MARKET RISK PROJECT REPORT

MARKET RISK

FINANCIAL ENGINEERING

2023-2024

Authors:	*Id :*
ROUDAUT Antoine	711964
THOMASSIN Pablo	719295

Course Coordinator:

Matthieu GARCIN

28th December 2023

Contents

1 ABSTRACT

This project presents our work for the Market Risk course of 2023 in financial engineering at ESILV. We present over this document, evaluation of risk measure using different methods, liquidation methods, and different methods to statistically compute parameters. We will develops our result using python and four modules : Numpy, Pandas, Scikit-Learn, Seaborn. Knowing the principal objectives is to develop the code without fully relying to the powerful methods of this four modules.

2 INTRODUCTION

Market risk refers to the potential for financial loss arising from adverse movements in market prices. It encompasses the uncertainty associated with fluctuations in interest rates, exchange rates, commodity prices, and equity values. Market risk is omnipresent in investment activities and portfolio management, influencing the profitability and solvency of financial institutions and investors alike. Efficiently assessing, mitigating, and managing market risk is essential for safeguarding financial stability and optimizing investment returns.

The importance of market risk evolves around : Preservation of capital, Optimal portfolio performance, Regulatory compliance (That aren't directly discussed in the subject).

To read this document, we highly recommend you to read the whole code and commentary, you will be able sometimes to find graphics in the appendix part, this will be mentioned directly in the document. Moreover in the code if some methods from numpy or pandas are used they will be explained and justified. Same for searborn, and small usage of scikit-learn.

3 QUESTIONS ANSWERS

3.1 PART I : HISTORICAL VALUE AT RISK

3.1.1 HISTORICAL VALUE AT RISK BASED ON KERNEL DENSITY

In this section we will compute the value at risk using a kernel density and in the same code we will run the threshold test this is due to the fact that when we wrote the code we used to work on a same DataFrame and sort the data depending the Date column first then the Returns column as such we need to do the thing in the right order. Some use of more performing function or notebook can resolve this problem.

First let's introduce big weigth kernel : $K(u) = \frac{15}{16}(1-u^2)^2 1_{|u|\leq 1}$. Then let's dive into the code :

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

#Preprocessing
data= pd.read_csv(r"Natixis stock (dataset TD12).txt", delimiter='\t',
    header=None, parse_dates=[0]) #We will use mainly dataframe to analyze
    data in our whole project
data.columns = ['Date', 'Value'] # We will tend to always rename columns
data['Date'] = pd.to_datetime(data['Date'], format="%d/%m/%Y") #
    Preprocessing of date-time
data['Value'] = data['Value'].str.replace(',', '.').astype(float) #Re type
    for float number as we read a txt, the  coma - point transition isn't
    immediate
data = data.sort_values('Date') # For now let's sort by date to calculate
    returns coherent we use build in sort algorithm.

#Compute of Returns in data frame and creating new column, the shift()
    function just go to the next row in the same column, we didn't use
    pct_change() a builded in function of pandas
def Returns(data):
    data['Returns'] = (data['Value'] - data['Value'].shift(1)) / data['
    Value'].shift(1)
    return data

```

```python
#Let's compute returns based on sorted date (which is the good sort)
data = Returns(data)
print(data)
```

```
            Date  Value   Returns
0     2015-01-02  5.621       NaN
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
...          ...    ...       ...
1018  2018-12-21  4.045 -0.001481
1019  2018-12-24  4.010 -0.008653
1020  2018-12-27  3.938 -0.017955
1021  2018-12-28  4.088  0.038090
1022  2018-12-31  4.119  0.007583
```

As we can see on this first data set we need to clean the first Nan value.

```python
#As we can see the first value of returns is NaN which is logical due to
    the definition of returns so we decide to erase the row from our dataset
data = data.dropna()
print(data)
```

	Date	Value	Returns
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
...	...	...	...
1018	2018-12-21	4.045	-0.001481
1019	2018-12-24	4.010	-0.008653
1020	2018-12-27	3.938	-0.017955
1021	2018-12-28	4.088	0.038090
1022	2018-12-31	4.119	0.007583

```python
# Let's define some value alpha is the probability level targeted and value
    portfolio is the actual value of the portfolio
alpha = 0.05
value_portfolio = 100000

#Now, on our sorted dataset, we will select only the date we wanted. the
   exercise for questions 1 and 2.
data_date_sort_2015 = data[(data["Date"] >= '2015-01-01') & (data["Date"] <
    '2016-12-31')]
data_date_sort_2017 = data[(data["Date"] >= '2017-01-01') & (data["Date"] <
    '2018-12-31')]
#Once the selection is done now we can sort returns !
data_date_sort_2015 = data_date_sort_2015.sort_values("Returns") # using
   build in sort function of pandas
data_date_sort_2017 = data_date_sort_2017.sort_values("Returns")
print(data_date_sort_2015) # quick check up
```

	Date	Value	Returns
378	2016-06-24	3.439	-0.171325
345	2016-05-10	4.083	-0.069083
358	2016-05-27	4.460	-0.061448
405	2016-08-02	3.422	-0.060664
277	2016-02-02	4.208	-0.058613
..	...	...	...
283	2016-02-10	4.101	0.062435
305	2016-03-11	5.083	0.063389
164	2015-08-25	5.755	0.064755
403	2016-07-29	3.685	0.075912
326	2016-04-13	4.785	0.078431

Now we will define correctly our Kernel density function and Kernel VaR for the time period of 2015 - 2016.

```
#Let's define kernel with regards to exercise and with respect to the indicatrice
def kernel(u):
    if abs(u)<=1 :
        return (15/16)*((1-(u)**2)**2)
    else:
        return 0

#Let's define our kernel density
def kernel_density(data,x):  # data is the whole column of returns and x is a particular return
    h= ((4 * data['Returns'].std())/(3 * len(data['Returns'])))**0.2 # h with respect to the thumb formula of silverman

    n=len(data['Returns'])
    somme_kernel_density=0
```

```

    for datas in data['Returns']:
        somme_kernel_density += kernel((x-datas)/h) # we sum based on our kernel function given precedently

    return (somme_kernel_density/(n*h)) # density function obtained mathematically


def VaR_Kernel(data, alpha):
    probabilities = [] # we want to create an array of all our probability

    #Let's evaluate every probability for each returns
    for datas in data['Returns']:
        probability = kernel_density(data, datas)
        probabilities.append(probability)

    total = sum(probabilities) # we sum all the probabilities cause we have to secure a condtion of the sum (probabilities ) == 1

    # Normalization in order to have the condition spoken about in the upper line
    if total != 0:
        probabilities = [prob / total for prob in probabilities]

    #Let's compute the quantile of distribution
    sorted_returns = sorted(data['Returns'])
    cumulative_prob = 0

    for i, prob in enumerate(probabilities):#Here we use enumerate to make each probabilities an indenpendent object  as such there is a new indexation which we can track with i
        cumulative_prob += prob
        if cumulative_prob >= alpha:
            resultat = sorted_returns[i]
            break

    return resultat

```

```
#Now let's compute our VaR with alpha = 0.05 on our data date selected,
    sorted on returns
Value_at_Risk = VaR_Kernel(data_date_sort_2015,0.05)
print(f"Value at Risk of Kernel : {Value_at_Risk} \n") # let's Print our
    result
```

```
Value at Risk of Kernel : -0.03478608556577359
```

First of all, we can see that the Value At Risk is negative, which is consistent since we are interested in losses and therefore in the left quantile of the distribution. What's more, this result is consistent when we compare it to the VaR obtained with other methods explained during the course.

3.1.2 VALUE AT RISK THRESHOLD

Let's now evaluate the percentage of data above a VaR threshold calculated with the kernel density for the 2017-2018 period. Let's first recall these two lines that first correctly select the time periods on the value sorted by date, and then calculate the returns. This gives us a sorted table of returns over the desired period.

```
data_date_sort_2015 = data[(data["Date"] >= '2015-01-01') & (data["Date"] <
     '2016-12-31')]
data_date_sort_2017 = data[(data["Date"] >= '2017-01-01') & (data["Date"] <
     '2018-12-31')]
#Once the selection is done now we can sort returns !
data_date_sort_2015 = data_date_sort_2015.sort_values("Returns") # using
    build in sort function of pandas
data_date_sort_2017 = data_date_sort_2017.sort_values("Returns") #same as
    previously
```

Then let's define properly the proportion of value that goes over the threshold for a VaR based on Kernel distribution computed in the 2017 range. As the subject was not clear if we had to compare 2017 data to a VaR of 2015 or 2017 data to a 2017 VaR : we have chosen to compare 2017 data to a 2015 VaR. As such we want to know if the distribution of the actual year can be explained by the previous year risk measure.

```
def Over_VaR(data, val_ref):
    somme=0 # initialization of the sum
    for i in data['Returns']: # Let's count all value of returns under the
    threshold
```

```
        if (i<val_ref):
            somme+=1

    proportion= (somme/len(data['Returns']))*100 # Let's compute the percentage

    return proportion

Over_Threshold_Returns = Over_VaR(data_date_sort_2017, Value_at_Risk) #We compare 2017 - 2018 data with 2015 VaR
print(f"Percentage over threshold for a VaR {Over_Threshold_Returns}")
```

```
Percentage over threshold for a VaR 1.768172888015717
```

This computed value indicate us that 1.76 percent of the values are over the threshold.Let's now decide if we validate the choice of this non-parametric VaR. We know that the kin of non parametric VaR are more performing on data with heavy tail distribution and when the underlying distribution of the returns isn't known. Which in our case seems to be the case due to the fact that for an $\alpha = 0.05$ which is 5 percent our over the threshold is 1.76 percent clearly less than what we wanted to tolerate to the expected level. As such we can deduce that we can validate this choice of non parametric VaR.

3.2 PART II : EXPECTED SHORTFALL

Using the code from the previous section, we will now calculate the Expected Shortfall for the 2017 periods using a VaR threshold from the 2015 periods(As the subject isn't quite precise on which VaR we have to consider we would like to advise that if we compare a 2015 periods with a 2015 VaR the conclusion are the same, even if we decide to take 2017), which is another measure of risk, this one being a cumulative function of the Value at Risk (relative to the price). Furthermore, Expected Shortfall is a more robust measure of risk than Value at Risk because it is consistent (guaranteeing sub-addivity) and more informative about the extent of losses. The code is strictly the same as in PART I.

