Market Risk Project

Academic Year 2024-2025

Autors:
Eloi Martin, Ahmed Mili

Institution: ESILV

Date: 30th December 2024

Contents

Note: All questions marked with an asterisk (*) are additional questions tackled by the authors.

1 Introduction

Risk management is a cornerstone of modern finance, playing a pivotal role in safeguarding portfolios and ensuring market stability. In an increasingly complex financial landscape, managing market risk effectively has become a critical endeavor for investors, institutions, and regulators alike. Market risk, defined as the potential for financial loss due to adverse movements in market variables such as prices, rates, or volatility, requires robust quantitative tools and methodologies to measure, analyze, and mitigate its impact.

This project represents the culmination of the market risk course, providing an opportunity to bridge theory and practice through the application of key concepts covered during lectures and tutorials. By tackling real-world problems using data-driven approaches, we aim to deepen our understanding of market risk and its implications for financial decision-making. The project is structured into a series of tasks, each targeting a specific aspect of market risk:

- **Value at Risk (VaR):** Estimation of historical and non-parametric VaR for Natixis stock returns, with a validation of its accuracy over out-of-sample data.
- **Monte Carlo VaR for Options:** Implementation of a Monte Carlo simulation to estimate the VaR for a call option, incorporating stochastic price dynamics modeled through a standard Brownian motion.
- **Extreme Value Theory (EVT):** Analysis of extreme returns using the Generalized Extreme Value (GEV) distribution and estimation of the extremal index via declustering methods.
- **Optimal Liquidation Strategies:** Application of the Almgren and Chriss framework to optimize asset liquidation while balancing market and liquidity risks.
- **Wavelet Analysis and Hurst Exponent:** Multiresolution correlation analysis of FX rates and estimation of the Hurst exponent to capture long-term memory effects and volatility scaling.

Python serves as our primary programming language, offering a robust ecosystem for data analysis and visualization. Key libraries such as `pandas`, `numpy`, `scipy.stats`, and `matplotlib` are extensively utilized for data preprocessing, numerical computation, statistical analysis, and graphical representation. Each task is approached methodically, ensuring the reproducibility and clarity of our results.

The Appendix section is used to show the rest of the code that we didn't consider relevant to include in the previous sections.

2 Question A

The Value at Risk (VaR) is a widely used metric in financial risk management. It quantifies the maximum probable loss of a portfolio over a specified time horizon, given a certain confidence level. For example, on a portfolio of €1 million, a VaR of €100,000 at a 95% confidence level means there is a 5% probability of losing €100,000 or more over the specified time horizon.

At a confidence level α, the VaR is mathematically defined as:

$$\text{VaR}_\alpha(L) = \inf\{l \in \mathbb{R} : P(L > l) \leq 1 - \alpha\},$$

where:

- L: the portfolio's loss,
- $\alpha \in [0, 1]$: the confidence level.

This corresponds to the $(1 - \alpha)$-quantile of the portfolio's losses.

In this exercise, we compute the historical VaR and non-parametric VaR using a logistic kernel. The kernel is defined as the derivative of the logistic function:

$$K(x) = \frac{e^{-x}}{(1 + e^{-x})^2}.$$

We use the daily price returns of the Natixis stock between January 2015 and December 2016:

```
# Convert the Date column to datetime format
data['Date'] = pd.to_datetime(data['Date'])

# Filter the data between January 2015 and December 2016
data_filtered = data[(data['Date'] >= '2015-01-01') & (data['Date'] <=␣
 ↪'2016-12-31')]
```

This non-parametric VaR will be validated by analyzing the proportion of price returns between January 2017 and December 2018 that exceed the previously computed VaR threshold:

```
# Filtrer les données pour la période entre janvier 2017 et décembre 2018
data_filtered_period = data[(data['Date'] >= '2017-01-01') & (data['Date']␣
 ↪<= '2018-12-31')]
```

The validation will determine whether the non-parametric VaR accurately represents the risk, based on the observed frequency of losses exceeding the threshold.

2.1 Historical VaR : Empirical VaR*

To compute the historical Value at Risk (VaR), we based our calculations directly on the empirical distribution of past returns, assuming that the historical distribution remains valid for the near future. Below is a step-by-step explanation of the process:

1. Calculation of Daily Returns: Daily returns were calculated using the formula:

$$\text{Return}_t = \frac{P_t - P_{t-1}}{P_{t-1}},$$

where:

- P_t: the price of the asset at time t,
- P_{t-1}: the price of the asset at time $t-1$.

This formula represents the relative change in price from one day to the next. The first row was dropped from the dataset since it contains a missing value (`NaN`) for the return calculation.

2. Calculation of Historical VaR: We set a confidence level, for example, 95%, which corresponds to the $(1-\alpha)$-quantile of the return distribution, where $\alpha = 0.95$. The Value at Risk was then computed as:

$$\text{VaR} = \text{Percentile}(\text{Returns}, (1-\alpha)\cdot 100).$$

This formula identifies the return threshold below which the worst $(1-\alpha)\cdot 100\%$ of observations lie. For instance, at a 95% confidence level, the VaR represents the maximum loss expected in the worst 5% of cases under normal market conditions.

3. Key Characteristics of Historical VaR:

- The historical VaR relies entirely on past data, assuming the historical distribution remains valid in the close future.
- Each observation is weighted equally, which can be a limitation when more recent data may better reflect current market conditions.
- The method often assumes returns are independent and identically distributed (i.i.d.), which may not always hold true in financial markets.

The calculated historical VaR provides an estimate of the maximum expected loss for the given confidence level based on historical price returns.

Using the daily stock prices of Natixis from January 2015 to December 2016 (as provided in TD1), a historical VaR will be estimated. The calculation is based on price returns at a one-day horizon for a given confidence level.

```
# Set a confidence level for VaR (e.g., 95% for 1-day horizon)
confidence_level = 0.95

# Calculate the VaR using the percentile of the returns distribution
VaR = - np.percentile(data_filtered['Return'], (1 - confidence_level) *␣
 ↪100)
print(f"\nHistorical VaR at {confidence_level * 100}% confidence level:␣
 ↪{VaR* 100:.3f}%")
```

```
Historical VaR at 95.0% confidence level: 3.778%
```

Using this method, we determine a VaR of 3.778% at a 95% confidence level. This means that, for a portfolio valued at €1000, there is a 5% chance of experiencing a loss exceeding €37.78.

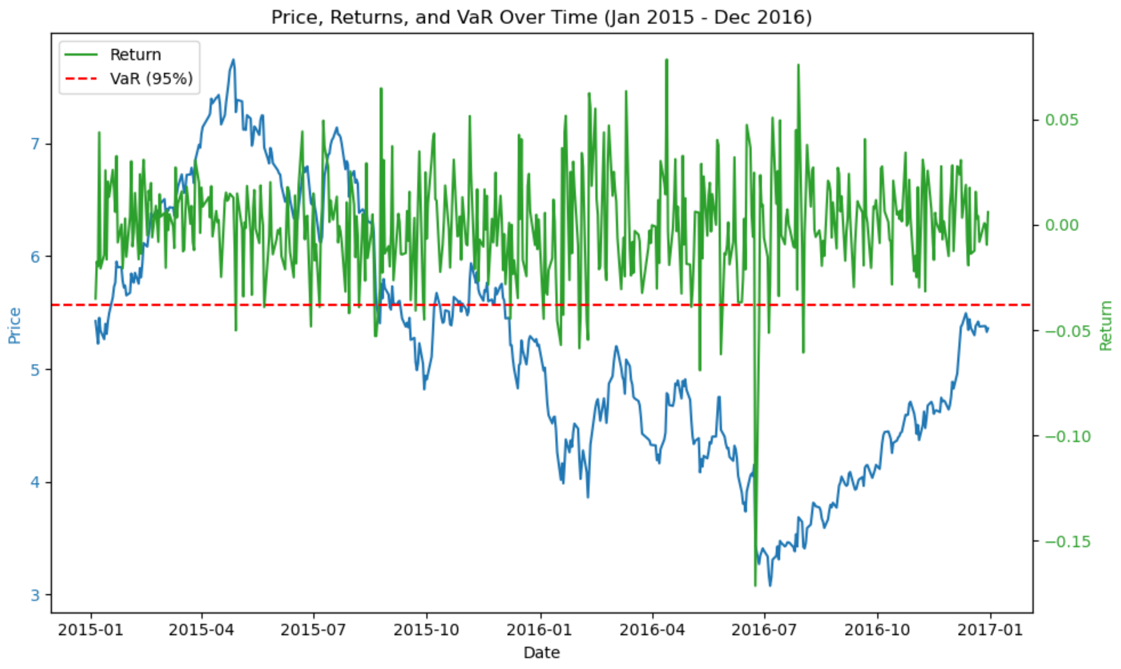

Figure 1.

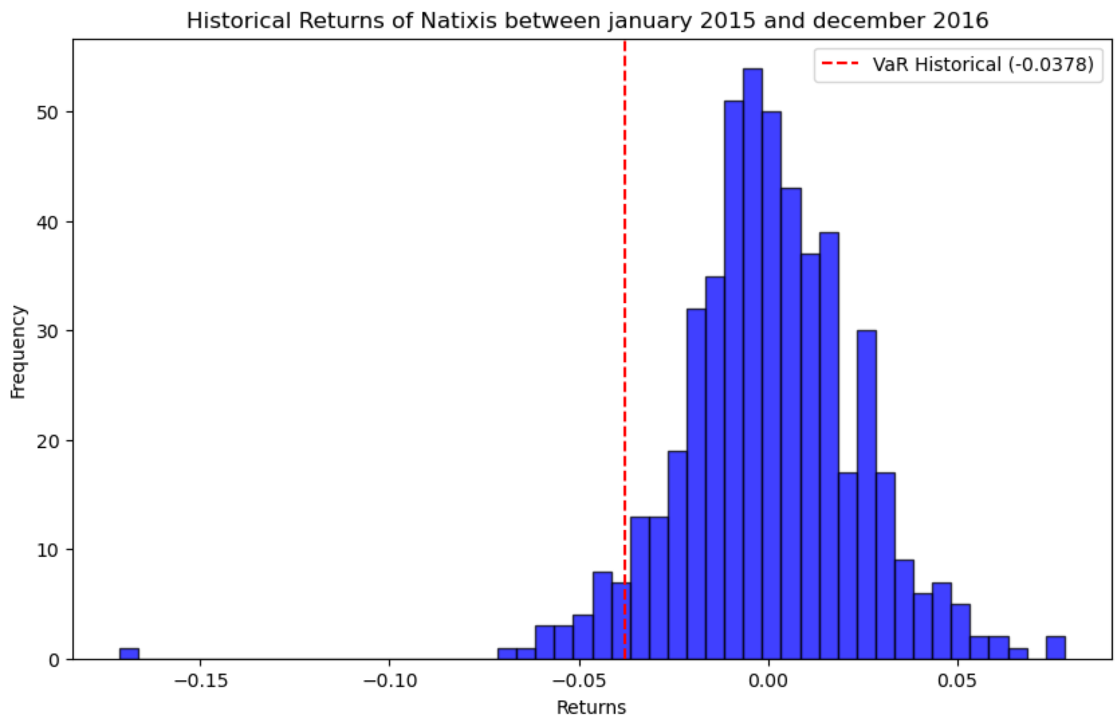

Figure 2.

2.2 Historical VaR : Non Parametric VaR

Non-parametric VaR is a method for calculating the Value at Risk (VaR) based solely on observed data, without making assumptions about the distribution of returns. Unlike parametric methods (which often assume a normal distribution), it uses techniques such as kernels to estimate the density of the data.
Unlike parametric methods, it does not rely on assumptions of normality or other specific distributions.

The VaR estimation will employ a non-parametric approach, specifically utilizing a logistic Kernel, where the kernel function K is defined as the derivative of the logistic function:

$$x \mapsto \frac{1}{1+e^{-x}}.$$

The function `logistic_kernel` implements the kernel function,derivative of the logistic function. Its formula is given by:

$$K(x) = \frac{e^{-x}}{(1+e^{-x})^2}$$

We consider a series of observed price returns $X_1, \ldots, X_n$.

The function `logistic_kernel` is defined as:

$$K(u) = \frac{e^{-u}}{(1+e^{-u})^2}.$$

It represents the derivative of the logistic function and is used to estimate the probability density.

Kernel Density Estimation Using the Logistic Kernel:

The function `kernel_density_estimate_logistic` uses the logistic kernel to estimate the probability density at a given point x. The mathematical formula underlying this estimation is:

$$\widehat{f}(x) = \frac{1}{n \cdot h} \sum_{i=1}^{n} K\left(\frac{x - X_i}{h}\right),$$

where:

- x: The point at which the density is estimated,
- h: The smoothing parameter (bandwidth),
- K: The logistic kernel,
- n: The number of observations,
- X_i: The observed data points.

Kernel-Based Empirical Cumulative Distribution Function (CDF):

The function `integrate_cdf` calculates the kernel-based empirical cumulative distribution function (CDF) at a given point x. The formula for the CDF is:

$$\widehat{F}(x) = \frac{1}{n} \sum_{i=1}^{n} \mathcal{K}\left(\frac{x - X_i}{h}\right),$$

where $\mathcal{K}$ here is the cumulative version of the logistic kernel.

To compute the Non-Parametric VaR using a logistic kernel, the process is as follows:

First, we define 1000 equally spaced points (x_{range}) between the minimum and maximum observed returns from the dataset. These points represent the range over which the kernel density is estimated.

Next, we calculate the kernel density estimates (KDE) for each point in x_{range} using the logistic kernel function and the chosen bandwidth h. The density values provide an estimate of the probability density at each point in the range.

We then compute the empirical cumulative distribution function (CDF) by numerically integrating the KDE values over x_{range}, using the function:

$$\text{cdf\_values} = \text{integrate\_cdf}(\text{kde\_values\_logistic}, x_{\text{range}}).$$

Finally, we determine the VaR at the 95% confidence level by finding the quantile corresponding to the 5% tail of the CDF. Specifically, we iterate through cdf_values to find the smallest value in x_{range} where:

$$\text{CDF}(x) \geq 1 - \text{probability\_level}.$$

This value represents the Non-Parametric VaR, which is the maximum expected loss at the 95% confidence level.

```python
# Logistic kernel function
def logistic_kernel(u):
    return np.exp(-u) / (1 + np.exp(-u))**2

# Kernel density estimation using the logistic kernel
def kernel_density_estimate_logistic(x, data, h):
    n = len(data)
    u = (x - data) / h
    kde = np.sum(logistic_kernel(u)) / (n * h)
    return kde

# Function to calculate the empirical CDF
def integrate_cdf(kde_values, x_range):
    dx = x_range[1] - x_range[0]
    cdf = np.cumsum(kde_values) * dx
    return cdf
```

```
# Bandwidth (Silverman's Rule of Thumb)
n = len(data_filtered['Return'])
# Silverman's rule of thumb
h = 1.06 * np.std(data_filtered['Return']) * (n ** (-1/5))

# Arbitrary bandwidth for comparison
h_small = 0.001

# Density estimation over a range of values
x_range = np.linspace(min(data_filtered['Return']),␣
 ↪max(data_filtered['Return']), num=1000)
kde_values_logistic = [kernel_density_estimate_logistic(x,␣
 ↪data_filtered['Return'], h) for x in x_range]
kde_values_small = [kernel_density_estimate_logistic(x,␣
 ↪data_filtered['Return'], h_small) for x in x_range]


# Empirical CDF calculation
cdf_values = integrate_cdf(kde_values_logistic, x_range)

# VaR calculation
probability_level = 0.95
non_parametric_var_logistic = None

for i, cdf in enumerate(cdf_values):
    if cdf >= (1 - probability_level):
        non_parametric_var_logistic = x_range[i]
        break

# Find the value of VaR corresponding to the desired confidence level
non_parametric_var_logistic_message = (
    f"Non-Parametric VaR (Logistic Kernel) at {probability_level * 100}%␣
 ↪confidence level: {-non_parametric_var_logistic*100:.3f}%"
)
print(non_parametric_var_logistic_message)
```

```
Non-Parametric VaR (Logistic Kernel) at 95.0% confidence level: 4.307%
```

Using this method, we determine a VaR of 4.307% at a 95% confidence level. This means that, for a portfolio valued at €1000, there is a 5% chance of experiencing a loss exceeding €43.07.

