them, such as the text within the element that I needed to find.

```
def findCompanyName(self, ticker):
    yFinancePage = f"https://finance.yahoo.com/quote/{ticker}"
    headers = {
    'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
    (KHTML, like Gecko) Chrome/123.0.0.0 Safari/537.36 Edg/123.0.0.0'
    }

    response = requests.get(yFinancePage, headers=headers)
    soup = BeautifulSoup(response.content, 'html.parser')
    companyName = soup.find('h1', class_='D(ib) Fz(18px)')

    return companyName.text
```

This method would take in a ticker, concatenate it with a url to create the link for my needed stock page, and then download the html of the page using the requests module. Whilst this method would work for most of the stocks that I tried for, any that had "Non-US" denoters (stocks that aren't listed on NASDAQ, instead with London Stock Exchange with .L or country specific Germany with .DE) would fail to have their page load and stored correctly. To combat this, I would employ the use of a User Agent, found at `https://user-agents.net/my-user-agent`, within the headers of the request, which would simulate the request to appear as if it was coming from a browser, and not a python script, successfully allowing me to download the HTML data.

I would then proceed to parse the html using BeautifulSoup, and find the `h1` element with the class I needed, and return the text within it, which would be the full company name. This method would now be called upon whenever a stock was added to the list, storing the full company name along with ticker inside the JSON file, and with a small adjustment to stocks, promptly displayed in tandem on the stock list, as seen in 18. This would allow for a much more user-friendly experience when navigating my app, I'm aware that it would be optimal to use a stock name when adding it to your portfolio but I could not envision this capability working vice versa due to the nature of the URL's that Yahoo Finance provides, since it is through this that I can use the ticker and find the stock names in the first place.

The new resultant holding information would look like this - `{"ADS.DE": {"name": "adidas AG (ADS.DE)", "ticker": "ADS.DE", "sharesOwned": "50", "initialPrice": 201.39999389648438}}`

Another thing to note within the figure 18 is the addition of a back button, which can be seen in the top right. I assumed with the `+` symbol for adding a stock and this `X` button, that the user would intuitively understand what each were trying to represent, since it adheres to the general design principles found within the creation of graphical interfaces [18]. On the Total Portfolio

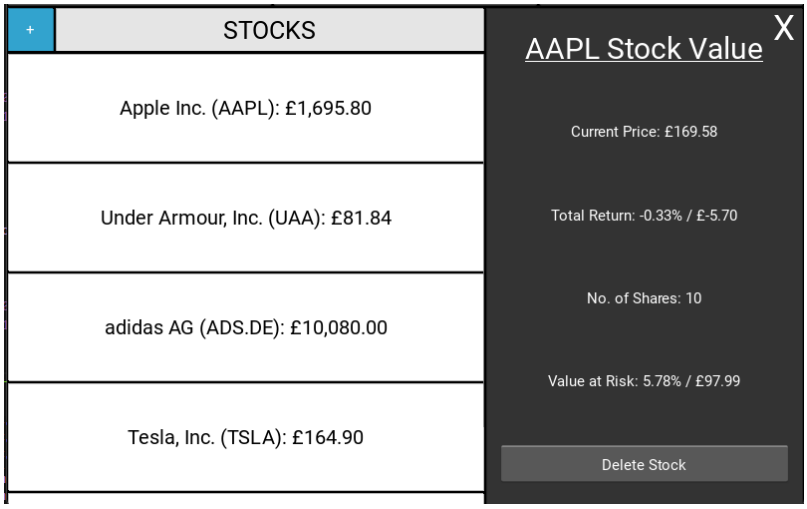

Figure 18: Stocks List with Company Names

display, the back button can still be seen in the same place, but with its opacity and ability to be pressed set to 0. When you run the `specificStockTotals` method, it would set the opacity to 1 and the ability to be pressed to 1, allowing for the user to return to the total portfolio view. And the code it runs when pressed is simply just `initialStockTotals()`, redrawing back over it and setting itself to invisible & untouchable again.

5.3.6 Improving Monte Carlo Simulation

Every time I would switch between my two Portfolio views, it would have to recalculate the VaR, thus running more Monte Carlo Simulations, and slightly lagging the app consistently. Whilst this method was efficient, I wanted to adapt is so that it didn't have to run through all the simulations extensively every time (since the more simulations produced more accurate results that I wanted to have displayed), resulting in me considering the use of a convergence point/threshold. With this, my method would continuously perform simulation steps, incrementing and producing a new VaR result every time, but performing many less simulations to begin with. And as soon as one of these simulations is within the same threshold (1%) as the last one, I could assume that this was a reasonably accurate estimation of the VaR, returning the latest value for it.

```
def monteCarloSim(self, totalValue, stocks):
    ...
    simNum = 5000
    convThreshold = 0.005
    previousVar = float('inf')
    converged = False

    while not converged and simNum <= 100000:
```

```
        ...
        currentVar = -np.percentile(sorted(portfoReturns), 100 * self.rlPercent)

        if previousVar != float('inf') and abs((currentVar - previousVar) /
                                               previousVar) < convThreshold:
            converged = True
        else:
            previousVar = currentVar
            simNum += 50004
```

I would set the starting simulation number to 5000, the threshold to 0.5% as it seemed to produce the best results with the speed the VaR would now be calculated at. Upon running the simulation, it would verify that the previous VaR was not infinity, and check if the current VaR was within the threshold of the previous one. If so, it would set the converged boolean to true and return this as the new VaR, and if not, it would set the previous VaR to the current one, increment the simulation number by 5000, and run the simulation again, until it either converges or reaches the point where it is performing 100,000 simulations again. With this convergence method, I was able to obtain the VaR results at a much faster speed, helping improve the app further.

5.3.7 Deleting Stocks

As can also be seen in figure 18, this time in the bottom right corner, I had implemented a delete button. This would only be visible when a user has selected a stock, since this indicates that the selected stock is the one that can be deleted. Since I found it so useful in creating its own formatted separate section to acquire information, I wanted to use another pop-up to confirm the deletion of a stock, since being able to do it with just one button press would be too much of a risk, especially when dealing with these assets. To do this, I would create a new class for the pop-up in my `Portfolio.py` file, as well as the subsequent one in my `Portfolio.kv` file, which I would be able to use to define the pop-ups layout, containing a title, warning and two buttons to press.

```
<ConfirmDelete>:
    size_hint: 0.5, 0.35
    title: 'Confirm Deletion'
    BoxLayout:
        orientation: 'vertical'
        spacing: 10
        Label:
            text: "Are you sure you want to delete this stock?\n\nIt will be permanent."
            halign: 'center'
            valign: 'center'
        BoxLayout:
            size_hint_y: None
            height: '50sp'
            Button:
                text: "Yes"
```

```
            on_release: root.on_confirm()
        Button:
            text: "No"
            on_release: root.dismiss()
```

In the process of formatting and changing the functionality of my add stocks pop-up, I discovered that by default, when you click off the pop-up, it would dismiss itself, which was the perfect behaviours for that pop-up, and could be well utilised here again, even if it does also have a no button that will accomplish the same thing. The on_confirm using the Yes button would send a signal to the root class, but since the button for the pop-up is found within my portfolio class, I had assigned the current ticker to it, so that when it is pressed, the ticker is passed in with it.

```
class ConfirmDelete(Popup):
  def __init__(self, portfolio, ticker=None,  **kwargs):
      super().__init__(**kwargs)
      self.ticker = ticker
      self.currentPortfolio = portfolio

  def on_confirm(self):
      JsonStore('holdings.json').delete(self.ticker)
      self.currentPortfolio.initialStockTotals()
      self.dismiss()
```

With the ticker being initialised and the current instance of my portfolio class also being passed in, the on_confirm method can find the ticker within my JSON file, delete it, and call the initialStockTotals method to update the stock list, and then dismiss itself. This would now remove the stock, whenever any of the Stock Totals methods are ran, they also call the `loadStocks` method, which would update the stock list, removing the deleted stock from the list. This happens dynamically within the list in kivy, so if you're scrolled down within the list when viewing and deleting a stock, it will keep you in the same position you were, not having to scroll back down to find where you were. The pop-up for this can be found below, as well as the portfolio screen display after deleting a stock.

5.3.8 Adjusting VaR Parameters

As seen in figure 20, in the exact place where the (Delete Stock) button was, there's another button called `Adjust VaR`. This is actually the same button but having the text within it changed based on the Portfolio or Specific stock view being used, the logic for which being in a function that checks what text is currently being used by the button to indicate what to do for it. In this instance, it is now used to open up a final pop-up, one used to help explain VaR and add the ability to adjust the parameters of the VaR calculation. My initial creation of it can be seen in figure 21.

```
class AdjustVaRPopup(Popup):
    def __init__(self, portfolio, varCalc, **kwargs):
        super().__init__(**kwargs)
```

Figure 19: Delete Stock Pop-Up

```
        self.varCalc = varCalc
        self.currentPortfolio = portfolio

    def submit(self):
        self.varCalc.timeHori = int(self.timeHoriInput.text)
        self.varCalc.rlPercent = float(self.riskLevelInput.text) / 100.0
        self.currentPortfolio.initialStockTotals()
        self.dismiss()
```

With this pop-up, it passed in the current instance of the portfolio class and the varCalc class being used inside it. Then, it could directly change the VaR calculator class by adjusting its parameters directly, as well as updating the new VaR totals to reflect these new changes, applicable for both Monte Carlo Simulation and Model (Variance-Covariance) methods. I knew that not everyone that uses this app could understand what Value at Risk even represents or means in general, so I decided to add a brief explanation of what it is in the form of a contextual sentence as can be seen in figure 22. I used colour coordination to make it easier to understand what text input relates to what portion of the sentence, making use of hint text colour, and for both pop-ups featuring inputted data (the other being `InputStock`), I added full verification and visual animations to help emphasise when either text input was incorrect. The numbers within this sentence will also update every time the pop-up is re-opened, so the user will always know what values are currently selected. I know that it would be better to show this on the Portfolio screen at all times, but I was not able to find a subsequent place that I could showcase such information, so it's current location will have to suffice.

5.4 The Finalised Portfolio Screen

Being focused more on the functionality of the program rather than its form, by using VaRChecker I was able to find a simple gray scheme, and whilst I continued to utilise a few of these colours for consistency, I settled on a range of different blues for the portfolio screens display. I have made a multitude of different styling and customisations to help with visual appearance and user

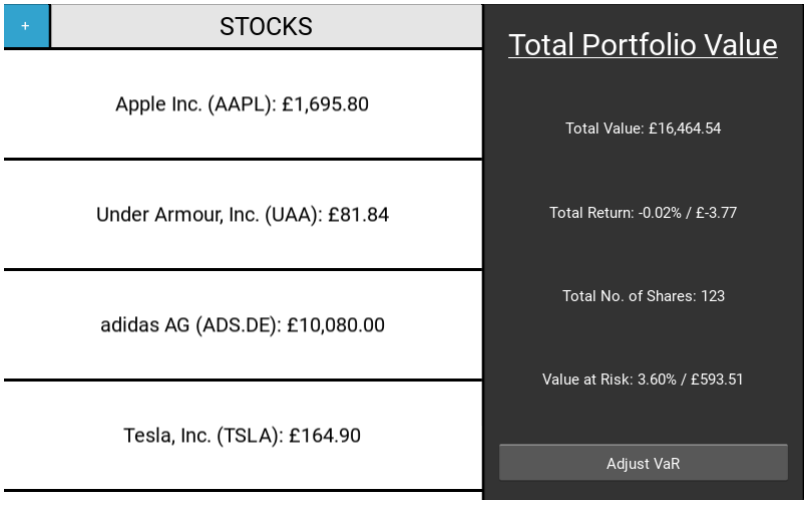

Figure 20: Stocks List after Deleting a Stock

understanding of where things are and what is important. The portfolio's code has been adapted to work when there are no stocks being held within it, and when there's only one stock being held as well, since yfinance downloads need to be defined differently afterwards when retrieving information since it gives it back in a different data-frame. For a while, I was dividing my VaR Method Simulation by 100 since I used the same code from my `VaRChecker.py` file, but after realising and changing this, I was able to get matching VaR results across the two screens, so both produce accurate single stock data. The screen is fantastically robust and can handle whatever stock storage scenario you want to throw at it, including storing large stock names and numbers. I will add a range of figures showing various parts of this screen below, this also including 22 since that is from the finished product as well and a previous green button design that I didn't think fit the visual style, as well as me not having changed the Stocks to a Blue Hue.

6 Chapter 5 — Graphical Visualisation

6.1 Convergence Analysis

After finishing my Portfolio screen, I realised that the term "Trends" for my final screen was not sufficient for what I wanted it to be. Thus, it was changed to "Graphs", particularly influenced by a brief detour I took within my recent development cycle, when I created a graphical representation of my Monte Carlo Simulation. I was trying to understand the convergence method I eventually used, since it would help me hone in on a more accurate result in faster time, improving my codes efficiency, but I didn't entirely understand what I was looking for. To help me visualise what I needed, I decided to employ the use of a library called MatPlotLib (`https://matplotlib.org/`), a plotting library for the Python programming language. When defined and called, it would create a new window with a graph of the data you had inputted, a feature I have used in the past for other coding projects, so I knew how useful it can be for displaying these forms of information.