```
def ES_nonparam_var(data, Value_at_Risk):

    over_threshold = data["Returns"][data["Returns"] < Value_at_Risk]
    ESnonparam=np.mean(over_threshold)
```

```
    return ESnonparam


ES_varnonparam = ES_nonparam_var(data_date_sort_2017,Value_at_Risk)
print(f"Expected Shortfall vaR non-param trique: {ES_varnonparam} \n")
```

```
Expected Shortfall vaR non-paramétrique: -0.04928369644923697
```

Let's compare our two results: The Expected Shortfall is slightly higher than the Value at Risk and negative, which suggests that the VaR overshoot values are not very far from them because our ES is close to the VaR. This additional measure tells us that larger-than-expected losses can occur. Thus, if our portfolio experiences extreme adverse events (i.e. beyond the VaR threshold), the average loss tends to be greater than indicated by the VaR. The expected shortfall gives more information about what happens if returns exceed the threshold; in our case, extreme adverse events have a more pronounced impact than the symmetrically expected loss.

3.3 PART III : VALUE AT RISK AND GENERALIZED EXTREME VALUE

For the following exercise we will go back to the start doing our code from scratch in order to be sure that the sorting that we did earlier have no impact on our parameters computing.

3.3.1 ESTIMATION OF GEV PARAMETERS WITH PICKHANDS ESTIMATORS

Let's begin by computing pickands estimator, for this we need to calculate our returns and sort them before calculating the GEV parameters for the left tail (losses) and right tail (profits) :

```
import numpy as np
import pandas as pd
#Preprocessing
data= pd.read_csv(r"Natixis stock (dataset TD12).txt", delimiter='\t',
    header=None, parse_dates=[0]) #We will use mainly dataframe to analyze
    data in our whole project
data.columns = ['Date', 'Value'] # We will tend to always rename columns
data['Date'] = pd.to_datetime(data['Date'], format="%d/%m/%Y") #
    Preprocessing of date-time
```

```python
data['Value'] = data['Value'].str.replace(',', '.').astype(float) #Re type for float number as we read a txt, the  coma - point transition isn't immediate
data = data.sort_values('Date') # For now let's sort by date to calculate returns coherent we use build in sort algorithm.
alpha = 0.05 # Definition of a confidence level for the begining of the program

#Compute of Returns in data frame and creating new column, the shift() function just go to the next row in the same column, we didn't use pct_change() a builded in function of pandas
def Returns(data):
    data['Returns'] = (data['Value'] - data['Value'].shift(1)) / data['Value'].shift(1)
    return data
#Let's compute returns based on sorted date (which is the good sort)
data = Returns(data)
print(data)
```

```
            Date  Value   Returns
0     2015-01-02  5.621       NaN
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
...          ...    ...       ...
1018  2018-12-21  4.045 -0.001481
1019  2018-12-24  4.010 -0.008653
1020  2018-12-27  3.938 -0.017955
1021  2018-12-28  4.088  0.038090
1022  2018-12-31  4.119  0.007583
```

```python
#Now let's drop our NaN Value
data = data.dropna()
print(data)
```

```
            Date  Value   Returns
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
...          ...    ...       ...
1018  2018-12-21  4.045 -0.001481
1019  2018-12-24  4.010 -0.008653
1020  2018-12-27  3.938 -0.017955
1021  2018-12-28  4.088  0.038090
1022  2018-12-31  4.119  0.007583
```

```python
#Let's define of a new column, profits with usage of the apply() methods of
    panda to apply selecting condition
def Profit(data):
    data['Profits'] = data['Returns'].apply(lambda x: x if x > 0 else  0)
    return data
data = Profit(data)
data = data.dropna()
data= data.sort_values("Profits")
#Print data cleaned and sorted for profits
print(data)
```

```
              Date  Value   Returns   Profits
1     2015-01-05  5.424 -0.035047  0.000000
584   2017-04-11  5.626 -0.012463  0.000000
585   2017-04-12  5.544 -0.014575  0.000000
586   2017-04-13  5.410 -0.024170  0.000000
594   2017-04-27  6.358 -0.007183  0.000000
..           ...    ...       ...       ...
305   2016-03-11  5.083  0.063389  0.063389
164   2015-08-25  5.755  0.064755  0.064755
403   2016-07-29  3.685  0.075912  0.075912
326   2016-04-13  4.785  0.078431  0.078431
591   2017-04-24  6.316  0.090281  0.090281
```

```python
#Let's now do the same for Losses
def Loss(data):
    data['Losses'] = data['Returns'].apply(lambda x: x if x < 0 else 0)
    return data
data = Loss(data)
data = data.dropna()
data= data.sort_values("Losses")
#And we can print our result
print(data)
```

```
           Date  Value   Returns   Profits    Losses
378   2016-06-24  3.439 -0.171325  0.000000 -0.171325
345   2016-05-10  4.083 -0.069083  0.000000 -0.069083
1016  2018-12-19  4.171 -0.063328  0.000000 -0.063328
358   2016-05-27  4.460 -0.061448  0.000000 -0.061448
405   2016-08-02  3.422 -0.060664  0.000000 -0.060664
...          ...    ...       ...       ...       ...
343   2016-05-06  4.361  0.006230  0.006230  0.000000
460   2016-10-18  4.392  0.006186  0.006186  0.000000
853   2018-05-04  6.838  0.006180  0.006180  0.000000
990   2018-11-13  5.124  0.006680  0.006680  0.000000
591   2017-04-24  6.316  0.090281  0.090281  0.000000
```

Extreme value theory is concerned with the tails of distributions and the rare events that make them up, which occur very rarely but can have a major impact. It is therefore necessary to supplement our risk measures with an EVT VaR. We will calculate an EVT VaR using Pickands Estimator. : $\epsilon^{P}_{k(n);n} = \frac{1}{\log(2)} \log \frac{X_{n-k(n)+1:n} - X_{n-2k(n)+1:n}}{X_{n-2k(n)+1:n} - X_{n-4k(n)+1:n}}$ GARCIN, 2023-2024

```
#This function compute the quantile of level 1 - alpha
def Quantile(series, p):
    sorted_series = series.sort_values(ascending=True) #let's sort the given series in case

    if 0 <= p <= 1:  # if p >=0 and p <= 1 then we can use the index computing as follow
        index = int(p * len(sorted_series))
        return sorted_series.iloc[index]
    else :
        if p < 0: #then the minimum we can go for the index is 0, in this cas we can do the same for p > 1 but was not needed further in the code
            return sorted_series.iloc[0]
        if p > 1 :
            return sorted_series.iloc[len(sorted_series)]

#Let's define pickands estimator with respect to formula of the course page 191
def Pickands_Estimator(data, alpha):
    n = len(data)

    k = Quantile(data, 1 - alpha)
    k_2n = Quantile(data, 1 - 2 * alpha)
    k_4n = Quantile(data, 1 - 4 * alpha)



    a = np.log(k - k_2n)
    b = 0
    #Let's define a security condition to make sure b is not zero and so we don't divide by zero
    if k_2n != k_4n  :
        b = np.log(k_2n - k_4n)
```

```python
    else:
        b = np.nan

    c = a - b

    return 1 / np.log(2) * c

# Selection profits and losses with value different of zero
profits = data[data['Profits'] != 0]['Profits']
losses = data[data['Losses'] != 0]['Losses']
# Estimate Pickands for profits
pickands_profits = Pickands_Estimator(profits, alpha)
# Estimate Pickands for losses
pickands_losses = Pickands_Estimator(losses, alpha)
#Printing our result
print(f"Pickands estimator for profits:{pickands_profits}")
print(f"Pickands estimator for losses:{pickands_losses}")
```

```
Pickands estimator for profits:0.7705750522956818
Pickands estimator for losses:-1.262301600266066
```

We obtain a Pickands estimator that is negative and greater than -1 for losses and positive and less than 1 for profits. According to the course, this suggests that the behaviour of losses does not conform to extreme value theory. This could suggest that extreme losses are not well modelled by the current distribution assumption or that the loss distribution has a lighter tail, indicating the low presence of extreme values. This view is reinforced by the Expected Shortfall found previously, as its value: the average of losses beyond VaR was not very far from the value of VaR itself. On the other hand, when we examine the Pickands estimator for profits, we observe a positive number, which tells us that the behaviour of the profits distribution is consistent with extreme value theory. In other words, the profits distribution has a heavy tail, indicating the presence of extreme values on a ”recurring” basis.

3.3.2 VALUE AT RISK BASED ON EVT WITH ASSUMPTION OF INDEPENDENT IDENTICAL DISTRIBUTED RANDOM VARIABLE

Now let’s compute the Value at Risk for pickands estimator with the assumpution of independent identical distributed random variable. So with respect to the course formula and the precedent code : $\mathrm{VaR}(p) = \frac{\left(\frac{k}{n(1-p)}\right)^{\epsilon^P} - 1}{1-2^{-\epsilon^P}} \left(X_{n-k+1:n} - X_{n-2k+1:n}\right) + X_{n-k+1:n}$ GARCIN, 2023-2024

```python
#Let's define many confidence level
alpha = np.array([0.01, 0.05, 0.10, 0.15, 0.20, 0.25, 0.30, 0.35, 0.50,
    0.75, 0.95])
#Let's select all the returns different of zero
returns = data[data['Returns'] !=0]['Returns']


#Let's now define the VaR for pickands estimator with respect to the course
     on slide 198
def VaR_Pickands(data,alpha):
    n=len(data)
    k =  Quantile(data, 1 - alpha)
    k_2n = Quantile(data, 1 - 2 * alpha)
    numerator = (1 /n*(1-alpha))**(Pickands_Estimator(data, alpha)) - 1 #
    with respect to the course page 197 we can deduce k = 1 is also true
    denominator= 1 - 2 **(-Pickands_Estimator(data,alpha))
    #If the denominator is 0 which can happen if the pickand estimator is 0
     can happen if the confidence level is too high
    if denominator !=0 :
        return (numerator/denominator)*(k - k_2n) + k
    else :
        return np.nan

# Compute VaR for returns using Pickands estimator for different alpha
    levels
var_returns_pickands = np.array([VaR_Pickands(returns, a) for a in alpha])
print(f"VaR using Pickands estimator for returns:{var_returns_pickands}")
```