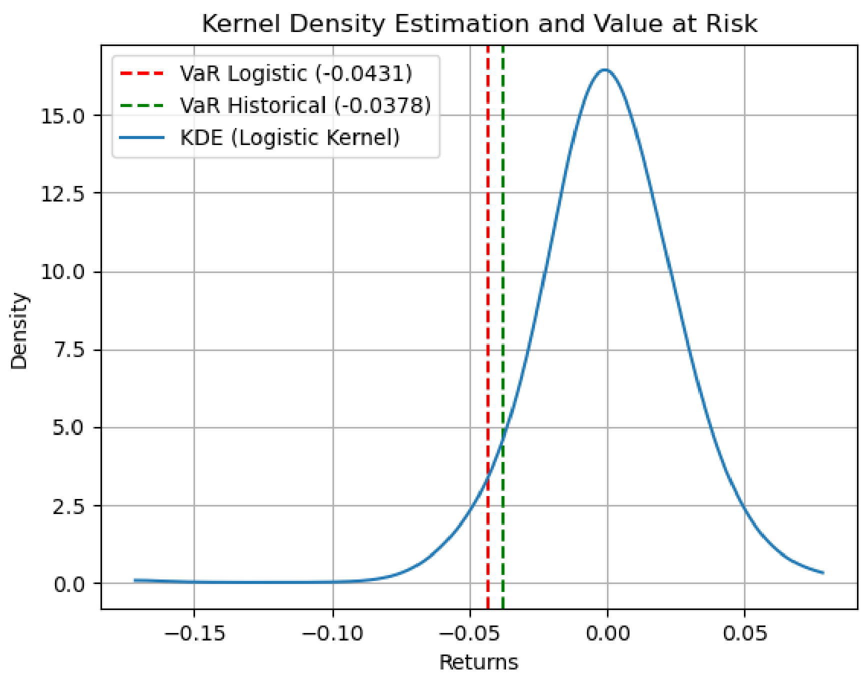

Figure 3.

The bandwidth (h) is a critical parameter that controls the smoothing in kernel estimation:

- **Small h:** The estimated density is highly sensitive to local variations in the data (overfitting), which can lead to excessive detection of details.
- **Large h:** The density becomes too smoothed and loses important details, potentially underestimating the tails of the distribution.

To find a good h, Silverman's rule of thumb is often used:

$$h = 1.06 \cdot \mathrm{std}(\mathrm{return}) \cdot n^{-1/5},$$

where n is the number of observations, and std(return) is the standard deviation of the returns. This rule balances the trade-off between precision and robustness in the estimation.

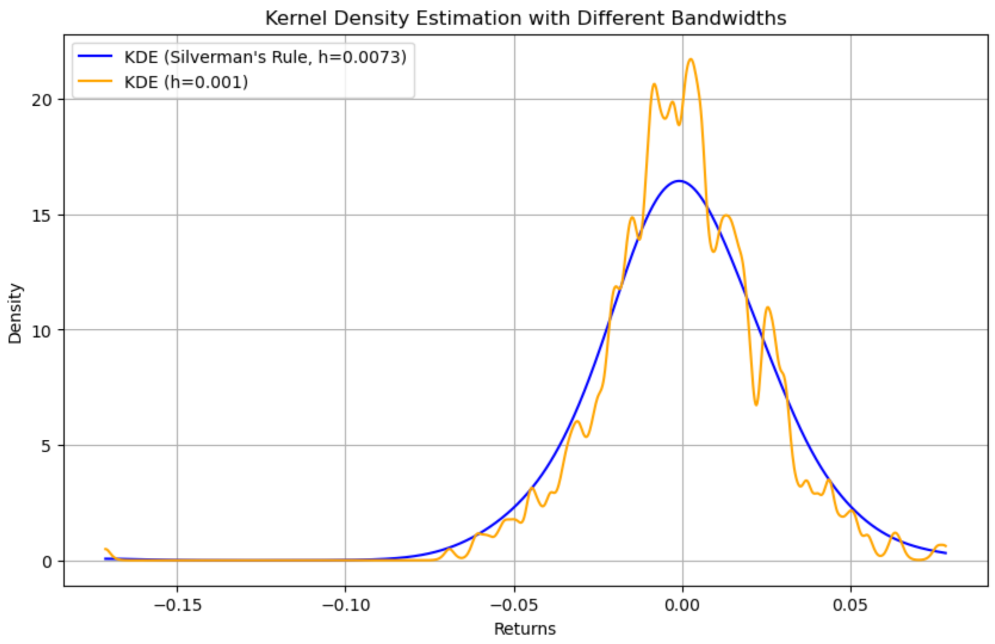

Figure 4.

2.3 Validation of the non-parametric VaR for the period 2017- 2018

To validate the Non-Parametric VaR estimated using data from 2015 to 2016, we calculate the proportion of returns between January 2017 and December 2018 that are less than or equal to the VaR threshold. The validation is performed by computing:

$$\text{Proportion Below VaR} = \frac{1}{n}\sum_{i=1}^{n} \mathbb{1}(R_i \leq \text{VaR Threshold}),$$

where:

- n is the total number of returns in the period,
- R_i is the i-th return in the dataset,
- $\mathbb{1}(R_i \leq \text{VaR Threshold})$ is the indicator function, which equals 1 if $R_i \leq$ VaR Threshold, and 0 otherwise.

If the calculated proportion of returns below or equal to the VaR threshold is less than or equal to 5%, we validate the Non-Parametric VaR model as it aligns with the theoretical confidence level of 95%.

```
# Calculer les returns si nécessaire
data_filtered_period['Return'] = data_filtered_period['Price'].pct_change()
# Définir la VaR donnée
VaR_threshold = non_parametric_var_logistic

# Calculer la proportion des returns inférieurs ou égaux à la VaR
returns = data_filtered_period['Return']
below_var = returns[returns <= VaR_threshold]
proportion_below_var = len(below_var) / len(returns)

# Afficher le résultat
print(f"Proportion of returns < VaR_threshold ({VaR_threshold}): 
 ↪{proportion_below_var * 100:.2f}%")
```

```
Proportion of returns < VaR_threshold (-0.043071874142037225): 1.37%
```

In our case, the calculated proportion of returns below or equal to the VaR threshold is 1.37%, which is well below the expected 5%. Therefore, we validate the Non-Parametric VaR model without any issue, as it aligns with the theoretical confidence level of 95%.

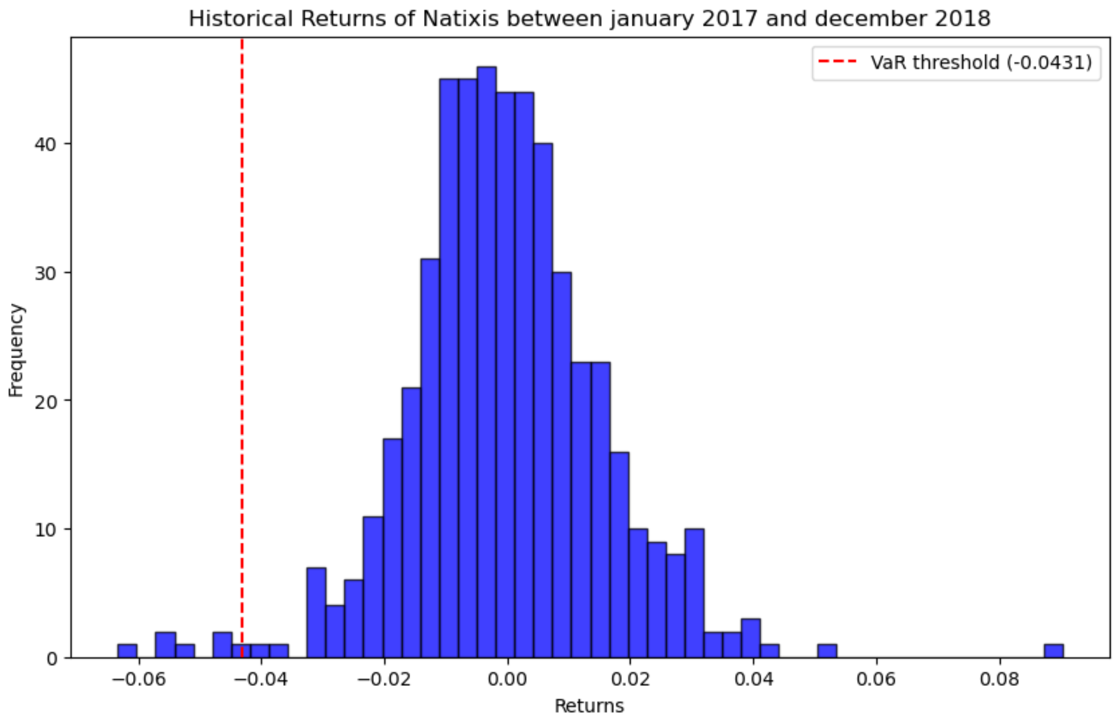

Figure 5.

2.4 Expected Shortfall*

Expected Shortfall (ES), also known as Conditional VaR (CVaR), is a risk measure used in finance to evaluate the average loss beyond a certain confidence level. It is often considered a more informative risk measure than Value at Risk (VaR), as it accounts for the severity of losses in the tail of the distribution.

If X is a random variable representing the returns of a portfolio, and α is the confidence level (e.g., 95%), then the Expected Shortfall is defined as:

$$\mathrm{ES}_\alpha = \mathbb{E}[X \mid X \leq -\mathrm{VaR}_\alpha]$$

In practice, ES provides the average loss in the worst-case scenarios, making it particularly useful for risk management during extreme market conditions.

```python
# Calculate the Expected Shortfall (ES) as the average of returns below␣
↪the VaR
expected_shortfall = data_filtered[data_filtered['Return'] <=␣
↪non_parametric_var_logistic]['Return'].mean()
print(f"ES for the non parametric VaR ({-non_parametric_var_logistic*100:.
↪3f}%): {-expected_shortfall* 100:.3f}%")
```

```
ES for the non parametric VaR (4.307%): 5.820%
```

The Expected Shortfall (ES) represents the average loss beyond the VaR threshold (4.3%). In this case, the average loss, conditional on being in the worst 5% of the distribution (i.e., $X \leq -0.4307$), is 5.820%.

- While VaR captures only the threshold for the worst-case events, ES provides a deeper understanding by quantifying the severity of these extreme losses.
- This suggests that once the portfolio enters the worst 5% of outcomes, the losses become significantly larger on average.

For a portfolio of \$1000, the losses in the worst 5% of scenarios are at least 43.07 dollars (VaR), but the average loss, considering the most extreme cases, amounts to 58.20 dollars (ES). This highlights the importance of Expected Shortfall as an effective risk management tool in extreme market scenarios.

3 Question B

3.1 Monte Carlo VaR for Call Options

The Monte Carlo method is a forward-looking approach for computing Value at Risk (VaR), making it particularly suitable for portfolios with non-linear instruments, such as options. By simulating potential future scenarios for the portfolio's value, it estimates the distribution of outcomes and identifies potential losses at a specific confidence level (e.g., 95%). This method is especially useful when the relationship between the portfolio's value and its underlying assets is complex or non-linear.

In our situation, we aim to calculate the VaR of a call option on the Natixis stock at a one-day horizon. Since the price of the call option is a non-linear function of the underlying stock price, the Monte Carlo method is an appropriate choice. Here's the detailed procedure:

1. Estimate the parameters of a standard Brownian motion on the Natixis stock between 2015 and 2018, using an exponential weighting of the data.
2. Simulate a number N of prices of the stock in a one-day horizon (we are working at the last date of 2018).
3. Transform each of these prices of the underlying in prices of the corresponding call (say at the money, with one-month maturity and 0 risk-free rate and dividend).
4. Pick the empirical quantile of these N call prices to build the VaR of the call.

In this process, we focused on estimating the parameters of a geometric Brownian motion by applying exponential smoothing to the log returns of Natixis stock prices. Starting with historical price data, we calculated logarithmic returns, which account for proportional changes and are more suitable for modeling in financial contexts.

```
# Calculate simple returns
data_filtered_carlo['Return'] = data_filtered_carlo['Price'].pct_change()

# Calculate logarithmic returns
data_filtered_carlo['Log_Return'] = np.log(data_filtered_carlo['Price'] /␣
↪data_filtered_carlo['Price'].shift(1))
data_filtered_carlo.dropna(inplace=True)

# Initialise the smoothing for the first date
data_filtered_carlo.loc[data_filtered_carlo['Date'] == '2015-01-05',␣
↪'Lissage_Log_Return'] = data_filtered_carlo.
↪loc[data_filtered_carlo['Date'] == '2015-01-05', 'Log_Return']
```

The formula for the smoothed log return is:

$$\text{Lissage\_Log\_Return}_t = \lambda \cdot \text{Log\_Return}_t + (1 - \lambda) \cdot \text{Log\_Return}_{t-1}$$

```
# Exponential smoothing parameter
lissage = 0.90
data_filtered_carlo['Lissage_Log_Return'] =␣
↪data_filtered_carlo['Lissage_Log_Return'].
↪fillna(data_filtered_carlo['Log_Return'] * lissage + (1 - lissage) *␣
↪data_filtered_carlo['Log_Return'].shift(1))
data_filtered_carlo.dropna(inplace=True)
```

Here, λ represents the smoothing parameter that determines the weight given to more recent observations. To identify the optimal λ, we used a time series cross-validation approach, testing several values $(0.85, 0.90, 0.94, 0.97)$ and selecting the one minimizing the cross-validated mean squared error (MSE). These values of λ are widely used in finance according to the empirical literature. In our case, the optimal λ was found to be 0.9.

```
# Testing different lambda values
lambdas = [0.85, 0.90, 0.94, 0.97]
# List to store cross-validation errors
errors_cv = []

# Cross-validation for time series
tscv = TimeSeriesSplit(n_splits=3)

for lissage in lambdas:
    mse_fold = []
    for train_index, test_index in tscv.split(data_filtered_carlo):
        train = data_filtered_carlo.iloc[train_index]
        test = data_filtered_carlo.iloc[test_index]

        # Calculate smoothing on training set
        train['Lissage_Test'] = train['Log_Return'] * lissage + \
                                (1 - lissage) * train['Log_Return'].
↪shift(1)

        # Apply smoothing to test set
        test['Lissage_Test'] = train['Lissage_Test'].iloc[-1] * lissage + \
                               (1 - lissage) * test['Log_Return'].shift(1)

        # Calculate the MSE for the current fold
        mse = mean_squared_error(test['Log_Return'].iloc[1:],␣
↪test['Lissage_Test'].iloc[1:])
        mse_fold.append(mse)

    # Add the average MSE for this lambda
    errors_cv.append(np.mean(mse_fold))

# Find the optimal lambda
optimal_lambda_cv = lambdas[np.argmin(errors_cv)]

print(f"Optimal ambda based on cross-validation : {optimal_lambda_cv:.2f}")
```

```
Optimal ambda based on cross-validation : 0.90
```

By emphasizing recent prices through exponential weighting, we better captured the current market dynamics. After applying the smoothing, we computed the mean ($\mu - \frac{\sigma^2}{2}$) and standard deviation (σ) of the smoothed log returns. These parameters are critical for simulating future prices using a geometric Brownian motion model, which is appropriate for financial assets as it ensures strictly positive prices and reflects exponential growth dynamics.