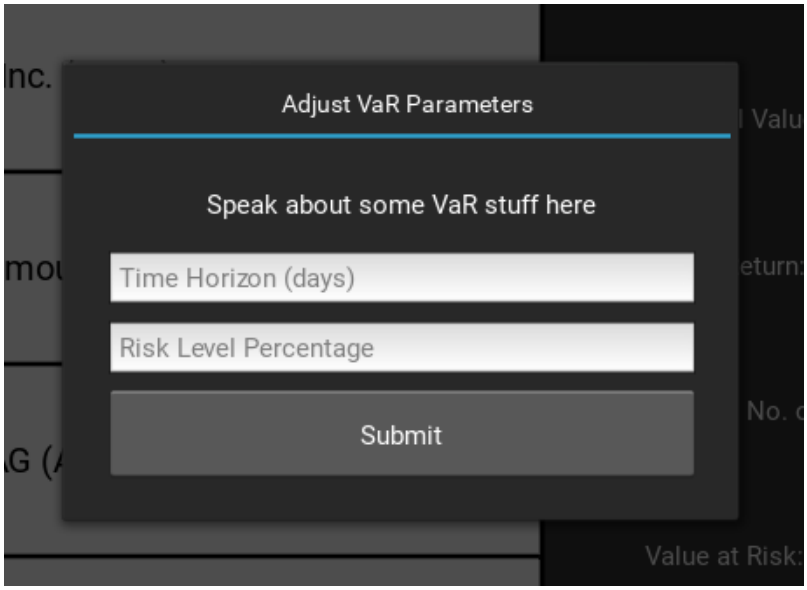

Figure 21: Original Adjust VaR Pop-Up

So whilst I was initially experimenting with the coding design found in section **??**, I used the codes results to store the VaR values into a list, and plotted these results on a graph, with the x-axis being the number of simulations (checkpoints) done for each VaR value, which was subsequently displayed on the y-axis. This would allow me to see if I was right to attempt this method of finding an incremental way to ensure that the VaR was converging, and if it was, how many simulations it would take to reach this point.

```
import matplotlib.pyplot as plt
plt.plot(checkpoints, varResults, marker='o-')
plt.xlabel('Number of Simulations')
plt.ylabel('Value at Risk (VaR)')
plt.title('Convergence Analysis of Monte Carlo Simulation')
plt.show()
```

The methods used for the library are quite self-explanatory, with the `plt.plot()` method plotting the data, the `plt.xlabel()` and `plt.ylabel()` methods setting the labels for the x and y axis, the `plt.title()` method setting the title of the graph, and with `plt.show()`, the graph is displayed. When the code is ran, a MatPlotLib pop-up window would appear, giving me the ability to visually perceive the convergence and perform my own analysis of it, resulting in me implementing a slightly altered version of this convergence method for my portfolio, but giving me the idea to have this be displayed within my Graphs screen, since its so useful in showing how the Monte Carlo Simulation works. I would go on to create many graphs within this screen, one of such being this one. Within the rest of this section, I will go into detail on each screen that was created, but here I will show what the Convergence Analysis graph looked like in figure 27.

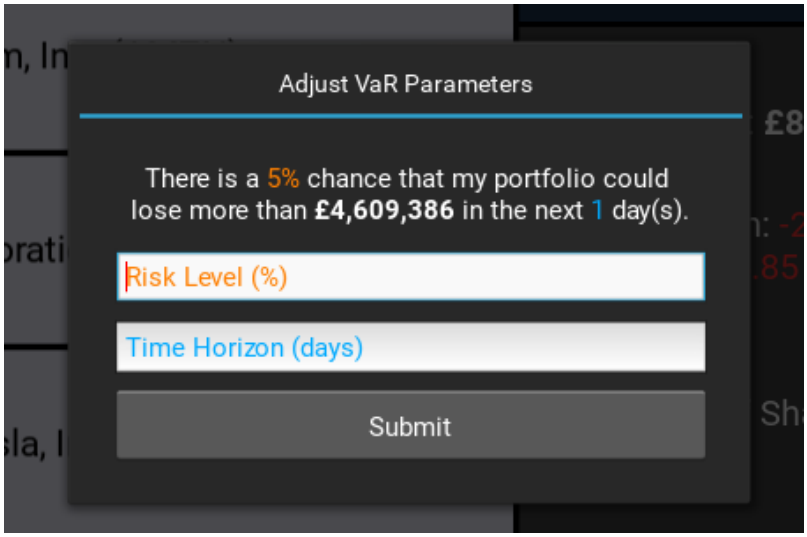

Figure 22: Completed Version of Adjust VaR Pop-Up

6.2 Matplotlib within Kivy

However, unlike my previous approach, having an external library forcefully open up its own custom pop-up was not what I was looking for in my singularly packaged application module I was looking to create. I knew that I wanted to be able to display this graphical interface within the confines of my program, and to do this I would need to utilise arguably the best feature that kivy has been able to integrate, `Garden`. Garden is a collection of widget and library addons that has been implemented and maintained by users, giving it full capability to be able to do whatever other users may have been able to implement within the kivy source material. This gives Kivy such an incredible wide range of capabilities, since community driven development can foster a lot of creativity and innovation, and upon looking through garden, the module I was in need of blossomed into view.

There was quite simply a MatPlotLib implementation of Kivy that was readily available within the Garden module, `https://github.com/kivy-garden/garden.matplotlib`, and whilst both garden and the matplotlib version would need a pip-install, those were the only limitations stopping me from being able to start utilising this framework within my project. Alas, after everything was installed and correctly referenced, there was no success in the communication between the garden matplotlib backend (which I had found the location of) and my ability to import the libraries into my python program. This resulted in me trying many techniques to initiate communication, even adding a custom `PATH` indicator for python to my system, but nothing seemed to work.
Fortunately, others online had been encountering the same issue as me, where they had full evidence that they had installed and referenced the library to within their knowledge, but could not prevail in overcoming the errors that were presented when trying to run the code. They had raised this issue in `Stack Overflow`, a notorious website for coders and developers alike for the amount of questions and answers contained within it, one such being someone asking for help with this same issue I was having. The answer they received was that it was a current bug, and you needed to

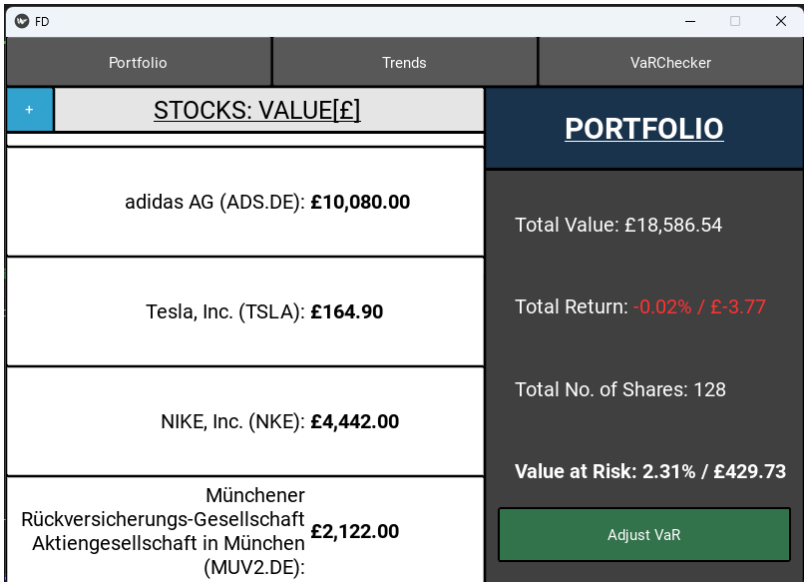

Figure 23: Previous Iteration of Portfolio Screen

install the library with a slightly different console command seen below:

```
python -m pip install https://github.com/kivy-garden/matplotlib/archive/master.zip
```

This would help install the newly update and fixed library needed from the proper github source. They also said to reference the import as `kivy_garden`, rather then with a "." between them. With my new full import being `from kivy_garden.matplotlib.backend_kivyagg import FigureCanvasKivyAgg`, I was able finally able to stop the errors that my IDE was presenting me with, and could now start on implement the MatPlotLib graphs within my Kivy application. The link to the answer that I used to help fix this issue was done by "John Anderson" and can be found here `https://stackoverflow.com/a/77911314`

6.3 Creating the Graphs Screen

For this screen, all I knew is that I wanted to show off a multitude of different graphs, and that if I could, I would want to represent them in a sorted, ranked format. This was due to me removing my previous rankings class, in favour of an overall graphing section, so I knew that it was still something that I had to desire to incorporate. So for the last time, I was able to draft and visualise what I needed to create before I started work on it, with pen and paper again.

As can be seen in figure 28, I originally had another scrollbar, on the right hand side this time, to navigate between the different graphical options, and the actual graphing section would take up the rest of the screen. But as you can also see with the red arrow below, I decided that moving it to the

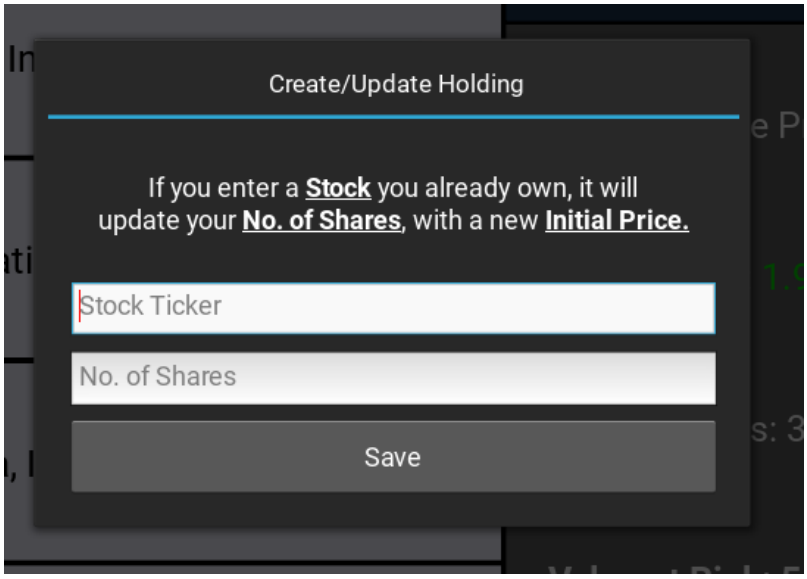

Figure 24: Final Create/Update Holding Pop-Up

bottom of the window would not only diversify it in comparison to my other screens, but allow for a better space to display my graphing portion, as it aligns to a wider rectangular shape, rather than a tall and thinner one. This would allow for a better visual representation of the data, and a more user-friendly experience when navigating the screen.

Below this drawing, I also specified the different types of graphs that I wanted to display, since it was good to plan ahead for what I wanted to accomplish. Whilst they can also be seen in figure 29, I will list what I initially envisioned here as well, since some are the main focus of this screen:

- Convergence Analysis for Current Portfolio with Monte Carlo Simulation
- Sorted FTSE100 Model VaR Plots (with hover functionality)
- Visualisation of Backtesting (in some regard)
- Past Theoretical Portfolio Value Over Time
- Past Single Stock Selected Price Over Time

As you may also be able to see, I jotted numbers next to relevant ideas to understand the x-axis limits that each would have to display, since this would alter how I develop them in the future, and I wanted to make sure each one was as clear as possible for understanding the resultant graphs. With this, I used kivy to layout my initial screen and created a basic plot to test the functionality of the libraries ability to display the graphs within my code.

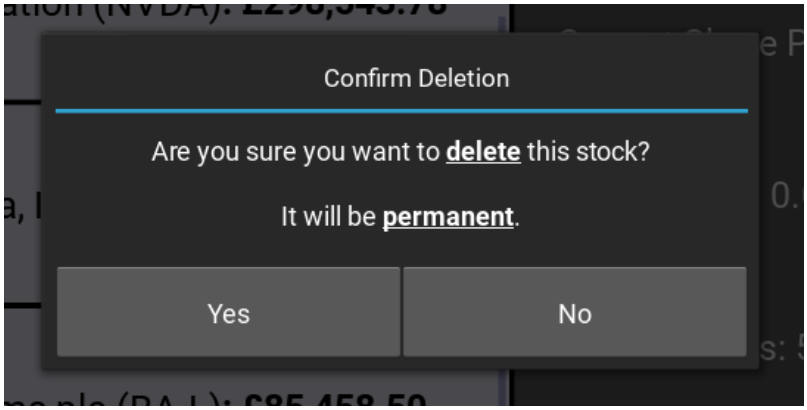

Figure 25: Final Confirm Delete Pop-Up

```
import matplotlib.pyplot as plt
from kivy\_garden.matplotlib.backend\_kivyagg import FigureCanvasKivyAgg
class Graphs(Screen):
    def __init__(self, **kwargs):
        super(Graphs, self).__init__(**kwargs)
        self.createGraph()

    def createGraph(self):
        plt.figure(figsize=(6, 4))
        plt.plot([1, 29, 12, 4])
        ...
        plt.tight_layout()
        self.ids.graphSection.add_widget(FigureCanvasKivyAgg(plt.gcf()))
```

During `Graphs` class initialisation, the `createGraph` method would be called, creating a new figure with a size of 6 by 4, plotting a simple line graph with the random data `[1, 29, 12, 4]`, and then adding it, using the `FigureCanvasKivyAgg` class, to the `graphSection` layout within the kivy file, where I had specified enough space for it to be rendered. This worked upon initial startup, and when I created the subsequent scroll wheel with simple buttons, binding one to this method, I was able to call it to be created again. However, I was yet to add a line clearing any of the widgets within the `graphSection` layout, so every time the button was pressed (and due to the line `plt.tight_layout()` being used to keep everything together within the graphic boundaries), the subsequent graphs would all be squashed together, lagging the program more and more. Fortunately, a simple addition of the line `self.ids.graphSection.clear_widgets()` removed it at the start of the method call, but you can see the results of this error in figures 30 and 31.

