

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	4
2	Chapter 1 - Introduction	6
2.1	What is VaR?	6
2.2	Aims and Goals	7
2.3	Milestone Plan	8
2.3.1	Timescale & Timeline	9
3	Chapter 2 — Understanding VaR	12
3.1	Historical Simulation	12
3.2	Model Building (Variance-Covariance)	13
3.3	Coding VaR Methods	14
3.3.1	Command Line — Historical Simulation	14
3.3.2	Command Line – Model Building (Variance-Covariance)	15
3.4	Back-Testing	17
3.5	Combining Implementations	19
4	Chapter 3 — Graphical User Interface	22
4.1	What is Kivy?	22
4.2	Conceptualisation of Initial Design Ideas	23
4.3	Initial GUI Creation	24
4.3.1	First Iteration	24
4.3.2	Final Iteration	26
4.4	Completed Initial Design	26
4.4.1	Enhanced Application Structure	26
4.4.2	Dynamic Stock List Generation	27
4.4.3	Value at Risk (VaR) Calculation	27
4.4.4	Back-testing Functionality	27
4.4.5	User Input Handling and Validation	28
4.4.6	Kivy Layout and Styling	28
4.5	Reflectional Adjustments	30
5	Chapter 4 — Portfolio Creation	30
5.1	Monte Carlo Simulation	30
5.1.1	Command Line — Monte Carlo Simulation	32
5.2	Kivy Screen Manager	33
5.2.1	Tabbing System and Workspace Structure	34
5.3	Developing a Portfolio Screen	36
5.3.1	Initial Structure and Pop-Up	38
5.3.2	Storing Stocks and Calculating Portfolio Value	39
5.3.3	Background App Refresh and Race Conditions	41
5.3.4	Value at Risk Calculators	42
5.3.5	Finding Company Names and Back Button	44
5.3.6	Improving Monte Carlo Simulation	46
5.3.7	Deleting Stocks	47

5.3.8	Adjusting VaR Parameters	48
5.4	The Finalised Portfolio Screen	50
6	Chapter 5 — Graphical Visualisation	50
6.1	Convergence Analysis	50
6.2	Matplotlib within Kivy	51
6.3	Creating the Graphs Screen	51
6.3.1	lots of these	52
6.3.2	lots of these	52
6.3.3	lots of these	52
6.3.4	lots of these	52
6.4	The Finalised Graphs Screen	52
7	Chapter 6 — Completed Deliverable	52
7.1	Features and Expected Use	52
7.2	Final GUI Design	52
7.3	P-Values and Graphical Analysis	52
7.4	Adaptations for Packaged Release	52
8	Chapter 7 — Software Engineering	52
8.1	Object Oriented Programming	52
8.2	Design Patterns	52
8.3	Testing and Documentation	55
8.4	Version Control (Git)	57
8.4.1	Branching and Tagging	57
9	Chapter 8 — Critical Analysis	57
9.1	Interim Evaluation	57
9.2	Final Report Evaluation	58
9.2.1	Coding Achievements and Success	58
9.2.2	Difficulties and Improvements	58
9.3	Conclusion	58
10	Professional Issues	58
10.1	Plagiarism and Using AI	58
10.2	Finance Industry Ethics	58
11	Bibliography	59
11.1	References	59
11.2	Literature Review	60

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:

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

Where:

N : Time horizon (in days)

X : Confidence level (in percentage)

L : Loss distribution over N 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 always 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 industry principles in regard to software engineering.

The ambition of this thesis 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 <ul style="list-style-type: none"> • 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 <ul style="list-style-type: none"> • Complete Project Plan • Continue researching VaR and Python • Begin project coding
Week 5–7	Coding and Data Preparation <ul style="list-style-type: none"> • 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 <ul style="list-style-type: none"> • Investigate methods and techniques for VaR back-testing • Start integrating back-testing into the project
Week 9–10	GUI Development <ul style="list-style-type: none"> • 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 <ul style="list-style-type: none"> • 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 <ul style="list-style-type: none"> • 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 <ul style="list-style-type: none"> • Start implementing the Monte Carlo simulation method and continue derivative implementation • Start researching Eigen & Cholesky decomposition to allow for however many stocks are needed within a portfolio
Week 5–7	GUI Finalisation <ul style="list-style-type: none"> • 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) <ul style="list-style-type: none"> • 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 <ul style="list-style-type: none"> • 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.