We found that $\mu - \frac{\sigma^2}{2} = 0.0003$ and $\sigma = 0.0186$.

```
mu_minus_sigma_carre = data_filtered_carlo['Lissage_Log_Return'].mean()
sigma = data_filtered_carlo['Lissage_Log_Return'].std()
```

```
mu_minus_sigma_carre
```

```
-0.0003084366224651973
```

```
sigma
```

```
0.018608142679754174
```

Using these parameters, we simulated 1000 prices of the Natixis stock in a one-day horizon using the formula of a geometric Brownian motion:

$$S_{t+1} = S_0 \cdot \exp\left(\mu - \frac{\sigma^2}{2} + \sigma B_t\right),$$

where $S_0 = 4.119$ (the last price of 2018) and $B_t \sim \mathcal{N}(0,1)$.

```
for i in range(1000):
    Bt = np.random.randn()
    Last_Price[i+1] = 4.119 * np.exp(mu_minus_sigma_carre + sigma * Bt)
```

```
Last_Price
```

```
0        4.119000
1        4.113530
2        4.115164
3        4.203774
4        4.119241
```

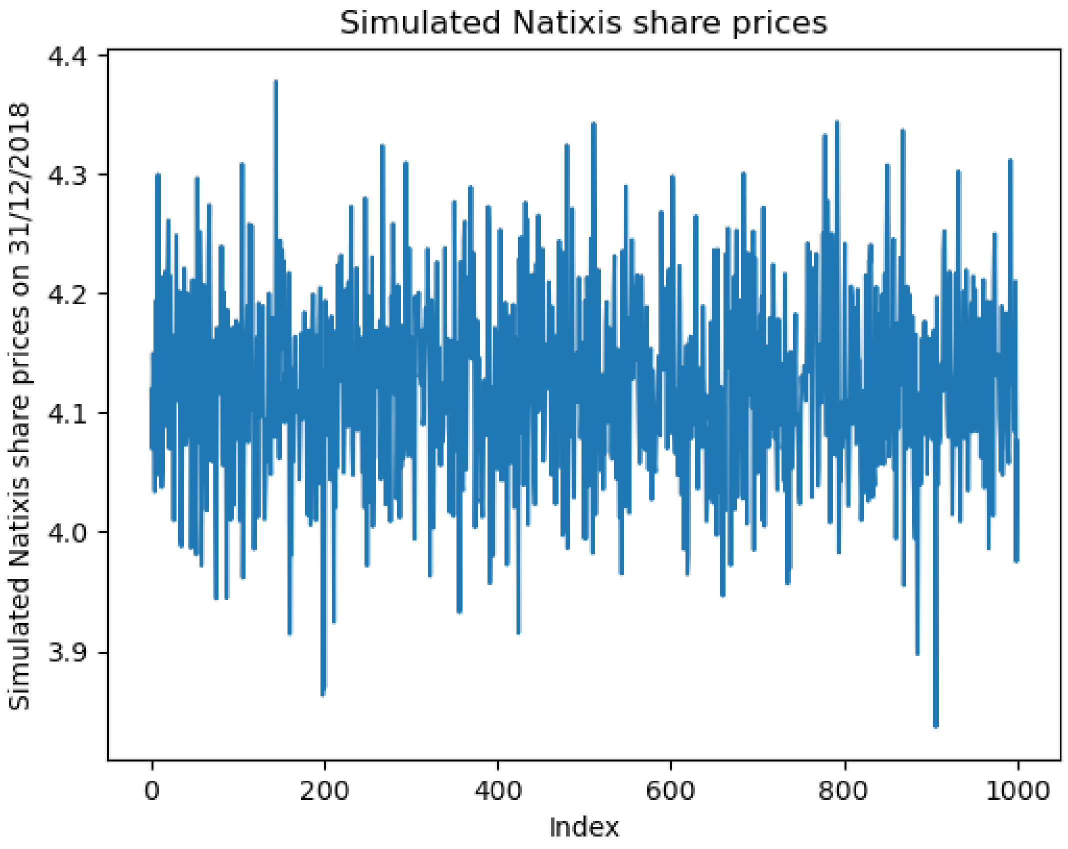

Figure 6.

The choice of $N = 1000$ simulations is a balance between computational efficiency and accuracy. According to the law of large numbers, the accuracy of Monte Carlo simulations improves as N increases. However, larger values of N come with higher computational costs. For $N = 1000$, the relative error, approximately $\frac{1}{\sqrt{1000}} \approx 3.16\%$, is acceptable for financial risk estimation. Additionally, 1000 simulations provide sufficient granularity for identifying the empirical quantiles required to compute the Value at Risk (VaR). While increasing the number of simulations (e.g., $N = 10,000$) reduces the error, it significantly increases the computational cost.

After simulating 1000 stock prices, we use the Black-Scholes formula to price the corresponding call options. The parameters are as follows:

- S_t: Simulated stock prices,
- $K = 4.119$: Strike price (last price of 2018),
- $\sigma = 0.0186$: Volatility,
- $T = 21$: Time to maturity in trading days,
- $r = 0$: Risk-free rate (assumed to be zero),
- $N(x)$: Cumulative distribution function (CDF) of the standard normal distribution.

The Black-Scholes formula for the call option price is:

$$C = S_t \cdot N(d_1) - K \cdot N(d_2),$$

where d_1 and d_2 are calculated as follows:

$$d_1 = \frac{\ln(S_t/K) + \left(r + \frac{\sigma^2}{2}\right)T}{\sigma\sqrt{T}}, \quad d_2 = d_1 - \sigma\sqrt{T}.$$

We use these formulas to compute the call option prices for each of the 1000 simulated stock prices.

```
black_scholes = pd.DataFrame()
# 4.119 is the last historical price
black_scholes["d1"] = (np.log(Last_Price/4.119) + (sigma**2) / 2 * 21) /␣
 ↪(sigma * np.sqrt(21))
black_scholes["d2"] = black_scholes["d1"] - sigma * np.sqrt(21)
black_scholes["Call Price"] = Last_Price * norm.cdf(black_scholes["d1"]) -␣
 ↪4.119 * norm.cdf(black_scholes["d2"])
```

To compute the Value at Risk (VaR) at a 95% confidence level, we first extract the simulated call option prices from the Black-Scholes model. The VaR is determined as the 5th percentile of the distribution of these prices, corresponding to the worst 5% of potential outcomes. Mathematically, this is done using the percentile function:

$$\text{VaR} = \text{Percentile}(1 - \text{confidence\_level}) \cdot 100$$

```
# Define the confidence level
confidence_level = 0.95

# Extract call prices
call_prices = black_scholes["Call Price"]

# Calculate Value at Risk (VaR)
VaR_percentile = np.percentile(call_prices, (1 - confidence_level) * 100)

# Display VaR
print(f"Value at Risk (VaR) at {confidence_level * 100}% confidence level:␣
 ↪{VaR_percentile}")
```

```
Value at Risk (VaR) at 95.0% confidence level: 0.08573364349493018
```

This quantile represents the maximum loss expected under normal market conditions with a 95% confidence level. In our case, the Value at Risk (VaR) at 95.0% confidence level is approximately 0.0857.

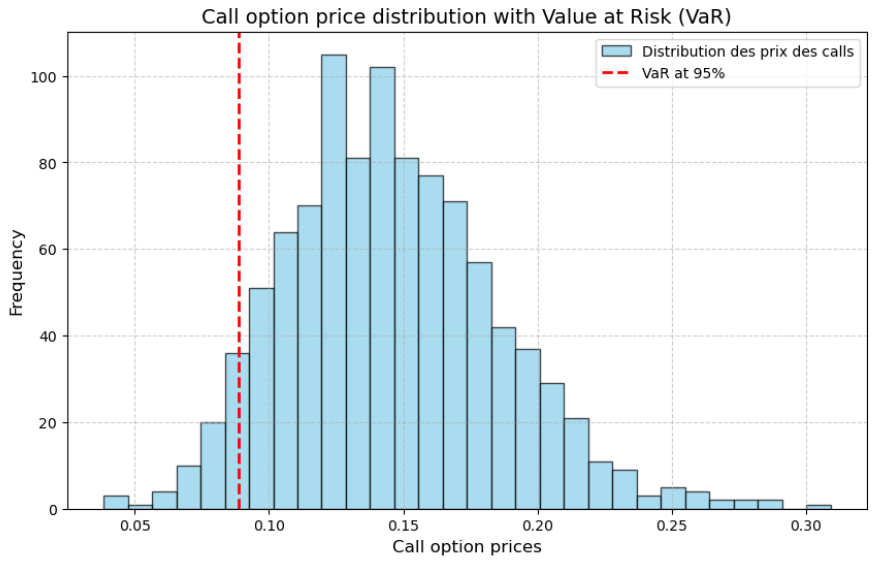

Figure 7.

4 Question C

Introduction

Extreme Value Theory (EVT) is a powerful statistical framework used to analyze the behavior of extreme events, particularly in financial risk management. This methodology is crucial for understanding and quantifying the risks associated with rare but impactful events, such as large market movements.

In this context, we aim to estimate the parameters of the Generalized Extreme Value (GEV) distribution for the tails of daily returns from Natixis data, focusing on both extreme gains and extreme losses. Specifically, we will calculate the Pickands and Hill estimators, as well as the extremal indices, to comprehensively assess the tail behavior of the return distribution.

4.1 GEV Parameters with the estimator of Pickands

Generalized Extreme Value (GEV) Distribution

The GEV distribution combines three types of extreme value distributions into a single formulation:

$$G_\xi(x) = \begin{cases} \exp\left(-\left(1+\xi x\right)^{-1/\xi}\right) & \text{if } \xi \neq 0, \\ \exp\left(-e^{-x}\right) & \text{if } \xi = 0, \end{cases}$$

where ξ is the shape parameter and $x \geq 0$ if $\xi \geq 0$, and $0 \leq x < -\frac{1}{\xi}$ if $\xi \leq 0$.
The nature of the distribution depends on ξ:

- $\xi > 0$: Fréchet (heavy-tailed distribution),
- $\xi = 0$: Gumbel (light-tailed distribution),
- $\xi < 0$: Weibull (finite upper limit for the tail).

Pickands Estimator Theorem

Let $\{X_n\}$ be a sequence of i.i.d. random variables with cumulative distribution function F, belonging to the max-domain of attraction of a GEV distribution with shape parameter $\xi \in \mathbb{R}$. Let $k(n)$ be a function of the sample size n such that:

$$\lim_{n\to\infty} k(n) = \infty \quad \text{and} \quad \lim_{n\to\infty} \frac{k(n)}{n} = 0.$$

The Pickands estimator is defined as:

$$\xi_P(k,n) = \frac{1}{\log(2)} \log\left(\frac{X_{n-k+1:n} - X_{n-2k+1:n}}{X_{n-2k+1:n} - X_{n-4k+1:n}}\right),$$

where $X_{n-k+1:n}$ denotes the $(n-k+1)$-th order statistic of the sample. Under the stated conditions, $\xi_P(k,n)$ converges in probability to ξ as $n \to \infty$.

Moreover, if:

$$\lim_{n\to\infty} \frac{k(n)}{\log(\log(n))} = \infty,$$

then the convergence of $\xi_P(k,n)$ to ξ is almost sure.

Steps for Estimation

1. Calculate Daily Returns: Using the Natixis_Data file, we first compute daily returns as:

$$\text{return}_t = \frac{\text{Price}_t - \text{Price}_{t-1}}{\text{Price}_{t-1}}.$$

2. Filter Returns: We separate positive returns (extreme gains) and negative returns (absolute values for extreme losses) then sort them in ascending order:

- $\text{return}_{\text{positive}} = \text{return}_t$ if $\text{return}_t > 0$,
- $\text{return}_{\text{negative}} = |\text{return}_t|$ if $\text{return}_t < 0$.

```python
# Create a DataFrame for positive returns
return_positive = pd.DataFrame()
return_positive= data_filtered_GEV['return_positive'].dropna()
sort_gains = np.sort(return_positive)

# Create a DataFrame for negative returns (absolute values)
return_negative = pd.DataFrame()
return_negative= data_filtered_GEV['return_negative'].dropna()
sort_losses = np.sort (abs(return_negative))
```

3. Estimate Shape Parameter with $k = \lfloor \log(n) \rfloor$: We apply the Pickands estimator for each tail:

$$\xi_P = \frac{1}{\log(2)} \log\left(\frac{X_{n-k+1:n} - X_{n-2k+1:n}}{X_{n-2k+1:n} - X_{n-4k+1:n}}\right),$$

where $k = \lfloor \log(n) \rfloor$.

```python
import numpy as np

def pickands_estimator_l(extreme_values):

    # Calculate the Pickands estimator for the shape parameter (xi) of the␣
 ↪GEV distribution.

    n = len(extreme_values)

    # Define k as log(n), ensure it's an integer and k > 0
    k = int(np.log(n))
    val_1 = n - k + 1
    val_2 = n - 2 * k + 1
    val_3 = n - 4 * k + 1

    # Pickands estimator formula
    xi = (1 / np.log(2)) * np.log(
        (extreme_values[val_1 - 1 ] - extreme_values[val_2 - 1 ]) /
        (extreme_values[val_2 - 1  ] - extreme_values[val_3 - 1 ])
    )
    return xi
```

4. Results: Using this method with $k = \lfloor \log(n) \rfloor$:

- The shape parameter for extreme gains is estimated as $\xi_{\text{upper}} = 0.5772$,
- The shape parameter for extreme losses is estimated as $\xi_{\text{lower}} = -0.5089$.

```python
# Estimate Xi for both tails
xi_upper_tail_pickands_log = pickands_estimator_l(sort_gains)
xi_lower_tail_pickands__log = pickands_estimator_l(sort_losses)
```

```python
xi_upper_tail_pickands_log
```

```
0.5772338569463286
```

```
xi_lower_tail_pickands__log
```

```
-0.508971577934174
```

The positive shape parameter for extreme gains ($\xi_{\text{upper}} > 0$) indicates a heavy-tailed Fréchet distribution, reflecting the potential for unbounded extreme gains. In contrast, the negative shape parameter for extreme losses ($\xi_{\text{lower}} < 0$) corresponds to a Weibull distribution, implying a finite upper bound for extreme losses.

This analysis highlights the asymmetry in the tails of the return distribution, with gains having potentially unbounded extremes, while losses are capped at a certain threshold.

3. Estimate Shape Parameter with $k = \lfloor \sqrt{n} \rfloor$: We apply the Pickands estimator for each tail:

$$\xi_P = \frac{1}{\log(2)} \log\left(\frac{X_{n-k+1:n} - X_{n-2k+1:n}}{X_{n-2k+1:n} - X_{n-4k+1:n}}\right),$$

where $k = \lfloor \sqrt{n} \rfloor$.

```python
import numpy as np

def pickands_estimator(extreme_values):

    # Calculate the Pickands estimator for the shape parameter (xi) of the␣
 ↪GEV distribution.

    n = len(extreme_values)

    # Define k as sqrt(n), ensure it's an integer and k > 0
    k = int(np.sqrt(n))
    val_1 = n - k + 1
    val_2 = n - 2 * k + 1
    val_3 = n - 4 * k + 1

    # Pickands estimator formula
    xi = (1 / np.log(2)) * np.log(
        (extreme_values[val_1 - 1 ] - extreme_values[val_2 - 1 ]) /
        (extreme_values[val_2 - 1  ] - extreme_values[val_3 - 1 ])
    )
    return xi
```

4. Results: Using this method with $k = \lfloor \sqrt{n} \rfloor$:

- The shape parameter for extreme gains is estimated as $\xi_{\text{upper}} = 0.7195$,
- The shape parameter for extreme losses is estimated as $\xi_{\text{lower}} = 0.4081$.

```
# Estimate Xi for both tails
xi_upper_tail_pickands_sqrt = pickands_estimator(sort_gains)
xi_lower_tail_pickands_sqrt = pickands_estimator(sort_losses)
```

```
xi_upper_tail_pickands_sqrt
```

```
0.7195506129332101
```

```
xi_lower_tail_pickands_sqrt
```

```
0.4081905706152293
```

The positive shape parameter for extreme gains ($\xi_{\text{upper}} = 0.7195$) indicates a heavy-tailed Fréchet distribution, reflecting the potential for unbounded extreme gains. In contrast, the positive shape parameter for extreme losses ($\xi_{\text{lower}} = 0.4081$) also corresponds to a heavy-tailed Fréchet distribution, suggesting that extreme losses, while significant, are not bounded by an upper limit. This suggests a high probability of significant extreme events, emphasizing the potential risk associated with these values.