```
VaR using Pickands estimator for returns:[-3.84494298e-01 -1.70967668e+00 -1.22921240e-01 -2.82525406
e-01
 -2.26934042e+00 -9.08229529e+08 -5.27751706e+08 -1.92512163e+08
            nan             nan             nan]
```

At first, we can observe that there is some nan value with respect to the alpha value being to high, it implies some zero in the Pickands estimator resulting in a value at risk not approximated. What we can deduce is that if alpha is too big(so if we aren't considering extreme values) the model implemented isn't working. In fact with respect to the course we can see that there is à totally new formula for this case. But here we would like to analyse extreme value. And so if alpha is not representing extreme value we would like to give a nan answer.

On the other hand we can see that the bigger alpha is (with respect to bigger meaning closer to one). The more the value at risk tend to to minus infinite, which is giving us as much as we don't consider extreme events the tolerated returns becomes greater. Thus rejoin the idea discussed earlier where at a certain level alpha we don't represent at all the extreme value and as such the model is less and less (as alpha is close to one) describing a good value at risk, and so a threshold to analyse the part of extreme event's occurring.

3.4 PART IV : ALMGREN AND CHRISS PORTFOLIO LIQUIDATION

3.4.1 PARAMETERS DETERMINATION

In order to compute, the parameters of the Almgren and Chriss portfolio let's explicitly express them :

- τ : The allocated periods of time
- Σ : The standard deviation
- λ : The risk aversion
- η : Transitory impact on the market
- γ : Permanent impact on the market
- X : The portfolio total value in dollars
- T : The lenght of our data-set
- ϵ : Half the average spread
- ν : The volume of transaction

Using our course knowledge we have that in fact : $g(\nu) = \gamma\nu$ GARCIN, 2023-2024. It's the linear dependency supposition of the Almgren and Chriss model. This will be usefull in order to get to the following result : $p_t - p_{t-1} = \sigma\sqrt{\tau} - \gamma\nu$ GARCIN, 2023-2024 ALMGREN and CHRISS, 1998. Using this result we can understand that in order to determine gamma with the linear hypothesis we have to do a linear regression.And if we further develop our idea of linearity in fact : $-p_t + p_{t-1} = \epsilon\nu + \frac{\eta}{\tau}\nu^2$ ALMGREN and CHRISS, 1998. The eta parameters is of the second order of the volume in order to determine η we have to look at the coefficient of order two and multiply it by τ.

Now let's dive in the code :

```python
import numpy as np
import pandas as pd
from sklearn.linear_model import LinearRegression
```

```python
from sklearn.metrics import mean_squared_error, r2_score
import matplotlib.pyplot as plt


# Load data from Excel file & Process the column of it
data = pd.read_excel(r"C:\Users\pablo\OneDrive - De Vinci\COURS\A4\S1\Market Risk\TD\TD_4\Dataset_TD4.xlsx")
data.columns = ['Date', 'Bid-Ask spread', 'Volume of transaction', 'Sign of the transaction', 'Price', '']
data = data.drop('', axis=1) # My local dataset may have been corrupted so I had to discard an empty column
data_copy = data # This is a copy of the data set for later usage.


# X & T Parameters
X_liquidate = 1E6
T = len(data)
# Here we define a return computing function, the shift() function is used to go down of one row in the same column, otherwise we could use .pct_change()
def Returns(data):
    data['Returns'] = (data['Price'].shift(1) - data['Price']) / data['Price']
    return data
data = Returns(data) # Adding a Returns column to our data set
print(data) # printing our data set
```

```
          Date  Bid-Ask spread  Volume of transaction  Sign of the transaction    Price   Returns
0     0.000202          0.1100                    8.0                       -1  100.000       NaN
1     0.001070          0.1030                    NaN                        1   99.984  0.000160
2     0.001496          0.1015                    NaN                       -1  100.029 -0.000450
3     0.003336          0.0920                    NaN                        1   99.979  0.000500
4     0.003952          0.1106                    NaN                        1  100.060 -0.000810
...        ...             ...                    ...                      ...      ...       ...
996   0.981441          0.0834                   79.0                        1  101.070 -0.000693
997   0.981875          0.1010                    NaN                       -1  101.120 -0.000494
998   0.986784          0.1007                    NaN                       -1  100.998  0.001208
999   0.991232          0.1153                    3.0                       -1  100.958  0.000396
1000  0.992002          0.1045                    NaN                        1  100.948  0.000099
```

We can observe that the Returns column has been created and that returns price has been computed for every row but not the first one. Moreover we took in account the good order of price for difference due to the fact that we have price before transaction in our Dataset.

```python
#Now we express our returns as a numpy array vector ignoring our first row
```

```python
data_returns = data['Returns'].values.astype(float)[1:]

#Using an expression in numpy array we compute our first parameters
average_spread = np.mean(data['Bid-Ask spread'].values.astype(float))
sigma = np.std(data_returns)
epsilon = average_spread / 2
tau = 1 / 24.0
lam = 2E-7 # This parameters has been arbitrarily choosed based on the
    course

#Now we introduce ou substration methods for price remembering the fact
    that it's price before transaction
def Delta_Price_gamma(data):
    data['Delta_Price_Gamma'] =  - data['Price'] + data['Price'].shift(-1)
    return data
#And according to the eta parameters of price we need to do the oposit
def Delta_Price_eta(data):
    data['Delta_Price_Eta'] =   data['Price'] - data['Price'].shift(-1)
    return data
# We apply our new function to our dataset
data = Delta_Price_gamma(data)
data = Delta_Price_eta(data)
print(data)
```

```
[1001 rows x 6 columns]
          Date  Bid-Ask spread  Volume of transaction  ...   Returns  Delta_Price_Gamma  Delta_Price_Eta
0     0.000202          0.1100                    8.0  ...       NaN             -0.016            0.016
1     0.001070          0.1030                    NaN  ...  0.000160              0.045           -0.045
2     0.001496          0.1015                    NaN  ... -0.000450             -0.050            0.050
3     0.003336          0.0920                    NaN  ...  0.000500              0.081           -0.081
4     0.003952          0.1106                    NaN  ... -0.000810              0.100           -0.100
...        ...             ...                    ...  ...       ...                ...
...
996   0.981441          0.0834                   79.0  ... -0.000693              0.050           -0.050
997   0.981875          0.1010                    NaN  ... -0.000494             -0.122            0.122
998   0.986784          0.1007                    NaN  ...  0.001208             -0.040            0.040
999   0.991232          0.1153                    3.0  ...  0.000396             -0.010            0.010
1000  0.992002          0.1045                    NaN  ...  0.000099                NaN
NaN
```

Now we can observe that we rightfully added the Delta Price column and the component are in their right place with respect to row due to the fact that once again we had price before transaction at the beginning.

```python
# Let's now clean our data set from it's NaN value in order to prepare the
    linear regression
data = data.dropna(subset=['Delta_Price_Gamma','Delta_Price_Eta','Volume of
     transaction'])
```

```python
print(data)
```

```
[1001 rows x 8 columns]
         Date  Bid-Ask spread  Volume of transaction  ...   Returns  Delta_Price_Gamma  Delta_Price_Eta
0    0.000202          0.1100                    8.0  ...       NaN             -0.016            0.016
6    0.004074          0.1294                   32.0  ... -0.000040              0.026           -0.026
16   0.014393          0.1141                    8.0  ...  0.001089             -0.019            0.019
28   0.022861          0.0978                  141.0  ...  0.000881             -0.075            0.075
51   0.037864          0.1291                  121.0  ...  0.000562             -0.086            0.086
..        ...             ...                    ...  ...       ...                ...              ...
988  0.968804          0.0896                   14.0  ... -0.001541             -0.022            0.022
989  0.969113          0.1105                  150.0  ...  0.000217             -0.101            0.101
990  0.971882          0.0929                   17.0  ...  0.000999             -0.025            0.025
996  0.981441          0.0834                   79.0  ... -0.000693              0.050           -0.050
999  0.991232          0.1153                    3.0  ...  0.000396             -0.010            0.010
```

We can observe a cleaned dataset with only 137 remaining value from the selection on the Volume of transaction and Delta Price. Now we will define our linear regression accordingly to our process

```python
# Now let's express our variable in the form of numpy array
data_delta_price_gamma = data['Delta_Price_Gamma'].values.astype(float)
data_delta_price_eta = data['Delta_Price_Eta'].values.astype(float)
# For volume we have (in order to get linear regression) to express the
    volume with it's sign so we make the product of two numpy array (same
    for the volume squarred)
volume_signed = (data['Volume of transaction'].values.astype(int)) * (data[
    'Sign of the transaction'].values.astype(int))
volume_signed_squared = (data['Volume of transaction'].values.astype(int))
    *(data['Volume of transaction'].values.astype(int)) *(data['Sign of the
    transaction'].values.astype(int))

# Linear regression parameter definition for gamma
X_gamma = volume_signed.reshape(-1,1) # here we are only looking for impact
     of order 1
y_gamma = data_delta_price_gamma

# Model's fitting
model = LinearRegression()
model.fit(X_gamma, y_gamma)

# Extraction of "coefficient directeur"
gamma = model.coef_[0] # 0 for the first coef with respect to volume_signed

#Let's compute predicted value
y_pred = model.predict(X_gamma)
# Let's compute R2 and Mean Squared Error
```

```python
mse_gamma = mean_squared_error(y_gamma, y_pred)
r2_gamma = r2_score(y_gamma, y_pred)


#definition of regression parameter for eta
X_eta = np.column_stack((volume_signed, volume_signed_squared)) # we want
    to look at impact of order two
y_eta = data_delta_price_eta
#Fitting model for eta
model.fit(X_eta,y_eta)
#Extraction of coefficient directeur but here at order of 2
eta = model.coef_[1] * tau # 1 for the second with respect to
    volume_signed_squared and due to quadratic approximation we have to
    multiply by tau
#Let's compute predicted value
y_pred = model.predict(X_eta)
# Let's compute R2 and Mean Squared Error
mse_eta = mean_squared_error(y_eta, y_pred)
r2_eta = r2_score(y_eta, y_pred)

# Printing all our computed & defined parameters
print(f"Estimated Gamma: {gamma}")
print(f"Estimated Eta: {eta}")
print("Sigma:", sigma)
print("Epsilon:", epsilon)
print("Tau:", tau)
print("Lambda", lam)
```

```
Estimated Gamma: 0.0005023780590409201
Estimated Eta: 2.2807061147258726e-08
Sigma: 0.0007380476897886703
Epsilon: 0.05026138861138861
Tau: 0.041666666666666664
Lambda 2e-07
```

With only this estimated value we can't say much about their importance, So let's proceed to analysis measurement.

```python
#Printing our computed measure to analyze our model
print(f'MSE_gamma : {mse_gamma}')
print(f'r2_gamma : {r2_gamma}')
print(f'MSE_eta : {mse_eta}')
print(f'r2_eta: {r2_eta}')
```

```
MSE_gamma : 0.0004907505222142957
r2_gamma : 0.9193112589657633
MSE_eta : 0.00023873091719548648
r2_eta: 0.9607480862831544
```

We can observe a R2 coefficient of 96 percent for η and 91 percent for γ, remembering the higher the R2 coefficient is the better the model explain our variable. Which in our case is a well enough percentage. And it confirms the linear hypothesis of Almgren and Chriss.
On the other hand, the mean squared error is putting up to light aberrant value in the process. In our case the MSE for both is really low as such our processing was good enough. But to go further in our determination we could have done something that Machine Learning is using a lot : Normalization. By normalizing our data we restrain them, forcing them in a way that's more usable for a dataset with a high level of disparate data.

3.4.2 TRAJECTORY FOR LIQUIDATION

Now we want to know how much we have to sell of our portfolio every hour to liquidate it, we will show that depending the amount we have to liquidate the higher our risk aversion have to be high (which in fact seems logical even on a liquid market if we want to sell a high amount in a day we have to expose our-self to risk).