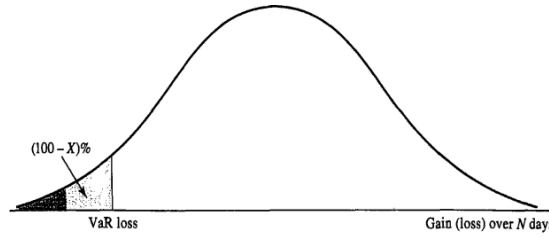


Figure 1: The Value at Risk (VaR) is calculated from the probability distribution of the variations in the portfolio's value, where gains are represented as positive values and losses as negative. The confidence level for this calculation is set at X%. [4]

- 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 out 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:

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

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 python's 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]))

```

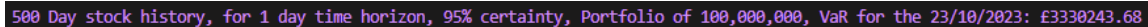
I then created an empty array called `diffs`, which would be used to store the percentage differences of the closing prices of the stock, followed by a `for` loop to iterate through each element of the `closes` array, calculating said percentage difference between the current element and the previous 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 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 than 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 values 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 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 your model has previously predicted the correct VaR, or more so, how many times they have predicted the VaR wrong. To do this, let's 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, let's 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.

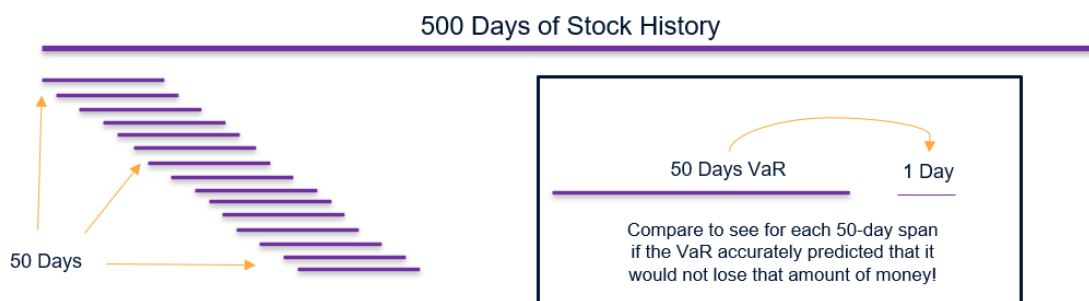


Figure 4: Back-Testing Diagram (Source: My Presentation Powerpoint)

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 code 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 than 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 Monet 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.

I have neglected to mention how the Time Horizon is 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:

$$N\text{-day VaR} = 1\text{-day VaR} \times \sqrt{N} \quad (3)$$

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.

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

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 a natural user interface (NUI). It is highly 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 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.

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 manner, 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.

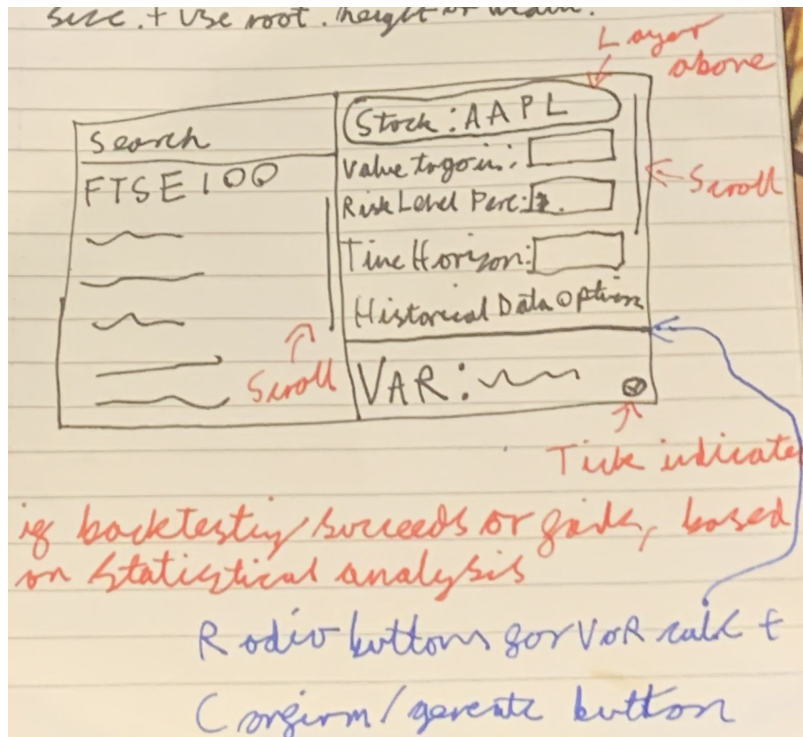


Figure 7: Initial GUI Design Sketch

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 and text input fields that are displayed in the GUI. The labels are populated with hopefully be populated 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()
```