Interpretation of Results for the Pickands Estimator

The Pickands estimator can vary significantly depending on the choice of the function k, as we have just observed in practice.
In practice, finance often favors using $\sqrt{n}$ as the function for k, as it provides a practical approach that ensures stable and reliable results. On the other hand, $\log(n)$ can be less stable, especially with smaller sample sizes, which is the case here. Therefore, we rely on the second result, where both shape parameters ($\xi_{\text{upper}} = 0.7195$ and $\xi_{\text{lower}} = 0.4081$) are positive, indicating that the distribution of extreme values follows a Fréchet distribution.

This conclusion implies that the risk of significant extreme values occurring is high, emphasizing the heavy-tailed nature of the distribution.

4.2 GEV Parameters with the estimator of Hill*

Hill Estimator Theorem

Let $\{X_n\}$ be a sequence of i.i.d. random variables with cumulative distribution function F, belonging to the max-domain of attraction of a GEV distribution with shape parameter $\xi > 0$. Let $k(n)$ be a function of the sample size n such that:

$$\lim_{n\to\infty} k(n) = \infty \quad \text{and} \quad \lim_{n\to\infty} \frac{k(n)}{n} = 0.$$

Then, the Hill estimator is defined as:

$$\xi_H(k,n) = \frac{1}{k} \sum_{i=n-k+1}^{n} \log\left(\frac{X_{i:n}}{X_{n-k+1:n}}\right),$$

where $X_{i:n}$ denotes the i-th order statistic of the sample, sorted in ascending order. Under the stated conditions, $\xi_H(k,n)$ converges in probability to ξ as $n \to \infty$.

Moreover, if:

$$\lim_{n\to\infty} \frac{k(n)}{\log(\log(n))} = \infty,$$

then the convergence of $\xi_H(k,n)$ to ξ is almost sure.

Steps for Estimation

1. Calculate Daily Returns: Using the Natixis_Data file, we first compute daily returns as:

$$\text{return}_t = \frac{\text{Price}_t - \text{Price}_{t-1}}{\text{Price}_{t-1}}.$$

2. Filter Returns: We separate positive returns (extreme gains) and negative returns (absolute values for extreme losses) then sort them in ascending order:

- $\text{return}_{\text{positive}} = \text{return}_t$ if $\text{return}_t > 0$,
- $\text{return}_{\text{negative}} = |\text{return}_t|$ if $\text{return}_t < 0$.

3. Estimate Shape Parameter with $k = \lfloor\sqrt{n}\rfloor$: We apply the Hill estimator for each tail:

$$\xi_H = \frac{1}{k} \sum_{i=n-k+1}^{n} \log\left(\frac{X_{i:n}}{X_{n-k+1:n}}\right),$$

where $k = \lfloor\sqrt{n}\rfloor$. Here, $X_{i:n}$ denotes the i-th order statistic of the sample, sorted in ascending order. The Hill estimator focuses on the largest k observations to estimate the shape parameter ξ, which characterizes the heavy-tailed nature of the distribution.

```python
import numpy as np

def hill_estimator(extreme_values):

    # Sort the extreme values in ascending order
    extreme_values = np.sort(extreme_values)

    # Length of the data
    n = len(extreme_values)

    # Define k as floor(sqrt(n))
    k = int(np.sqrt(n))

    # Hill estimator formula
    top_k_values = extreme_values[n - k:]
    reference_value = extreme_values[n - k]
    log_ratios = np.log(top_k_values / reference_value)
    xi_hill = (1 / k) * np.sum(log_ratios)

    return xi_hill
```

4. Results: Using this method with $k = \lfloor \sqrt{n} \rfloor$:

- The shape parameter for extreme gains is estimated as $\xi_{\text{upper}} = 0.2153$,
- The shape parameter for extreme losses is estimated as $\xi_{\text{lower}} = 0.2195$.

```python
# Estimate Xi for both tails
xi_upper_tail_hill_sqrt = hill_estimator(sort_gains)
xi_lower_tail_hill_sqrt = hill_estimator(sort_losses)
```

```python
xi_upper_tail_hill_sqrt
```

```
0.21537827843774737
```

```python
xi_lower_tail_hill_sqrt
```

```
0.21952614389640798
```

Interpretation of Results for the Hill Estimator

Using the Hill estimator with $k = \lfloor\sqrt{n}\rfloor$, the shape parameters for the tails of the distribution are estimated as follows:

- **Positive Tail (ξ_{upper}):** The shape parameter for extreme gains is estimated as $\xi_{\text{upper}} = 0.2153$, suggesting a moderately heavy-tailed behavior for extreme gains.
- **Negative Tail (ξ_{lower}):** The shape parameter for extreme losses is estimated as $\xi_{\text{lower}} = 0.2195$, indicating a similar moderately heavy-tailed behavior for extreme losses.

Both shape parameters being positive indicates that the tails of the distribution follow a Fréchet distribution, which is characterized by heavy tails. While the values suggest moderate extremity, the risks associated with extreme events remain significant. This result is consistent with the Pickands estimator using the same function $k = \lfloor\sqrt{n}\rfloor$, further validating the heavy-tailed nature of the distribution.

4.3 Extremal Index using Runs declustering

In extreme value theory, the extremal index (θ) quantifies the clustering of extreme events in a time series. It is estimated using declustering methods, such as block declustering and run declustering.

Here, we focus on run declustering, where clusters are identified based on a sliding window of size r.

Run declustering concept:

- The method divides the data into clusters based on the spacing between extreme events.
- A cluster ends, and a new one starts, if no extreme event is observed within a window of length r.
- This method is well-suited for time series with dependence, such as financial returns.

The extremal index is linked to the ratio of the number of extreme clusters to the total number of extremes. For run declustering, it is estimated as:

$$\hat{\theta}_n^R(u;r) = \frac{\sum_{i=1}^{n-r} 1(X_i > u, M_{i,i+r} \leq u)}{\sum_{i=1}^{n-r} 1(X_i > u)}$$

Where:

- u: Threshold for defining extreme events.
- r: Run length or sliding window size.
- X_i: Observations (e.g., returns).
- $M_{i,i+r} = \max\{X_{i+1}, \ldots, X_{i+r}\}$: Maximum within the window $[i+1, i+r]$.
- $1(\cdot)$: Indicator function, which equals 1 if the condition inside is true, otherwise 0.

Interpretation:

- If the variables are independent, each extreme constitutes its own cluster, leading to $\theta = 1$.
- If every extreme is followed by another extreme, the clustering reduces the extremal index, resulting in $\theta < 1$.

```
def estimate_theta(returns, threshold_percentile=95, run_length=5):
    # Define the threshold
    threshold = np.percentile(returns, threshold_percentile)

    # Identify extreme indices
    extreme_indices = np.where(returns > threshold)[0]

    # Identify clusters
    clusters = np.split(extreme_indices, np.where(np.diff(extreme_indices)␣
 ↪> run_length)[0] + 1)

    # Calculate θ
    N_e = len(extreme_indices)
    N_c = len(clusters)

    return N_c / N_e if N_e > 0 else 0
```

For our analysis, we chose a run length of $r = 5$, which is commonly used in financial studies to capture short-term clustering of extreme events. The threshold was set at the 95% percentile, focusing on the top 5% of the most extreme values. This choice is widely adopted in finance to analyze tail risks, as it balances the need to isolate rare events while retaining sufficient data for meaningful statistical analysis.

```
return_negative = abs(return_negative)
```

```
# Estimate θ for positive returns
theta_positive = estimate_theta(return_positive, threshold_percentile=95,␣
↪run_length=5)
print(f"Extreme index (θ) for extreme gains: {theta_positive:.4f}")

# Estimate θ for negative returns
theta_negative = estimate_theta(return_negative, threshold_percentile=95,␣
↪run_length=5)
print(f"Extreme index (θ) for extreme losses:  {theta_negative:.4f}")
```

```
Extreme index (θ) for extreme gains: 0.6000
Extreme index (θ) for extreme losses:  0.5769
```

Interpretation of the extremal index:

The calculated extremal indices (θ) provide insight into the clustering of extreme returns. For extreme gains, $\theta = 0.6000$ indicates that approximately 60% of extreme positive returns occur as isolated events, while 40% are part of clusters. This suggests a moderate level of clustering for gains, consistent with market uptrends where positive returns are sporadic but exhibit some clustering.

For extreme losses, $\theta = 0.5769$ shows that around 57.7% of extreme negative returns are isolated, while the remaining 42.3% occur in clusters. This reflects slightly stronger clustering of losses, a common feature of financial markets during crises or periods of high volatility, where sharp downturns often lead to consecutive extreme negative events.

Overall, the results indicate moderate clustering for both gains and losses, with losses exhibiting slightly stronger clustering. This aligns with the observation that market downturns tend to be sharper and more persistent than uptrends, highlighting the importance of robust risk management during periods of market stress.

5 Question D

The Almgren and Chriss model is a mathematical framework designed to optimize the liquidation of assets in a portfolio. Its primary objective is to find a balance between:

- **Minimizing market impact**: reducing the effect of liquidation orders on the asset price.
- **Reducing market risk exposure**: limiting losses due to price volatility during the liquidation period.

The total quantity X to be liquidated is divided into tranches $n_1, n_2, \ldots, n_k$, where:

$$x_k = X - \sum_{j=1}^{k} n_j$$

x_k represents the remaining quantity to be liquidated after k steps.

Liquidation occurs in discrete time, with a time step τ. This means liquidation decisions are made at regular intervals.

Optimization Problem

The model seeks to minimize an objective function that combines:

- The expected value of the loss ($\mathbb{E}(x)$), which includes permanent and temporary market impact.
- The variance of the loss ($\mathrm{Var}(x)$), representing the risk due to market volatility.

The objective function is defined as:

$$\mathbb{E}(x) + \lambda \mathrm{Var}(x)$$

where λ is a risk aversion parameter that balances cost minimization and risk reduction.

The expected value of the loss is given by:

$$\mathbb{E}(x) = \sum_{k=1}^{N} \tau x_k \, g\left(\frac{n_k}{\tau}\right) + n_k h\left(\frac{n_k}{\tau}\right)$$

where:

- $g\left(\frac{n_k}{\tau}\right)$: the *permanent impact function*, which depends on the trading speed or rate of trading $\frac{n_k}{\tau}$.
- $h\left(\frac{n_k}{\tau}\right)$: the *temporary impact function*, reflecting temporary price changes.
- n_k: the size of the transaction at time k.
- τ: the trading interval.

The variance of the loss is given by:

$$\mathrm{Var}(x) = \sigma^2 \sum_{k=1}^{N} \tau x_k^2$$

where:

- σ^2: the variance of the asset price, representing its volatility.

Liquidation Strategies

By liquidating at a constant speed ($n_k = X/N$), market impact is minimized. However, the variance can be high because the portfolio remains exposed to market risk throughout the liquidation period.

By liquidating everything at the first step ($n_1 = X$ and $n_2 = \ldots = n_N = 0$), the variance is eliminated ($\mathrm{Var}(x) = 0$), but the market impact is maximized. In this case, the expected cost is:

$$\mathbb{E}(x) = \xi X + \eta \frac{X^2}{\tau}$$

Between these two extremes lies an efficient frontier for various levels of risk aversion (λ). This corresponds to the solution of the Lagrangian problem, solved using Euler-Lagrange equations. The optimal strategy is given by:

$$x_k = \frac{\sinh\left(K\left(T - \left(k - \frac{1}{2}\tau\right)\right)\right)}{\sinh(KT)} X$$

where:

- $K \sim \sqrt{\frac{\lambda\sigma^2}{\eta}} + \mathcal{O}(\tau)$ as $\tau \to 0$.
- T: the total liquidation time.

5.1 Parameter Estimation of the Almgren and Chriss Model

In this analysis, we need to estimate three key parameters of the Almgren and Chriss model: the volatility (σ), which reflects the asset's price uncertainty; the permanent market impact parameter (γ), which quantifies the lasting effect of trades on the price; and the temporary market impact parameter (η), which captures the transient price changes caused by trading.

Estimation of Annualized Volatility

Our dataset contains hourly data, allowing us to calculate the hourly returns using the following formula:

$$r_t = \frac{P_t - P_{t-1}}{P_{t-1}}$$

where:

- P_t is the price at hour t,
- P_{t-1} is the price at the previous hour $(t-1)$,
- r_t represents the hourly return.

```
returns = (Grenchriss['Price (before transaction)'] - Grenchriss['Price␣
↪(before transaction)'].shift(1)) /  Grenchriss['Price (before␣
↪transaction)'].shift(1)
returns.dropna(inplace=True)
```

Once the hourly returns are calculated, we estimate the hourly volatility by computing the standard deviation of hourly returns, denoted as $\text{std}(r_t)$.

To obtain an annualized volatility, we adjust this hourly volatility by considering the number of trading hours per day (24) and the number of effective trading days per year (252):

$$\sigma = \text{std}(r_t) \times \sqrt{24} \times \sqrt{252}$$

Assuming that markets operate 24 hours a day, this method converts the hourly volatility into an annualized measure.

```
sigma = np.std(returns)*np.sqrt(24)*np.sqrt(252)
sigma
```

```
0.057397522889744174
```

Using this method, we obtain an annualized volatility (σ) of 5.7%, reflecting the uncertainty associated with the asset's price fluctuations on an annual basis.

Estimation of the permanent market impact parameter (γ)

The Almgren and Chriss model suggests that the permanent market impact is directly proportional to the trading rate ($\frac{n_k}{\tau}$). To estimate the parameter γ, a linear regression analysis is performed.