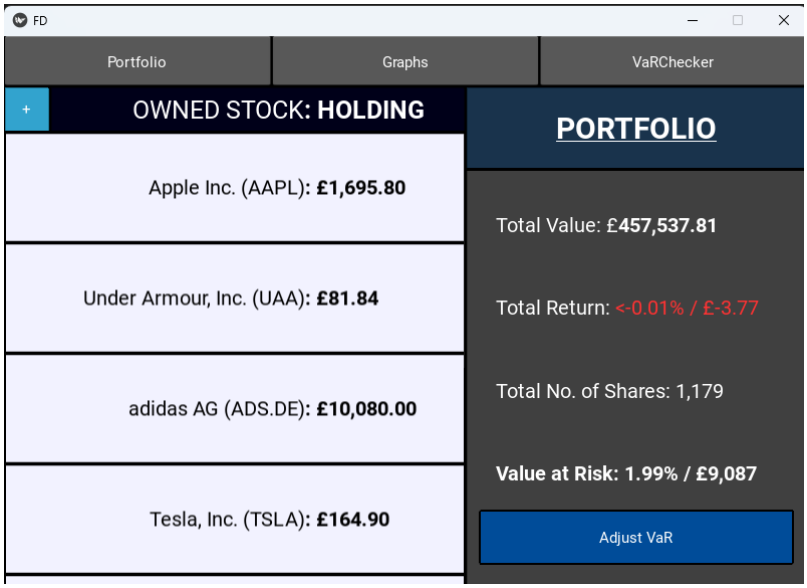

Figure 26: Completed Portfolio Screen

6.3.1 Layout and Dynamic Pop-Up Labels

Something that can also be seen within the figures 30 and 31 is the crudely drawn graph repeated many times across the buttons at the bottom of the screen. My goal was to have distinct and vibrant graphs by the end of this section, so my goal was to take these, add a blur effect as to make any text in front of them more distinct, helping the user know which graph they are going to generate when the buttons are pressed by just being able to vaguely see when scrolling through. Of course this would have to be added at the end of the screens development, when all the other graphs have achieved a distinct appearance, so the "stock" stock image that I created in paint was used as a placeholder for the majority of this classes development going forward. My original full code for `Graphs.kv` can be seen in figure **??**, since I haven't had opportunity to showcase much of the kivy layout. In it, you can see me define the

With my layout defined and the buttons set up to create these stock graphs, I knew that now would be the best time for me to attempt to implement one of the most important features, having the ability to hover over points on the graph, and have a little display for the information it is being plotted with. Since there was little conventional documentation for this implementation of MatPlotLib, as well as a few failed attempts to try and trial and error this to work, I decided to see if anyone in the small community that helped to develop for this user ran library had any more success with this than I was having. I once again found myself on Stack Overflow, where someone was having a problem getting their interpretation of the issue to work, and once again someone in the comments had offered a solution. Whilst I didn't take the exact code that they presented, I was heavily inspired by the logic that they used within their solution, and I want to fully acknowledge that without this original help I would not have been able to expand and adapt on this informational pop-up throughout the rest of my screens development. The link to the answer that

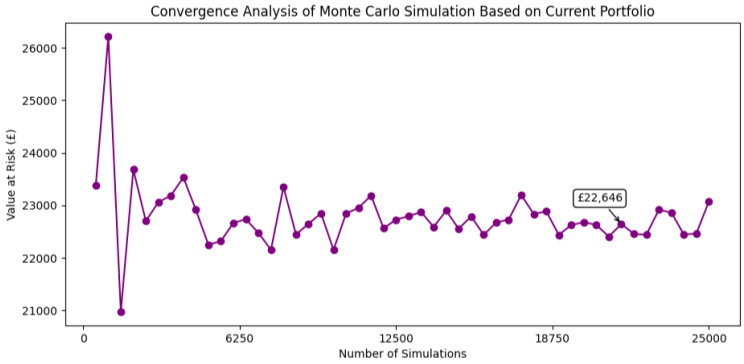

Figure 27: Convergence Analysis of Monte Carlo Simulation

helped inspire my adapted solution, given by `Justin CR`, can be found here `https://stackoverflow.com/a/55184676`, with my adapted version also being visible below, split into sections to explain how it works.

```
def Graph1(self):
    x = list(range(1, 101))
    y = [random.randint(i, 1000) for i in x]
    self.createGraph(x, y, 'some random linear numbers',
         'more random linear numbers', 'Random Graph 1')
```

I've now separated the creation of the graph into its own method, so that I can call it with different data and labels, something I am continusly planning to do for other graphs, and have it create a new plot each time. This was just a random test one with some randomly generated numbers and default labels.

```
def createGraph(self, x, y, xlabel, ylabel, title):
    self.ids.graphSection.clear_widgets()
    self.fig, self.ax = plt.subplots()
    self.currentLine, = self.ax.plot(x, y, 'o-')
    ...
    self.infoPopup = self.ax.annotate("", xy=(0, 0), xytext=(-20, 20), textcoords=
    "offset points",bbox=dict(boxstyle="round", fc="w"), arrowprops=dict(arrowstyle="->"))
    self.infoPopup.set_visible(False)

    canvas = FigureCanvasKivyAgg(self.fig)
    canvas.mpl_connect("motion_notify_event", self.mouse_hover)

    canvas = FigureCanvasKivyAgg(self.fig)
    self.ids.graphSection.add_widget(canvas)
```

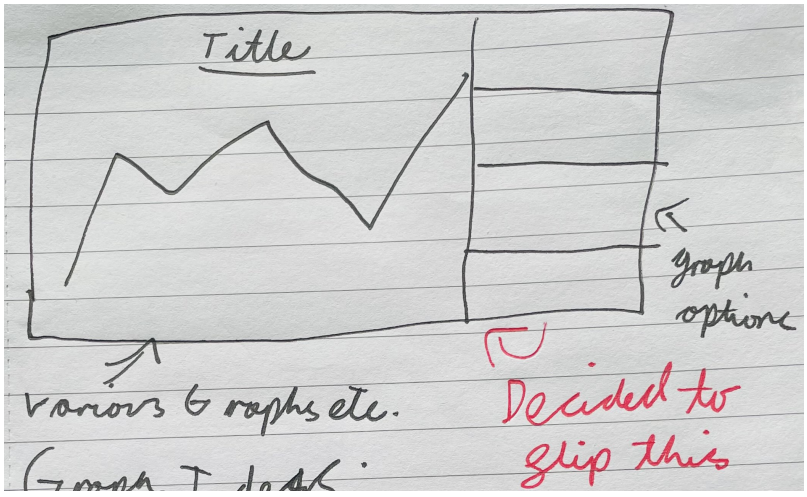

Figure 28: Draft of the Graphs Screen

The `createGraph` method is now called with the data and labels that I want to display being passed into it, proceeding to create a new figure and axis, plotting all the data. Next, it will carefully construct the informational pop-up that will be displayed when hovering over the graph, and set it to be invisible by default. The other parameters within it are used to define the style of the pop-up, and the arrow that will point to the data point being hovered over. The `FigureCanvasKivyAgg` class is then used to display the graph within the kivy layout, and the `mpl_connect` method is used to connect the mouse hover event to the `mouse_hover` method, something that will be defined underneath this.

```
def mouse_hover(self, event):
    if event.inaxes == self.ax:
        cont, ind = self.line1.contains(event)
        if cont:
            pos = ind['ind'][0]
            x, y = self.line1.get_data()
            self.showPopup(x[pos], y[pos])
        else:
            self.hidePopup()

def hidePopup(self):
    self.infoPopup.set_visible(False)
    self.fig.canvas.draw_idle()
```

The `mouse_hover` method is called whenever the mouse is moved over the graph, constantly checking to see if the mouse is within the graph's axis. If it is, it will check if the mouse is hovering over any data point by using the `contains` method, and if it so, it will pass the x and y data points

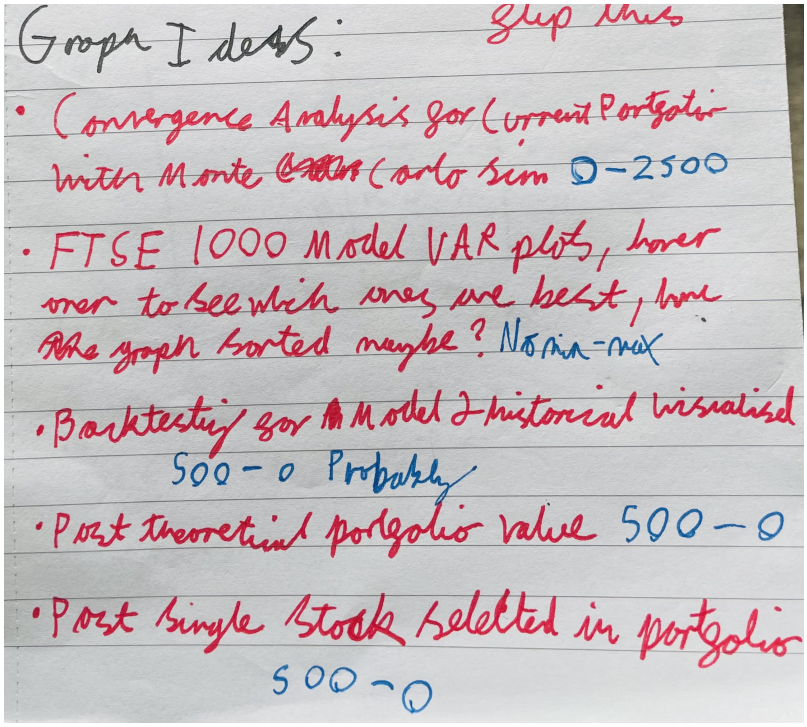

Figure 29: Draft of the Graphs Screen Continued

to another method called `showPopup`. If the mouse is not hovering over any data points, it will call the `hidePopup` method, which will set the pop-up to be invisible, seen beneath the other method.

```
def showPopup(self, x, y):
    text = f"({x}, {y})"
    self.infoPopup.set_text(text)
    self.infoPopup.xy = (x, y)
    self.infoPopup.set_visible(True)
    self.fig.canvas.draw_idle()
```

And finally, the `showPopup` method will set the text of the pop-up to currently be the x and y data points that the mouse is hovering over. I will need to change this to be able to display different values in the future, since the x values are rarely relevant to what I want to display for my proposed graphical ideas. It then sets the position of the pop-up to be at the x and y data points, sets it to be visible, and then redraw the canvas to display the pop-up. With this working in tandem with the other methods, I am able to create a fully functioning hover method that will display the values of whatever graph is entered into the `createGraph` method, and will be able to be used for all future graphs that I create.

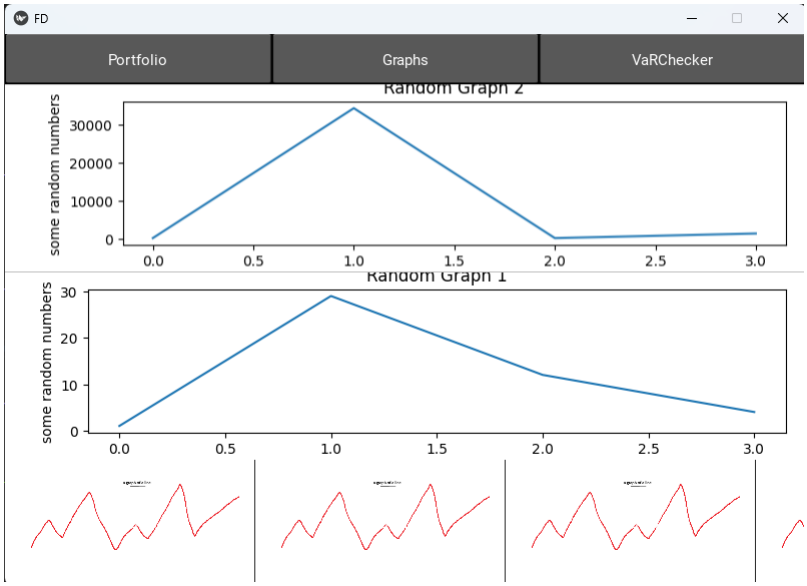

Figure 30: Graphs Screen with 2 Graphs Plotted

6.3.2 Threading

With this newfound ability to see specific relevant information for specific points on graphs, I knew the first one I should properly try to implement was my convergence analysis graph, since I already knew how to define it for MatPlotLib, and would especially useful in comparing the converging points to the Monte Carlo Simulation result found in my Portfolio screen. After porting over the code I had previously used and getting it to run independently from the varCalculators class in the portfolio screen, I was able to have the simulation checkpoints generated and displayed within my Kivy application. This was fantastic, but did cause a massive issue, even with the optimised simulation I had created, the amount of time I had set for the simulation to run unaltered (for 75k simulations) would lag the program for around 10+, where you couldn't move the program would become completely unresponsive. This was no way for any form of industry standard software to run, since I knew regardless of how fast I can get the simulations to generate, if I could still make the program usable at the same time, it could work as a background task before its completed and displayed.
To do this, I would need to implement threading, something that I had not needed to include before now due to the lack of any significant hindrances to the applications running speed. Threading is part of pythons core library, so all I would need to do is import it, and then create a new thread that would run the simulation in the background, whilst the main thread would continue to run the program as normal. This would allow for the program to be responsive, and the simulation to run in the background, with the results being displayed when they are ready. I also tried for a while to implement a module called asyncio, but this was to no avail.

```
def graph3(self):
    threading.Thread(target=self.monteCarloConvSim).start()
```

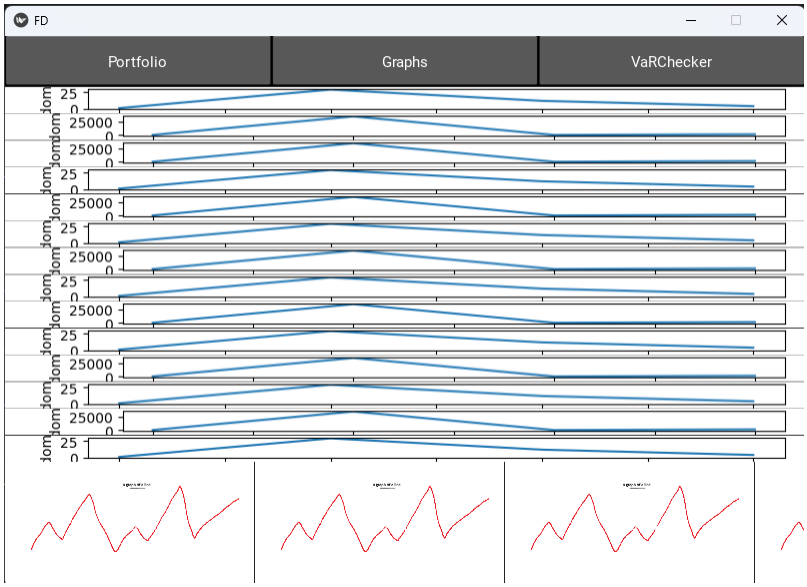

Figure 31: Graphs Screen with Many Graphs Plotted

```
  def monteCarloConvSim(self):
      stocks = self.portfolio.tempDownload
      store = JsonStore('holdings.json')
      ...
      totalValue = float(self.portfolio.totalValue.text.split('£')[1].replace(',', '')[:-4])

@mainthread
    def createGraph(self, x, y, xlabel, ylabel, title):
    ...
```

My `graph3` method would be called when the button was pressed, and would start a new thread that would run the `monteCarloConvSim` method. This method would then run the simulation, and then call the `createGraph` method when it had ended to display the results. The `@mainthread` decorator would then be used to ensure that the graph is created on the main thread, since the simulation would have been subsequently running on a separate parallel one. This would allow for the program to be be more responsive, and the simulation to run in the background, but due to the intensity of the simulation, I had to implement a delay of 1 second in-between checkpoints just to let the program run semi-properly. It was apparent that even with this solution, I would still need to change something in the future to get this graph to be a more consistent viable option.