```python
# determination of K coefficient using the hypothesis that tau = 1/24 is
    sufficiently near of zero in order to have this simplification
K = np.sqrt(((sigma**2) * lam) / (eta)) # Here we note that lambda has to
    be positive

def trajectory(X, T, K, tau, data):
    ans = []
    indexer = 0
    for t in range(T):
```

```
        if (data['Date'].values.astype(float))[t] >= indexer * tau: #This
    condition make sure that we sell a certain amount of our portfolio only
    every hour
            indexer = indexer + 1
            x = ((np.sinh(K * (T - (t - (1 / 2) * tau))) / np.sinh(K * T) *
 X))
            ans.append(x)
        else:
            x = ans[-1] if ans else 0  # Use 0 if ans is empty but normaly
    no because we start at t_0 for the first liquidation

    return np.array(ans)

trajectory_X0_K0 = trajectory(X_liquidate, T, K, tau, data_copy) #Here we
    use data_copy because data has been corrupted due to too many
    modification on it's number of row

# Plot the trajectory
plt.plot(trajectory_X0_K0)
plt.xlabel('Time')
plt.ylabel('Number of shares held in dollars')
plt.title('Optimal Liquidation Trajectory')
plt.show()
```

Now we will print graphics for different price and value of λ. Referring to the graphics chart in Appendix we can see that for both different price the liquidation steps is highly dependent of λ and when it's near zero as $\lambda = 2E - 15$ we can see that independently of the price we follow a linear distribution of the liquidate asset over the time. This confirm the Almgren and Chriss model.

On another topic we want in fact to liquidate the portfolio faster but we don't want to have too much convexity on the curve also we don't want to go over boundaries for the time allocated (here one day). As such we try to found lambda, in our example $\lambda = 2E - 6$ seems to be a good candidate but if we want to proceed with a quantitative method we can use dichotomy with respect to our exigence. Certainly it's a naive method but it's a first way to do.

```
#Definition of the amount to trade every hour up to lambda = 2E-6 and X = 1
    E6
```

```python
def Strategie(trajectory):
    strat = []
    for i in range(0, len(trajectory) - 1): # we can't consider last index as it's a rest
        diff = trajectory[i] - trajectory[i+1]
        strat.append(diff)
    strat.append(trajectory[len(trajectory)-1]) # we add the rest
    return strat
strategie = Strategie(trajectory_X0_K0)
strategie_trade = strategie[:len(strategie)-1]
rest = strategie[len(strategie)-1:len(strategie)]
print(f"\nTrade to be made every hour in order {strategie_trade}\n") # This represent the amount in dollar to liquidate at every hour with respect to the hypothesis we made a trade at t_0
print(f"Rest at the end of trade : {rest}")
```

```
 Trade to be made every hour in order [325658.99271134834, 270044.96020391927, 149906.67888044516, 89855.04303438572, 34489.
55755620934, 40557.26383651978, 24417.442810796383, 20324.503043942612, 10379.970790748404, 7801.471618967989, 7565.25616923
9678, 5439.085378582653, 3128.8993098192714, 2433.5973160250314, 1807.178757821157, 1649.6966790376518, 752.4662878409522, 1
081.2746258269744, 663.2043245907557, 551.1183746383676, 532.3895368720923, 420.5430561905562, 408.68572113761314]

Rest at the end of trade : [274.7177029288644]
```

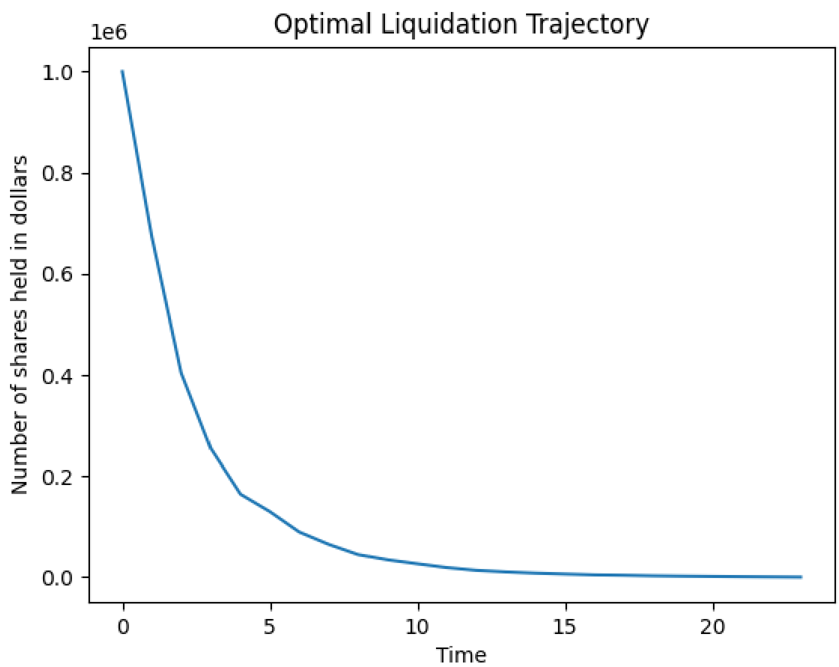


Figure 1: $X = 1E6$ and $\lambda = 2E-6$

We can see that at the end there is a rest which can be in the last trade or can't depending on

if we can or can't do a trade at over 24hour period. Thus it can be considered a loss, but knowing the amount of the loss comparing to the portfolio size in dollars this is irrelevant. As well the rest existing has to be minimize without having a too big impact on market, thus resulting in the problem of minimizing rest with the convexity of the curve being minimal also. Maybe some better approximation of lambda than dichotomy can approach this better.
To conclude we have shown that the risk exposure rate is important for the liquidation strategies and it's impact and value as to be measured with the amount we have to liquidate and we gave a clean strategy to liquidate the portfolio .

3.5 PART V: WAVELET, HURST AND VOLATILITY

To answer this questions we will first present the first part of our code which is related to Pre-processing :

```
import numpy as np
import  pandas as pd
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
import seaborn as sns

#Preprocessiing
data = pd.read_excel(r"C:\Users\pablo\OneDrive - De Vinci\COURS\A4\S1\
    Market Risk\TD\TD_5\Dataset TD5.xlsx")
data = data.drop(data.index[0]) # we will drop first two row as it's not
    pure data
data = data.drop(data.index[0])

data.columns = ['GBPEUR_Date', 'GBPEUR_HIGH', 'GBPEUR_LOW', 'Unamed_1', '
    SEKEUR_Date',
        'SEKEUR_HIGH', 'SEKEUR_LOW', 'Unamed_2', 'CADEUR_Date', 'CADEUR_HIGH
    ',
        'CADEUR_LOW']
data = data.drop('Unamed_1', axis=1) # Due to spaces between data there is
    empty column to drop
data = data.drop('Unamed_2', axis=1)

#Let's print our data
print(data)
```

```
                        GBPEUR_Date GBPEUR_HIGH GBPEUR_LOW  ...                 CADEUR_Date CADEUR_HIGH CADEUR_LOW
2      2016-03-07 08:59:59.990000      1.2932     1.2917  ... 2016-03-07 08:59:59.990000      0.6842     0.6829
3             2016-03-07 09:15:00       1.294      1.293  ...        2016-03-07 09:15:00      0.6849     0.6841
4             2016-03-07 09:30:00      1.2943     1.2922  ...        2016-03-07 09:30:00      0.6844     0.6837
5             2016-03-07 09:45:00       1.293     1.2913  ...        2016-03-07 09:45:00      0.6844     0.6839
6             2016-03-07 10:00:00      1.2931     1.2921  ...        2016-03-07 10:00:00       0.684     0.6835
...                           ...         ...        ...  ...                        ...         ...        ...
12926         2016-09-07 17:00:00      1.1879     1.1867  ...        2016-09-07 17:00:00      0.6897     0.6893
12927         2016-09-07 17:15:00      1.1883     1.1874  ...        2016-09-07 17:15:00      0.6902     0.6895
12928         2016-09-07 17:30:00       1.188     1.1874  ...        2016-09-07 17:30:00      0.6902     0.6898
12929         2016-09-07 17:45:00      1.1874     1.1866  ...        2016-09-07 17:45:00      0.6902     0.6901
12930         2016-09-07 18:00:00       1.187     1.1869  ...        2016-09-07 18:00:00      0.6901     0.6901
```

```python
#Let's compute average price and create new column in our data set to use
    them here we use numpy built in function to preserve as much as possible
     significative numbers
data['GBPEUR_Avg_Price'] = np.divide(np.add(data['GBPEUR_HIGH'] ,data['
    GBPEUR_LOW']),2)
data['SEKEUR_Avg_Price'] = np.divide(np.add(data['SEKEUR_HIGH'] ,data['
    SEKEUR_LOW']), 2)
data['CADEUR_Avg_Price'] = np.divide(np.add(data['CADEUR_HIGH'] ,data['
    CADEUR_LOW']),2)
#Let's observe new data set
print(data)
```

```
                        GBPEUR_Date GBPEUR_HIGH GBPEUR_LOW  ... GBPEUR_Avg_Price SEKEUR_Avg_Price CADEUR_Avg_Price
2      2016-03-07 08:59:59.990000      1.2932     1.2917  ...          1.29245         0.107225          0.68355
3             2016-03-07 09:15:00       1.294      1.293  ...           1.2935         0.107225           0.6845
4             2016-03-07 09:30:00      1.2943     1.2922  ...          1.29325         0.107225          0.68405
5             2016-03-07 09:45:00       1.293     1.2913  ...          1.29215         0.107245          0.68415
6             2016-03-07 10:00:00      1.2931     1.2921  ...           1.2926          0.10722          0.68375
...                           ...         ...        ...  ...              ...              ...              ...
12926         2016-09-07 17:00:00      1.1879     1.1867  ...           1.1873         0.105335           0.6895
12927         2016-09-07 17:15:00      1.1883     1.1874  ...          1.18785         0.105355          0.68985
12928         2016-09-07 17:30:00       1.188     1.1874  ...           1.1877          0.10537             0.69
12929         2016-09-07 17:45:00      1.1874     1.1866  ...            1.187         0.105365          0.69015
12930         2016-09-07 18:00:00       1.187     1.1869  ...          1.18695          0.10537           0.6901
```