Linear Regression Model: The regression is based on the following model:

$$P(t+2) - P(t+1) = \gamma(\frac{n_k}{\tau})$$

where:

- $P(t+2)-P(t+1)$: The change in the asset's price between times t and $t+2$, representing the permanent impact on price.
- $\frac{n_k}{\tau}$: The trading rate, defined as the volume of trades n_k executed over a time interval τ, divided by the interval duration.
- γ: The parameter quantifying the permanent impact of trading on price.

```python
# Calculation of differences with every second element skipped
price_difference = [
    price.iloc[i + 1] - price.iloc[i]
    for i in range(0, len(price) - 1, 2)
]

# Retrieve the corresponding indices
index_difference = price.index[::2][:len(price_difference)]

# Create a DataFrame with the original indices
price_difference_df = pd.DataFrame({
    'Price Difference': price_difference
}, index=index_difference)
```

In practice, the trading rate ($\frac{n_k}{\tau}$) is calculated as the product of the trade volume n_k and the *sign of the transaction*, where:

- $+1$: For buy transactions.
- -1: For sell transactions.

```
# Multiply the elements of 'volume' and 'sign_Transaction' while␣
↪preserving the indices
signed_volume = {
    idx: volume[idx] * sign_Transaction[idx]
    for idx in volume.index
    if idx in sign_Transaction.index
}

# Convert to DataFrame for a more readable structure
signed_volume_df = pd.DataFrame(list(signed_volume.items()),␣
↪columns=['Index', 'Signed Volume']).set_index('Index')
```

```
# Filter impact_df to retain only the indices present in signed_volume_df
filtered_impact_df = price_difference_df[price_difference_df.index.
↪isin(signed_volume_df.index)]
signed_volume_df = signed_volume_df[signed_volume_df.index.
↪isin(filtered_impact_df.index)]
```

```
# Extract the 'Permanent Impact' column from filtered_impact_df
# Convert to NumPy array
Price_diff = filtered_impact_df['Price Difference'].values
Produit_signe = signed_volume_df['Signed Volume'].values
```

The slope of the regression line, denoted as γ, quantifies the permanent market impact caused by trading. A higher γ indicates a stronger price impact per unit of trading rate.

```
def linear_regression_np(X, y):
    # Means of the data
    X_mean = np.mean(X)
    y_mean = np.mean(y)

    # Calculation of the slope
    slope = np.sum((X - X_mean) * (y - y_mean)) / np.sum((X - X_mean) ** 2)

    # Calculation of intercept
    intercept = y_mean - slope * X_mean

    return slope, intercept
```

```
slope, intercept = linear_regression_np(Produit_signe,Price_diff)
```

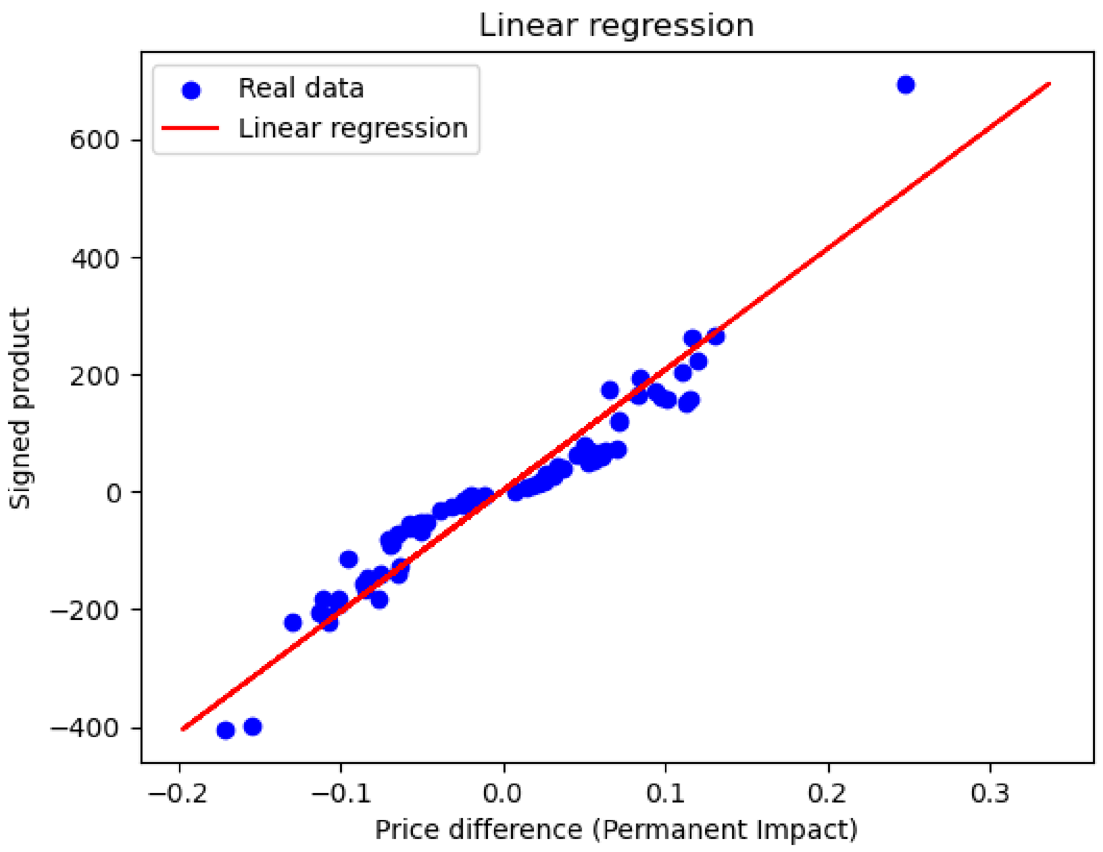

Figure 8.

After performing the linear regression to estimate the parameter γ, the following results were obtained:

```python
import numpy as np

# Calculate the MSE (Mean Squared Error)
mse_gamma = np.mean((Price_diff - y_pred) ** 2)

# Calculation of the coefficient of determination R^2
r2_gamma = 1 - np.sum((Price_diff - y_pred) ** 2) / np.sum((Price_diff -␣
↪np.mean(Price_diff)) ** 2)
```

- Estimated γ (slope): 0.000486,
- Mean Squared Error (MSE): 0.000505,
- Coefficient of Determination (R^2): 0.919.

These results are highly satisfactory for several reasons:

1. High R^2 ($R^2 = 0.919$): This indicates that approximately 91.9% of the variance in price changes ($P(t+2) - P(t)$) is explained by the trading rate ($\frac{n_k}{\tau}$). A high R^2 value suggests that the model is well-fitted to the data.

2. Low Mean Squared Error (MSE): The MSE, which measures the average squared difference between predicted and observed values, is very low (0.000505). This indicates that the model's predictions are close to the actual values, providing a highly accurate estimation of γ.

3. Precise estimation of γ: The estimated value of γ (0.000486) reflects a proportional relationship between price changes and trading rate. This result aligns with the Almgren and Chriss model, where γ quantifies the permanent price impact per unit of trading rate.

Estimation of the temporary market impact parameter (η)

To estimate the parameter η, which represents the transient impact in the Almgren and Chriss model, we performed a linear regression where:

- The dependent variable is $P(t+1) - P(t+2)$, representing the transient price impact over two time steps.

- The explanatory variable is $\left(\frac{n_k}{\tau}\right)^2$, defined as the square of the transaction volume multiplied by the sign of the transaction ($+1$ for a buy and -1 for a sell).

```
price_difference_transitoire = - price_difference_df
# Filter impact_df to keep only the indices present in signed_volume_df
filtered_impact_tr =␣
↪price_difference_transitoire[price_difference_transitoire.index.
↪isin(signed_volume_carre.index)]
signed_volume_carre = signed_volume_carre[signed_volume_carre.index.
↪isin(filtered_impact_tr.index)]
```

```
# Multiply the elements of 'volume' and 'sign_Transaction' while␣
↪preserving the indices
signed_volume_carre = {
    idx: volume[idx]*volume[idx] * sign_Transaction[idx]
    for idx in volume.index
    if idx in sign_Transaction.index
}

# Convert to DataFrame for a more readable structure
signed_volume_carre = pd.DataFrame(list(signed_volume_carre.items()),␣
↪columns=['Index', 'Signed Volume Carre']).set_index('Index')
```

```
# Extract the 'Permanent Impact' column from filtered_impact_df
# Convert to NumPy table
Price_diff_trans = filtered_impact_tr['Price Difference'].values
Produit_signe_carre = signed_volume_carre['Signed Volume Carre'].values
```

```
slope_d, intercept_d =␣
 ↪linear_regression_np(Produit_signe_carre,Price_diff_trans)
```

```
# Predictions based on regression
y_pred_d = slope_d * Produit_signe_carre + intercept_d
```

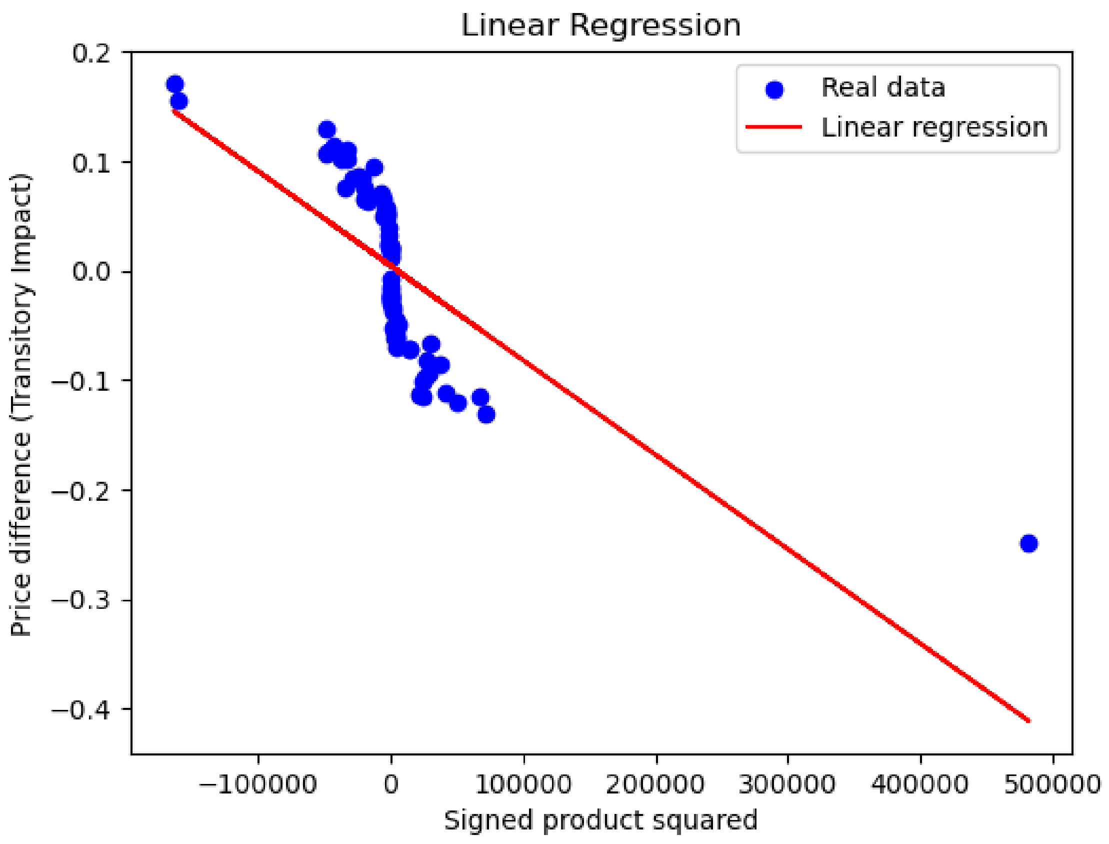

Figure 9.

The results obtained are as follows:

```
# Calculate the MSE (Mean Squared Error)
mse_eta = np.mean((Price_diff_trans - y_pred_d) ** 2)

# Calculation of the coefficient of determination R^2
r2_eta = 1 - np.sum((Price_diff_trans - y_pred) ** 2) / np.
 ↪sum((Price_diff_trans - np.mean(Price_diff_trans)) ** 2)
```

- **Estimated η (slope):** -8.64×10^{-7},
- **Mean Squared Error (MSE):** 0.00315,
- **Coefficient of Determination (R^2):** -2.76.

These results indicate that the linear regression provides unsatisfactory outcomes:

- The slope (η) is very close to zero, indicating an almost nonexistent relationship between $P(t+1) - P(t+2)$ and $\left(\frac{n_k}{\tau}\right)^2$.
- A negative R^2 (-2.76) suggests that the model predicts the variations worse than a simple mean of the data. This reflects poor model specification or invalid linear assumptions.

Given the unsatisfactory results from the linear regression, we transitioned to a multivariate regression approach to better capture the complexity of the relationships between variables. We will adopt a non-linear approach by integrating the quadratic cost function of the initial linear regression with the first equation. This results in the formulation of the following multivariate regression model:

$$h\left(n_k^2 \cdot \text{Sign}(n_k)\right) = \eta\left(n_k^2 \cdot \text{Sign}(n_k)\right) = P(t+1) - P(t+2)$$

Where $h\left(n_k^2 \cdot \text{Sign}(n_k)\right)$ represents the price change due to the transitory market impact, and $n_k^2 \cdot \text{Sign}(n_k)$ is the squared signed volume of the trade.

In this setup:

- The dependent variable remains $P(t+1) - P(t+2)$, which represents the transient price impact over two time steps.
- The explanatory variables include:
  - signed_Product = Volume $\times$ Transaction Sign, which captures the directional impact of trading activity.
  - signed_Product_square = Volume^2 $\times$ Transaction Sign, which models the nonlinear relationship between the transaction volume and the transient impact.

The features were combined into a design matrix, X_{tharray}, defined as:

$$X_{\text{tharray}} = \begin{bmatrix} \text{signed\_Product}_1 & \text{signed\_Product\_square}_1 \\ \text{signed\_Product}_2 & \text{signed\_Product\_square}_2 \\ \vdots & \vdots \end{bmatrix}$$

We performed the multivariate regression using:

$$\text{Price\_diff\_trans} = P(t+1) - P(t+2)$$

```python
import numpy as np

def linear_regression_multivariate(X, y):
    # Variable averages
    X_mean = np.mean(X, axis=0)
    y_mean = np.mean(y)

    # Covariance matrix of X
    X_cov = np.dot(X.T, X) - np.sum(X, axis=0) * X_mean

    # Inverse of the covariance matrix
    X_cov_inv = np.linalg.inv(X_cov)

    # Calculation of cross products between X and y
    X_X_mean_y_mean = np.dot(X.T, y) - np.sum(X, axis=0) * y_mean

    # Calculation of coefficients (slopes)
    coefficients = np.dot(X_cov_inv, X_X_mean_y_mean)

    # Calculation of intercept
    intercept = y_mean - np.dot(coefficients.T, X_mean)

    return coefficients, intercept
```

```python
Xtharray = np.column_stack((Produit_signe, Produit_signe_carre))
```

```python
coefficients, intercept_e =␣
 ↪linear_regression_multivariate(Xtharray,Price_diff_trans)
```

```
# Calculation of predictions
y_pred_tha = np.dot(Xtharray, coefficients) + intercept_e

# Sort the values of Product_sign_carre and y_pred_multivariate
sorted_indices = np.argsort(Produit_signe_carre)
Produit_signe_carre_sorted = Produit_signe_carre[sorted_indices]
y_pred_sorted = y_pred_tha[sorted_indices]
```

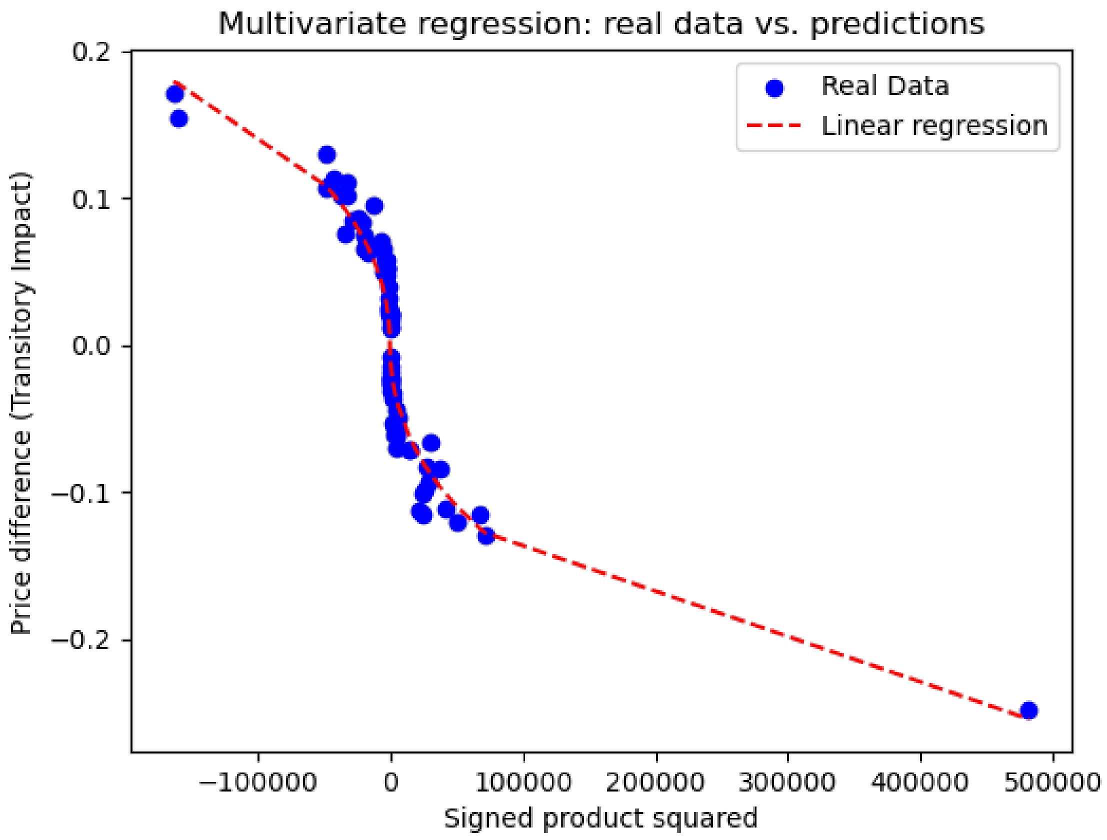

Figure 10.

