Portfolio Risk Assessment Using 'Value-at-Risk' and 'Expected Shortfall' Risk Measures

Term Paper for IRFA Program "C++" Course

Tomas SOMOZA, Meriem HAFID and Sarbjit GILL

Submitted: January 29, 2023

Table of Contents

Portfolio Risk Assessment Using 'Value-at-Risk' and 'Expected Shortfall' Risk Measures

1. Project Topic Introduction & Related Literature

Risk assessment and management is a vital part of finance and insurance, from both a macroprudential and individual business perspective, among varied others (McNeil, Frey & Embrechts, 2015). From the systemic perspective, it is in the interest of the society and a crucial role of governments to ensure that finance and insurance systems function smoothly to protect household savings and investments. From the viewpoint of a finance or insurance entity, managing risk properly can increase the value of the corporation, its shareholder value, its portfolio returns, its profits, and win the firm more business. Thus, for finance and insurance entities, managing risk, especially portfolio risk, is an important topic.

Value-at-Risk (VaR) and Expected Shortfall (ES) are two risk measures that can be employed to assess market risk of a portfolio (Frikha, 2022). VaR, introduced in the 1990s by J.P. Morgan - RiskMetrics, is now required for financial institutions by the Basel Committee on Banking Supervision (BCBS)*. In layman's terms, for a given time-period (1 trading day, 1 week, 10 days, etc) and given probability, which is our confidence level (90%, 95%, etc), the VaR metric calculates the portfolio amount that is at the risk of loss by (100 – probability)% during that time-period (Holton, 2014). It is the worst loss expected for a given time-period. Mathematically, at confidence level $\alpha \in (0, 1)$, VaR is defined as the threshold that is exceeded by 1-α % for a certain time horizon T, for example: $\alpha = 99.5\%$, 1-$\alpha = 0.05\%$, T = 1 trading day (Frikha, 2022). European banks and insurance companies are required to hedge themselves against these potential losses as part of market risk capital requirements under Solvency II (Brenda, 2021) and Basel III (The Bank for International Settlements, 2019). However, a major drawback of VaR is that for any non-spherical distribution of returns (non-Gaussian), it is unable to satisfy the sub-additivity property of coherent risk measures: $\varrho(a+b) \leq \varrho(a) + \varrho(b)$, which says that the combined risk of two assets is lower than the individual risk of the assets (Frikha, 2022).

*The current market risk capital requirements will be replaced with the Fundamental Review of the Trading Book (FRTB) risk measures from 2023 onwards (Bloomberg Professional Services, 2022). The deadline for European institutions is January 2025.

To address this issue, BCBS proposed a fully coherent market risk measure in 2012: **Expected Shortfall (ES)**, also known as the 'conditional value-at-risk,' which in simple terms is defined as "the average of all potential losses exceeding the VaR at a given confidence level" (The Bank for International Settlements, 2019). Mathematically, at confidence level $\alpha \in (0, 1)$ of a loss L, Frikha (2022) defines $ES_\alpha(L)$ as:

$$E[L \mid L \geq VaR_\alpha(L)] = \frac{1}{1-\alpha} \int_{\alpha}^{+\infty} VaR_a(L)\, da$$

The aim of introducing ES by BCBS was to better capture the tail losses during times of extreme turmoil (MSCI, 2014). However, due to the back-testing problem pertaining to the elasticity property of ES, in 2013 BCBS recommended that financial institutions compute expected losses using ES but continue to back-test using VaR. In the near future, by January 2025 for European financial institutions, the current market risk measures will be replaced by BCBS's Fundamental Review of the Trading Book (FRTB) measures.

Our project, coded in the C++ programming language, will use current market risk measures: VaR and ES, to assess the risk of portfolio returns for two models: a toy model and a (slightly more) complex model. For the toy model we'll work with an arbitrary portfolio containing a vector of randomly generated returns spanning 2 years of data. For the (slightly more) complex model, we'll work with a portfolio containing 1 asset and build a vector of returns spanning 2 years from the asset's historical data. Given these aims, we share our Monte Carlo approach for arriving at VaR and ES estimates based on our models in Section 2 of this report. In Section 3, we explain how we coded the MC method for our toy model problem and share sample results we derived. In Section 4, we do the same for the (slightly more) complex model. We conclude the report with Section 5, where we discuss our work and suggest topics for furthering this work in the future.