```python
#Let's define a function to compute returns and creating new column for
    each of our Fx once again we use numpy function to preserve
    significative number
def Returns(data, column, stocks):

     returns_name = 'Returns_' + str(stocks)
     data[returns_name] = np.divide(np.subtract(data[column] ,data[column].
    shift(1)),data[column].shift(1))
     print(data)
     return data
#Let's apply it
data = Returns(data, 'CADEUR_Avg_Price', 'CADEUR')
```

```python
data = Returns(data, 'GBPEUR_Avg_Price', 'GBPEUR')
data = Returns(data, 'SEKEUR_Avg_Price', 'SEKEUR')
#Let's observe our data
print(data)
```

```
                  GBPEUR_Date GBPEUR_HIGH GBPEUR_LOW  ... SEKEUR_Avg_Price CADEUR_Avg_Price Returns_CADEUR
2      2016-03-07 08:59:59.990000      1.2932     1.2917  ...         0.107225          0.68355            NaN
3             2016-03-07 09:15:00       1.294      1.293  ...         0.107225           0.6845        0.00139
4             2016-03-07 09:30:00      1.2943     1.2922  ...         0.107225          0.68405      -0.000657
5             2016-03-07 09:45:00       1.293     1.2913  ...         0.107245          0.68415       0.000146
6             2016-03-07 10:00:00      1.2931     1.2921  ...          0.10722          0.68375      -0.000585
...                           ...         ...        ...  ...              ...              ...            ...
12926         2016-09-07 17:00:00      1.1879     1.1867  ...         0.105335           0.6895       0.000363
12927         2016-09-07 17:15:00      1.1883     1.1874  ...         0.105355          0.68985       0.000508
12928         2016-09-07 17:30:00       1.188     1.1874  ...          0.10537             0.69       0.000217
12929         2016-09-07 17:45:00      1.1874     1.1866  ...         0.105365          0.69015       0.000217
12930         2016-09-07 18:00:00       1.187     1.1869  ...          0.10537           0.6901      -0.000072

                  GBPEUR_Date GBPEUR_HIGH GBPEUR_LOW  ... CADEUR_Avg_Price Returns_CADEUR Returns_GBPEUR
2      2016-03-07 08:59:59.990000      1.2932     1.2917  ...          0.68355            NaN            NaN
3             2016-03-07 09:15:00       1.294      1.293  ...           0.6845        0.00139       0.000812
4             2016-03-07 09:30:00      1.2943     1.2922  ...          0.68405      -0.000657      -0.000193
5             2016-03-07 09:45:00       1.293     1.2913  ...          0.68415       0.000146      -0.000851
6             2016-03-07 10:00:00      1.2931     1.2921  ...          0.68375      -0.000585       0.000348
...                           ...         ...        ...  ...              ...            ...            ...
12926         2016-09-07 17:00:00      1.1879     1.1867  ...           0.6895       0.000363      -0.000589
12927         2016-09-07 17:15:00      1.1883     1.1874  ...          0.68985       0.000508       0.000463
12928         2016-09-07 17:30:00       1.188     1.1874  ...             0.69       0.000217      -0.000126
12929         2016-09-07 17:45:00      1.1874     1.1866  ...          0.69015       0.000217      -0.000589
12930         2016-09-07 18:00:00       1.187     1.1869  ...           0.6901      -0.000072      -0.000042

                  GBPEUR_Date GBPEUR_HIGH GBPEUR_LOW  ... Returns_CADEUR Returns_GBPEUR Returns_SEKEUR
2      2016-03-07 08:59:59.990000      1.2932     1.2917  ...            NaN            NaN            NaN
3             2016-03-07 09:15:00       1.294      1.293  ...        0.00139       0.000812            0.0
4             2016-03-07 09:30:00      1.2943     1.2922  ...      -0.000657      -0.000193           -0.0
5             2016-03-07 09:45:00       1.293     1.2913  ...       0.000146      -0.000851       0.000187
6             2016-03-07 10:00:00      1.2931     1.2921  ...      -0.000585       0.000348      -0.000233
...                           ...         ...        ...  ...            ...            ...            ...
12926         2016-09-07 17:00:00      1.1879     1.1867  ...       0.000363      -0.000589       0.000142
12927         2016-09-07 17:15:00      1.1883     1.1874  ...       0.000508       0.000463        0.00019
12928         2016-09-07 17:30:00       1.188     1.1874  ...       0.000217      -0.000126       0.000142
12929         2016-09-07 17:45:00      1.1874     1.1866  ...       0.000217      -0.000589      -0.000047
12930         2016-09-07 18:00:00       1.187     1.1869  ...      -0.000072      -0.000042       0.000047

                  GBPEUR_Date GBPEUR_HIGH GBPEUR_LOW  ... Returns_CADEUR Returns_GBPEUR Returns_SEKEUR
2      2016-03-07 08:59:59.990000      1.2932     1.2917  ...            NaN            NaN            NaN
3             2016-03-07 09:15:00       1.294      1.293  ...        0.00139       0.000812            0.0
4             2016-03-07 09:30:00      1.2943     1.2922  ...      -0.000657      -0.000193           -0.0
5             2016-03-07 09:45:00       1.293     1.2913  ...       0.000146      -0.000851       0.000187
6             2016-03-07 10:00:00      1.2931     1.2921  ...      -0.000585       0.000348      -0.000233
...                           ...         ...        ...  ...            ...            ...            ...
12926         2016-09-07 17:00:00      1.1879     1.1867  ...       0.000363      -0.000589       0.000142
12927         2016-09-07 17:15:00      1.1883     1.1874  ...       0.000508       0.000463        0.00019
12928         2016-09-07 17:30:00       1.188     1.1874  ...       0.000217      -0.000126       0.000142
12929         2016-09-07 17:45:00      1.1874     1.1866  ...       0.000217      -0.000589      -0.000047
12930         2016-09-07 18:00:00       1.187     1.1869  ...      -0.000072      -0.000042       0.000047
```

```python
# Due to Nan value on first row let's avoid it by dropping it
data = data.dropna()
print(data)
```

```
[12929 rows x 15 columns]
            GBPEUR_Date GBPEUR_HIGH GBPEUR_LOW  ... Returns_CADEUR Returns_GBPEUR Returns_SEKEUR
3      2016-03-07 09:15:00       1.294      1.293  ...        0.00139       0.000812            0.0
4      2016-03-07 09:30:00      1.2943     1.2922  ...      -0.000657      -0.000193           -0.0
5      2016-03-07 09:45:00       1.293     1.2913  ...       0.000146      -0.000851       0.000187
6      2016-03-07 10:00:00      1.2931     1.2921  ...      -0.000585       0.000348      -0.000233
7      2016-03-07 10:15:00      1.2926     1.2921  ...           -0.0      -0.000193      -0.000093
...                    ...         ...        ...  ...            ...            ...            ...
12926  2016-09-07 17:00:00      1.1879     1.1867  ...       0.000363      -0.000589       0.000142
12927  2016-09-07 17:15:00      1.1883     1.1874  ...       0.000508       0.000463        0.00019
12928  2016-09-07 17:30:00       1.188     1.1874  ...       0.000217      -0.000126       0.000142
12929  2016-09-07 17:45:00      1.1874     1.1866  ...       0.000217      -0.000589      -0.000047
12930  2016-09-07 18:00:00       1.187     1.1869  ...      -0.000072      -0.000042       0.000047
```

Here end the pre-processing now we will involve implementing Haar wavelet. Implementation of the Haar wavelet and transformation methods of our data with respect to Haar mother wavelet. The function $\psi(t)$ is defined as:

$$\psi(t) = \begin{cases} -1 & \text{if } 0 \leq t < \frac{1}{2}, \\ 1 & \text{if } \frac{1}{2} \leq t < 1, \\ 0 & \text{otherwise.} \end{cases}$$

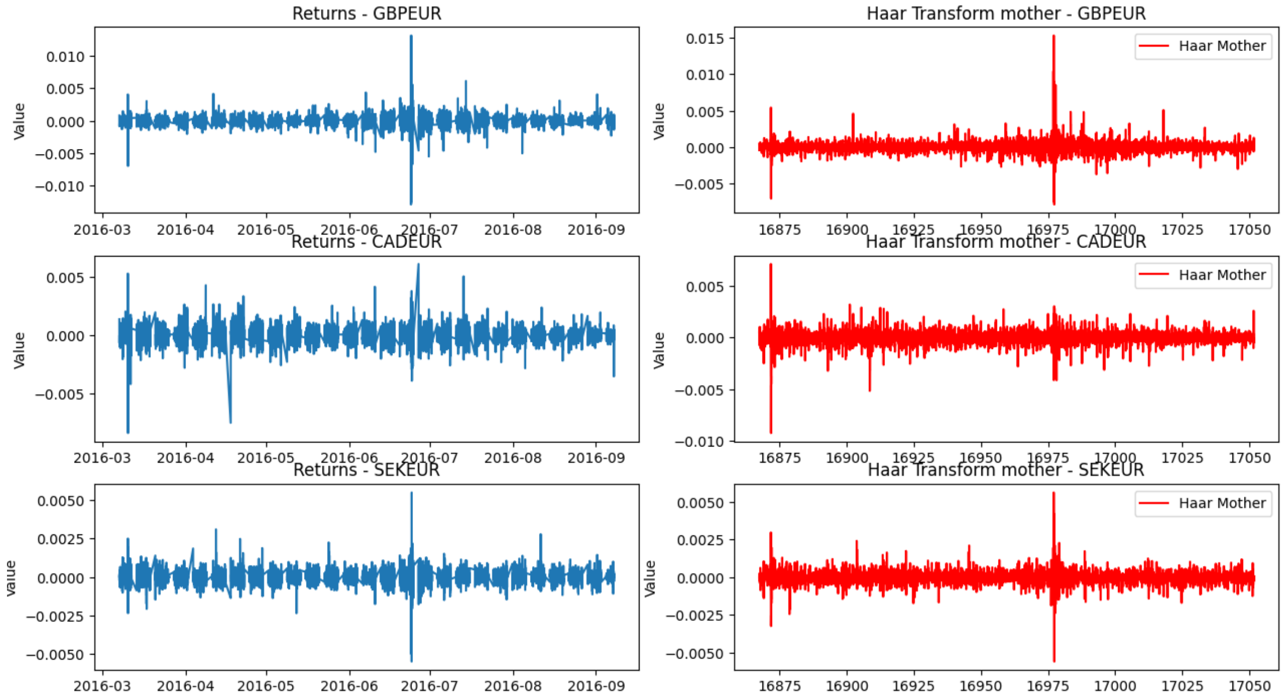
GARCIN, 2023-2024

```python
# Now let's define our mother walette with respect to the cours page 302
def haar_mother_wavelet(x):
    return np.where((x >= 0) & (x < 0.5), 1, np.where((x >= 0.5) & (x < 1),
     -1, 0))