The results obtained from the regression are significantly improved:

```
# Calculation of evaluation metrics
mse_tha = np.mean((Price_diff_trans - y_pred_tha) ** 2)
r2_tha = 1 - sum((Price_diff_trans - y_pred_tha) ** 2) /␣
↪sum((Price_diff_trans - np.mean(Price_diff_trans)) ** 2)
```

```
tau = 1/24
etha = coefficients[1] * tau
etha
```

- **Estimated η:** 1.091×10^{-8},
- **Mean Squared Error (MSE):** 0.000314,
- **Coefficient of Determination (R^2):** 0.950.
- The much higher R^2 value (0.950) indicates that the multivariate regression explains approximately 95% of the variance in $P(t+1) - P(t+2)$, demonstrating a significantly better fit compared to the linear regression.
- The lower MSE (0.000314) confirms that the model predictions are much closer to the observed data, reducing prediction errors.
- The estimated η (1.091×10^{-8}) quantifies the transient price impact, capturing both linear and nonlinear effects of transaction volume and direction.

By incorporating both linear and nonlinear effects into the explanatory variables, the multivariate regression provides a robust and accurate estimation of η. This approach addresses the shortcomings of the initial linear regression and highlights the importance of considering more complex relationships in modeling transient market impacts.

5.2 Liquidation Strategy

The Almgren and Chriss model provides a framework for determining optimal liquidation strategies. The model balances the trade-off between market risk (exposure to price volatility over time) and liquidity risk (the cost of trading large volumes quickly). This trade-off is controlled by the risk aversion parameter λ, which reflects the trader's tolerance for these competing risks.

Examined Scenarios: We compute and analyze liquidation trajectories for three different values of λ: $\lambda = 1 \times 10^{-5}$, $\lambda = 1 \times 10^{-4}$, and $\lambda = 1 \times 10^{-3}$. The liquidation is modeled over a 24-hour period, with T representing the fractional part of the day.

For each λ, the parameter K, which governs the liquidation trajectory, is calculated using the formula:

$$K = \sqrt{\frac{\lambda \sigma^2}{\eta}}$$

where:

- σ: The volatility of the asset, estimated as 5.7% annually.
- η: The transient market impact parameter, estimated as 1.091×10^{-8}.
- λ: The risk aversion parameter.

The number of shares to be liquidated at each time step, denoted as x_k, is determined by the following formula:

$$x_k = \frac{\sinh\left(K\left(T - \left(k - \frac{1}{2}\tau\right)\right)\right)}{\sinh(KT)} X$$

where:

- X: The total volume to be liquidated.
- T: The total liquidation time.
- τ: The time step interval (1 hour in this case).
- k: The time step index.
- K: The parameter governing the liquidation trajectory, as defined above.

```
# Parameters
X = 1000000 # Initial volume to be liquidated
T = [(1 / 24) * i for i in range(1, 25)]  # Liquidation horizon in days
Lambda_values = [1e-5, 1e-4, 1e-3] # Different lambda values
sigma = 0.057
tha = 1.0911524540649665e-08

# Plot optimal liquidation paths
plt.figure(figsize=(10, 6))

for Lambda in Lambda_values:
    # Calculate K for each Lambda
    K = np.sqrt(Lambda * (sigma**2) / tha)
    print(f"K for λ={Lambda:.1e}: {K}")

    # Calculate strategic trajectories
    strat = [np.sinh(K * (1 - t)) * X / np.sinh(K) for t in T]
    # Plot the trajectory
    plt.plot(T, strat, label=f'λ = {Lambda:.1e}')

# Title, axes and legend
plt.title("Optimal Liquidation Trajectories for Different Values of λ")
plt.xlabel("Time (days)")
plt.ylabel("Number of Shares")
plt.legend()
plt.grid(True)
plt.show()
```

```
K for λ=1.0e-05: 1.7255682401912498
K for λ=1.0e-04: 5.456725897052854
K for λ=1.0e-03: 17.255682401912498
```

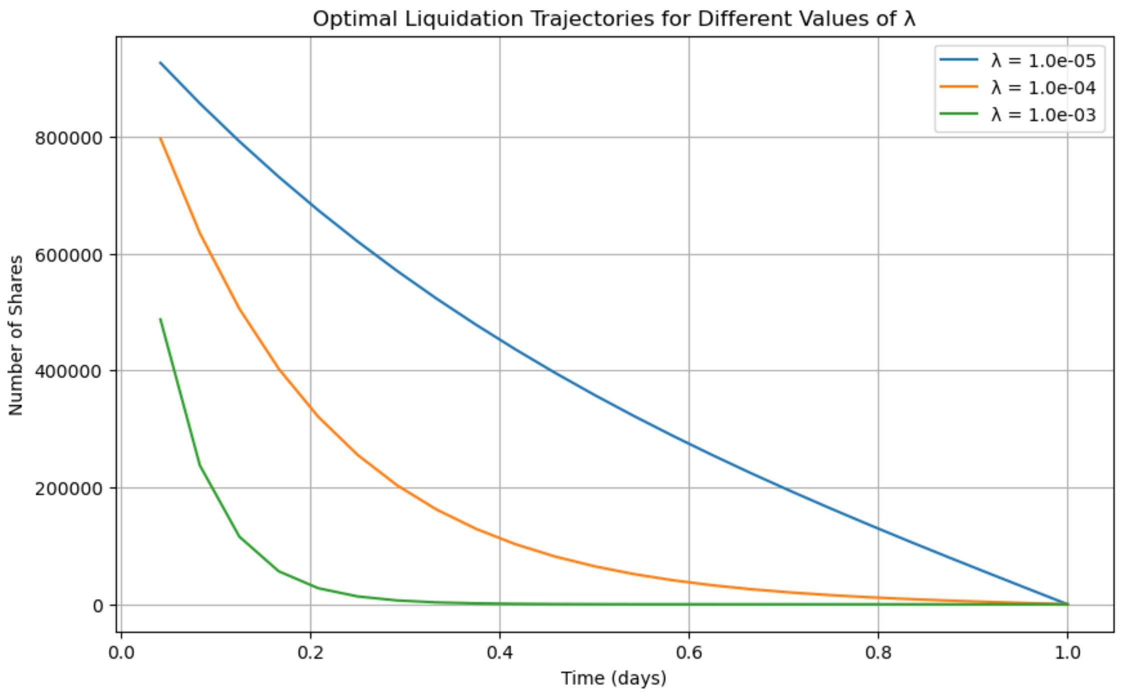

Figure 11.

The computed trajectories illustrate how λ influences the liquidation strategy:

- **For higher λ:** A larger λ (e.g., 1×10^{-3}) reflects greater risk aversion to market volatility. This leads to more aggressive liquidation at the beginning of the trading period, as the trader seeks to minimize exposure to market risk by accepting higher liquidity costs.
- **For lower λ:** A smaller λ (e.g., 1×10^{-5}) reflects lower risk aversion to market volatility. This results in a more gradual liquidation, prioritizing reduced liquidity costs at the expense of increased market risk exposure.

These strategies highlight the importance of calibrating λ based on market conditions and the trader's objectives. In practice, a trader with access to hourly transactions would adapt the liquidation trajectory according to the selected λ to balance the liquidity-market risk trade-off effectively.

Conclusion: The Almgren and Chriss framework demonstrates that the optimal liquidation strategy is not static but depends on the trader's risk preferences, encapsulated by λ. By choosing appropriate λ values, the trader can either prioritize faster liquidation to mitigate market risk or opt for slower liquidation to reduce liquidity costs.

6 Question E

Haar wavelets are among the simplest wavelet functions, widely used for signal processing, data compression, and multiresolution analysis. The Haar wavelet transform decomposes a time series into approximations and details at various scales, enabling the study of relationships between variables across different resolutions. This makes wavelets particularly useful in financial data analysis, where patterns and correlations can vary significantly depending on the time scale.

In this context, Haar wavelets allow us to analyze the multiresolution correlation between foreign exchange (FX) rates. Specifically, we aim to determine the multiresolution correlation between three currency pairs: GBPEUR, SEKEUR, and CADEUR, using their average prices (calculated as the mean of the highest and lowest prices for each time step). This approach helps capture finer details of correlations across different levels, offering insights into how market interactions vary across time scales.

6.1 Multiresolution Correlation between the pairs of FX rates

The mean prices for each currency pair were calculated using the high and low prices:

$$\text{Mean\_Price} = \frac{\text{High\_Price} + \text{Low\_Price}}{2}.$$

```
GBPEUR['Mean_Price'] = (GBPEUR['HIGH_GBPEUR'] + GBPEUR['LOW_GBPEUR'])/2
SEKEUR['Mean_Price'] = (SEKEUR['HIGH_SEKEUR'] + SEKEUR['LOW_SEKEUR'])/2
CADEUR['Mean_Price'] = (CADEUR['HIGH_CADEUR'] + CADEUR['LOW_CADEUR'])/2
```

Daily returns were then computed as:

$$\text{Returns} = \frac{\text{Mean\_Price}_t - \text{Mean\_Price}_{t-1}}{\text{Mean\_Price}_{t-1}}.$$

```
GBPEUR['GBPEUR_RETURNS'] = ((GBPEUR['Mean_Price'] - GBPEUR['Mean_Price'].
 ↪shift(1)) / GBPEUR['Mean_Price'].shift(1)).dropna()
SEKEUR['SEKEUR_RETURNS'] = ((SEKEUR['Mean_Price'] - SEKEUR['Mean_Price'].
 ↪shift(1)) / SEKEUR['Mean_Price'].shift(1)).dropna()
CADEUR['CADEUR_RETURNS'] = ((CADEUR['Mean_Price'] - CADEUR['Mean_Price'].
 ↪shift(1)) / CADEUR['Mean_Price'].shift(1)).dropna()

GBPEUR.dropna(inplace=True)
SEKEUR.dropna(inplace=True)
CADEUR.dropna(inplace=True)
```

The resulting series of returns were used as inputs to the Haar wavelet transform.

The Haar wavelet transform was applied to the return series of each currency pair. At each iteration, the transform computed:

- Low-frequency components (averages), capturing the trend at the current resolution level.
- High-frequency components (differences), capturing short-term fluctuations.

```
def haar_transform(data):
    data = np.array(data, dtype=float)  # Ensure the data is in a mutable␣
 ↪array
    n = len(data)
    output = np.zeros(n)  # Initialize output array
    while n > 1:
        n = n // 2
        for i in range(n):
            # Compute averages (low frequencies)
            output[i] = (data[2 * i] + data[2 * i + 1]) / 2
            # Compute differences (high frequencies)
            output[n + i] = (data[2 * i] - data[2 * i + 1]) / 2
        # Update data for the next iteration
        data[:2 * n] = output[:2 * n]
    return output
```

The transform continues until the resolution level matches the size of the dataset. For our analysis, we used all available data, with $2^{14} = 16,384$ as the maximum resolution level.

To investigate the correlations between the currency pairs at different resolution levels, we used the following steps:

1. Truncation at Each Level: At each resolution level l, the Haar-transformed series were truncated to the first 2^l elements to ensure equal lengths.
2. Correlation Calculation: The Pearson correlation coefficient was computed for each pair of Haar-transformed series:

$$\text{Correlation} = \frac{\text{Cov}(\text{Haar1}, \text{Haar2})}{\sigma_{\text{Haar1}}\sigma_{\text{Haar2}}}.$$

The correlation matrix at each level was computed for the three currency pairs. For each level:

- The Haar-transformed series were grouped into a list.
- Pairwise correlations were calculated and stored in a symmetric matrix.
- The diagonal elements were set to 1, as they represent the self-correlation of each series.

```
# Multiresolution correlation with data validation
def multiresolution_correlation(haar1, haar2, level):
    length = 2 ** level
    if length > len(haar1) or length > len(haar2):
        raise ValueError(f"Level {level} exceeds the length of Haar␣
↪transforms.")

    truncated_haar1 = haar1[:length]
    truncated_haar2 = haar2[:length]

    # Check whether the series are constant or empty
    if np.std(truncated_haar1) == 0 or np.std(truncated_haar2) == 0:
        return 0

    return np.corrcoef(truncated_haar1, truncated_haar2)[0, 1]
```

```
# Correlation matrix at the given level
def correlation_matrix_at_level(level, haar_transforms):
    import pandas as pd  # For better formatting of the output

    matrix_size = len(haar_transforms)
    correlation_matrix = np.zeros((matrix_size, matrix_size))

    print(f"\n--- Correlation Matrix at Level {level} ---")

    for i in range(matrix_size):
        for j in range(i + 1, matrix_size):
            try:
                correlation =␣
↪multiresolution_correlation(haar_transforms[i], haar_transforms[j],␣
↪level)
            except ValueError as e:
                print(f"Error at level {level}, pair ({i}, {j}): {e}")
                correlation = 0
            correlation_matrix[i, j] = correlation_matrix[j, i] =␣
↪correlation

    np.fill_diagonal(correlation_matrix, 1)  # Set diagonal to 1␣
↪(self-correlation)

    # Convert matrix to DataFrame for better visualization
    df_correlation = pd.DataFrame(
        correlation_matrix,
        columns=[f"Series {i+1}" for i in range(matrix_size)],
        index=[f"Series {i+1}" for i in range(matrix_size)]
    )

    # Print formatted correlation matrix
    print(df_correlation.to_string(float_format="{:0.2f}".format))

    return correlation_matrix
```

```
level = 14
# Group the Haar transformations in a list
haar_transforms = [gbpeur_haar_r, sekeur_haar_r, cadeur_haar_r ]

# Calculate the correlation matrices for each level
correlation_matrices = [correlation_matrix_at_level(lvl, haar_transforms)␣
 ↪for lvl in range(1, level)]
```

```
correlation_matrices
```

```
--- Correlation Matrix at Level 1 ---
          Series 1  Series 2  Series 3
Series 1      1.00      1.00     -1.00
Series 2      1.00      1.00     -1.00
Series 3     -1.00     -1.00      1.00

--- Correlation Matrix at Level 2 ---
          Series 1  Series 2  Series 3
Series 1      1.00      0.37      0.29
Series 2      0.37      1.00     -0.43
Series 3      0.29     -0.43      1.00

--- Correlation Matrix at Level 3 ---
          Series 1  Series 2  Series 3
Series 1      1.00      0.33      0.17
Series 2      0.33      1.00      0.60
Series 3      0.17      0.60      1.00

--- Correlation Matrix at Level 4 ---
          Series 1  Series 2  Series 3
Series 1      1.00      0.14      0.45
Series 2      0.14      1.00      0.59
Series 3      0.45      0.59      1.00

--- Correlation Matrix at Level 5 ---
          Series 1  Series 2  Series 3
Series 1      1.00      0.41      0.22
Series 2      0.41      1.00      0.43
Series 3      0.22      0.43      1.00

--- Correlation Matrix at Level 6 ---
          Series 1  Series 2  Series 3
Series 1      1.00      0.48      0.31
Series 2      0.48      1.00      0.45
Series 3      0.31      0.45      1.00

--- Correlation Matrix at Level 7 ---
          Series 1  Series 2  Series 3
Series 1      1.00      0.35      0.25
Series 2      0.35      1.00      0.36
Series 3      0.25      0.36      1.00

--- Correlation Matrix at Level 8 ---
          Series 1  Series 2  Series 3
Series 1      1.00      0.33      0.34
Series 2      0.33      1.00      0.37
Series 3      0.34      0.37      1.00

--- Correlation Matrix at Level 9 ---
          Series 1  Series 2  Series 3
Series 1      1.00      0.35      0.22
Series 2      0.35      1.00      0.24
Series 3      0.22      0.24      1.00

--- Correlation Matrix at Level 10 ---
          Series 1  Series 2  Series 3
Series 1      1.00      0.16      0.31
Series 2      0.16      1.00      0.26
Series 3      0.31      0.26      1.00

--- Correlation Matrix at Level 11 ---
          Series 1  Series 2  Series 3
Series 1      1.00      0.24      0.37
Series 2      0.24      1.00      0.29
Series 3      0.37      0.29      1.00
```

```
--- Correlation Matrix at Level 12 ---
          Series 1  Series 2  Series 3
Series 1      1.00      0.26      0.34
Series 2      0.26      1.00      0.25
Series 3      0.34      0.25      1.00

--- Correlation Matrix at Level 13 ---
          Series 1  Series 2  Series 3
Series 1      1.00      0.16      0.34
Series 2      0.16      1.00      0.23
Series 3      0.34      0.23      1.00
```