Later on, I would define my threads with `thread.daemon = True`, this boolean value allowing threads to run in the background as usual, but if the main thread is closed, the daemon threads will also be closed. This was important, as there were times when threads were running, and I wouldn't

```
<Graphs>:
BoxLayout:
    orientation: 'vertical'
    BoxLayout:
        id: graphSection
        orientation: 'vertical'
        size_hint_y: 0.75
        canvas.before:
            Color:
                rgba: 0.2, 0.2, 0.2, 1
            Rectangle:
                pos: self.pos
                size: self.size
    ScrollView:
        do_scroll_x: True
        do_scroll_y: False
        size_hint_y: 0.25
        GridLayout:
            cols: 10
            size_hint_x: None
            width: self.minimum_width
            canvas.before:
                Color:
                    rgba: 0.2, 0.2, 0.2, 1
                Rectangle:
                    pos: self.pos
                    size: self.size
            Button:
                text: 'Option 1'
                background_normal: 'graph reference.png'
                size_hint_x: None
                width: 250
                ...
            Button:
                text: 'Option 6'
                background_normal: 'graph reference.png'
                size_hint_x: None
                width: 250
```

Figure 32: Graphs Screen Original `.kv` (Kivy) File

want to see the final result, opting to shut the program down. Since the thread was running separately from the main app initiating its shutdown, this would cause the whole app to crash, resulting in an unnecessary situation for this sort of software to encounter. To fix this, with the daemon value being set to true, as soon as the main thread closes down, the threads will all close down with it, resulting in smooth exits from the application.

6.3.3 Theoretical Portfolio Value Over Time

For my first graph made specifically for this screen, I wanted to go with the Theoretical Portfolio Value Over Time, as I understood the logic for it, and knew that with it completed, I would be able to create the specific stock graph easily afterwards. I had altered the info popup to now only display the more relevant y-value, along with a passed in symbol (this being a £sign), allowing for a much more visually understandable graph. For all the graphs that I have developed for this screen, I will show a significant portion of the code, allowing me to explain the logic behind each one.

```
def graph1(self):
    stocks = self.portfolio.tempDownload
    store = JsonStore('holdings.json')
    closePrices = stocks['Adj Close'].tail(500)
    totalValues = []
    for i in range(len(closePrices)):
        dailyTotal = 0
        row = closePrices.iloc[i]
        for stockKey in store:
            stockData = store.get(stockKey)
            dailyTotal += row[stockData['ticker']] * float(stockData['sharesOwned'])
        totalValues.append(dailyTotal)
    x = list(range(0, 500, 15))
    y = totalValues[::15]
    self.createGraph(x, y, 'Days', 'Total Portfolio Value', 'Portfolio Value Over Time')
```

My new `graph1`, since I thought that it would be the best one to see when the screen is originally loaded, will directly communicate with the portfolio screen for its information. I set it up so that whenever the stocks are downloaded, to reduce the need to have them downloaded any further time, I would store them within a variable, which can be cross referenced between classes. The store would also be loaded from it's usual file, and the subsequent adjusted close prices of the stocks being calculated too. The total values would then be computed for each day by iterating through each stock and multiplying these daily adjusted close prices by the amount of shares owned (stored within holdings), appending the results to a list. The x values would be set to be every 15 days, as this produced a well enough structured graph, and the y values would be set to be every 15th value in the total values list to match. The `createGraph` method would then be called with these values, and the graph would be displayed.

6.3.4 Single Stock Price Over Time and Vectorisation

Succeeding this, I could adapt this same code to be able to display the Single Stock Price Over Time, since the logic was very similar, and I could use the same data that I had already downloaded. For this, the user would select a stock within their portfolio screen, then go over to the graphical screen to select this graph to be rendered. Since every time I would display a specific stock, I would store its downloaded info in a variable, that would also be set to none if there totals are being displayed, so I can call this variable into this program to access the data.

```
tempStockInfo = self.portfolio.tempStockInfo
if tempStockInfo is not None:
    stocks = self.portfolio.tempDownload
    ...
    for i in range(len(closePrices)):
        dailyTotal = 0
        row = closePrices.iloc[i]
        ...
        totalValues.append(dailyTotal)
    x = list(range(500, 0, -15))
    y = totalValues[::15]
self.createGraph(x, y, 'Last 500 Days', self.portfolio.tempStockInfo['ticker']...)
```

If the `tempStockInfo` variable is not none, the stocks variable will retrieve the rest of the temporarily stored downloaded stocks. The total values would then be computed in the same way as the previous graph, minus the need to multiply by shares and added together, for the specific stock, with the x and y value being set with almost the same metric. I decided that since these graphs are showing the progression of the stock over the last 500 days, then the graph should indicate this, with numbers ascending from 500 to 0. Now normally, this goes against how the library is plotted, since it usually defaults to putting the x-axis in ascending order. However, you can call the `invert_xaxis()` method to help flip is when you display. Implementing this into my `createGraph` method, I was able to get it to check in what order the passed in x values where in after I plotted the values with the original x-y coordinates defaulting to ascending, allowing for the plot to look like normal, but for the x-axis to be displayed in a reverse descending order.

During this time, I was becoming increasingly annoyed by the ridiculous amount of time it would take to generate the simulations for my convergence graph (unlike how it stops itself when converging within my main program), so I decided to do some experimentations with the code, utilising the time module within python to directly time certain portion of the code, and see how me changing my algorithm affected it. However, this timer wasn't even necessary to comprehend the improvement that I found when adjusting one of my Numpy functions.

```
weightings = weightings.reshape(1, -1)
portfoReturns = np.sum(optimisedSim * weightings, axis=-1)
```

By using np.sum with the added additional parameter of summing them across the last axis, I was able to speed up the overall simulation sequence by 100% almost. This was due to vectorisation,

with the optimisedSim being a 2D array, numpy would let me then sum across its last axis, reducing it to a 1D array, and effectively replacing the previous hard coded python loop. To match this sum, I needed to slightly adjust the dimensions of weightings beforehand, but with this newfound method, I could generate the simulations needed for the graph without even the need for threading, but it was still kept due to reduce the program that can have anything capable of blocking. I implemented this revised and improved method for my normal convergence within my portfolio class as well, it bringing the same results, at a much faster rate.

6.3.5 Creating Ranking Graphs and Displaying Names

When I first thought about implementing rankings graphs, I knew that I would have to completely adapt the way my code would display its popup, since there wouldn't be much relevance in seeing the y-axis, if you don't know what each individually plotted points represents. It becomes less about the value, and more about each points relative position to each other. With this knowledge, I knew I had to find some way to not only get all the names of the stocks without having to use my previous method, due to the slow speed done in retrieving that many request responses. But I also needed to find a way to get my hover feature to the names for each plotted point, especially since the would be extremely bunched together when displaying 100 different stocks for it.

For this, I decided to create an entirely new graph creation method called `createRankingGraph`, that would be called when a ranking graph was selected. It would take in a list of strings and a list of values, create an x-axis plot based on the lengths of the list, and plot these y values equally spaced out along the x-axis.

```
def createRankingGraph(self, tickers, vars):
      self.fig, self.ax = plt.subplots()
      self.currentLine, = self.ax.plot(range(len(vars)), vars, 'o-')
      self.ax.get_xaxis().set_ticks([])
      ...
      self.tickerHover = tickers # Save the tickers for the hover function
      canvas.mpl_connect("motion_notify_event", self.FTSEonHover)
      self.threadRunning = False
```

It would remove the ticks from the x-axis, as they hold no relevance to the graph, other then to place a point dependant on the given y values, that would be sorted in an order before being passed in. The tickers would need to be saved for the hover function to work in the future for this method, as it can't be derived from a coordinate position, and the `mpl_connect` method would be used to connect to this new hover event called `FTSEonHover`. Finally, individual thread checks were set up for any methods using threading, as to not have the user click on the button multiple times and load up multiple threads in succession, this way, it locks them out of this ability until the thread has finished running.

```
def FTSEonHover(self, event):
  ...
      pos = ind['ind'][0]
```

```
        tickerName = self.tickerHover[pos]
        self.showFTSE(tickerName, event.xdata, event.ydata)
    else:
        self.hidePopup()
```

Done the same way as the other hover method, the `FTSEonHover` method would check if the mouse was hovering over any data points, and if so, it would pass the ticker name to the `showFTSE` method, along with the x and y data points. This would then display then use the x and y points to know where to display the popup, with it not containing the ticker name within it. Later on I changed this to be the full names of the companies, as it is not good practice to ask users to be able to remember what company relates to what ticker symbol. This was done by accessing the same pandas imported wikipedia table, that contained the stocks names as well as the tickers, with both needing to be mapped with each other. If the mouse was not hovering over any data points, the previous `hidePopup` method could still be utilised for the same purpose.

```
def ftse100Ranking(self):
    ftse100 = [ticker + ".L" for ticker in self.varChecker.ftse100['Ticker'].tolist()]
    ftse100Names = self.varChecker.ftse100['Company'].tolist()
    tickerToName = dict(zip(ftse100, ftse100Names))  # mapping between tickers and names
    stockData = yf.download(ftse100, period="500d")
    VaRs = {}
    for stock in stockData.columns.levels[1]:
        VaRs[tickerToName[stock]] = float(self.portfolio.varCalc.modelSim(1000, stockData['Adj Clo
    sortedVar = dict(sorted(VaRs.items(), key=lambda item: item[1]))
    self.createRankingGraph(list(sortedVar.keys()), list(sortedVar.values()))
```

The zipped dictionary that maps each ticker to its company is stored to be used for later. The FTSE100 stocks are then downloaded all at once, much faster then doing each stock iteratively within a loop, and their VaRs are calculated for each stock using the modelSim in the VaRCalculators class, with these results also being stored in a dictionary. This dictionary is then sorted by its values, and the keys and values are passed into the `createRankingGraph` method, which will display the graph using the methods described above.

With this, I was able to plot all 100 of the lists stocks onto a graph, and with little degree of difficulty, move my mouse over each one to see the exact name of the company that the name was displaying for. This similar system was also used for creating a ranking graph for your portfolio, helping display the relative risk that each stock contains individually, and providing the user with a versatile tool to help them choose and understand the risk that they are taking on.

6.4 Final Design

7 Chapter 7 — Software Engineering

7.1 Object Oriented Programming

Object-Oriented Programming (OOP) helps form the backbone of the design and structure of my Value at Risk (VaR) application. Utilising certain principles, such as encapsulation, inheritance, and polymorphism, I have been able to organise my codebase into classes and objects, allowing for modular, reusable, and scalable software architecture.

I use encapsulation to bundle the data (attributes) and code (methods) that operates on the data into a single unit, or class. For instance, the `VaRCalculators` class encapsulates the logic for VaR calculations, including both historical simulation and the Monte Carlo simulation methods. This encapsulation ensures that the calculation logic is abstracted away from the rest of the application, promoting code reuse and simplicity.

Inheritance is a core part of developing within Kivy, since its has predefined classes used for GUI creation, enabling you to adapt them into subclasses that inherit attributes and methods from these parent classes. In my codebase, inheritance is prominently used for all my graphical user interface (GUI) components. For example, the `Portfolio`, `Graphs`, and `VaRChecker` classes inherit from the Kivy `Screen` class, inheriting essential GUI functionalities while also allowing for the extension and customisation I have done specifically for each screen within the application.

Polymorphism helps allow for the use of a single interface to represent different underlying forms (data types). Through method overriding, the application can process objects differently depending on their class. An example of polymorphism is observed when handling different types events (e.g. button presses) within the GUI, since the same event listener interface is used, but the specific actions performed are dependent on the context in which the event occurs, such as switching between different views or initiating a VaR calculation etc.

Since I originally used a more linear approach, and then decided to adopt OOP principles, it has helped significantly contributed to the project's success by:

- Enhancing code maintainability and readability, thereby reducing the complexity involved in extending the application.
- Facilitating the division of the project into distinct, manageable sections, each encapsulating specific functionalities.
- Allowing for the efficient handling of GUI events and interactions, therefore improving user experience.

Since one of my main goals was to create software that would adhere to industry standards, using OOP in the development of financial applications such as this one, aligns with industry best practices. OOP's emphasis on modularity, reusability, and scalability is particularly beneficial in the fast-paced financial sector, where these applications need to be robust, secure, and adaptable to ever-changing market conditions. Moreover, the OOP approach aids in collaborative development environments, facilitating easier code reviews, testing, and integration, thus contributing to the

overall efficiency and quality of software development projects within the industry, even if I wasn't able to utilise these benefits as much within my own development cycle.

7.2 Design Patterns

In developing my Initial Design for my Graphical User Interface (GUI) to calculate VaR, I have employed several design patterns that facilitate a robust, scalable, and maintainable codebase. Below are key design patterns utilized within my application:

Model-View-Controller (MVC)

For my final iteration of my Initial GUI Design, the architecture loosely follows the MVC pattern, segregating the application logic (Model), user interface (View), and input control (Controller) into separate components. This separation enhances the application's ability to handle user interactions and data manipulation independently.

- The **Model** is represented by the data retrieval and VaR calculation logic, which fetches financial data and computes the VaR and back-testing estimation.
- The **View** is the user interface, designed with Kivy's layout and widget system, providing a responsive and interactive experience.
- The **Controller** is evident in the event-handling methods, like `populateList` and `generateVaR`, which respond to user inputs and trigger model updates.

Observer Pattern

The Observer pattern is present in the way the application monitors the state of user inputs. For instance, toggle buttons for selecting the VaR calculation method employ event listeners that update the system's state when user interaction occurs.

```
hButton.bind(on_press=self.simMethodPressed)
mButton.bind(on_press=self.simMethodPressed)
```

This pattern decouples the system's components, allowing for independent updates and scalability.

Command Pattern

The Command pattern is utilized in encapsulating the action to be performed in response to user interactions. For example, when a user selects a stock or initiates a VaR calculation, the corresponding actions are enclosed within command methods, such as `populateList` or `generateVaR`.

```
button.bind(on_release=lambda btn, i=i: ...)
```

This approach allows for more flexible and extensible execution of actions within the application.

Singleton Pattern

While not explicitly implemented, the Singleton pattern is conceptually used in the management of the application state. There is only one instance of the `ApplicationView` class, which maintains a single state throughout the application's lifecycle.

```
class IDTApp(App):
    def build(self):
        return ApplicationView()
```

This ensures that there is a consistent state that is accessible throughout all parts of the application.

These design patterns contribute significantly to the application's robustness, allowing for efficient risk assessment and providing a foundation for future enhancements and scalability. By adhering to these established practices, I have ensured that my software remains maintainable and adaptable to the evolving needs that my graphical interface my undergo in the future.

Going forward, when developing my final deliverable, I introduced and explored even more design patterns, each one helping build and maintain a different section of my code base, making it more efficient and easier to understand and work with.