#Here is the transform function obtained from the course
def Haar_transform(data, t):
    haar_transform = []
    N = len(data) # The total number of data available

    for k in range(N // t): #We use floor division which is a division that
     takes the rounded number after dividing
        start_idx = k * t # begining of value considered
        end_idx = start_idx + t # end of value considered with respect to t
     the time scale
        scale_factor = 1 / np.sqrt(t)

        # Let's compute approximation coefficient
        approx_coeff = scale_factor * np.sum(haar_mother_wavelet(np.arange(
```

```
start_idx, end_idx) / N) * data[start_idx:end_idx]) # np.arrange is there to make sure we are taking our value in the good order
        haar_transform.append(approx_coeff)

        # Let's compute detail coefficient
        detail_coeff = scale_factor * np.sum(haar_mother_wavelet(np.arange(start_idx + 0.5 * t, end_idx + 0.5 * t) / N) * data[start_idx:end_idx])
        haar_transform.append(detail_coeff)

    # Approximation coefficients for the remaining parts that is if N is a multiple of our time scales only
    if N % t != 0:
        remaining_coeff = scale_factor * np.sum(haar_mother_wavelet(np.arange(N - (N % t), N) / N) * data[-(N % t):]) / (N % t)
        haar_transform.extend([remaining_coeff] * (N % t))# Here we add elements at the end of the list

    return np.array(haar_transform)# we give the result under the form of a numpy.array which will be usefull later

t=int(30/15) # We take a time intervalle of 30 minuts and due to our data spacing being 15 minuts we divide by this here for the begining t = 2 which will give a really precise result of our modelisation

# Apply Haar wavelet transform to returns we make sure to pass numpy array with .values and with .astype(float) we make sure the type of the data is float
haar_transform_CADEUR_mother = Haar_transform(data['Returns_CADEUR'].values.astype(float), t)
haar_transform_GBPEUR_mother = Haar_transform(data['Returns_GBPEUR'].values.astype(float), t)
haar_transform_SEKEUR_mother = Haar_transform(data['Returns_SEKEUR'].values.astype(float), t)

# Create a 3x2 grid of subplots to print our result
fig, axs = plt.subplots(3, 2, figsize=(14, 10))

# Plot for GBPEUR
axs[0, 0].plot(data['GBPEUR_Date'], data['Returns_GBPEUR'])
```

```
axs[0, 0].set_title('Returns - GBPEUR')

# Convert datetime to numerical values due to the possibility of t = 1 we have to make sure the date value correspond to the good haar value obtained
date_values_GBPEUR = mdates.date2num(data['GBPEUR_Date'])
haar_x_values_GBPEUR = np.linspace(date_values_GBPEUR[0], date_values_GBPEUR[-1], len(haar_transform_GBPEUR_mother))
axs[0, 1].step(haar_x_values_GBPEUR, haar_transform_GBPEUR_mother, color='red', label='Haar Mother')
axs[0, 1].set_title('Haar Transform mother - GBPEUR')
axs[0, 1].legend()
# Plot for CADEUR
axs[1, 0].plot(data['CADEUR_Date'], data['Returns_CADEUR'])
axs[1, 0].set_title('Returns - CADEUR')
# Convert datetime to numerical values
date_values_CADEUR = mdates.date2num(data['CADEUR_Date'])
haar_x_values_CADEUR = np.linspace(date_values_CADEUR[0], date_values_CADEUR[-1], len(haar_transform_CADEUR_mother))
axs[1, 1].step(haar_x_values_CADEUR, haar_transform_CADEUR_mother, color='red', label='Haar Mother')
axs[1, 1].set_title('Haar Transform mother - CADEUR')
axs[1, 1].legend()
# Plot for SEKEUR
axs[2, 0].plot(data['SEKEUR_Date'], data['Returns_SEKEUR'])
axs[2, 0].set_title('Returns - SEKEUR')
# Convert datetime to numerical values
date_values_SEKEUR = mdates.date2num(data['SEKEUR_Date'])
haar_x_values_SEKEUR = np.linspace(date_values_SEKEUR[0], date_values_SEKEUR[-1], len(haar_transform_SEKEUR_mother))
axs[2, 1].step(haar_x_values_SEKEUR, haar_transform_SEKEUR_mother, color='red', label='Haar Mother')
axs[2, 1].set_title('Haar Transform mother - SEKEUR')
axs[2, 1].legend()
# Add a common y-axis label
for ax in axs.flat:
    ax.set(ylabel='Value')
# Adjust layout to prevent clipping of ylabel
fig.tight_layout()
```

```
# Show the plot
plt.show()
```



Here we have chosen $t = 2$ which refer to a really close time space (close to the minima which is $t = 1$). As such our data fitting is really close to the return observed. If you run the code you will be able to zoom in the red graphic and you will be able to see that Haar transformation is a stair based transformation. This is in respect to the theory.

3.5.1 CORRELATION MATRIX, VOLATILITY WITH RESPECT TO HURST EXPONENT, COVARIANCE

Using the code exposed previously to compute the Haar transformation of a given data-set. We will now expose how to compute correlation matrix between the coefficient of Approximation and Details :

First let's introduce the different time scales :

```
#Now let's introduce differents time scales with respect to 10080 minuts
    representing a week. But our data set starting not at 00:00:00 we will
    consider a little bit more of value than necessary
time_minutes = [15, 30, 60, 120, 240, 480, 960, 1440, 2880, 4320, 10080]
scales = [int(i/15) for i in time_minutes]
print(f"Our different time scales for the haar wavelette {scales}")
```

```
Our different time scales for the haar wavelette [1, 2, 4, 8, 16, 32, 64, 96, 192, 288, 672]
```

Then let's introduce our new portfolio made of the FX average prices normalized :

```python
# Calculate portfolio prices by taking the sum of the three avg FX pric and dividing it by 3
data['Prices_Portfolio'] = (data['GBPEUR_Avg_Price'] + data['SEKEUR_Avg_Price'] + data['CADEUR_Avg_Price']) / 3
data = Returns(data, 'Prices_Portfolio', 'Portfolio') # Then we compute their returns (average on the normalized of the average price of the three FX)
data = data.dropna()
print(data)
```

```
                GBPEUR_Date GBPEUR_HIGH GBPEUR_LOW  ... Returns_SEKEUR Prices_Portfolio Returns_Portfolio
4       2016-03-07 09:30:00      1.2943     1.2922  ...           -0.0         0.694842         -0.000336
5       2016-03-07 09:45:00       1.293     1.2913  ...       0.000187         0.694515          -0.00047
6       2016-03-07 10:00:00      1.2931     1.2921  ...      -0.000233         0.694523          0.000012
7       2016-03-07 10:15:00      1.2926     1.2921  ...      -0.000093         0.694437         -0.000125
8       2016-03-07 10:30:00       1.293     1.2906  ...            0.0         0.694237         -0.000288
...                     ...         ...        ...  ...            ...              ...
...
12926   2016-09-07 17:00:00      1.1879     1.1867  ...       0.000142         0.660712         -0.000219
12927   2016-09-07 17:15:00      1.1883     1.1874  ...        0.00019         0.661018          0.000464
12928   2016-09-07 17:30:00       1.188     1.1874  ...       0.000142         0.661023          0.000008
12929   2016-09-07 17:45:00      1.1874     1.1866  ...      -0.000047         0.660838          -0.00028
12930   2016-09-07 18:00:00       1.187     1.1869  ...       0.000047         0.660807         -0.000048
```

Now let's introduce our method to compute the correlation matrix, this one relies on a good understanding of the Approximation and Details coefficients, then we use the numpy corrcoef methods to compute them. This methods is just computing Covariance of the two sliced of the data set, then dividing it by the product of the standard deviation of both sliced data :

```python
#Let's now define correlation matrix for each scale defined earlier
def wavelet_correlation_matrix(data, scales):
    correlation_matrices = []

    for scale in scales:
        # Apply Haar wavelet transform to portfolio returns
        haar_transform_portfolio = Haar_transform(data, scale)
        # Combine approximation and detail coefficients for correlation calculation because we want to look at the correlation of this two coefficient to understand up and down of our data
        half_len = len(haar_transform_portfolio) // 2 #Depending of the half length of the data set there is disjunction cas in order to avoir division by zero
```

```python
        if len(haar_transform_portfolio) % 2 == 0:
            combined_coefficients = np.vstack([
                haar_transform_portfolio[:half_len],
                haar_transform_portfolio[half_len:]
            ])
        else:
            combined_coefficients = np.vstack([
                haar_transform_portfolio[:half_len],
                haar_transform_portfolio[half_len:-1]
            ])
        #We used Vstack to create a vertical array of both our coefficient
        # Calculate the correlation matrix for the current scale
        correlation_matrix = np.corrcoef(combined_coefficients) #Here we use numpy.corrcoef, which is using a covariance classical formula and dividing it by the var of both the two set considered here our splitted set

        # We had each matrix of different scales in a bigger vector to have them at our disposition in order
        correlation_matrices.append(correlation_matrix)
    #We return all the matrices in order
    return correlation_matrices

#Let's compute all the correlation matrices with once again a numpy array given on the returns of portfolio
Correlation_Matrices = wavelet_correlation_matrix(data['Returns_Portfolio'].values.astype(float), scales)
print(f"Here is the correlation matrix: {Correlation_Matrices}")
```

```
Here is the correlation matrix: [array([[ 1.        , -0.00645],
       [-0.00645,  1.        ]]), array([[ 1.        , -0.0157578],
       [-0.0157578,  1.        ]]), array([[ 1.        , -0.0319397],
       [-0.0319397,  1.        ]]), array([[ 1.        , -0.0254295],
       [-0.0254295,  1.        ]]), array([[1.        , 0.00218832],
       [0.00218832, 1.        ]]), array([[1.        , 0.00802792],
       [0.00802792, 1.        ]]), array([[ 1.        , -0.12931199],
       [-0.12931199,  1.        ]]), array([[ 1.        , -0.00878005],
       [-0.00878005,  1.        ]]), array([[ 1.        , -0.17796262],
       [-0.17796262,  1.        ]]), array([[ 1., nan],
       [nan, nan]]), array([[1.00000000e+00, 5.11811851e-17],
       [5.11811851e-17, 1.00000000e+00]])]
```

What we can observe first is that all our matrices are symmetrical, which is a good start.

Moreover the anti-diagonal is made of 1 which is the correlation of a coefficient with itself. What we cans observe is that when the scales is near 1 week the correlation matrices are having really low coefficient, thus implying that between our two coefficient it began too difficult to put a relation between them. As well as that the two coefficient are negatively correlated.
Now we will compute the hurst exposant for the normalized prices. To do that let's recall :

$$H = \frac{1}{2}\log_2\left(\frac{M_2'}{M_2}\right)$$

With :
- $M_2' = \frac{2}{NT}\sum_{i=1}^{NT/2}|X(2i/N) - X\left(\frac{2(i-1)}{N}\right)|^2$
- $M_k = \frac{1}{NT}\sum_{i=1}^{NT}|X(i/N) - X\left(\frac{(i-1)}{N}\right)|^k$

GARCIN, 2023-2024

Now let's code it :