2. Monte Carlo Approach to Estimating 'Value-at-Risk' and 'Expected Shortfall' of Portfolio Returns

The Monte Carlo (MC) method, at its core, "encompasses any technique of statistical sampling employed to approximate solutions to quantitative problems" (Holton, 2014). It predicts possible outcomes of an uncertain event (Amazon Web Services, 1978). Given enough repeated random simulations on data generated with known values of the parameters, the MC method produces an estimate to our problem that is close to real-life results. In Finance, the MC method has many applications: from long-term stock price forecasts to options pricing to estimating VaR and ES of portfolio returns.

There are several approaches to arriving at VaR and ES estimates using the MC method, depending on the models chosen and the complexity. In this project, we're borrowing from the MC method detailed in the "Handbook of Financial Risk Management" (Roncalli, 2020, p. 90) to build code for toy and (slightly more) complex models that we'll use to address our problem. Our approach is detailed below.

Table 1: Steps and assumptions for estimating VaR and ES using the Monte Carlo Method

MC Steps & Assumptions: Toy Model	**MC Steps & Assumptions: (A Slightly More) Complex Model**
1. Problem definition and user inputs: ● MC VaR and ES estimates of an arbitrary portfolio with a vector of randomly generated returns spanning 2 years of data, with user choice for distribution of returns: Gaussian, Pareto or student's-T.*	1. Problem definition and user inputs: ● MC VaR and ES estimates of a portfolio composed of 1 asset with historical returns data spanning over 2 years, with user choice for distribution of returns: Normal, Pareto or student's-T.*
2. Selection of MC simulation parameters: ● Time horizon: 2 years ● Confidence level $\alpha \in (0, 1)$	2. Selection of MC simulation parameters: ● Time horizon: 2 years ● Confidence level $\alpha \in (0, 1)$
3. Selection of number of simulations: ● 100,000	3. Selection of number of simulations: ● 100,000
4. Estimating VaR and ES: ● After running our problem using steps and assumptions above, we sort the return values from all simulations in ascending order and choose (1-α) percentile which is our VaR. Expected Shortfall follows.	4. Estimating VaR and ES: ● Same as on the left

**Table 2: Justification for the selected returns distributions to address our problem:*

Returns Distribution	**Justification**
Normal (Gaussian)	No matter the preliminary distribution, the sum of a statistically large number of independent or weakly correlated random variables with finite second moment converges to a normal distribution according to the Central Limit Theorem (Fischer, 2011). Though the justification is mathematically sound, the portfolio returns typically do not follow a normal distribution and have fat tails (Lachowicz, 2015). However, normal distribution is easy to work with given its prevalence in mathematics and its properties, giving us a decent first approximation.
Student's-t	Student's-t distribution is a type of a normal distribution used to infer from small samples and unknown population variance. It has fatter tails, thus allowing for "higher dispersion of variables, as there is more uncertainty" (Valchanov, 2021).
Pareto	Pareto is a power law distribution that has proved useful in explaining portfolio returns (Macro Ops, 2017). In this project, we are testing Pareto distribution results and comparing them to normal and Student's-t results out of curiosity.

In the next two sections of this report, we walk the reader through our C++ code for the above-stated models and share sample results.

3. C++ Code & Sample Results: A Toy Model to Estimate VaR and ES

> **An Important Note:** We will share select code below to demonstrate our work, however, we invite you to take a look at our complete code and all files at this link (format .cpp; Google Drive link).