Decorator Pattern

The Decorator Pattern is utilised within the graphical components of my application, particularly for the customisation of graphical user interface (GUI) elements. For instance, in `Graphs.kv`, `BlackScroll` acts as a decorator around the standard `ScrollView` component provided by Kivy. This decoration enhances the functionality of `ScrollView` by customizing its visual appearance, such as modifying the scroll bar's color and width to be more visually distinct and viewable, something I did not do with my initial design. The decorator pattern allows for dynamic additions to the functionality of an object without altering its structure, since I could leave the logic in place and just enhance its visual appearance.

```
<BlackScroll@ScrollView>:
  bar_width: 5
  bar_color:  0, 0, 0.05, 1
  ...
```

Dependency Injection Pattern

The creation of an object of another class within the initialisation of a class, such as the instantiation of `VaRCalculators` within the `Portfolio` class, aligns with the Dependency Injection Pattern. This pattern allows for a form of inversion of control where a class receives its dependencies from external sources rather than creating them internally, helping facilitate increased modularity and making the code more maintainable and extensible.

```
class Portfolio(Screen):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.varCalc = VaRCalculators()
```

By injecting the `VaRCalculators` dependency into the `Portfolio` class, the application's components become loosely coupled, enhancing the flexibility and reusability of the code. This can be seen when I reference the `varCalc` object within the `Portfolio` class in my `Graphs` screen,

allowing for the VaR calculations to be performed for data retrieved from another class, meshing them all together coherently.

Accessor and Mutator Methods (Getter/Setter)

The use of `@property` decorators in Python, seen at the start of my Graphs screen, embodies the Accessor (Getter) and Mutator (Setter) methods pattern, by providing controlled access to certain attributes of another class. This decorator allows a method to be accessed like an attribute, enabling encapsulation and the hiding of internal representation from the class's external interface.

```
class Graphs(Screen):
    @property
    def portfolio(self):
        return self.manager.get_screen('Portfolio') if self.manager else None
```

The implementation of getter and setter methods through this helps ensure that object attributes are accessed and modified in a controlled manner, helping not only safeguard the data integrity of the class but also enhancing the flexibility of changing the attribute's internal representation without affecting external access to it. Using the Accessor and Mutator Methods pattern through the `@property` decorators significantly helps contribute towards the robustness and maintainability of the program, helping promote one of the best practices within object-oriented programming.

Wrapper Pattern

I was very happy with my use of the Wrapper Pattern employed in `graphs.py` through the use of a decorator function, `checkForStocks`, which serves as a pre-invocation check for methods responsible for generating graphs. This wrapper ensures that the underlying graph generation methods are called only if there are stocks present in the portfolio, thereby preventing runtime errors or undesirable behaviours when attempting to render the graphs without data. This pattern is an excellent example of adding pre-processing steps to method invocations without altering the core functionality of the methods themselves, and I plan to use this far more in future projects due to its simplicity and usefulness.

```
def checkForStocks(func):
    def wrapper(*args, **kwargs):
        holdings = JsonStore('holdings.json')
        if len(holdings) == 0:
            print("Cannot create graph due to lack of stocks in portfolio.")
            return
        return func(*args, **kwargs)
    return wrapper
```

This implementation of the Wrapper Pattern ensures that each graph generation checks the state of the portfolio before execution. By doing so, it gracefully handles scenarios where the portfolio is empty, enhancing the application's robustness, verification and user experience.

These additional design patterns utilised throughout all code within my program helps underscore the application's well-structured architecture, helping achieve a high degree of modularity, extensibility, robustness, and characteristics that are all essential for maintaining and scaling complex software systems such as this.

7.3 Testing and Documentation

Testing is a critical component of software development, essential for ensuring that the application meets its requirements and is free of errors. Mainly for my project, the primary mode of testing involved running the application and observing its behaviours for errors, crashes, GUI issues or incorrect VaR calculations.

This testing approach used can be classified as Ad Hoc Testing and Exploratory Testing. These methods are informal and unstructured, as I have been relying on my intuition and experience to identify issues.

- **Ad Hoc Testing**: This type of testing is carried out without any formal test plan or test case documentation. It is a random examination of the application, often useful for discovering glaring issues quickly.
- **Exploratory Testing**: This approach combines learning, test design, and test execution into a single activity. It is particularly effective in situations where there are no detailed specifications, and tests need to be devised on the fly.

While these methods can be effective for identifying obvious faults, they lack reliability and cannot provide a systematic approach that other testing methods would. They are often more suited to the early stages of development, but have been sufficient enough with my knowledge and expertise to produce a robust final product regardless.

One of the main limitations of my current testing approach is the absence of **Test-Driven Development (TDD)**. TDD is a modern software development process where requirements are turned into very specific test cases, then the software is improved to pass the new tests. This approach was not followed, as I was unable to justify what I would be testing for (I had planned out unit testing, but I especially did not know how to incorporate it within a Kivy application, as I have not done TDD with Python or Kivy before), so I continued with the above testing style, which worked and allowed me to create my program efficiently, but not up to the industry standard that I know I should be aiming for.

If I was to continue to develop my final deliverable, with its scalability, for future development cycles, I would plan on incorporating more testing strategies to enhance the quality and reliability of the application, such as:

- **Unit Testing**: Develop tests for individual components or functions to ensure they work correctly in isolation. (Had already planned, now I shall be able to implement)
- **Integration Testing**: Test the interactions between different parts of the application to verify that they work together as intended (Happened during my Ad Hoc Testing often, but need to be able to explicitly show that that's what I was testing for).
- **Automated Regression Testing**: Implement an automated test suite that can be run regularly to catch regressions early in my next GUI's development cycle.
- **Performance Testing**: Evaluate the application's performance, especially when handling large datasets or performing complex calculations, to ensure that it meets the necessary speed and efficiency requirements, especially when I incorporate multiple stocks, portfolios, and full searching capabilities into future applications.
- **User Acceptance Testing (UAT)**: Involve end-users to validate the functionality and usability of the application in real-world scenarios (Will bring in external individuals to help be observed, providing a wide range of feedback that I can take note of and apply to my application).

By integrating these practices, my application could achieve a higher level of quality assurance and deliver a more robust and user-trusted financial product.

Another key aspect to consider when developing software is the use of documentation. Documentation is essential for ensuring that the codebase is understandable for future users and mainly developers, so they can easily navigate, comprehend and possibly scale the application going forward. For my project, since it was created in Python, I could have utilised a library called PyDoc (`https://docs.python.org/3/library/pydoc.html`), which automatically generates documentation from Python modules.

This would have helped me easily create documentation for my codebase, providing a clear and structured overview of the application's components, classes, methods, and attributes. However, due to the nature of my codes development alongside this reports structure, I believed that the level of detail that I have been able to display of multiple parts within my codes development cycle are acceptable in understanding key aspects of the application. But there are still many things that I did not touch on within this report, so having a more detailed documentation system in place would have been beneficial, and something that I would have welcomed implementing in the future if development was to continue with this project.

7.4 Version Control (Git)

Version control is a system that records changes to a file or set of files over time so that specific versions can be recalled later. In this project, I have had Git employed as the version control system for everything I have done. Git is a distributed version control system, which means that

the complete codebase and history are available on every developer's computer, allowing for easy branching and merging. My approach to using Git was straightforward:

- Regular Commits: Consistent commits were made to the repository, with detailed messages that clearly described the changes and their impact on the project.
- Backup and Security: All the code was backed up on OneDrive as well (a personal decision), providing an additional layer of security and ensuring that no work would be lost in case of system failures.
- Remote Repository: The code was pushed to a remote repository on GitHub, enabling me to work on the project from multiple devices when necessary.

The practice of making regular commits with detailed messages is incredibly invaluable, thus me making sure to regularly comply with this methodology. It enabled the tracking of changes and helps simplify the process of understanding the evolution of the codebase. For me, Git has been essential for several reasons:

- **Reversion**: Git can allow me to revert to previous states of the codebase, which is crucial when a new feature can accidentally break functionality.
- **Traceability**: With Git, all my changes are traceable to a specific commit, making it easier to track when and why changes were made.
- **Backup and Restore**: My files are backed up on a remote repository, providing a fail-safe in case the my local repository suffers a setback.
- **Future Referencing**: The commit history can be looked back upon to help understand the evolution of the codebase and the rationale behind specific decisions.

While my development with this project's version control strategy was effective to a degree, it also highlighted areas for improvement. I acknowledge that I did not engage with using the branching feature for my program, which would have been easily implemented due to the nature of working on one specific screen at any given time, since I would be able to easily isolate these periods of development into feature branches. This was a lapse in my ability when using Git, I was more focused on the outcome of the overall product that I chose not to consider branching, especially since I was the only developer on this project. However, this is still not an excuse and I am rightly disappointed in my lack of its usage.

Another example of me failing to leveraging the full suite of features offered by Git, which I should have implemented to ensure a more efficient and error-preventative development workflow, would be the use of tagging. Tagging is a feature that allows you to mark specific points in the repository's history as being significant, such as a release or a version update. This would have been useful for marking the completion of each screen, or the completion of each iteration of the GUI, so that I could easily refer back to these points in the future, especially when using Git interfaces such as `GitHub Desktop`, since it shows the tags in a linear visual manner, used to easily identify your significant commits. However, I do understand that tags are mainly used for different iterations of deployed software, a stage that my application did not reach until the very end of its development, so I think that my use of a tag for its final release is justifiable in this instance.

8 Chapter 8 — Critical Analysis

8.1 Interim Evaluation

In the end, for this term, I am still happy with the progress I made. I learnt some very valuable lessons, such as needing to make sure that I do all of the documentation alongside the coding, since with the gap in progress, it can become tedious to try and explain it all properly in the future. I feel that I accomplished the goals set out within my original timeline for what I wanted to achieve this term, for this interim report, and whilst there are sections of this report that I am unhappy with the result of (especially the final iteration of the GUI), I am still satisfied with the report as a whole in displaying what I had learnt and the code I was able to create, especially with the inconsistency with my physical and mental health experienced over the last few months. I also am aware of my lack of testing that I have performed, I spent a large chunk of time just testing everything myself to make sure it works, when I should have been following a better framework, I plan to learn from this and change the way I develop and document the code come next term. I know the Latex is not great in this document, its still the first time I've ever done large scale documentation like this before, but hopefully by the time my final report is complete I will have given myself enough time for it to be at an acceptable level. I also know I haven't used enough sources, the sources that I did use were so useful that I kept referring back to them, which makes my ability to reference them rather bland, so I will also make sure to diversify with many more references in the future. With how the final design for the initial GUI came out, I'm extremely happy with its functionality, its very fast and easy to use and produces the exact results you need whilst still being robust. There's still a lot to work on for it in the future, like adding different windows/views, incorporating multiple stocks VaR calculations (I wrote a program to do multiple stocks but I didn't have the time to include that in this documentation either, can be found within my Git project/submitted programs) and allowing you to access entire portfolio's within the application. What I made this term was a great proof of concept for what I can do with Kivy, for my final report I want to be able to do so much more, since that is my main goal, the development of an industry grade application for Value at Risk as well as financial management in general. Finally, there were many planned sections of this report that were never completed, I aim to be able to structure my documentation so that I can complete everything I need in the following term, so to conclude, I am extremely happy with everything I was able to do, but I plan on doing everything much better in the future, as I know it was not optimal enough for what I want to achieve...

8.2 Final Report Evaluation

8.2.1 Coding Achivements and Dificulties

8.3 Conclusion

9 Professional Issues

9.1 Finance Industry Ethics

The finance industry, at its core, has consistently, for the last few centuries, played an incredibly important role in the global economy, influencing everything from peoples individual livelihoods to the stability of entire nations. However, its complexity and the significance has unfortunately

introduce a grand myriad of ethical considerations. With the increasing integration of technology and computer science within this sector, especially in areas such as algorithmic trading, risk assessment models, and financial prediction systems, the ethical implications have started to become consistently more pronounced.

One of these primary ethical concerns within the industry is about the concept of risk and how to manage it. Financial institutions often employ complex computer models to predict market trends and assess risk levels associated with various financial products; whilst these models which are designed to minimise risk and maximise returns, they can sometimes fail spectacularly, leading to significant financial losses and, more importantly, affecting the lives of millions and millions of people. The 2008 financial crisis serves as a stark reminder of what can happen when risk management models are not adequately designed or ethically considered [14], highlighting these consequences of prioritising profit over the potential societal impact it can bring.

Furthermore, the finance industry's reliance on technology and algorithms has raised further questions about transparency and accountability, with financial algorithms, created by computer scientists and mathematicians, forming a sort of "black box" with their inner workings known only to a select few. This lack of transparency can lead to companies and wealthy individuals exploiting and performing unethical practices, going unnoticed until the resultant negative consequences come to fruition.

This ethical use of technology in finance can also extend to the issue of data privacy and security. Financial institutions collect vast amounts of personal data, and the ethical handling of this data is paramount [15]. There have been numerous instances of data breaches in major financial institutions, leading to the exposure of sensitive personal information. The ethical responsibility of protecting this data should never be overstated, as the repercussions of such breaches can be devastating for individuals affected, evaporating people's livelihoods.

Following on from this, the integration of Artificial Intelligence (AI) in financial services not only brings about innovation and efficiency but also introduces another broad range of ethical and privacy concerns. To try and mitigate these specific kind of risks, corporations have started establishing Institutional Review Boards (IRB) [15] so they can ensure ethical data practices and help guide further ethical decision-making with these new technologies.

These AI technologies, with their vast data processing and computational capabilities, amplify the risks of data abuse, so upholding them to ethical scrutiny should be imperative. An IRB can function as an early-warning system, identifying potential ethical and privacy issues before they escalate and proactively helping ensure that any AI-driven decisions align with ethical standards and general societal values.

The ethical deployment of AI within more and more financial services must require a consistent commitment to transparency, accountability, and it's continuous ethical evaluation. By establishing these rules and boards under scrutiny, financial institutions can help navigate the ethical complexities of AI, ensuring that technological advancements contribute positively to society while safeguarding the right and privacy of individuals.

This approach not only aligns with the ethical guidelines provided by professional bodies such as the BCS and ACM, but also reflects a broader commitment to ethical professionalism in the

computing field. As future computing professionals, understanding and implementing ethical practices in the development and deployment of AI technologies is paramount, ensuring that we contribute to a technologically advanced yet ethically responsible future.