```
#With respect to the formula page 263 of the course
def Hurst_Exponent(data):
    N = 1 #Spacing variable to make [0, T] sliced with respect to 1/N
    k = 2 # With the respect to fomula page 263 involving moment of order 2
    T = len(data)
    M_2, M_2_prime = 0, 0
    # We compute the two empirical absolute moments and make sure to round
   up indexs in order to make them coeherent with a discrete selection
    for i in range(0, int(N*T)):
        M_2 += abs(data[int(i/N)] - data[int((i-1)/N)])**k
    for i in range(0, int(N*T/2)):
        M_2_prime += abs(data[int(2*i/N)] - data[int(2*(i-1)/N)])**k

    M_2 = M_2 / (N*T)
    M_2_prime = (2 / (N*T) ) * M_2_prime

    return 0.5 * np.log2(M_2_prime/M_2)


# Calculate Hurst exponent with Prices normalized
hurst_value = Hurst_Exponent(data['Prices_Portfolio'].values.astype(float))
print(f"Hurst Exponent: {hurst_value}")
```

```
Hurst Exponent: 0.5964094404939776
```

Now that we obtained our Hurst exponent based on all the normalized prices. Let's compute our volatility vector with scaled with the Hurst exponent.As Hurst $\geq 1/2$ it signifies a persistence of the trends in the series. To do that we have to identify the scale as the same scale used for our Haar transformation :

```python
# Function to calculate volatility vector with scaling based on Hurst exponent
def Volatility_vector(data, hurst, scales):
    sigma = data.std() # Here we use the standard deviation included in numpy looking at the sqrt of the var
    volatility_vector = [sigma * (scale**hurst) for scale in scales] #Each member of the vector is the given volatility for the given scale (which is the same that is used for our haar function) up to a common hurst exponent
    return volatility_vector

# Calculate volatility vector on Returns of the portfolio
volatility_vector = Volatility_vector(data['Returns_Portfolio'].values.astype(float), hurst_value, scales)
print(f"Volatility Vector: {volatility_vector}")
```

```
Volatility Vector: [0.0004570791931982273, 0.0006910804118744917, 0.001044878311644113, 0.00157980267
90875541, 0.002388581021387l853, 0.003611412596810045, 0.005460271528417262, 0.0069540327094718005, 0
.010514142539335124, 0.013390486306416153, 0.022195308118231314]
```

In this vector there is each volatility with respect to each scales (with respect to order). Now we will have to generate a diagonal matrix out of each volatility for each scale and to follow the formula :

Covariance Matrix = Diagonal Matrix(Volatility Vector) $\times$ Correlation Matrix

$\times$Diagonal Matrix(Volatility Vector)

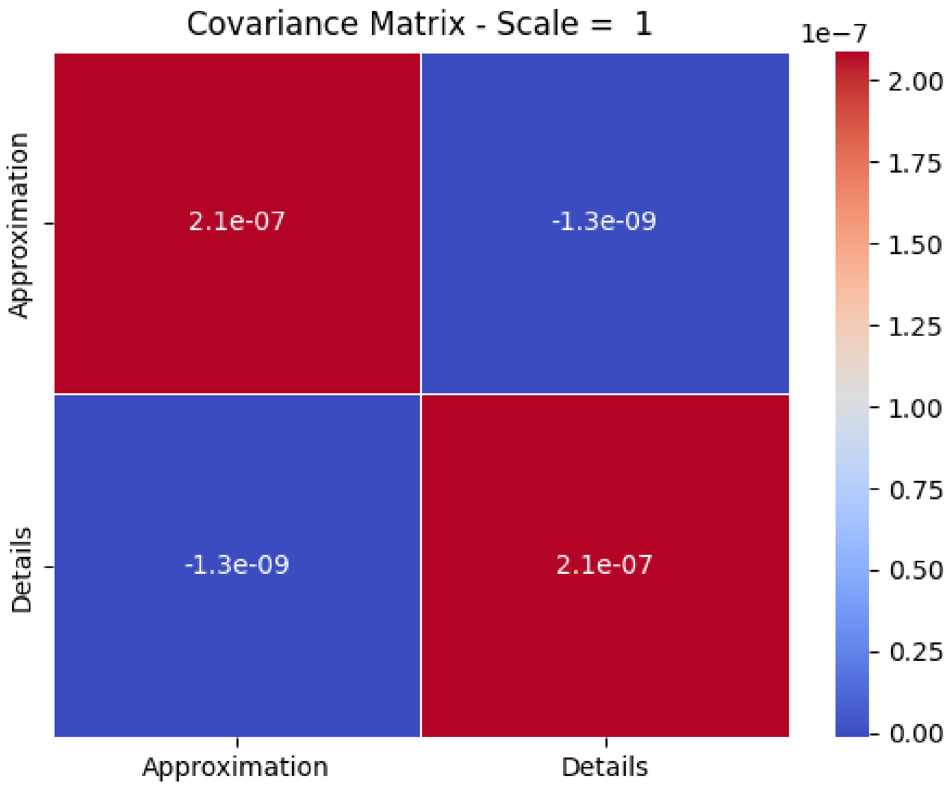
Let's recall that the correlation matrix is a 2X2 matrix, thus because we only have two coefficient, the diagonal volatility is made of the same volatility on the diagonal and is also a 2X2 matrix. As well as that the last matrix is equal to it's transposed matrix :

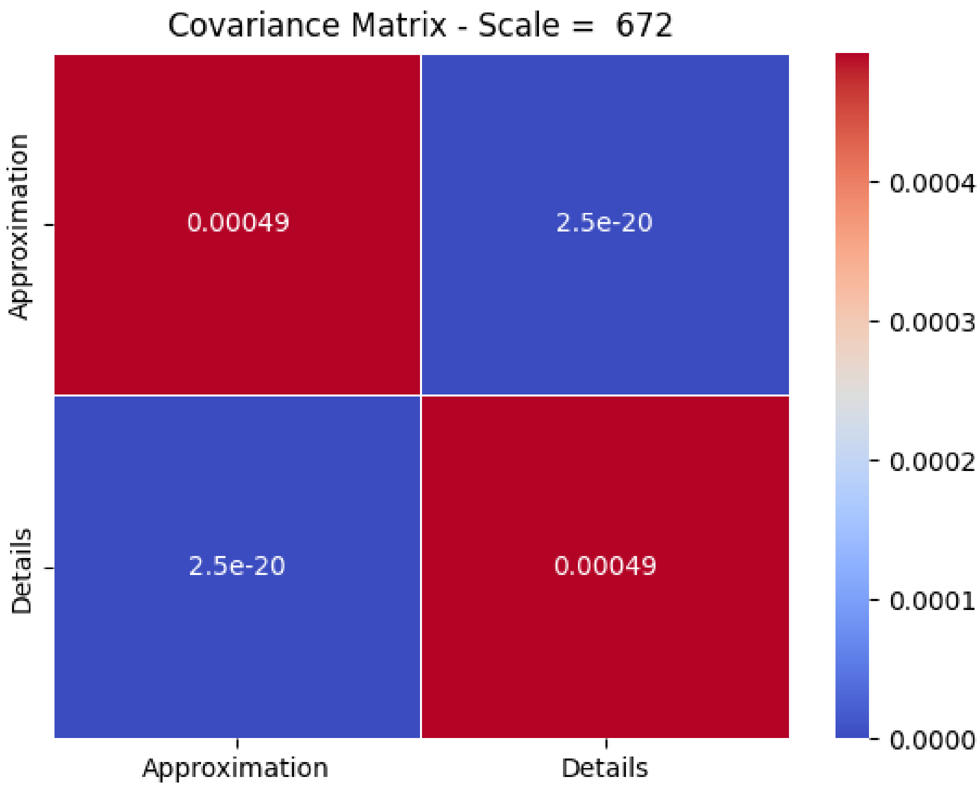
```python
#Let's define our covariance matrix with respect to mathematical formula
def Covariance_Matrices(Correlation_Matrices, volatility_vector, scales):
    num_matrices = len(Correlation_Matrices) # the number of correlation
```

```
matrix we need to itterate uppon which is the same number as the len(scales)
    covariance_matrices = [] #stocking vectors for each covariance matrix in order with respect to scales

    for i in range(num_matrices):
        vol_matrix = np.diag([volatility_vector[i], volatility_vector[i]]) # If we recall that we have symmetrical correlation matrices we need to developp a 2X2 diagonal matrix made of on the diagonal of the probability for the given scales, which is the case because they are all ordered in the same way.

        correlation_matrix = Correlation_Matrices[i] # We select a Correlation Matrix

        # Matrix multiplication to get the covariance matrix with np.dot which is used for multiplying matrix (for information it's using hadamard product) more over because vol_matrix is diagonal and of the same six as correlation_matrix we don't need to take the transpose at the end
        covariance_matrix = np.dot(vol_matrix, np.dot(correlation_matrix, vol_matrix))
        #We stock each covariance matrix in a vector
        covariance_matrices.append(covariance_matrix)

        # Display the covariance matrix using seaborn, this module offer the opportunity to easly print heat map which are more readable than any other things
        sns.heatmap(covariance_matrix, annot=True, cmap="coolwarm", linewidths=.5,
                    xticklabels=['Approximation', 'Details'], yticklabels=['Approximation', 'Details'])
        plt.title(f"Covariance Matrix - Scale =  {scales[i]}") # here it will print the scale considered so t = int(15/15) = 1 for example
        plt.show()
    #Let's return all our covariance_matrices
    return covariance_matrices
#Let's compute the covariance matrices for our already computed parameter
covariance_matrices = Covariance_Matrices(Correlation_Matrices,
```

```
    volatility_vector, scales)