This analysis demonstrates the application of the Haar wavelet transform to compute multiresolution correlations between financial time series. By analyzing correlations across different scales, we gain a deeper understanding of the relationships between assets, particularly how short-term and long-term dynamics interact. The use of all available data at the chosen resolution level ensures a comprehensive analysis.

6.2 Epps effect

The decrease in correlation at short time intervals, as described by the Epps Effect, is primarily due to several market dynamics that distort the true relationships between asset prices. At very high frequencies, asynchronous trading plays a significant role, as different assets are not traded simultaneously, leading to desynchronized price movements that weaken observed correlations. Additionally, at these short intervals, price changes are dominated by market microstructure noise, such as bid-ask spreads, transaction costs, and order book fluctuations, which are unrelated to fundamental market factors. This noise obscures the true co-movements between assets. Over longer time intervals, these short-term distortions diminish, prices have more time to adjust to new information, and systemic factors dominate, allowing correlations to stabilize and better reflect the underlying relationships between assets.

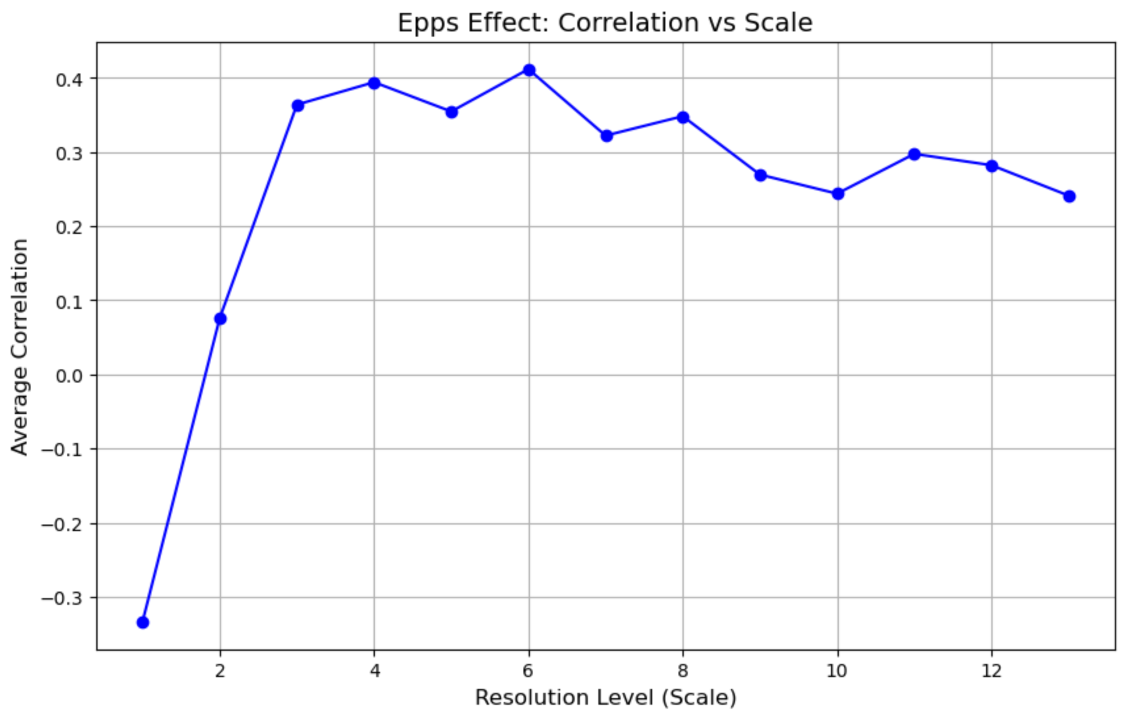

Figure 12.

The graph demonstrates the Epps Effect, where correlations initially increase with the scale, reflecting the diminishing influence of noise and asynchronous trading. As the scale continues to grow, the correlation stabilizes at approximately 0.2, representing the true relationship between the assets using the entirety of our dataset. This stabilization, achieved using the entirety of our dataset, reflects the true underlying relationships between assets as noise and asynchronous trading effects diminish at larger scales.

6.3 Hurst Exponent

To estimate the Hurst exponent H of a fractional Brownian motion (fBm), one can use the absolute moments of increments. The estimator of H is defined as:

$$H = \frac{1}{2\log(2)} \cdot \log\left(\frac{M_2'}{M_2}\right),$$

where:

$$M_2 = \frac{1}{N_T}\sum_{i=1}^{N_T}\left|X\left(\frac{i}{N}\right) - X\left(\frac{i-1}{N}\right)\right|^2,$$

and:

$$M_2' = \frac{2}{N_T}\sum_{i=1}^{N_T/2}\left|X\left(\frac{2i}{N}\right) - X\left(\frac{2(i-1)}{N}\right)\right|^2.$$

```python
def Hurst(serie):
    T = len(serie)
    M2 = np.mean((abs(serie[1:] - serie[:-1])) ** 2)
    M2_prime = np.mean(abs((serie[2:] - serie[:-2])) ** 2)
    H_est = 0.5 * np.log2(M2_prime / M2)
    return H_est
```

- $H < \frac{1}{2}$:
  - Indicates a negatively correlated process, also referred to as anti-persistence.
  - The increments tend to reverse direction. Specifically, if the process increases in one step, it is more likely to decrease in the next step, and vice versa.
  - Such behavior is characteristic of systems with a tendency to oscillate or revert to the mean.
- $H = \frac{1}{2}$:
  - Corresponds to a standard Brownian motion (or random walk).
  - The increments are uncorrelated and independent, showing no memory of past movements.
  - The process is purely stochastic, with no preference for continuation or reversal of trends.
- $H > \frac{1}{2}$:

  - Indicates a positively correlated process, also referred to as persistence.
  - The increments tend to continue in the same direction. For example, if the process increases in one step, it is more likely to continue increasing in the subsequent step.
  - Such behavior is observed in systems with long-term memory or trend-following characteristics.

The series `GBPEUR_Average`, `SEKEUR_Average`, and `CADEUR_Average` represent the averages of the *low* and *high* prices, calculated as follows:

$$\text{Average Price} = \frac{\text{Low Price} + \text{High Price}}{2}.$$

These series allow us to analyze the central dynamics of prices without being influenced solely by extreme values.

```
hurst_gbpeur = Hurst(GBPEUR_Average)
hurst_sekeur = Hurst(SEKEUR_Average)
hurst_cadeur = Hurst(CADEUR_Average)
```

- $H_{\textbf{GBPEUR}} = 0.6796$**:**
  - The Hurst exponent is greater than 0.5, indicating a persistent behavior.
  - Variations in the average GBP/EUR prices tend to continue in the same direction (increase or decrease) over the observed period.
- $H_{\textbf{SEKEUR}} = 0.6673$**:**
  - This value is also greater than 0.5, indicating a similar persistent behavior as observed for GBP/EUR.
  - The average SEK/EUR prices show significant trends with a higher likelihood of past movements (increase or decrease) influencing future movements.
- $H_{\textbf{CADEUR}} = 0.6681$**:**
  - This value, like the previous ones, is greater than 0.5, indicating persistent behavior.
  - Variations in the average CAD/EUR prices also tend to maintain their previous direction, reflecting a dynamic with some degree of memory.

All calculated H values are greater than 0.5, suggesting that the three average price series (GBP/EUR, SEK/EUR, and CAD/EUR) exhibit positive correlation or persistence in their movements. In practical terms, this means that observed price trends are more likely to continue than to reverse.

These results show that the average prices calculated from the low and high prices for these three currency pairs display market behavior that is not purely white noise or a random walk

($H = 0.5$). Such persistent behavior can be exploited to develop trading strategies based on the continuity of trends.

6.4 Annualised Volatility with daily volatility and Hurst Exponent

1. Data Description

The dataset consists of price data recorded every 15 minutes. To compute the daily returns and subsequently annualize the volatility, we extracted the daily closing prices from the dataset.

```
df_sekeur = pd.DataFrame(SEKEUR)

# Convert the 'Date' column to datetime
df_sekeur['Date'] = pd.to_datetime(df_sekeur['Date'])

# Extract only the date (without the time)
df_sekeur['Day'] = df_sekeur['Date'].dt.date
# Find the last price for each day
last_prices_sekeur = df_sekeur.groupby('Day')['Mean_Price'].last().
 ↪reset_index()
```

2. Methodology

Step 1: Calculation of Daily Returns

The daily returns were computed using the formula:

$$\text{Returns} = \frac{P_t - P_{t-1}}{P_{t-1}},$$

where P_t is the closing price at day t and P_{t-1} is the closing price of the previous day.

Step 2: Daily Volatility

The daily volatility was calculated as the standard deviation of the daily returns:

$$\text{Daily Volatility} = \sigma_{\text{returns}} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(r_i - \bar{r})^2},$$

where r_i are the daily returns, $\bar{r}$ is the mean of the returns, and N is the total number of observations.

```
def calculate_daily_volatility(series):
    # Calculate daily returns
    returns = (series[1:] - series[:-1]) / series[:-1]
    # Calculate daily volatility
    return np.std(returns)
```

Step 3: Annualized Volatility Using the Hurst Exponent

The annualized volatility was computed using the daily volatility and the Hurst exponent:

$$\text{Annualized Volatility} = \text{Daily Volatility} \cdot (252^H),$$

where H is the Hurst exponent, and 252 represents the average number of trading days in a year.

```
def annualized_volatility(daily_vol, hurst_exponent):
    return daily_vol * (252 ** hurst_exponent)
```

3. Results

The results of the calculations are as follows:

```
gbpeur_daily_vol = calculate_daily_volatility(last_prices_gbpeur_average)
sekeur_daily_vol = calculate_daily_volatility(last_prices_sekeur_average)
cadeur_daily_vol = calculate_daily_volatility(last_prices_cadeur_average)
```

```
gbpeur_annual_vol = annualized_volatility(gbpeur_daily_vol, hurst_gbpeur)
sekeur_annual_vol = annualized_volatility(sekeur_daily_vol, hurst_sekeur)
cadeur_annual_vol = annualized_volatility(cadeur_daily_vol, hurst_cadeur)
```

- **GBP/EUR:**
  - Daily Volatility: $\sigma_{\text{GBP/EUR}} = 0.0006$
  - Annualized Volatility: $\text{Annualized Volatility}_{\text{GBP/EUR}} = 0.0267$
- **SEK/EUR:**
  - Daily Volatility: $\sigma_{\text{SEK/EUR}} = 0.0003$
  - Annualized Volatility: $\text{Annualized Volatility}_{\text{SEK/EUR}} = 0.0131$
- **CAD/EUR:**
  - Daily Volatility: $\sigma_{\text{CAD/EUR}} = 0.0005$
  - Annualized Volatility: $\text{Annualized Volatility}_{\text{CAD/EUR}} = 0.0204$

The annualized volatilities reflect the yearly variability of the exchange rates, scaled using the Hurst exponent to account for market memory effects. For GBP/EUR, the annualized volatility is 2.67%, indicating moderate variability, while SEK/EUR shows lower variability at 1.31%. CAD/EUR exhibits slightly higher annualized volatility at 2.04%, highlighting differing levels of risk across currencies.

7 Conclusion

The results of our analyses not only reinforce the theoretical concepts studied throughout the course but also provide actionable insights into risk mitigation strategies. By addressing real-world scenarios with a rigorous and methodical approach, this project bridges the gap between academic learning and practical application in the field of market risk management.

As we navigated through each question, we highlighted the importance of various risk measures, such as Value at Risk (VaR), Monte Carlo VaR, and Extreme Value Theory (EVT), in understanding and mitigating market risk. Furthermore, the implementation of advanced methodologies, including wavelet analysis and the Almgren and Chriss liquidation model, showcased the depth and versatility required for comprehensive risk management.

This project is a synthesis of knowledge gained during the course and its application to complex financial challenges. It underscores the critical role of statistical analysis, computational tools, and financial modeling in identifying and managing risks in increasingly volatile markets.

We would like to extend our sincere gratitude to Mr. Matthieu Garcin and Mrs. Julie Gamain for their invaluable guidance throughout the course and tutorial classes.
Their expertise and dedication have significantly enriched our understanding of market risk and its practical implications. Their support has been instrumental in the successful completion of this project.

Finally, the insights gained through this work emphasize the necessity of robust risk management practices in today's financial markets. The analytical tools and frameworks explored here will undoubtedly serve as a foundation for addressing future challenges in the field of finance.

8 Appendix

8.1 Clean Dataset TD1 for 2.1

```
import warnings

# To ignore warnings
warnings.filterwarnings("ignore")

# To display all warnings again (if needed later)
# warnings.filterwarnings("default")
```

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm
```

```
# Load the data
file_path = 'Natixis_MG.xlsx'
xls = pd.ExcelFile(file_path)

# Load the content of the first sheet
data = pd.read_excel(xls)
data
```

```
           Date  Price
0    2015-01-02  5.621
1    2015-01-05  5.424
2    2015-01-06  5.329
3    2015-01-07  5.224
4    2015-01-08  5.453
...         ...    ...
1018 2018-12-21  4.045
1019 2018-12-24  4.010
1020 2018-12-27  3.938
1021 2018-12-28  4.088
1022 2018-12-31  4.119

[1023 rows x 2 columns]
```

```
# Calculate daily returns: (P_t - P_{t-1}) / P_{t-1}
data_filtered['Return'] = data_filtered['Price'].pct_change()
```

```
# Remove the first row since it contains NaN for the return
data_filtered.dropna(inplace=True)

# Display the results: Price, Return, and VaR
print("Filtered Data with Returns:")
data_filtered
```

```
Filtered Data with Returns:

          Date  Price    Return
1   2015-01-05  5.424 -0.035047
2   2015-01-06  5.329 -0.017515
3   2015-01-07  5.224 -0.019704
4   2015-01-08  5.453  0.043836
5   2015-01-09  5.340 -0.020723
```

8.2 Figure 1.