To derive MC VaR and ES estimates using the toy model, we first defined our main function *(see Figure 1 below)*, which upon initialization produces the estimates. Within our main function, we set the number of simulations; ask user to input the confidence level $\alpha \in (0, 1)$ and distribution type; and then the function runs Monte Carlo simulations. The MC method is nested in a class of its own.

```
//getting constant pi just to avoid calling another library for one thing
double pi = 3.141592;

int main() {
    // defining number of simulations (in general has to be bigger or equal than this)
    int num_samples = 100000;

    // asking the user for the confidence interval
    double alpha;
    std::cout << "\nEnter confidence level (1-alpha): \n";
    std::cin >> alpha;

    // Initialising the parameters for the distributions
    // caveat here: we need to initialise them for every distribution

    double mu = 0.0;
    double sigma = 0.2;
    double alpha1 = 2.0;
    double xmin = 1.0;
    double nu = 5.0;

    // now the user decides the distribution type
    std::string dist_type;
    std::cout << "Enter distribution type (t, normal or pareto): ";
    std::cin  >> dist_type;

    //now we use the methods run() and show() of our Montecarlo class
    MC portfolio(num_samples, mu, sigma, alpha, alpha1, xmin, nu, dist_type);
    portfolio.run();
    portfolio.show();

    return 0;
}
```

Figure 1: Main function to estimate MC VaR and ES for the toy model

In the Monte Carlo class *(see figure 2 below)*, we have two of the three methods explicitly asked in the project description: run() and show(). The method called sample() in the project description for us will be a nested method in another class "densities_1.cpp" that we will be accessing in the MC as if it was a proper method.

```
MC::MC(int num_samples_,double mu_,double sigma_,double alpha_,double alpha1_,double xmin_, double nu_, std::string dist_type_):
    //  initialising variables
    num_samples(num_samples_),
    mu(mu_),
    sigma(sigma_),
    alpha(alpha_),
    alpha1(alpha1_),
    xmin(xmin_),
    nu(nu_),
    dist_type(dist_type_)
{
}
    // this method simulates a return given that the distribution matches some user choice
    // and adds it to a vector of returns
    void MC::run() {
        if (dist_type == "t") {
            dist t_dist(nu, 1.0, dist::student);
            for (int i = 0; i < num_samples; i++) {
                double r = t_dist.compute();
                returns.push_back(r);
            }
        }
        else if (dist_type == "normal") {
            dist normal_dist(mu, sigma, dist::normal);
            for (int i = 0; i < num_samples; i++) {
                double r = normal_dist.compute();
                returns.push_back(r);
            }
        }
        else if (dist_type == "pareto"){
            dist pareto_dist(xmin, alpha1, dist::normal);
            for (int i = 0; i < num_samples; i++) {
                double r = pareto_dist.compute();
                returns.push_back(r);
            }
        }
        else {
            std::cout << "Invalid distribution type. Please choose 't', 'normal' or 'pareto': " << std::endl;
            return;
        }
    }

    double MC::calculateVaR() {
        std::sort(returns.begin(), returns.end()); // sorting in ascending order
        int index = (int) (num_samples * (1 - alpha)); // getting the index
        return returns[index];
    }

    double MC::calculateES() {
        std::sort(returns.begin(), returns.end());
        int index = (int) (num_samples * (1 - alpha));
        double VaR = returns[index];
        // once that we get the VaR we need the expected value above the VaR.
        double sum = 0;
        int count = 0;
        for (int i = 0; i < index; i++) {
            if (returns[i] < VaR) {
                sum += returns[i];
                count++;
        }
    }
    return sum / count;
}

    // method to show results
    void MC::show() {
        double VaR = calculateVaR();
        std::cout << "Value at Risk (VaR) at " << (1 - alpha) * 100 << "%: " << VaR << std::endl;
        double ES = calculateES();
        std::cout << "Expected Shortfall at " << (1 - alpha) * 100 << "%: " << ES << std::endl;
    }

    //}
```