As a computer science student, hopefully soon to be graduating and entering this field or one similar with the same problems to do with artificial Intelligence, it is crucial that I recognize the profound societal impact of what the people in the industry are able to achieve and shape. The aforementioned codes of conduct and ethical guidelines are there to emphasise integrity, competence, and a commitment to the interests of not only stakeholder, but to all the people reliant on these products and services that we provide. Adhering to these guidelines ensures that myself, and all others within this industry as professionals, can positively help contribute to it's ethical landscape.

In conclusion, being aware of the intersection between finance and technology means acknowledging that it is fraught with ethical challenges. From the management of risk and livelihoods, to the transparency of algorithms and the security of data, and especially when dealing with the newfound technological advances and access granted to AI, the decisions made by computer science professionals have further reaching consequences than even. I believe that it is our ethical duty to approach these challenges with responsibility and caution, making certain we can ensure that with our contributions to the finance industry, we can not only provide innovation, but also balance this with moral values, hopefully resulting in bringing benefits for society in general.

9.2 Plagiarism and Using AI

I felt that I had made my point clear with the section above, but I also wanted to explore briefly one other ethical issue that I think is paramount within not only my project and the computer science industry as a whole, but especially for university students.

The first thing I want to address if for the issue of plagiarism in academia and the industry, since it has far-reaching implications for students, educators, and professionals alike. The problem for touching on this with the context of computer science and software development, the is an incredible proliferation of readily available code and resources online, which can consistently blur the lines between inspiration and outright plagiarism. These forms of ambiguity pose significant challenges when trying to ensure the academic integrity of ones work, as well as misplacing genuine innovation if not handled correctly.

Plagiarism for computer science is not just limited to copying code without proper attribution; it can also encompass the unauthorized use of ideas, algorithms, and documentation when not correctly addressed or credited. The nature of programming and it's development, where certain solutions and patterns are common place, only helps further exacerbate the issue. Attempting to distinguish between commonly accepted practices and plagiarism requires a ever nuanced understanding of both the technical and ethical aspects of software development, becoming harder and harder with modern technological innovation producing more ways to circumvent this integrity.

Following on as such, Artificial Intelligence (AI) tools have introduced a new dimension for attempting to monitor and judicate plagiarism. It is a very conflicting subject, as on one hand, AI can be a powerful tool for learning, informational exploration and increasing the efficiency that a programmer is able to tackle development, helping aid students and professionals in understanding complex concepts and developing innovative solutions. On the other hand, the ease with which AI

can produce work that appears original raises questions about the authenticity of anyone's contributions.

The ethical use of AI in educational and professional settings all hinges on the individuals transparency and intentions with it. It's imperative to disclose the use of AI-generated content and if so, ensuring that the content is used as a supplement to one's own understanding and creativity, rather than as a replacement. Educational institutions and industries, taking influence from coding institutions, must develop clear guidelines on the use of AI tools, for the continued sake of academic integrity.

10 Bibliography

10.1 References

1. Alexander, C. (2008). *Market Risk Analysis: Value-at-Risk Models.* Hoboken, NJ: Wiley. [Online] Available at: `https://ebookcentral-proquest-com.ezproxy01.rhul.ac.uk/lib/rhul/reader.action?docID=416450`.
   This book covers various aspects that I want to approach in this project, such as historical simulation, Monte Carlo simulation and various forms of testing that could be highly useful for my project.
2. Arbuckle, D. (2017). *Daniel Arbuckle's Mastering Python: Build Powerful Python Applications.* Birmingham, England: Packt. [Online] Available at: `https://learning.oreilly.com/library/view/daniel-arbuckles-mastering/9781787283695/?ar=`.
   I chose this reference as it helps with python packaging, being able to turn a python code and all its GUI and libraries into a single executable file, which is something I want to be able to do for this project.
3. Choudhry, M. (2006). *An Introduction to Value-at-Risk.* Chichester: John Wiley & Sons Limited. [Online] Available at: `https://learning.oreilly.com/library/view/an-introduction-to/9780470017579/`.
   Covers similar topics to the first reference, but seems to go into more detail on specific issues that I may need to address in this project.
4. Duffie, D. and Pan, J. (2019). *An Overview of Value at Risk.* [Online] Available at: `http://web.mit.edu/people/junpan/ddjp.pdf`.
   A much shorter reference, it is a paper that covers the basics of Value at Risk, which I will be using to help me understand the fundamentals of the topic since I have been able to follow it better then the other references.
5. Föllmer, H. and Schied, A. (2016). *Stochastic Finance: An Introduction in Discrete Time.* Berlin: de Gruyter. [PDF]
   A recommended reference from my supervisor, it is a book that covers higher level concepts relating to my project but also provides an overview of stochastic finance in general, which I have seen to be useful in understanding the topic.
6. Hull, J.C. (2008). *Options, Futures, and Other Derivatives.* Upper Saddle River, NJ: Prentice Hall. [PDF]
   This is my main reference for the project, as it is the main textbook suggested. It has a relevant chapter on Value at Risk, which explores many of the different aspects that I will need to look into for this project.
7. Pritsker, M. (1997). "Evaluating Value at Risk Methodologies: Accuracy versus Computational Time." *Journal of Financial Services Research* 12: 201-242. [Online] Available at: `https://doi.org/10.1023/A:1007978820465`.
   Another short reference, this is a paper that compares the accuracy and computational time of various Value at Risk methodologies, which I will be using to help me understand the different methodologies and how I can apply them to my project in the most efficient manner.
8. Raman, K. (2015). *Mastering Python Data Visualization: Generate Effective Results in a Variety of Visually Appealing Charts Using the Plotting Packages in Python.* Birmingham: Packt Publishing Limited. [Online] Available at: `https://learning.oreilly.com/library/view/mastering-python-data/9781783988327/?ar=`.
   This reference covers various aspects of data visualisation, which I will be using to help me understand how to best display important information in a visually appealing way for my project.

9. Ulloa, R. (2015). *Kivy - Interactive Applications and Games in Python - Second Edition.* Packt Publishing. [Online] Available at: https://learning.oreilly.com/library/view/kivy-interactive/9781785286926/?ar=.
This reference is not being used for its content on game development, rather Kivy is a framework that can help create a GUI that works on both desktop operating systems as well as mobile operating systems, which I think I would like to explore the possibility of when developing the application.

10. Weiming, J.M. (2015). *Mastering Python for Finance: Understand, Design, and Implement State-of-the-Art Mathematical and Statistical Applications Used in Finance with Python.* Birmingham, England: Packt Publishing. [Online] Available at: https://learning.oreilly.com/library/view/mastering-python-for/9781784394516/.
This reference covers various ways of handling financial data within python, which I will ensure to help me understand how to best handle the data I will be using within my project.

11. Wasserstein, R.L., & Lazar, N.A. (2016). *The ASA Statement on p-Values: Context, Process, and Purpose.* The American Statistician, 70(2), 129-133. [Online] Available at: https://www.tandfonline.com/doi/full/10.1080/00031305.2016.1154108.
This reference is crucial for understanding the nuanced interpretation and use of p-values in statistical hypothesis testing, since it is relevant for the back-testing aspect of my project involving the VaR model validation.

12. Casarin, R., Chang, C.-L., Jimenez-Martin, J.-A., McAleer, M., Pérez-Amaral, T. (2013). *Risk management of risk under the Basel Accord: A Bayesian approach to forecasting Value-at-Risk of VIX futures.* Mathematics and Computers in Simulation, 94, 183-204. ISSN 0378-4754. [Online] Available at: https://doi.org/10.1016/j.matcom.2012.06.013.
This reference is useful for understanding the Bayesian approach to forecasting Value-at-Risk of VIX futures and its relevance to risk management in financial institutions.

13. Kivy. (2023). *Kivy Documentation.* [Online] Available at: https://kivy.org/doc/stable/.
This reference is the official documentation for the Kivy framework, which I consistently used to aid me in my understanding and ability to develop the GUI to such a high degree.

14. Boatright, J.R. (2010). *The Ethics of Risk Management: A Post-Crisis Perspective.* Business Ethics Quarterly, 20(4), 621-642. [Online] Available at: https://www.bbvaopenmind.com/en/articles/the-ethics-of-risk-management-a-post-crisis-perspective/.
This reference is a paper that explores the ethical considerations in risk management, which I utilised to help me understand the ethical implications of risk management in financial institutions.

15. Weis, V. (2023). *How Financial Services Firms Can Inform Decision-Making with Ethical Data Handling.* Atrium. [Online] Available at: https://atrium.ai/resources/how-financial-services-firms-can-inform-decision-making-with-ethical-data-handling/.
This reference is another article that discusses ethical data handling in financial services, it also touches briefly onto the use of AI in the industry, leading me to understand and explore the further ethical impact that AI has in finance.

16. Hong, L.J., Hu, Z., Liu, G. (2014). *Monte Carlo Methods for Value-at-Risk and Conditional Value-at-Risk: A Review.* ACM Computing Surveys, 46(2), 1-29. [Online] Available at: https://dl.acm.org/doi/pdf/10.1145/2661631.
This is the reference paper that I used to help explain and comprehend the Monte Carlo Simulation method, a key aspect within the most important aspects of my projects development cycle.

17. Holman, J.O., Hacherl, A. (2022). *Teaching Monte Carlo Simulation with Python.* Journal of Statistics Education, 30(1), 33-44. [Online] Available at: https://www.tandfonline.com/doi/epdf/10.1080/26939169.2022.2111008?needAccess=true.
This reference is a paper that discusses teaching Monte Carlo simulation within Python, helping me understand how to best implement this method for my projects language of choice.

18. Trinidad, B. (2021). *A User-Centered Design Approach to Improving Financial Intelligence and Analytical Software.* Universidad Politécnica de Madrid. [Online] Available at: `https://oa.upm.es/68940/1/TFM_BEATRICE_TRINIDAD.pdf`.
This reference is a thesis that discusses user-centered design in financial software, which I briefly used to help guide me on a few user interface design decisions.

10.2 Literature Review

10.2.1 Bibliography Summary

The bibliography that I have provided helps me to summarise some of the key literature surrounding Value at Risk (VaR), encompassing foundational theories, methodological advancements, and practical aspects of VaR in financial risk management, as well as touching on python and its plethora of libraries. It integrates perspectives from seminal texts, empirical studies, and modern computational approaches.

Foundational knowledge on VaR can be derived from Hull (2008) [6] and Choudhry (2006) [3], focusing on its mathematical underpinnings and historical context. Alexander (2008) [1] provides detailed insights into various VaR methodologies, complemented by Pritsker's (1997) [7] empirical analysis on their competence.

The practical implementation of VaR models using Python is guided by Arbuckle (2017) [2] and Weiming (2015) [10], highlighting Python's utility in financial data handling and visual representation. Ulloa (2015) [9] adds a modern perspective with multi-platform graphical user interface applications that are applicable to help display and increase the accessibility for financial risk assessments associated with VaR.

Recent discussions around VaR and statistical analysis are enriched by Wasserstein and Lazar's (2016) [11] exploration of p-values in statistical hypothesis testing, relevant for VaR model back-testing. Casarin et al. (2013) [12] introduce a Bayesian approach to VaR, showcasing the evolving methodologies in response to financial market dynamics.

Overall, I think the references I have chosen really help encapsulate the multifaceted aspects of VaR found in the modern age, highlighting its theoretical, methodological, and practical scale. It highlights the continuous evolution in VaR modelling, helping set the context for my further research into this field, as well as helping enhance my knowledge for user interface design and general industry standard financial risk software development.

10.2.2 General Literature

The general literature within the industry around the development of Python software found online, particularly for applications in finance and risk management, helps to exemplify its simplicity, efficiency, and robustness for being utilised for the medium. Python's apparent ascent as a preferred programming language for financial software development is largely attributed to its easy readability, expansive library ecosystem, and the persistently supportive community surrounding it. As noted when I was looking into Van Rossum (1995) [1], the creator of Python, the language's design philosophy helps to emphasise code readability and syntax simplicity, allowing developers to express concepts in fewer, more readable lines of code compared to other programming languages.

Furthermore, the work of Oliphant (2007) [2] on NumPy, an essential library for the majority of all numerical computations within Python, has significantly propelled its adoption into data-intensive and computational fields such as finance. Oliphant details how NumPy's array objects and optimised functionalities for mathematical operations lay the groundwork for more specialised financial and statistical libraries, which has resulted in packages such as pandas and SciPy, this scalability python provides making i such an indispensable tool for the financial analysis and algorithmic exploration.

With McKinney's (2012) [3] seminal work on pandas, a Python library providing high-performance, easy-to-use data structures, and data analysis tools, the literature surrounding the integration of Python in finance has been extensively explored in both academic and industry contexts. This only serves to once again underscore the language's capability to address the complex data manipulation necessary to analyse the needs of financial markets in a way that few other languages are capable of. McKinney notes that Pandas' development was partly motivated by the finance industry's requirement for powerful analytical and data manipulation tools that are both flexible and accessible, being implemented within Python only helping to further that point.

In addition to libraries, Python's role in financial modelling and risk management can be seen detailed by Hilpisch (2014) [4], who highlights Python's application in quantitative finance, from derivatives modelling all the way to risk management strategies. The book exemplifies how Python's incredible ecosystem, encompassing libraries such as MatPlotLib for data visualisation and IPython for interactive computing, help to facilitate an end-to-end workflow for financial analysis.

To follow up from my previous observations, may I note how `O'Reilly` Media has played a pivotal role in my dissemination of knowledge across various domains, including the computer science, financial, and data analysis aspects of my research. With its rich history of publishing comprehensive, well-authored books, `O'Reilly` has continue to establish itself as a beacon for learners and professionals seeking to enhance their understanding and skills. Their commitment to quality content, coupled with the ease of access I can take advantage of through digital publishing and online learning platforms, have undeniably helped contribute to the widespread accessibility of these educational resources.

Moving on to exploring financial specifics, the calculation of Value at Risk (VaR) perhaps exemplifies one of the most critical aspects of risk management, helping to quantify the potential loss in value of a portfolio over a defined period for a given confidence interval (as I have mentioned throughout my report). Jorion (2007) [5] writes comprehensively about the metric, elucidating its foundational methodologies and concepts, showcasing its applications in many circumstances in financial risk management. Jorion's work emphasizes VaR's singular role in quantifying market risk and showcases its adoption by financial institutions for compliance and internal risk management.