```
# Plot the price and returns over time
fig, ax1 = plt.subplots(figsize=(10, 6))

# Plot the Price
ax1.set_xlabel('Date')
ax1.set_ylabel('Price', color='tab:blue')
ax1.plot(data_filtered['Date'], data_filtered['Price'], color='tab:blue',␣
 ↪label='Price')
ax1.tick_params(axis='y', labelcolor='tab:blue')

# Create a secondary axis to plot Returns
ax2 = ax1.twinx()
ax2.set_ylabel('Return', color='tab:green')
ax2.plot(data_filtered['Date'], data_filtered['Return'], color='tab:
 ↪green', label='Return')
# Add VaR line
ax2.axhline(y=VaR, color='red', linestyle='--', label=f'VaR␣
 ↪({confidence_level*100:.0f}%)')
ax2.tick_params(axis='y', labelcolor='tab:green')

# Add a legend for VaR
ax2.legend(loc='upper left')
```

```
# Title and plot
plt.title('Price, Returns, and VaR Over Time (Jan 2015 - Dec 2016)')
fig.tight_layout()
plt.show()
```

8.3 Figure 2.

```
plt.figure(figsize=(10, 6))
plt.hist(data_filtered['Return'], bins=50, alpha=0.75, color='blue',␣
 ↪edgecolor='black')
plt.axvline(x=VaR, color="red", linestyle="--", label=f"VaR Historical␣
 ↪({VaR:.4f})")
plt.title('Historical Returns of Natixis between january 2015 and december␣
 ↪2016 ')
plt.xlabel('Returns')
plt.ylabel('Frequency')
plt.legend()
plt.show()
```

8.4 Figure 3.

```
plt.figure(figsize=(10, 6))
plt.plot(x_range, kde_values_logistic, label=f"KDE (Silverman's Rule, h={h:
 ↪.4f})", color='blue')
plt.plot(x_range, kde_values_small, label=f"KDE (h=0.001)", color='orange')

plt.title("Kernel Density Estimation with Different Bandwidths")
plt.xlabel("Returns")
plt.ylabel("Density")
plt.legend()
plt.grid(True)
plt.show()
```

8.5 Figure 4.

```
import matplotlib.pyplot as plt

plt.axvline(x=non_parametric_var_logistic, color="red", linestyle="--",␣
 ↪label=f"VaR Logistic ({non_parametric_var_logistic:.4f})")
```

```
plt.axvline(x=VaR, color="green", linestyle="--", label=f"VaR Historical␣
 ↪({VaR:.4f})")
plt.plot(x_range, kde_values_logistic, label="KDE (Logistic Kernel)")

plt.title("Kernel Density Estimation and Value at Risk")
plt.xlabel("Returns")
plt.ylabel("Density")
plt.legend()
plt.grid(True)
plt.show()
```

8.6 Figure 5.

```
plt.figure(figsize=(10, 6))
plt.hist(data_filtered_period['Return'], bins=50, alpha=0.75,␣
 ↪color='blue', edgecolor='black')
plt.axvline(x=VaR_threshold, color="red", linestyle="--", label=f"VaR␣
 ↪threshold ({VaR_threshold:.4f})")
plt.title('Historical Returns of Natixis between january 2017 and december␣
 ↪2018')
plt.xlabel('Returns')
plt.ylabel('Frequency')
plt.legend()
plt.show()
```

8.7 Clean Dataset TD1 for 3.1

```
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import TimeSeriesSplit
```

```
# Filter the data between January 2015 and December 2016
data_filtered_carlo = data[(data['Date'] >= '2015-01-01') & (data['Date']␣
 ↪<= '2018-12-31')]
data_filtered_carlo
```

```
           Date  Price
0    2015-01-02  5.621
1    2015-01-05  5.424
2    2015-01-06  5.329
3    2015-01-07  5.224
4    2015-01-08  5.453
```

```
...         ...    ...
1018 2018-12-21  4.045
1019 2018-12-24  4.010
1020 2018-12-27  3.938
1021 2018-12-28  4.088
1022 2018-12-31  4.119

[1023 rows x 2 columns]
```

8.8 Figure 6.

```
# Plot option prices
plt.plot(Last_Price)
plt.title("Simulated Natixis share prices")
plt.xlabel("Index")
plt.ylabel("Simulated Natixis share prices on 31/12/2018")
plt.show()
```

8.9 Figure 7.

```
# Draw the histogram of call option prices
plt.figure(figsize=(10, 6))
plt.hist(call_prices, bins=30, color='skyblue', edgecolor='black', alpha=0.
 ↪7, label='Distribution des prix des calls')

# Add a vertical line for VaR
plt.axvline(x=VaR_percentile, color='red', linestyle='--', linewidth=2,␣
 ↪label=f'VaR at {confidence_level*100:.0f}%')

# Add details to the graph
plt.title("Call option price distribution with Value at Risk (VaR)",␣
 ↪fontsize=14)
plt.xlabel("Call option prices", fontsize=12)
plt.ylabel("Frequency", fontsize=12)
plt.legend(fontsize=10)
plt.grid(True, linestyle='--', alpha=0.6)

# Display the graph
plt.show()
```

8.10 Clean Dataset TD1 for 4.1

```
data_filtered_GEV = data
data_filtered_GEV['return']=(data_filtered_GEV['Price']-data_filtered_GEV['Price'].
↪shift(1))/data_filtered_GEV['Price'].shift(1)
data_filtered_GEV['return_positive'] = data_filtered_GEV['return'].
↪where(data_filtered_GEV['return'] > 0, np.nan)
data_filtered_GEV['return_negative'] = data_filtered_GEV['return'].
↪where(data_filtered_GEV['return'] < 0, np.nan)
data_filtered_GEV
```

```
           Date  Price    return  return_positive  return_negative
0    2015-01-02  5.621       NaN              NaN              NaN
1    2015-01-05  5.424 -0.035047              NaN        -0.035047
2    2015-01-06  5.329 -0.017515              NaN        -0.017515
3    2015-01-07  5.224 -0.019704              NaN        -0.019704
4    2015-01-08  5.453  0.043836         0.043836              NaN
...         ...    ...       ...              ...              ...
1018 2018-12-21  4.045 -0.001481              NaN        -0.001481
1019 2018-12-24  4.010 -0.008653              NaN        -0.008653
1020 2018-12-27  3.938 -0.017955              NaN        -0.017955
1021 2018-12-28  4.088  0.038090         0.038090              NaN
1022 2018-12-31  4.119  0.007583         0.007583              NaN

[1023 rows x 5 columns]
```

8.11 Clean Dataset TD4 for 5.1

```
#Load the data
file_path = 'Greenchriss.xlsx'
xls = pd.ExcelFile(file_path)

# Load the content of the first sheet
Grenchriss = pd.read_excel(xls)
Grenchriss
```

```
      transaction date (1=1day=24 hours)  bid-ask spread  \
0                               0.000202          0.1100
1                               0.001070          0.1030
2                               0.001496          0.1015
3                               0.003336          0.0920
4                               0.003952          0.1106
```

```
...                                  ...           ...
996                             0.981441        0.0834
997                             0.981875        0.1010
998                             0.986784        0.1007
999                             0.991232        0.1153
1000                            0.992002        0.1045

      volume of the transaction (if known)  Sign of the transaction  \
0                                      8.0                       -1
1                                      NaN                        1
2                                      NaN                       -1
3                                      NaN                        1
4                                      NaN                        1
...                                    ...                      ...
996                                   79.0                        1
997                                    NaN                       -1
998                                    NaN                       -1
999                                    3.0                       -1
1000                                   NaN                        1

      Price (before transaction)
0                        100.000
1                         99.984
2                        100.029
3                         99.979
4                        100.060
...                          ...
996                      101.070
997                      101.120
998                      100.998
999                      100.958
1000                     100.948

[1001 rows x 5 columns]
```

```
volume = []
price = []
sign_Transaction = []
returns = []
```

```
volume = Grenchriss['volume of the transaction (if known)'].dropna()
volume
```

```
0        8.0
6       32.0
16       8.0
28     141.0
51     121.0
        ...
988     14.0
989    150.0
990     17.0
996     79.0
999      3.0
Name: volume of the transaction (if known), Length: 137, dtype: float64
```

```
price = Grenchriss['Price (before transaction)'].dropna()
price
```

```
0       100.000
1        99.984
2       100.029
3        99.979
4       100.060
          ...
996     101.070
997     101.120
998     100.998
999     100.958
1000    100.948
Name: Price (before transaction), Length: 1001, dtype: float64
```

```
sign_Transaction = Grenchriss['Sign of the transaction'].dropna()
sign_Transaction
```

```
0      -1
1       1
2      -1
3       1
4       1
       ..
996     1
997    -1
998    -1
999    -1
```

```
1000    1
Name: Sign of the transaction, Length: 1001, dtype: int64
```

8.12 Figure 8.

```python
import matplotlib.pyplot as plt

# Predictions based on regression
y_pred = slope * Produit_signe + intercept

# Plot the actual data and the regression with inverted axes
plt.scatter(Price_diff, Produit_signe, color='blue', label='Real data')
plt.plot(y_pred, Produit_signe, color='red', label='Linear regression')
plt.xlabel("Price difference (Permanent Impact)")
plt.ylabel("Signed product")
plt.title("Linear regression")
plt.legend()
plt.show()
```

8.13 Figure 9.

```python
# Plot real data and regression with inverted axes
plt.scatter(Produit_signe_carre, Price_diff_trans, color='blue',␣
 ↪label='Real data')
plt.plot(Produit_signe_carre, y_pred_d, color='red', label='Linear␣
 ↪regression')
plt.xlabel("Signed product squared")
plt.ylabel("Price difference (Transitory Impact)")
plt.title("Linear Regression")
plt.legend()
plt.show()
```

8.14 Figure 10.

```python
# Plot the actual data
plt.scatter(Produit_signe_carre, Price_diff_trans, color='blue',␣
 ↪label='Real Data')
# Plot the linear regression curve with sorted data
plt.plot(Produit_signe_carre_sorted, y_pred_sorted, color='red',␣
 ↪linestyle='--', label='Linear regression')
```

```python
# Add labels and a title
plt.xlabel('Signed product squared')
plt.ylabel('Price difference (Transitory Impact)')
plt.title('Multivariate regression: real data vs. predictions')
plt.legend()
plt.show()
```

8.15 Figure 12.

```python
import numpy as np
import matplotlib.pyplot as plt

# Function to calculate the average correlation in a matrix
def calculate_avg_correlation(correlation_matrix):
    n = correlation_matrix.shape[0]
    upper_triangle = correlation_matrix[np.triu_indices(n, k=1)]
    return np.mean(upper_triangle)

# Calculate the average correlations at each level
avg_correlations = [calculate_avg_correlation(matrix) for matrix in␣
 ↪correlation_matrices]

# Plot the Epps Effect
plt.figure(figsize=(10, 6))
plt.plot(range(1, level), avg_correlations, marker='o', linestyle='-',␣
 ↪color='b')
plt.xlabel('Resolution Level (Scale)', fontsize=12)
plt.ylabel('Average Correlation', fontsize=12)
plt.title('Epps Effect: Correlation vs Scale', fontsize=14)
plt.grid(True)
plt.show()
```

8.16 Cleaning Dataset TD5 for 6.1

```python
import numpy as np
import pandas as pd

# Load the data
file_path = 'Wavelete.xlsx'
xls = pd.ExcelFile(file_path)
```

```
# Load the content of the first sheet
Wavelets = pd.read_excel(xls)
```

```
import pandas as pd

# Clean up column names to avoid errors caused by extra spaces
Wavelets.columns = Wavelets.columns.str.strip()

# Initialise empty DataFrames
GBPEUR = pd.DataFrame()
SEKEUR = pd.DataFrame()
CADEUR = pd.DataFrame()

# Delete rows with missing values
GBPEUR['Date'] = Wavelets['Date']
GBPEUR['HIGH_GBPEUR'] = Wavelets['HIGH_GBPEUR']
GBPEUR['LOW_GBPEUR'] = Wavelets['LOW_GBPEUR']
GBPEUR.dropna(subset=['Date', 'HIGH_GBPEUR', 'LOW_GBPEUR'], inplace=True)

SEKEUR['Date'] = Wavelets['Date']
SEKEUR['HIGH_SEKEUR'] = Wavelets['HIGH_SEKEUR']
SEKEUR['LOW_SEKEUR'] = Wavelets['LOW_SEKEUR']
SEKEUR.dropna(subset=['Date', 'HIGH_SEKEUR', 'LOW_SEKEUR'], inplace=True)

CADEUR['Date'] = Wavelets['Date']
CADEUR['HIGH_CADEUR'] = Wavelets['HIGH_CADEUR']
CADEUR['LOW_CADEUR'] = Wavelets['LOW_CADEUR']
CADEUR.dropna(subset=['Date', 'HIGH_CADEUR', 'LOW_CADEUR'], inplace=True)
```

8.17 Clean Dataset TD5 for 6.4

```
df_gbpeur = pd.DataFrame(GBPEUR)

# Convert the 'Date' column to datetime
df_gbpeur['Date'] = pd.to_datetime(df_gbpeur['Date'])

# Extract only the date (without the time)
df_gbpeur['Day'] = df_gbpeur['Date'].dt.date

# Find the last price for each day
last_prices_gbpeur = df_gbpeur.groupby('Day')['Mean_Price'].last().
 ↪reset_index()
```

```
# Display the result
print(last_prices_gbpeur)
```

```
            Day  Mean_Price
0    2016-03-07     1.29480
1    2016-03-08     1.29115
2    2016-03-09     1.29255
3    2016-03-10     1.27785
4    2016-03-11     1.29025
..          ...         ...
154  2016-09-02     1.19165
155  2016-09-04     1.19230
156  2016-09-05     1.19380
157  2016-09-06     1.19345
158  2016-09-07     1.18695

[159 rows x 2 columns]
```

```
last_prices_gbpeur_average = df_gbpeur['Mean_Price'].values
last_prices_gbpeur_average
```

```
array([1.2935 , 1.29325, 1.29215, ..., 1.1877 , 1.187  , 1.18695])
```

```
df_sekeur = pd.DataFrame(SEKEUR)

# Convert the 'Date' column to datetime
df_sekeur['Date'] = pd.to_datetime(df_sekeur['Date'])

# Extract only the date (without the time)
df_sekeur['Day'] = df_sekeur['Date'].dt.date
# Find the last price for each day
last_prices_sekeur = df_sekeur.groupby('Day')['Mean_Price'].last().
 ↪reset_index()
```

```
last_prices_sekeur_average = df_sekeur['Mean_Price'].values
```

```
df_cadeur = pd.DataFrame(CADEUR)

# Convert the 'Date' column to datetime
df_cadeur['Date'] = pd.to_datetime(df_cadeur['Date'])
```

```
# Extract only the date (without the time)
df_cadeur['Day'] = df_cadeur['Date'].dt.date

# Find the last price for each day
last_prices_cadeur = df_cadeur.groupby('Day')['Mean_Price'].last().
 ↪reset_index()
```

```
last_prices_cadeur_average = df_cadeur['Mean_Price'].values
```

References

1. Garcin, Matthieu. *Market Risk*. ESILV Lecture Notes, 2024.