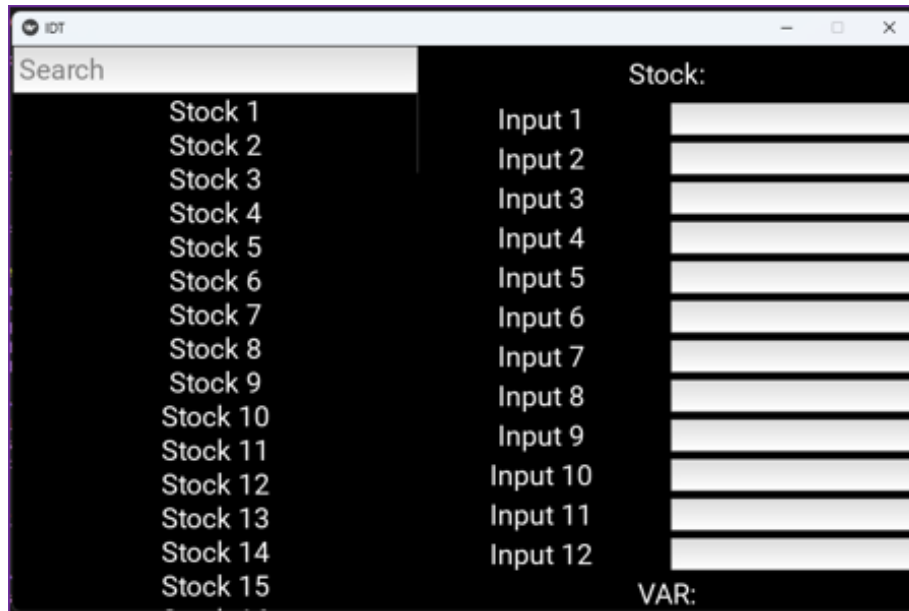


Figure 8: Initial Design Test - First Iteration

4.3.2 Final 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 has been expanded to include new properties and functionalities that support a richer user experience and more sophisticated data analysis capabilities:

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

These new properties include a default portfolio value, risk level percentage, time horizon for analysis, simulation method preference, and the currently selected stock ticker. This setup allows for a dynamic and customizable analysis based on user inputs.

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 code snippet showcases the application’s ability to fetch and display a list of companies from a live data source (Wikipedia), with each company represented as a button within the user interface.

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:

```
def generateVaR(self):
    ...
    stock = yf.download(self.currentTicker, startDate, endDate).tail(500)
    ...
    if self.simMethod == "Historical":
        VaR = np.percentile(closeDiffs, self.rlPercent)...
    else:
        VaR = (-self.portfolio*norm.ppf(self.rlPercent/100, ...))...
    ...
    setattr(self.valAtRisk, 'text', " VaR: £" + VaR)
```

This function demonstrates complex financial analysis by downloading stock data using ‘yfinance’, calculating percentage changes in the stock’s closing price, and determining the VaR based on the user’s selected risk assessment method.

4.4.4 Back-testing Functionality

The ‘backTest’ method evaluates the performance of the VaR calculation against historical data to assess its accuracy:

```
def backTest(self, stock):
    ...
    for i in range(1, len(stock) - adjust - 1):
        ...
        if VaR > nextDay:
            count += 1
        ...
    if pValue > self.rlPercent/100:
        ...
    else:
        ...
```

This section of the code uses historical stock price data to perform a back-test, comparing calculated VaR values against actual market movements to generate a statistical p-value representing the model's reliability.

4.4.5 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):
    ...
    if varName == 'portfolio':
        ...
    elif varName == 'rlPercent':
        ...
    else:
        ...
    self.populateInputs()
```

This function validates user inputs for the portfolio value, risk level percentage, and time horizon, applying constraints and default values as necessary. It showcases the application's robust error handling and user feedback mechanisms.

4.4.6 Kivy Layout and Styling

The Kivy language file (.kv) defines the layout and appearance of the application, separating the design from the Python logic:

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

This .kv file segment outlines the application's main layout, demonstrating the use of ScrollView for the stock list and BoxLayout for organizing interface elements. This separation of concerns facilitates easier maintenance and updates to the UI without altering the backend logic.

Conclusion: The final iteration of the application incorporates advanced data processing, user interaction, and financial analysis features. Through detailed explanations of key code sections, we have illustrated how the application leverages Python's capabilities and Kivy's framework to provide a rich user experience and robust financial analysis tools. Further elaboration may be required to fully cover all aspects of the code's functionality and design decisions.

Youtube Video: Presentation Demonstration For Final Initial Design - FYP

Stock List: FTSE100

3i
Admiral Group
Airtel Africa
Anglo American plc
Antofagasta plc
Ashtead Group
Associated British Foods
AstraZeneca
Auto Trader Group
Aviva
B&M
BAE Systems
Barclays
Barratt Developments
Beazley Group

No Selected Stock

Enter Portfolio Value (£):

Enter Risk Level Percentage (%):

Enter Time Horizon (No. of Days):

Select Method:

Historical
Model

Current Info Given (Default)
Portfolio Value: £100,000,000
Risk Level Percentage: 5%
Time Horizon: 1 day(s)

Back-Testing

Generate VaR:

Figure 9: Initial Design Test - Final Iteration 1

Stock List: FTSE100

informa
International Airlines Group
Intertek
JD Sports
Kingfisher plc
Land Securities
Legal & General
Lloyds Banking Group
London Stock Exchange Group
M&G
Marks & Spencer
Melrose Industries
Mondi
National Grid plc
NatWest Group
Next plc

Stock: LSEG

Enter Portfolio Value (£):

Enter Risk Level Percentage (%):

Enter Time Horizon (No. of Days):

Select Method:

Historical
Model

Current Info Given
Portfolio Value: £65,564,196
Risk Level Percentage: 1%
Time Horizon: 10 day(s)

Back-Testing

Generate VaR: £6,586,092

PASSED: 17.0%
(p-value)

Figure 10: Initial Design Test - Final Iteration 2

4.5 Reflectional Adjustments

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 averages 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 stock's 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.

Given a portfolio consisting of n assets, let $r = (r_1, r_2, \dots, r_n)$ 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.
- $w = (w_1, w_2, \dots, w_n)$ 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:

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

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:

$$\{\tilde{R}_1, \tilde{R}_2, \dots, \tilde{R}_N\} = \{w \cdot \tilde{r}_1^T, w \cdot \tilde{r}_2^T, \dots, w \cdot \tilde{r}_N^T\} \quad (7)$$

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:

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

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:

$$VaR_{\text{Monetary}} = P \times VaR_\alpha \quad (9)$$

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.