Among the various methods for calculating VaR, the historical simulation approach, method (variance-covariance) approach, and Monte Carlo Simulation are all predominantly used, all of which I have delved into and thoroughly explored. For Monte Carlo, it's flexibility and applicability to complex financial instruments make it a valuable tool for risk assessment, albeit with computational intensity that Python's scientific libraries coupled with intelligent programming help mitigate.

To conclude, this wide range of literature detailed above and within my reports bibliography helps to underscore Python's significant role in financial software development, a role that I have had the

pleasure of utilising myself, particularly with helping produce competent applications for VaR calculation. The language's simplicity, combined with its powerful libraries and the active development community, positions Python as the key enabler in innovation for risk management software. As the financial industry continues to evolve amidst growing data volumes, complexities and technological advancements, Python's relevance is only expected to increase further.

10.2.3 Literature Review References

1. Van Rossum, G. (1995). *Python Reference Manual.* Amsterdam: Centrum voor Wiskunde en Informatica. [Online] Available at: `https://stuff.mit.edu/afs/athena/course/18/18.417/doc/pydocs/ref.pdf`.
   This reference provides the creators insight into Python's design philosophy and its emphasis on readability and simplicity, which make it a fantastic tool for use within financial software development.
2. Oliphant, T.E. (2007). *Python for Scientific Computing.* Computing in Science & Engineering, 9(3), 10-20. [Online] Available at: `https://ieeexplore.ieee.org/abstract/document/4160250`.
   This reference discusses NumPy's role in scientific computing and its impact on Python's quick adoption into finance due to its data-intensive capabilities.
3. McKinney, W. (2012). *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython.* Sebastopol, CA: O'Reilly Media. [Online] Available at: `https://books.google.co.uk/books?hl=en&lr=&id=v3n4_AK8vu0C&oi=fnd&pg=PR3&dq=McKinney,+W.+(2012).+*Python+for+Data+Analysis:+Data+Wrangling+with+Pandas,+NumPy,+and+IPython.*+&ots=riEGbmwwnC&sig=6yVYKAcKhqcwpXXLiib29nFF2tc&redir_esc=y#v=onepage&q&f=false`.
   This reference explores pandas' functionalities for data analysis and manipulation, highlighting Python's utility for financial data processing.
4. Hilpisch, Y. (2014). *Python for Finance: Analyze Big Financial Data.* Sebastopol, CA: O'Reilly Media. [Online] Available at: `https://learning.oreilly.com/library/view/python-for-finance/9781491945360/`.
   This reference showcases hoe applicable Python can be for quantitative programming, emphasizing its role in helping compute financial modelling and risk management.
5. Jorion, P. (2007). *Value at Risk: The New Benchmark for Managing Financial Risk.* New York, NY: McGraw-Hill. [Online] Available at: `https://merage.uci.edu/~jorion/answer.pdf`.
   This reference provides a comprehensive look at Value at Risk, and how significant it has been for managing financial risk.

11 Appendix

11.1 Diary

FINAL YEAR PROJECT DIARY

28/9/23 - Start of Project
Today, I am starting this project. I have finally fully recovered from a health issue I was having at the start of term, but I am still dealing with an ongoing emotional issues that I will have to push through to complete this project successfully (heartbreak).

01/10/23 - Setting Up LaTeX
I have been able to get Latex working for my Documentation, after having various problems with the version of LaTeX I installed not working with VSCode. I finally used Latex Live and was able to

get it working, thus have created a test document to ensure it works. I have started researching and acquiring sources for my project, and will need to ensure I complete the Project Plan accordingly.

17/10/23 - Fixing my DIARY
Due to illnesses (that are still ongoing), I have not been able to properly update this diary to a satisfactory level. When my illness has rescinded, I will be able to start continuously updating this diary with my projects progress...

11/11/23 - Finally being able to get on with working
Have finally gotten illnesses and emotional issues under control, and have been able to start working on my project. I've used the sources I originally found to help me understand VaR for historical simulation and variance covariance model methods, whilst also using trial and error to create two test programs to test the methods. I need to start on the documentation for the project tomorrow with creating the latex for it and making sure to document my progress in understanding var as well.

12/11/23 - Setting Up Documentation
I copied over my documentation for the Project Plan, and used it as a baseline for me to establish what I need to be in my interim report. Taking brief reference from an email that Argyrios sent, and combining it with what I already had from my previous plan (timeline), as well as making sure I include everything needed from the mark scheme, I created the contents and headings for my documentation going forward, so I know what I need to add into each section up until the completion of the interim report. I now need to start populating the information before I continue to progress with the project too much, however for this week in my original plan I was supposed to research and understand back testing, however I think I will wait slightly so that I can catch up on documentation, since I really should have been documenting my testing on getting the VaR methods to work in the first place, so I will make sure to complete the prior sections (I think I'll skip on the introduction documentation at the moment too so I can focus more on getting the documentation done for the code, so that I can then progress with more coding).

13/11/23 - Back-testing
I said that I would do more documentation next, but I wanted to briefly read on back testing. Upon doing so, I saw that it didn't look to challenging, so I thought I'd quickly try and implement it for historical simulation. I was able to do so, which means I should be able to adapt it for the model method tomorrow quite easily, I also found that the back testing seemed to think my VaR calculated was accurate at the 5% level, but when changing it to 1% level it was not as accurate, so I will need to look into this further tomorrow as well, see if its to do with assuming that my returns are normally distributed, or if its something else.

14/11/23 - Combined Program for Both Methods, for Single Stock
At the start of the day, I implemented the back testing for the model building method, tested it to see if it worked, and saw that it would mainly still fail the back testing. I don't really know what I'm supposed to do with the back testing at the moment, but I will research more into it in the future. I then decided to make a program that I could input all the relevant details into to get VaR results, only accounting for a portfolio of just one stock type, I wanted to have it so the user could choose their one stock from a choice of multiple stocks, so I displayed the FTSE100, let the user choose that and input their portfolio amount, as well as other values, and at the end they would chose what VaR method they would want to use to calculate the VaR. The program displays the

VaR and back tests it afterwards, but almost 90% of the time it would fail the back test. I assume this is due to these methods not being the most robust for calculating VaR, or for I'm performing the back testing wrong, but regardless everything else in the program seems to work, and this program can help me in the future when I implement a GUI, my initial one will most likely only handle one stock type, so I can use this program as a base for that.

24/11/23 - GUI + Finalising some statistical analysis + Documentation
Have not been able to update the last week or so due to illness again, have gotten around to continuing to work on the project. I have been using a reference book to research on how to use Kivy for the GUI, I am giving myself a few days to understand and implement that. I have been in contact with my supervisor, and they have helped me in better understanding the statistical analysis that I need to do when back testing, I still don't 100% have a grasp of it, but I know that I will need to be able to utilise it better in the future. Finally, I think I've finalised what I need to be in the documentation, and I really need to start on it soon, so I will do do that soon too hopefully.

29/11/23 - Initial GUI close to completion
I had been struggling to get started on the GUI, as in I have been working through the book/recourse I had on it but I still didn't know where to start. So I experimented with what I had learnt and managed to create an initial design based on a sketch I made for the application. With this initial design, which was not populated with any functionality, I have been gradually adding bit by bit of what I need to be able to generate my single stock VaR results. I know it's not multi-stock but after being able to create this initial GUI design and learning Kivy, I'll be much more confident in creating further kivy applications in the future. The final thing I need to do for it now is to have it actually generate the VaR results, display that it has done back testing and some indication of the significance of the result, as well as text box validation so it is easier for the user to use. Then I will have to quickly create my presentation, which will involve slides explaining VaR, what my aims are for the project as well as a showcase of what my final GUI can do, I think, I will need to go over the specification for it in more detail tomorrow. Still a lot to do and I'll have to work as hard as I can over the next week to ensure I complete it all appropriately.

30/11/23 - Implemented VaR and Back Testing and fixed a lot of problems
I was having issues when I used my laptop to create the application, since it broke all the formatting. Upon testing around, I discovered that depending on the monitor size (since they have the same resolution the two displays I use), it will change the size of the kivy application. I didn't want this, so I found out using DPI aware will mitigate this issue and allow to be more accessible everywhere. I implemented my VaR code from the command line program, and it works well. I also discovered that many of the stocks don't work from the FTSE or maybe the trackers are out of date. Regardless, I'm thinking of maybe displaying a different list of stocks, but I will decide on that tomorrow. I have implemented back-testing, but I don't think its all working properly and I need to use tomorrow morning to get it to be represented in some fashion on the application. After that I think I am happy with the initial GUI design and I will begin work on the presentation, since with the GUI complete, I can make sure to be able to show it off.

02/12/23 - Completed GUI and Created presentation
After performing more test on the back-testing, I found that me thinking that it wasn't working properly was an outlier that gave a very large p-value, in general the back testing appeared to be very accurate and helped a lot with the statistical analysis and being able to disprove/approve the

hypothesis's proposed with each VaR. I also added better verification to the text inputs, so that when you write any numbers or letters that don't make sense, it will just default the values. I also had previously commented on not being able to access many of the other stocks with yahoo finance, but I had seen evidence of all the FTSE100 being listed on the yahoo finance website, so after doing even more research I discovered that a lot of the stocks have a .L listed with their ticker, so I got it so that every time it would fail to download a stock, it would attempt to re-download the stock with the additional '.L', and this managed to work for every single FTSE Stock, so now the program feels more complete now that every stock is accessible. I managed to add a few more bits of verification, to help with some of the stocks returning bizarre results, so they can be accounted for in the future if it happens as well, and to make sure my program never crashes. Finally, I was able to create my presentation, I am happy with how it looks even though I did have to rush certain sections but it still looks great and I should be able to describe everything well. All I have to do now is most of my documentation, I had been severely behind on it, so I'll see if I can catch up. Also comment on forgetting to use a different branch for the GUI.

06/12/23 - Documentation
I have spent the last few days working hard on the documentation. I regret not being able to do this as I progressed with the project, since now it will be hard for me to show the development of certain aspects properly. Regardless, I do not have much more time before this is due, so I wouldn't have been able to say/show everything I wanted for each section anyway. I am sad that I have so much that I need to show off/write about within my report(s), but I feel that due to the circumstances with my health and emotional issues this term, I was unable to create work that I feel is satisfactory to the standard I hold myself to. I've been able to mostly complete the first section of my documentation, I think, and I've gotten onto where I am finally talking about code. I want to include a lot of screenshots to show things visually, but using latex is a lot slower then word, with you having to save your pictures in a certain directory, and then having to reference then within latex, with word just letting you copy and paste in comparison. Maybe I would've been able to do more consistent documentation alongside my coding if I was able to do it within word, I will keep that in mind for next term, maybe use word whilst I go along with the development, and then transfer it over to latex at the end. I will continue to work on the documentation tomorrow, and hopefully I will be able to complete it all by the end of the day, so I can submit it and then perform my presentation the following day.

07/12/23 - Finale
I have finally completed the documentation, I am unhappy with the fact I had to miss out on so many sections that I had planned, but in the end I'm rather blown away that I was able to consolidate and present so much in such a small amount of time. I'm proud of the result, of this report and the code I created this term, but I do plan to spend the majority of my holiday catching up on the time I feel that I lost for my project throughout the term due to my illnesses and heartbreak. I need to compile all the programs I think are relevant for my submission, and put them all together within a zip file to be submitted. I am also dreading my presentation, as I know I will be quite sleep deprived for it, but I am happy with the slides I made, and I hope to be able to justify my project well, especially after I was able to justify it so well within the interim report/documentation. There are many changes that I want to make, but for now I will finally take a bit of a break to recover from the excessive amount of work that I have put in before the deadline to achieve this. Thank you for reading for now...
29/01/24 - Start of Term 2

I have discovered testing over the winter break that it displays properly on my normal monitor, but not on my laptop, even though I have fixed the resizing issues. I need to make sure to take screenshot references for this, and write about it within my FYP documentation.

05/02/24 - Fixing the issue from the last term
I worked out what the issue was with my program. Essentially, I had hard coded the structure of a lot of the layouts, by not using size_hints for everything, since that would keep everything consistent no matter what window size. Because of this, I need to change everything to implement size_hints, which helped fix overall layout, but there would still be problems with text and other objects that I had hard-coded on the Python side that I couldn't use hints for, such as font size. This is where sp (scale independent pixels) came in handy, since I could use this to change any of my hard-coded values into ones that scale (which I could've done to fix the previous problem but it was better for me to get used to using hinting with the change). After attributing the dp to all integer size values, I had essentially gotten the program to work on both of my monitors, so I decided to stop using the DPIaware command, just let my program adjust naturally. I think I want to have the window be resizable in the future, but for now I want to get the tabbing system working next. I also need to quickly fix the back-testing title label I have for it, since I know where I want it to be, its just hard to add it to the layout properly...

19/03/24 - Finished Researching Monte-Carlo simulation
After a busy month, with lots of other coursework and external factors, I have now finally been able to return to focusing on my project for the final stretch needed. I have now researched and understood the Monte-Carlo simulation, so I will start by creating a simple Monte-Carlo command line program to test that my knowledge lines up with creating it within Python. After that, I will start implementing tabs for the program, so I can move on to adding more utilities for the program, as well as implementing Monte-Carlo within it too.

23/03/24 - Implemented MonteCarlo
I created a separate command line application to test that I could properly get MonteCarlo Sim to work, which I was able to. I'm planning to implement it into the main program after I've done the tabbing system.

25/03/24 - Implemented Tabbing System and OOP + Folders
I managed to find a source that explained how screens work within Kivy, helping me set up a screen manager, and attempt to swap between multiple screens. At first, this absolutely broke my deliverable from last term when I ported it into a screen, but I realised that it didn't have the original box layout support that I had defined it with on the python side. So I had to implement that within the kv file and it seemed to work with the same functionality. I set up generic buttons for tabs at the top, but I don't like how they looks, so I will try and change them if I have time in the future. I like the custom animation that Kivy seems to have when switching between screens, but I don't like that it only moves in one direction, so I may also try and adjust it so that it will slide a certain direction depending on which screen you're navigating from to.
Finally, I decided that I wanted to make the project and coding more Object Oriented, to allow it to be easier to understand and maintain the codebase in the future. To do this, I realised I can easily import the python code into a main python file, which allowed me to separate all the screens I now wanted all into their own individual files, including last terms deliverable, which I have now called a VaRChecker (which I still need to implement Monte-Carlo Sim into). I also, after performing some

tests to make sure it worked properly, have been able to separate up the kivy (kv) files so they all communicate but are still separate files, so everything is split, communicates, and runs properly. I assume that when I compile this all into a working application in the end it will all be fine but I will have to see then. For now, I am going to start working on the Portfolio page as fast as I can, since I feel like it is one of the most important, complex features and I need it to work robustly.