Figure 2: The Monte Carlo class

The Monte Carlo method takes distribution parameters input from the "densities_1.cpp" file (see figure 3 below), where we use the random number generator to create random returns for the three distributions.

```
std::default_random_engine generator;
std::uniform_real_distribution<double> distribution(0.0, 1.0);
std::normal_distribution<double> standard_normal(0.0, 1.0);

//initialise parameters
dist::dist(double param1_, double param2_, DisType TheDisType_):
    param1(param1_),
    param2(param2_),
    TheDisType(TheDisType_)
{
}

double dist::compute()
{
    // using switch to see different cases
    switch (TheDisType)
    {
        case normal:{ //using the brackets to avoid problems with variable definition
            double z = standard_normal(generator);
            return param1 + param2*z;}
        case pareto:{ // a future improvement will be to change this to generalized Pareto typpe IV
            double u = distribution(generator);
            double x = param1 / pow(u, 1.0 / param2);
            return x;}
        case student:{
            std::chi_squared_distribution<double> chi_squared(param1);
            double v = standard_normal(generator);
            double t = v / sqrt(chi_squared(generator) / param1);
            return t;}
        default:
            throw std::runtime_error("Unknown distribution type found.");
    }
    return 0;
}
```

Figure 3: Code for generating random returns from user's choice of distribution

Toy model sample results: After building our toy model code, we tested it on different confidence levels and distributions to derive sample VaR and ES estimates, shared in the table below. Please note that 1-α is the confidence level, VaR (in %) and ES (in %) are the percent change in returns that we have the chance to lose with the given confidence level, and VaR (in €) and ES (in €) is the euro amount that we stand to lose with the given confidence level and initial portfolio value of 1 million euros.

Table 3: Toy model sample MC VaR and ES estimates.

Distribution	**Parameters**	**1-α**	**VaR** (in %)	**VaR** (in €)	**ES** (in %)	**ES** (in €)
Normal	μ=0, σ=0.2	0.95	0.3307%	€ 3,307	0.4158%	€ 4,158
Student's-t	ν=5.0	0.99	3.3933%	€ 33,933	4.5130%	€ 45,130
Pareto	x_{min}=1.0, α=2.0	0.90	1.5769%	€ 15,769	2.5323%	€ 25,323

4. C++ Code & Sample Results: A Complex Model to Estimate VaR and ES

> **An Important Note:** We will share select code below to demonstrate our work, however, we invite you to take a look at our complete code and all files at this link (format .cpp; Google Drive link).

To derive MC VaR and ES estimates using the complex model, we again defined our main function *(see Figure 4 below)*, which upon initialization produces the estimates. Within our main function, we read the CSV file with historical returns data, call upon the stock function to transform the data and estimate distribution parameters, set the number of simulations; ask user to input the confidence level $\alpha \in (0, 1)$ and the distribution type; and then the function runs Monte Carlo simulations. The MC method is nested in a class of its own.

```
double pi = 3.141592;

int main() {

    // Read data from CSV file
    Stock stock("C:/Users/tomso/Desktop/QEM/2nd year/C++/^SSMI V2.csv");
    // compute returns
    auto returns = stock.computeReturns();
    // compute mean
    double mean = stock.computeMean(returns);
    // compute variance
    double variance = stock.computeVariance(returns, mean);

    double stddev = sqrt(variance);

    int num_samples = 100000;

    double alpha;
    std::cout << "\nEnter confidence level (1-alpha): \n";
    std::cin >> alpha;

    std::string dist_type;
    std::cout << "Enter distribution type (if we do not use normal the program we will use predetermined values for the other distributions): ";
    std::cin  >> dist_type;

    double alpha1 = 2.0;
    double xmin = 1.0;
    double nu = 5.0;

    MC portfolio(num_samples, mean, stddev, alpha, alpha1, xmin, nu, dist_type);
    portfolio.run();
    portfolio.show();

    return 0;
}
```

Figure 4: Main function to estimate MC VaR and ES for the complex model

The stock function *(see figure 5 below)* reads from a CSV file and extracts the closing price from each line of the file. We start by creating an input file stream (ifstream) object called 'file'. First we declare a variable named 'line' of type std::string which is a standard C++ class that represents a sequence of characters. One of the main advantages of using std::string is that it manages the memory for the string automatically, which makes it less prone to memory-related bugs. It also provides a variety of methods that can be used to manipulate strings. We are using find() which returns the position of a substring within the string and substr() returns a substring of the string.