```
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("{:,.2f}".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 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 the different 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:

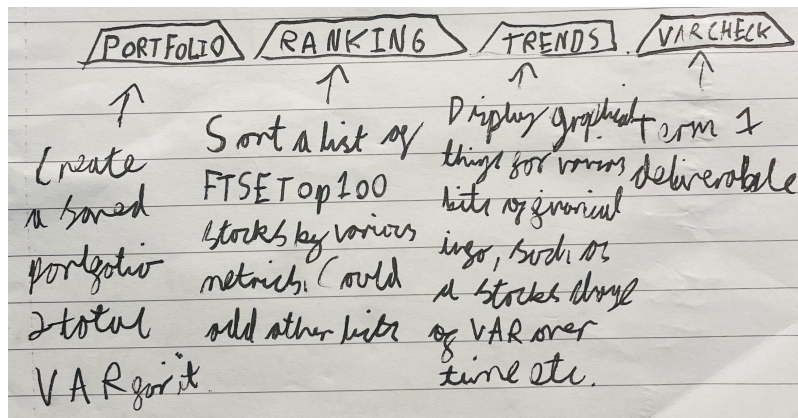


Figure 11: Tabbing Conceptualisation for Switching Between Screens

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 constraints 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
```

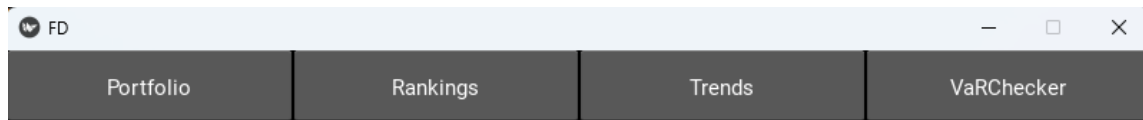


Figure 12: Tabs Implemented into the Program

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 its 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:

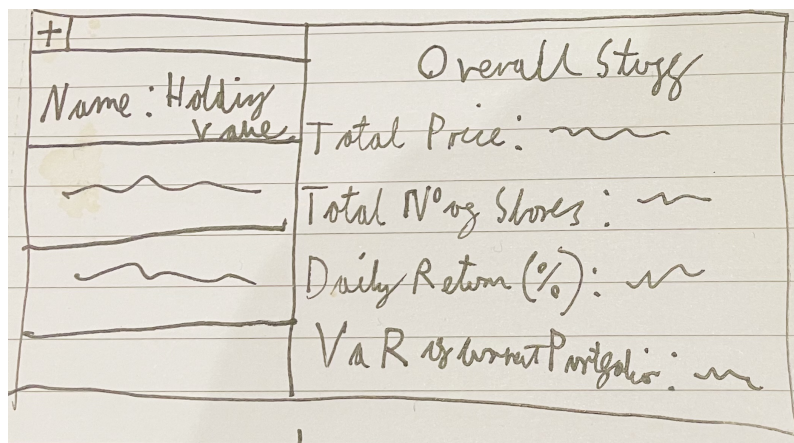


Figure 13: Draft Portfolio Screen - Top

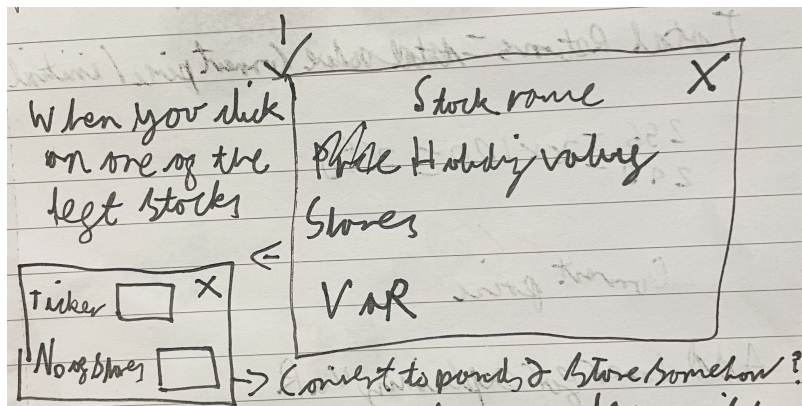


Figure 14: Draft Portfolio Screen - Bottom

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 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 to inputs for ticker (since this is how yfinance would download it) and no's 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 projects 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

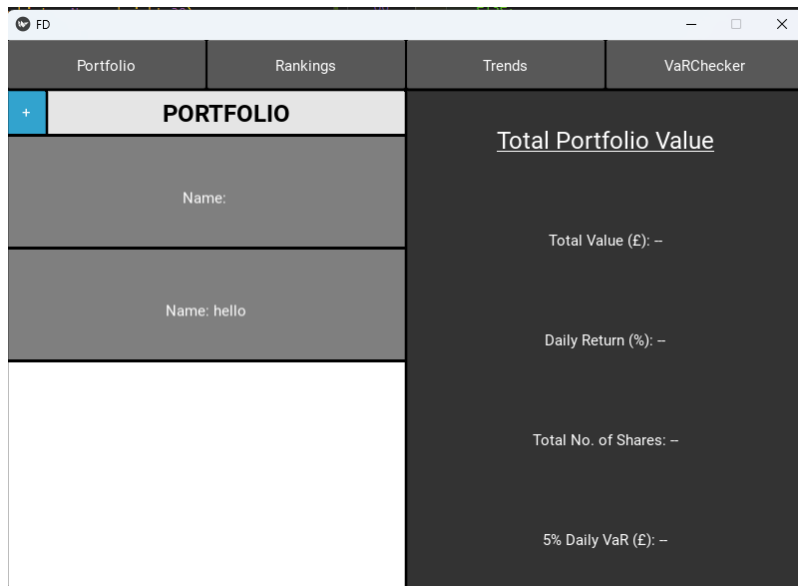