28/03/24 - Started creating Portfolio Screen
Today I started working on the portfolio screen, I wanted to make sure the functionality and aesthetics of this part were satisfactory but I am bad at making the aesthetic work properly, so for now I'm just focusing on functionality. I used some conceptual sketches that I will probably add to my report, and I've made a rough version of what I want to screen to look like, based off of it. Its very early stage, so it doesn't look very good but I'm hoping to add to the functionality, so it can work sufficiently.

03/04/24 - More Portfolio Screen + Removing Rankings Screen + Caching storage
I found out you can use JSON.store to create a way to essentially cache portfolio information that I want to be retained within the program. This is amazing, as now I can store bits of information and keep it there, so the program can have initial stocks prices, and compare back to them whenever. I also realised that I just do not have the time to be able to create the 3 extra screens that I was planning to make for this term, so I'm just going to have to create 2, to be honest, the third screen I was going to make I didn't have a very good plan/idea for either, so doing it this way has streamlined the options that I have available, which is good, I'm feeling more confident that I can complete my goals.
The two screens I'm keeping are trends and portfolio (the main one), and I am not going to be doing rankings any more. I was having a problem with labels not displaying when being put on my stock buttons, but I fixed this by directly adding text to the button, although it doesn't look very good regardless so I'll definitely have to change it in the future. I was then able to finally integrate yfinance into the current code, there was a massive error to do with the .info section, when researching online, there were no ways to fix the issues other then pulling custom code from other people's GitHub to fix, but I realised that I can just download the stocks and grab the latest value, and that gave me the results I needed anyway, so bizarre. I wanted to get it so that I would get consistently updating information every minute for fresh stock data, whether I was looking at an individual stock or all stocks.
TO do this, I used a clock feature that would loop, and set up variable checkers to see if the other one was active or not, if so, it would turn off one clock and start the other, making sure only one can loop at a time and acquire fresh stock information. I've got it so it creates the totals, the returns and everything by comparing current prices to the initial prices, this is great as I can leave the stocks in their storage for now and it will only help show off my program more in the future. I need to get more information shown on the right hand side, like current individual stock price, rather then just the owned amount by timesing by the shares owned. I also have yet to implement any VaR stuff, so I will do that next as well.

04/04/24 - Adding Monte Carlo + Convergence
I first implemented MonteCarlo sim, this was initially a bit annoying since I had to re-download all the stocks so I could recreate the weightings of each of them, but I changed the structure I displayed everything with so that I generate the stocks once at the start, and then get them passed in for every other bit of information. I also first realised that I can't use monte carlo sim for single

stock simulations, which was fine since I could just re-incorporate my model method that I used within VaR checker. I got both of these to work so that when the total stocks were being displayed, monte carlo sim was used to calculate it, and model was used for the individual stocks, but I didn't like the amount of fluctuation that my monte carlo sim was displaying when I would get it to run 10000 simulations. It was also incredible laggy, I implemented a timer feature to see it took 4 seconds to generate every time.
After researching some more, I realised that numpy can generate all the simulations at once with one of its methods, returning it in a structured format, and then I can work out the rest with all the simulations prone, this was so much faster, taking 0.07 seconds for the same amount of simulations, 10000. But I still got the results that were too different for every set of simulations, so I decided to adjust the code again so that it would do multiple bursts of simulations, and depending on how close each burst was to the previous, only when it starts to converge, will I display the result. This convergence point indicates that the simulations are giving similar, and subsequently more accurate results towards a specific outcome, which is what we're looking for. I set the convergence point low (0.5%) to make sure the result was good, and made it so it can perform up to 100,000 simulations if need be, but actually this resulted in my simulations usually being a lot faster, and only sometimes slower then usual, but from what I could to tell, a lot more accurate. Finally, I decided to verify how I thought by using Matplotlib to display results of a simulation that continues to run until 100000, thus giving me a good overview to see how accurate it becomes when converging on a certain point over time, and it was good to see this implemented, as the first graphical change I had shown. This was great, but its not a section that I need here on this screen, so I will leave it for now and probably implement it within the trends screen in the future. It got me thinking that I should do the same for the back-testing within my var-checker screen as well, so I will try and do all of that when I work on the next screen hopefully tomorrow. I made a few UI adjustments, fixed up how exiting the pop-up worked and a bunch of other functionality things like being able to click on an invisible button when you select a specific stock to take you back to the total stock display. Overall, a big day for getting back-end stuff to work, now I just need to get this section to look visually better, add a little more information, and I had forgotten to add a way to delete stocks so I definitely need to implement that. I'm aiming to get started on the next screen tomorrow, so hopefully that is how it turns out.

05/04/24 - StockNameFinder, Delete Stocks, and Portfolio Screen Functionality Completion
I started today by being tired of displaying tickers for everything, its practical because its how I want them to be added into my portfolio, but I wanted to find a way to get the name to display it, it would look way nicer. To do this, without using yfinance.info since its broken, I tried web scrapers, first pandas since it can find info in tables in pages, but I couldn't find any pages that would get me the relevant info. So instead, I used beautiful soup, which would help me, when I had sent a request to a webpage, find certain html elements in the page, which let me make requests to specific yahoo websites and look for the html code containing the title of the stock on every page. This could only be done since the ticker for each stock is part of the url, which was perfect since they'd all match as its how I get my stock data within python anyway. With this, I was able to retrieve most of the names for my stocks, but it wouldn't work for certain ones for foreign stock exchange, and I fixed it by adding a user-agent to have my request mimic a browser, which seemed to fix the situation. I could now display the names on the stock buttons, and I also got it so that whenever the stock totals (for specific or initial) are displayed with their looping background app refresh, it would also run the load stocks function, passing in the downloaded stock data, so I could

then display the current prices of all the stocks without having to re-download them again every time, having everything work together at the same time. Then I added the delete stock functionality, this wasn't too bad, as it just involved me pushing the button originally, and it would see what specific stock is being stored and delete it.
But since this is deleting, I wanted to add extra verification, so I moved stuff around, had it so a popup appears, the screen info is passed into the class so it can run the initialStockTotals only when a stock is deleted, and made it run all fluidly. This looked great and I don't think I need to change anything about this in the future, it works well so I'm happy. Finally, since this button would disappear whilst I was on the totals page, since it would only need to delete a specific stock when its selected, there was a gap on the screen when it was hidden. I felt that, since my project is about VaR, even though I have a very good VaR working for the screen, I need to give the user the ability to adjust it to some extent, and I think initiating another popup gives me the most freedom to implement and have this happen, since I understand it more after making 2 already.
I wanted to have two buttons on top of each other that would boolean change which one is active or not at any given time, but this wouldn't let me click one of them, so instead I made it the same button that would send you to a function whenever you clicked on it, and depending on the text at the time (which changed depending on initial or specific stocks), it would perform a different action, either the previous delete functionality, or the new VaR adjustment functionality. The new adjustment one would take you to a new popup, that would have the calculator class passed into it, the one that's initialised and used within the portfolio screen, and you could directly change the time horizon and risk level being used for the VaR. I need to add a lot more verification, finish up that popup and add a lot more visual improvements and quality of life, but functionality wise the screen is looking fantastic.

06/04/24 - Finishing Portfolio and Starting Graphs Screen
I started today by finally getting around and doing proper verification for my inputs. I started with the save stock input, since it needs to check if you're passing in a valid ticker and a bunch of other stuff. This wasn't too hard, since I had verification set up for a similar kind of thing for my previous screen, but to get the logic working perfectly it took a while and some nesting. I also wanted to give visual feedback on entering something wrong, where I discovered the animations feature for kivy, which essentially transitions from one colour to another, for me at least. This was great, because I could quickly get red to flash up and fade, showing they got it wrong. I was having so much trouble with focus shifting not working, not letting me tab from one input to another, but I was able to get it so that it could do that when I clicked the enter key which is good enough for now. Because I liked the animation feature so much, I wanted to add it as a small background transition for how the portfolio looks when going from portfolio to specific stock totals, which looks smooth and helps the user understand whats happening a bit better.
I wanted to find a way to explain VaR a bit better, so I used the adjustment pop-up for it to write out and explain it, colour coordinating the values displayed to be the same as the text inputs, making it even easier for the user to see what they need to input. I also added verification for these as well, which was even easier to do, since I found out you can use input_filtering, so I would only let people enter integers for these inputs. All I really needed to do for my screen then was to just fix formatting, I did a bunch of stuff like stopping the stocks from being buttons, instead they're a grid-layout with two columns, so I could wrap the text to be side by side based on the ":", since I thought this would look the most visually appealing, which it does. I had to give them an on click/on press whatever it is, so they could still function like before.

I changed a bunch of colours all day, going back and forth between different visual ideas, adding boldness and underlining, making the top right have its own border and section, adjusting how close the right and side information is to each other and the left hand side etc. It was all just a bunch of formatting changes, so I think everything looks a lot better now, but I'm not very good at making stuff look pretty, but it will have to do for now. Very proud of the functionality, I added a massive stock name to the portfolio and the app would still display it correctly, works robust. Finally, I've started on the next and final screen that I want to complete for this project. It was originally trends but since I'm not only going to have trends display don the screen, I thought I would change it to be called graphs now.
I got the original layout working, after I had to look online to find a way to get the imports for having matplotlib stuff within my kivy applications working, they gave me a pip-install and a different way to import the kivy gardens module, but with it working, it all seems to be good. I have a basic graph, displayed, with a scrollable bar underneath, with stock images being used for each option. I wanted to change up the style of having the up and down scrollable sections in my previous screens, so I like how this is, and honestly if I can get this section looking nice, I think I can breeze through the logic of it tomorrow, and finish up, ready to start on the write-up properly by Monday, I HOPE :))

07/04/24 - Graphs Screen
I made an example second graph and tried to add it to the same graph area, but I didn't remove the previous one, so they all got compiled together and it looked very comical haha, they would squish up. Easily solved by adding a clear widget which I had forgot to. I then knew I wanted to get it so that when you hover over points, it would display what those points are, since I knew I would want to show a list of stocks and, to be able to tell which stock is which, I want to display it when you hover over it. This is ambitious, but I really wanted to add the functionality. Unfortunately, its not built into the kivy side of matplotlib with garden, compared to just the matplotlib side, so I was struggling to get anything to work effectively. After doing some research online, I found a single page that had an answer that said would work, so I implemented it and adapted it based on what they had done, and, with a lot of trial and error, I finally got it functioning properly. This looked great, its not 100% since if you have a graph that has a lot of value near the top, it can sometimes cut into the title and be impossible to see, but I think I'll fix that later if necessary. I was creating a graph every time I had a function, and knew that this wouldn't be efficient, so I took out the creating a graph logic, and made it its own method, making sure to pass the information needed in when it is called.
This was great because I knew I would have a lot of big complicated code within most graph sections, that I could just call create graph and have that handle the information properly. I essentially created a factory design pattern I think :D I then tired to implement the Monte Carlo Simulation Analysis Visualisation for Convergence that I had made as a method in my Portfolio file, within the VaR Calculators class, but I didn't want to run it from there, I wanted to have all the graphical logic be within the graphs file, so I moved it over. I still needed information from my portfolio page since I was attempting to just create a method to pass it in, so I found a way to call the portfolio and have the information passed in and working for the code, which I then just displayed the values of, working perfectly, I even created a temporary save of the downloaded stocks, so I could just use that data rather then call for stock data again. However, this code, for it to be able to display the amount of simulations I want to show the convergence visually, is incredible slow, making the program freeze for over 10+ seconds every time its clicked. I was so

annoyed by this, so I wanted to at least be able to use the rest of my program whilst I waited. To do this, I knew I had to incorporate threading/asynchronous programming
logic, so I originally tried asyncio, but could not get it working at all, even after updating all my files to be running on the latest version of kivy (I originally had it on 1.9.0 I think became the originally guide that taught me kivy used that), so I decided to use normal threading instead, and it worked, but I needed to use a sleep method to have my code logic run slow enough to even make the rest of the program barely usable whilst it ran. It would also now take a minute, all highly un-optimal but I couldn't do anything about it if I wanted my simulations to look good. This was also annoying since when the threading was running, I couldn't leave the application without it crashing, so I implemented a check to see if I'm trying to close the application, and if so, stop threading. I knew I would want to call portfolio multiple times in my codes future, so I made it a property, that will handle the logic, since it would need to happen after all the screens were initialised, so I can't have it as like a "self" variable, but I also did an on_enter thing so maybe I can actually, I may try and change that in the future... I disliked the scroll wheel that my portfolio had, so I made a custom one, I need to add it to my original VaRChecker.
I also took the stock download and displayed a past theoretical total if I had kept these amount of shares with stock pricing from the last 500 days downloaded. This looked great but would display the numbers at the bottom in a confusing way, and I got stuck forever trying to fix it, since it defaults to displaying the numbers from 0-x, but I implemented a check to flip the axis which finally seemed to work. There were also a bunch of gaps between points on the graph, after printing out I saw there were nan values, so I created some code to check for this and create assumed values that make logical sense to me and look good on the graph. I also wanted to display pounds and other symbols with the hover over, so I added that as something saved and shown as well. On a whim, I decided to research a way to get my monte carlo sim to run a bit faster, since it was horrifically slow without using the convergence stopping method I was using in my portfolio code, upon testing each section of it for time taken, I realised it was mainly the for loop nested inside the outer for loop, the one calculating returns.
Seeing that it was done with numpy, I realised that I can implement the built in vectorisation to calculate this faster, so after adapting my weightings to fit the shape of my optimised simulation for them to be summed together, it got it to achieve the same results without the loop. This made the simulation honestly 100 times faster, which is great now but embarrassing for my previous code, so I'm glad I looked into this. Now it runs extremely fast on both instances of the simulation, so even the threading isn't necessary now but I'll still keep what I have, but I won't add threading to anything else at the moment. And after checking, my VaR results are the same, so everything is looking great in that regard. Finally, I created a graph that would only appear when clicked if you had selected a specific stock (added a check in portfolio for this) and it would display that stocks single share price over time, which is a standard thing for these applications to have, so I liked how it looked. I also wanted to make sure that my final value for this and the portfolio graphs matched the ones being displayed on the portfolio page, so I hard coded it. I'm annoyed that I haven't finished up on this section, since I'm worried about having enough time on the report, but I reckon its still doable for now, so I wish myself luck for the future.