We define the function 'ComputeReturns' that takes a reference to an ifstream object as input, and returns a vector of doubles containing the returns. The while loop reads each line of the file into the 'line' variable. Then the closing price is extracted from the 'line' variable using the following steps:

1. Find the last occurrence of *'comma'* using the find_last_of() method and save the index in a variable 'idx'.
2. The closing price is located after the second last *'comma'*. So, the find_last_of() method is called again with the index returned in the previous step as the first argument and a *'comma'* as the second argument.
3. Extract the substring from the index returned by the above method + 1. This will be the closing price.
4. Try to convert the substring to a double using the std::stod() function. If it fails, it will throw an invalid_argument exception.
5. If the exception is caught, the line is skipped and a message is printed to the console.
6. If the exception is not caught, the closing price is added to the 'data' vector.

The while loop continues until there are no more lines to read in the file and we get a vector of returns. From this, we calculate the mean and variance of the daily returns, which then is used by the Monte Carlo class to run simulations *(see figure 5 below)*.

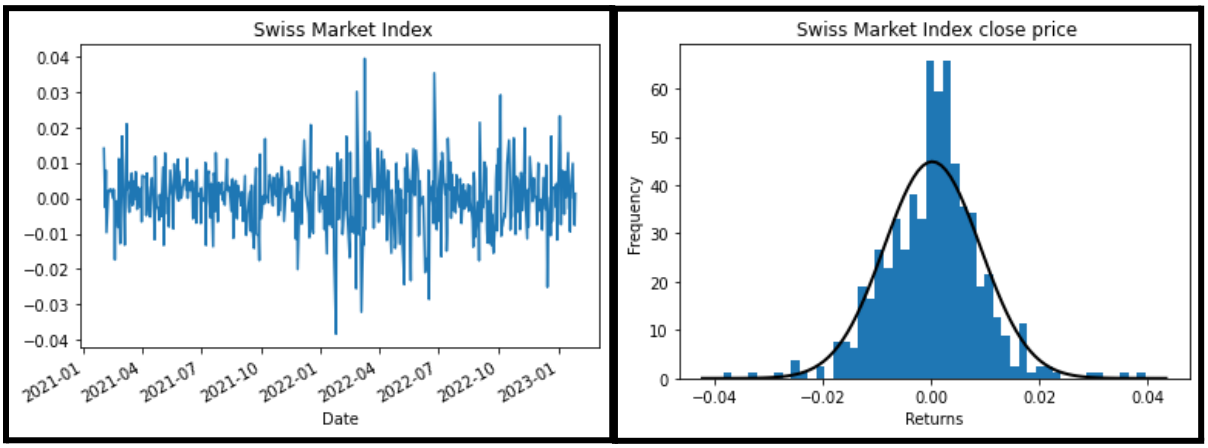
```
Stock::Stock(std::string file_path_):
    file_path(file_path_)
{
}

    std::vector<double> Stock::computeReturns() {
        // Read data from CSV file
        std::ifstream file(file_path);
        std::string line;

        // Store data in a vector
        std::vector<double> data;
        while (std::getline(file, line)) {
            // Extract closing price from line
            double closing_price;
            try {
                closing_price = std::stod(line.substr(line.find_last_of(",", line.find_last_of(",") - 1) + 1));
                data.push_back(closing_price);
            } catch (const std::invalid_argument& e) {
                // skip the line if the closing price is not a number like the first line in our data
                std::cerr << "Skipping invalid line: " << e.what() << std::endl;
            }
        }

        // Clean data (e.g. remove missing values)
        data.erase(std::remove(data.begin(), data.end(), 0), data.end());

        // Compute returns
        std::vector<double> returns;
        for (std::size_t i = 1; i < data.size(); ++i) {
            returns.push_back((data[i] - data[i - 1]) / data[i - 1]);
        }

        return returns;
    }

    double Stock::computeMean(std::vector<double> returns){
        // Compute mean of returns
        double sum = 0;
        for (auto &returns : returns)
        {
            sum += returns;
        }

        double mean = sum / returns.size();
        std::cout << "Mean of Returns: " << mean << std::endl;
        return mean;

    }
    double Stock::computeVariance(std::vector<double> returns, double mean){
        // Compute variance of returns
        double variance = 0;
        for (auto &returns : returns)
        {
        variance += (returns - mean) * (returns - mean);
        }

        variance = variance / returns.size();
        std::cout << "Variance of Returns: " << variance << std::endl;
        return variance;


    }
```