Figure 15: Early Portfolio

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]
```

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 it's 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 stock's 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.

vspace0.3cm

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))

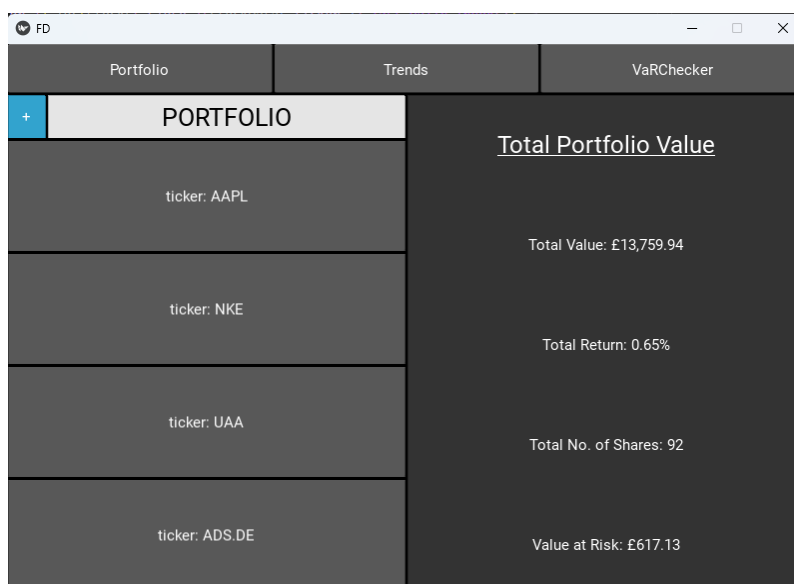
    return "{:,.2f}".format(-np.percentile(sorted(portfoReturns), 100
        * self.rlPercent)*totalValue)
```

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 statment. 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 Company 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 17 (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.

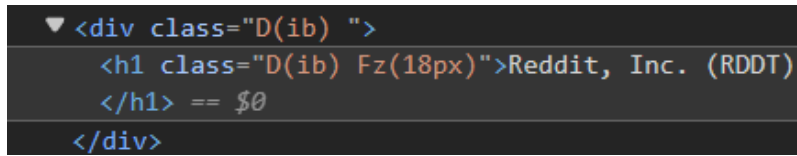


Portfolio	Trends	VaRChecker
<div>+</div> <div>PORTFOLIO</div>		<div>Total Portfolio Value</div>
<div>ticker: AAPL</div>		<div>Total Value: £13,759.94</div>
<div>ticker: NKE</div>		<div>Total Return: 0.65%</div>
<div>ticker: UAA</div>		<div>Total No. of Shares: 92</div>
<div>ticker: ADS.DE</div>		<div>Value at Risk: £617.13</div>

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)`.

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