Figure 5: Code for estimating parameters from historical returns

Complex model sample results: After building our complex model code, we tested it on different confidence levels to derive sample VaR and ES estimates, shared in the table below. Please note that 1-α is the confidence level, VaR (in %) and ES (in %) are the percent change in returns that we have the chance to lose with the given confidence level, and VaR (in €) and ES (in €) is the euro-amount that we stand to lose with the given confidence level and initial portfolio value of 1 million euros. Kindly also note that if the user does not choose the Gaussian distribution to run the complex model, the code will automatically use predetermined values for other distributions.

Table 4: Complex model sample MC VaR and ES estimates for Swiss Market Index.

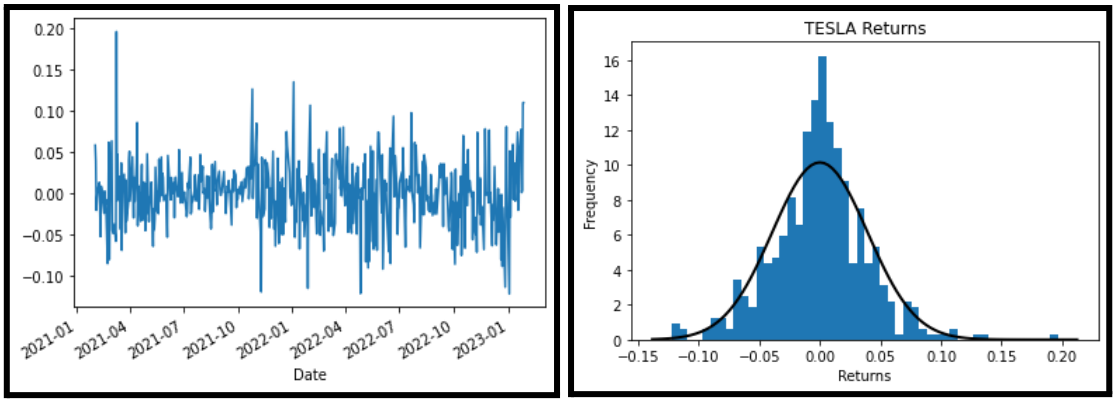
Distribution	**Parameters**	**1-α**	**VaR** (in %)	**VaR** (in €)	**ES** (in %)	**ES** (in €)
Normal	μ=0.000176 & σ=8.0438e-0.5	0.99	0.020%	€ 200	0.023%	€ 230
Normal		0.95	0.015%	€ 150	0.018%	€ 180
Normal		0.90	0.011%	€ 110	0.015%	€ 150



Figures 6 and 7: Time-series (left) and distribution (right) of Swiss Market Index returns

Table 5: Complex model sample MC VaR and ES estimates for Tesla.

Distribution	**Parameters**	**1-α**	**VaR** (in %)	**VaR** (in €)	**ES** (in %)	**ES** (in €)
Normal	μ=-1.75e-05 & σ=0.001547	0.99	0.0928%	€ 928	0.1060%	€ 1,060
Normal		0.95	0.0651%	€ 651	0.0818%	€ 818
Normal		0.90	0.0507%	€ 507	0.0695%	€ 695



Figures 8 and 9: Time-series (left) and distribution (right) of Tesla returns

5. Discussion, Conclusions and Recommendations

In this document we have shown two approaches for estimating the Value-at-Risk and the Expected Shortfall of portfolio returns using the Monte Carlo method.

The initial toy model with its simplicity allowed us to understand how to work with files and classes in C++, and it also became a fundamental building block for a more realistic estimation in the second file.

For the user, the clarity in the toy model allows them to tune the parameters of the model with a simple inspection of the code. Parameters available for users to manipulate are distribution type and confidence level.