```

▼ <div class="D(ib) ">
  <h1 class="D(ib) Fz(18px)">Reddit, Inc. (RDDT)
</h1> == $0
</div>

```

Figure 17: Ticker as Stock Names

`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 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}}`

+	STOCKS	AAPL Stock Value X
	Apple Inc. (AAPL): £1,695.80	Current Price: £169.58
	Under Armour, Inc. (UAA): £81.84	Total Return: -0.33% / £-5.70
	adidas AG (ADS.DE): £10,080.00	No. of Shares: 10
	Tesla, Inc. (TSLA): £164.90	Value at Risk: 5.78% / £97.99
		Delete Stock

Figure 18: Stocks List with Company Names

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 genreal design principles found within the creation of graphical interface. (IDK ADD A REFERENCE HERE TO UCD MAYBE). On the Total Portfolio 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 behaviors 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.

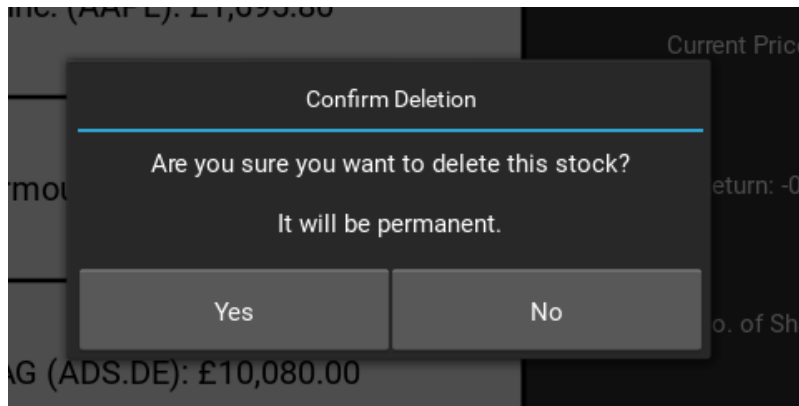


Figure 19: Delete Stock Pop-Up

```
class AdjustVaRPopup(Popup):
    def __init__(self, portfolio, varCalc, **kwargs):
        super().__init__(**kwargs)
        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. With this, 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 its current location will have to suffice.

+	STOCKS	Total Portfolio Value
	Apple Inc. (AAPL): £1,695.80	Total Value: £16,464.54
	Under Armour, Inc. (UAA): £81.84	Total Return: -0.02% / £-3.77
	adidas AG (ADS.DE): £10,080.00	Total No. of Shares: 123
	Tesla, Inc. (TSLA): £164.90	Value at Risk: 3.60% / £593.51
		Adjust VaR

Figure 20: Stocks List after Deleting a Stock

5.4 The Finalised Portfolio Screen

I am inherently not the best at creating visually appealing software, I struggle with finding colour schemes that are coherent and I don't know how to always make things look right. With my VaRChecker, I was able to find a bland 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 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

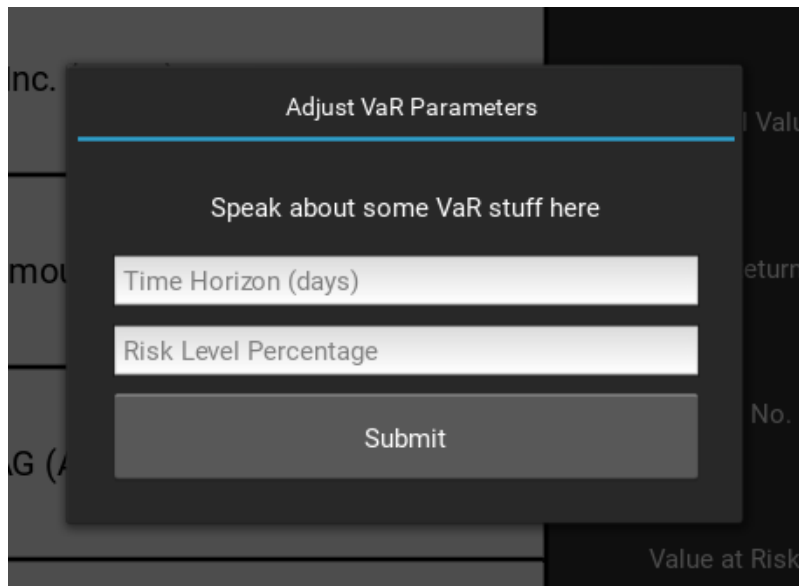


Figure 21: Original Adjust VaR Pop-Up

of my Monte Carlo Simulation.

6.2 Matplotlib within Kivy

6.3 Creating the Graphs Screen

Start with putting my initial drawing for this here.

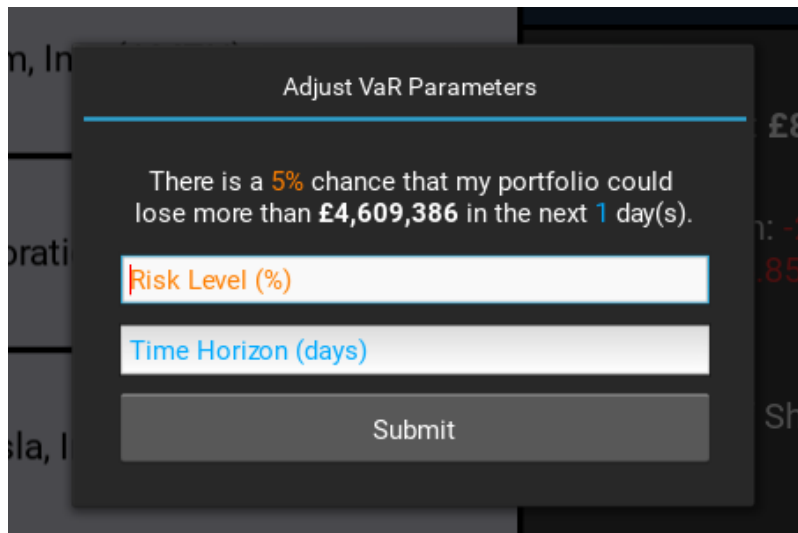


Figure 22: Completed Version of Adjust VaR Pop-Up

6.3.1 lots of these

6.3.2 lots of these

6.3.3 lots of these

6.3.4 lots of these

6.4 The Finalised Graphs Screen

7 Chapter 6 — Completed Deliverable

7.1 Features and Expected Use

7.2 Final GUI Design

7.3 P-Values and Graphical Analysis

7.4 Adaptations for Packaged Release

8 Chapter 7 — Software Engineering

8.1 Object Oriented Programming

8.2 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:

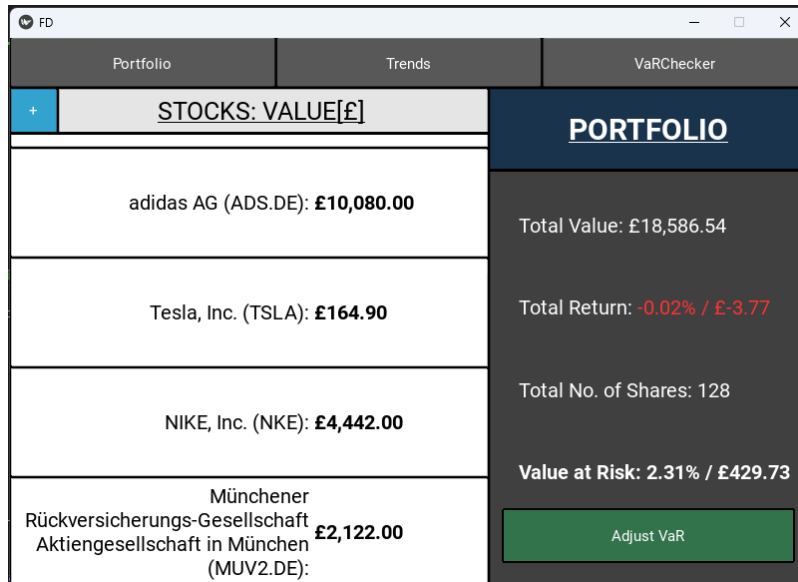


Figure 23: Previous Iteration of Portfolio Screen

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.