A key conclusion from our results is that we are obtaining more extreme values for non-normal distributions. This result could be happening both because of our parameter choice and because these distributions can model fatter tails. With real data values, in the complex model we obtained smaller losses for the Swiss index than for a stock like TSLA. This reflects the idea that an index as a mixture of many stocks helps dilute the risk of the individual stocks. We see that in general, losses in the case of TSLA amount to approximately 5 times the ones that could happen with the index.

Future work would include implementing a more complex portfolio, i.e. a weighted sum of random variables, more complex distributions, or a thorough estimation of the parameters minimizing some functional form.

Also, it is worth mentioning that we tried to implement a volatility estimation based on GARCH models but the code was purely based in the Eigen library (to work with matrices), thus the complexity to really understand what was going on beneath the code was getting increasingly difficult. With more available time we could have defined our own file with matrix operations to be able to implement this code successfully.

Finally, the last idea that we could have implemented was to apply **inheritance** to our code. Especially in the class of distributions, we could have defined a base class of distributions and then tried to implement an inherited class with a new distribution such that we do not need to modify the file each time we want to consider a new distribution. However, given the time constraints and the scope of this project, we recommend this implementation as part of future work.

We hope to continue and build on this work.

References

1. Amazon Web Services (1978) *What is Monte Carlo Simulation?*, *Amazon*. Available at: https://aws.amazon.com/what-is/monte-carlo-simulation/?nc1=h_ls (Accessed: January 28, 2023).

2. Bloomberg Professional Services (2022) *Fundamental Review of the trading book (FRTB): Where do we stand?*, *Bloomberg.com*. Bloomberg. Available at: https://www.bloomberg.com/professional/blog/fundamental-review-of-the-trading-book-frtb-where-do-we-stand/ (Accessed: January 26, 2023).

3. Brenda, P. (2021) *Calculation of the Minimum Capital Requirement - EIOPA European Commission*, *Eiopa*. Available at: https://www.eiopa.europa.eu/rulebook/solvency-ii/article-2353_en (Accessed: January 26, 2023).

4. Fischer, H. (2011) *A history of the central limit theorem*. New York, NY: Springer New York.

5. Frikha, N. (2022). *Market Risk Measures - Chapter 2: Value-at-Risk and Expected Shortfall (Conditional Value-at-Risk)* [Class notes]. Université Paris I Panthéon-Sorbonne.

6. Holton, G.A. (2014) *Value-at-Risk: Theory and practice, second edition*. Available at: https://www.value-at-risk.net/ (Accessed: January 26, 2023).

7. Lachowicz, P. (2015) *Student t Distributed Linear Value-at-Risk*, *Quant at Risk*. Available at: https://quantatrisk.com/2015/12/02/student-t-distributed-linear-value-at-risk/ (Accessed: January 29, 2023).

8. Macro Ops (2017) *Pareto's law, 90/10, and the distribution of returns*, *Macro Ops*. Available at: http://operators.macro-ops.com/wp-content/uploads/2016/09/Paretos-Law-Vault-.pdf (Accessed: January 29, 2023).

9. McNeil, A.J., Frey, R. and Embrechts, P. (2015) *Quantitative risk management: Concepts, techniques and tools*. Princeton, NJ: Princeton University Press.

10. MSCI (2014) *Expected shortfall story*, *MSCI*. Available at: https://www.msci.com/expected-shortfall-story (Accessed: January 26, 2023).

11. Roncalli, T. (2020) *Handbook of Financial Risk Management*. Boca Raton: CRC Press.

12. The Bank for International Settlements (2019) *Minimum capital requirements for market risk*, *The Bank for International Settlements*. Available at: https://www.bis.org/bcbs/publ/d457.htm (Accessed: January 26, 2023).

13. Valchanov, I. (2021) *What is the student's t distribution?*, *365 Data Science*. Available at: https://365datascience.com/tutorials/statistics-tutorials/students-t-distribution/ (Accessed: January 29, 2023).

Appendix I: Project CPP Files

Project CPP files can be accessed at this link (format .cpp; Google Drive link).