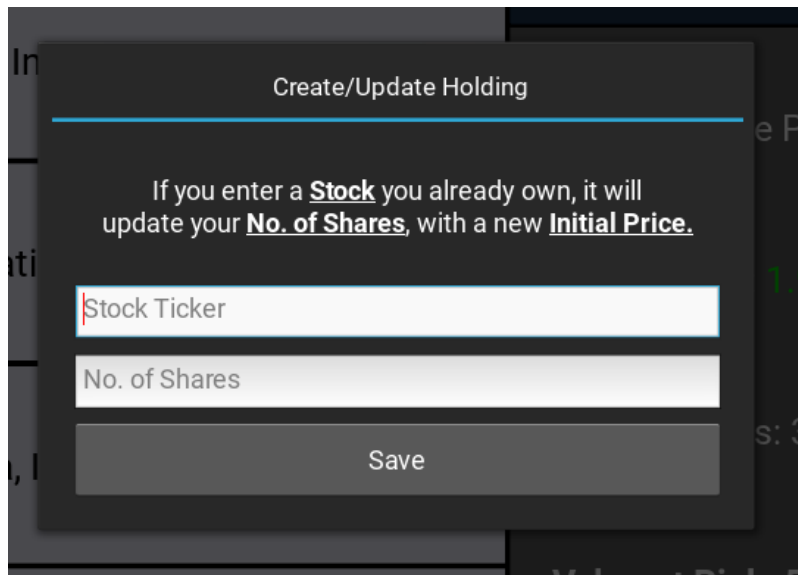


Figure 24: Final Create/Update Holding Pop-Up

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 may undergo in the future.

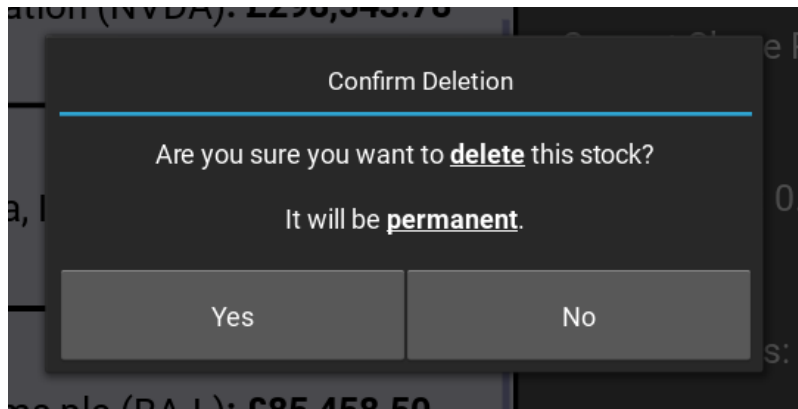


Figure 25: Final Confirm Delete Pop-Up

8.3 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 behavior for errors 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 which this project is still in, thus me using them so far.

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.

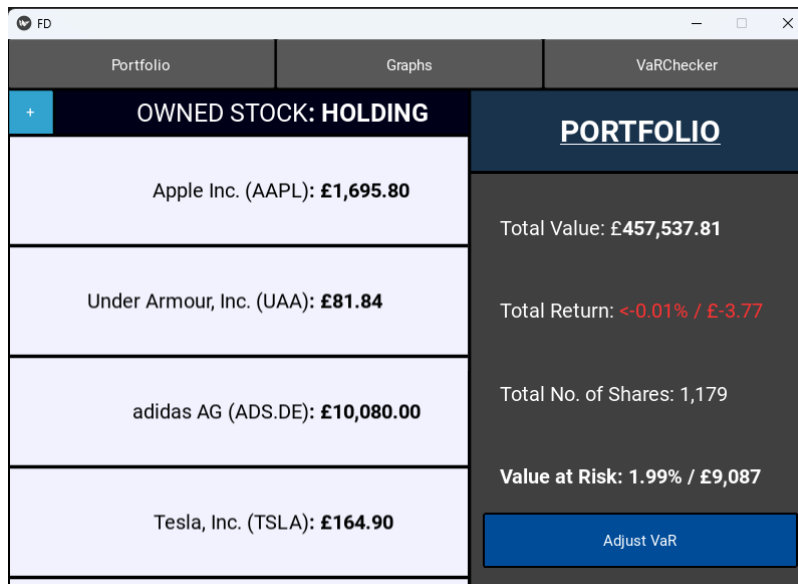


Figure 26: Completed Portfolio Screen

For future development cycles, I 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 project can achieve a higher level of quality assurance and deliver a more robust and user-trusted VaR product.

8.4 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.
- **Branching Strategy:** While currently the project has not utilised extensive branching, I acknowledging the benefits of feature branches and plan to rigorously employ them in the future.

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.

While my development with this project's version control strategy was effective to a degree, it also highlighted areas for improvement. Adopting a more structured approach to branching and leveraging the full suite of features offered by Git will be a focus in future endeavours to ensure a more efficient and error-preventative development workflow.

8.4.1 Branching and Tagging

9 Chapter 8 — Critical Analysis

9.1 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...

9.2 Final Report Evaluation

9.2.1 Coding Achivements and Success

9.2.2 Difficulties and Improvements

9.3 Conclusion

10 Professional Issues

10.1 Plagiarism and Using AI

10.2 Finance Industry Ethics

11 Bibliography

11.1 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 than 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/>.

11.2 Literature Review

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.

TALK ABOUT THIS STUFF MORE, ADD MORE ABOUT IDK NEW REFERENCES AND GO INTO MORE DETAIL, LIKE CALLUM