# Access individual covariance matrices and print them to make sure the
    result is coherent with seaborn heatmap print
for i, cov_matrix in enumerate(covariance_matrices):
    print(f"Covariance Matrix - Scale = {scales[i]}:\n{cov_matrix}")
```

As we can see in the code there is a lot of printing, we will just put here the covariance matrix for $Scale = 1$ and the one for $Scale = 672$. If you want to see the numerical print or all the Covariance matrices with respect to scale in the ascending order please got to APENDIX.





At first what we can observe is that as the scale increases, the values in the covariance matrices also increase. This is expected, as larger scales imply a broader time window, leading to larger variations in the returns.Moreover the covariance is dominated by it's main diagonal. the variances of individual components are much larger than the covariance between them.
Thus leading us to the fact that as the scale increases, the impact of the volatility vector becomes more pronounced it is with respect to the fact the volatility vector is directly incorporated into the covariance matrices through the diagonal matrix multiplication.
Then we can conclude that the volatility is sensitive to the scale at which we observe of Haar transformation. Moreover with the Hurst exponent being between 0.5 and 1 it signifies a persistent time series with long-range dependence. Which we confirmed earlier.

3.5.2 VOLATILITY ESTIMATION DIRECTLY ON THE DATA

First as the question suggest it we will discuss about the overlapping or not return.

With overlapping returns :

- We choose a time interval and our return calculation will share points between adjacent time interval.

- *Advantages* : is that we will generate more points, thus resulting in a more precise analysis.And so for short-terms price we will have a better approach.

- *Disadvantages* : is that the correlation will increase drastically between adjacent returns that are sharing data .

With non - overlapping returns :

- Now our returns don't share any common point between each interval in our calculation.

- *Advantages* : we make it so that each returns is from a distinct data points, and so the correlation is diminished.

- *Disadvantages* : We have a less precise analysis and so we can have a hard time to catch short-terms movement.

Now previously we used the Hurst exponent to compute the volatility vector, the objectives of the Hurst exponent is to catch trends over a series and it work as a reminder of how the trend of the series should go at a long term range. With this statement we can consider that if we decide to proceed to a short term analysis with overlapping results the Hurst exponent need to be clarified if it will help by looking at it.

Let's recall that :

```
time_minutes = [15, 30, 60, 120, 240, 480, 960, 1440, 2880, 4320, 10080]
scales = [int(i/15) for i in time_minutes]
print(f"Our different time scales for the haar wavelette {scales}")
```

```
Our different time scales for the haar wavelette [1, 2, 4, 8, 16, 32, 64, 96, 192, 288, 672]
```

Is the scale, and so the number of point previously defined that we will consider.

So let's compute it for a restrained number of price :

```
for scale in scales:
    hurst_value = Hurst_Exponent(data['Prices_Portfolio'].values.astype(float)[:scale]) # here we restrain the numpy array to only number of data equal to scale.
    print(f"Hurst Exponent for Time Scale {scale}: {hurst_value}")
```

```
Hurst Exponent for Time Scale 1: nan
c:\Users\pablo\OneDrive - De Vinci\COURS\A4\S1\Market Risk\TD\TD_5\TD_5.py:183: RuntimeWarning: divid
e by zero encountered in log2
  return 0.5 * np.log2(M_2_prime/M_2)
Hurst Exponent for Time Scale 2: -inf
Hurst Exponent for Time Scale 4: 0.27121053230679626
Hurst Exponent for Time Scale 8: 0.30082810730885917
Hurst Exponent for Time Scale 16: 0.23523900258801375
Hurst Exponent for Time Scale 32: 0.5375555848479342
Hurst Exponent for Time Scale 64: 0.4181875863384873
Hurst Exponent for Time Scale 96: 0.5940782141195596
Hurst Exponent for Time Scale 192: 0.6063314831924959
Hurst Exponent for Time Scale 288: 0.5356950819919261
Hurst Exponent for Time Scale 672: 0.6827200491127478
```

If we look from $Scale = 4$ due to the fact that the first result are aberrant due to lack of points or division by zero.

We can observe that on small segmented number of point (in comparison to the 12000+ entry of the data set) the Hurst exponent is $\leq 1/2$. At least until we consider 96 points in the data set. But for larger number of segmented point it tend to converge to a value $\geq 1/2$.

If we decided to go for a short term analysis, with overlapping results to for example give us more points, depending of the scale considered (so the number of point) we will consider different Hurst exponent to reflect the trends in the segmented series.

Let's consider $scale = 64$ and so an analysis for a time periods of 16 hours, 960 minutes :

Now let's implement our Method :

Step 1 - We will compute overlapped returns to smooth the volatility over short-term periods.

Step 2 - We will consider the number of points shared as our scaling coefficient for the scaled volatility with the Hurst exponent computed on the restricted data segment to better capture short-term response in our analysis.

Step 3 - We will consider only a restricted number of points in our data set, starting from the beginning, to emphasize short-term analysis.

Step 4 - After computing volatility for each restricted number of points, where each segment shares no points when we slice the whole data set, we will compute the average weighted volatility.

```
def Overlapping_Returns(arr, points_shared):
    returns = []
```

```
    n = len(arr)

    for i in range(n - points_shared + 1):# we consider this sum until this
     index in order for points to make groups of points_shared
        subset1 = arr[i:i + points_shared]# we define the price at t for a
    certain numbers of points_shared
        subset2 = arr[i + 1:i + points_shared + 1]# we define the price at
    t +1  for a certain numbers of points shared)

        # Calculate returns for the overlapping subsets
        returns.append((subset2[-1] - subset1[0]) / subset1[0])

    return np.array(returns)

def calculate_volatility(data, points_shared, hurst):
    returns = Overlapping_Returns(data, points_shared)  # Compute returns
    for overlapping points
    volatility = np.std(returns) # standard deviation for the returns
    series

    # Scale volatility based on the Hurst exponent
    scaled_volatility = volatility * (points_shared ** hurst) # here we
    choosed to used point_shared as the scaling factor in order to catch a
    better response on short term analysis but if we wanted to look at
    longer term we could have considered scaling = 64

    return scaled_volatility

scaling = 64 #the length of each segment
shared_points = 5 # the number of points shared in price calculation
Price_Portfolio_length = len(data['Prices_Portfolio'].values.astype(float))
     # The total length of the column considered
volatility_vector_2 = [] #Stocking array for our volatility

for i in range(0, Price_Portfolio_length, scaling): # here the argument in
    python go as follow, begin, start, steps
    prices_slice = data['Prices_Portfolio'].values.astype(float)[i:i+
    scaling] # the sliced interval

```

```
    #Compute the hurst exponent value for each slice
    hurst_value = Hurst_Exponent(prices_slice)

    # Calculate volatility for different time scales with overlapping
   returns and scaling based on Hurst exponent
    volatility_scaled = calculate_volatility(prices_slice, shared_points,
   hurst_value)

    volatility_vector_2.append(volatility_scaled)

print(f"Here is my volatility for {scaling} points steps considered with
   returns sharing {shared_points} points for each intervals : {
   volatility_vector_2}")
```

You will be able to fin the print at the end of the appendix. But what we computed is all the short terms smoothed volatility for our data set, when we say short term we want to say with a 16 hour horizon. Moreover we considered each independent Hurst exponent to reflect, the trends of each segment, which may be not the same as the whole trends of the data-set.But we recall that here our objectives is to emphasize with a short-term analysis. Let's now compute the weighted average volatility for our whole data-set, we decided to use weighting but in our case every sliced segment of our data set share the same number of points. As such it will be equally weighted. But if someone want to do a multi level analysis depending of the day or the periods of the year (for this data set) then they will be able to obtain coherent results with weighting :

```
# Calculating the weighted average of volatilities here the weight is the
   same for all the segment because they have the same lenght but we can
   imagine it's not the case
weighted_volatility_sum = 0
total_points = 0

for i in range(len(volatility_vector_2)):# the size of each segment is
   scaling
    segment_size = scaling
    total_points += segment_size
    weighted_volatility_sum += segment_size * volatility_vector_2[i]

# Calculating the total volatility as the weighted average
```

```
total_volatility = weighted_volatility_sum / total_points

print(f"Total volatility for the entire dataset: {total_volatility}")
```

```
Total volatility for the entire dataset: 0.002437036637597957
```

Finally this method gave us a result for a short term oriented analysis with respect to some smoothing in order to have more points to consider and to give a better result. The result given has to be compared with other methods to know if the data set considered (or the restricted one in our case) is highly volatile or not. This type of short term volatility leads to short terms risk analysis which is really important on market where the correlation is highly involved leading to a major difficulty in order to hedge a book of asset. The part of the course beginning to explain this subject is with the application of fractional Brownian motion.

4 CONCLUSION

To conclude, we tried our best not to use advanced package from python limiting our self to : numpy, pandas, scikit-learn(for linear regression, R2, MSE), and seaborn(for a heat map).
We tried to develops as much as possible our own functionality but sometimes due to other priorities we used methods from pandas and numpy mainly.
We made some references in order to follow our reasoning and the code is mainly explained as a form of commentary, thus leading to an educational approach.
We tried to give formal explanation of our result and to explain them with respect to the knowledge we gain from the course as much as possible. As it is an educational project there may be some errors or wrong doing, but we would like to advise that the author at that time are still learning and beginning their journey on market risk.
Finally for clarity we furnished this report as a LaTex document and emphasize the result in a pleasant way as much as we could. As well as that the workload was evenly shared this project is the result from a group project.

References

ALMGREN, R., & CHRISS, N. (1998). Optimal liquidation.

GARCIN, M. (2023-2024). Market risk. *ESILV*.

5 APPENDIX

5.1 PART IV : GRAPHICS FOR TRAJECTORY WITH RESPECT TO RISK AVERSION

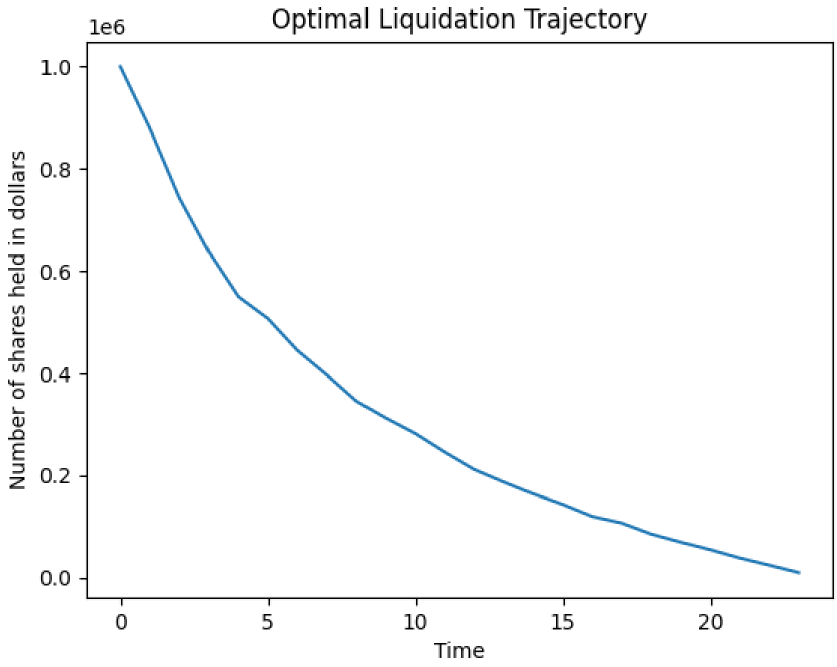


Figure 2: $X = 1E6$ and $\lambda = 2E - 7$

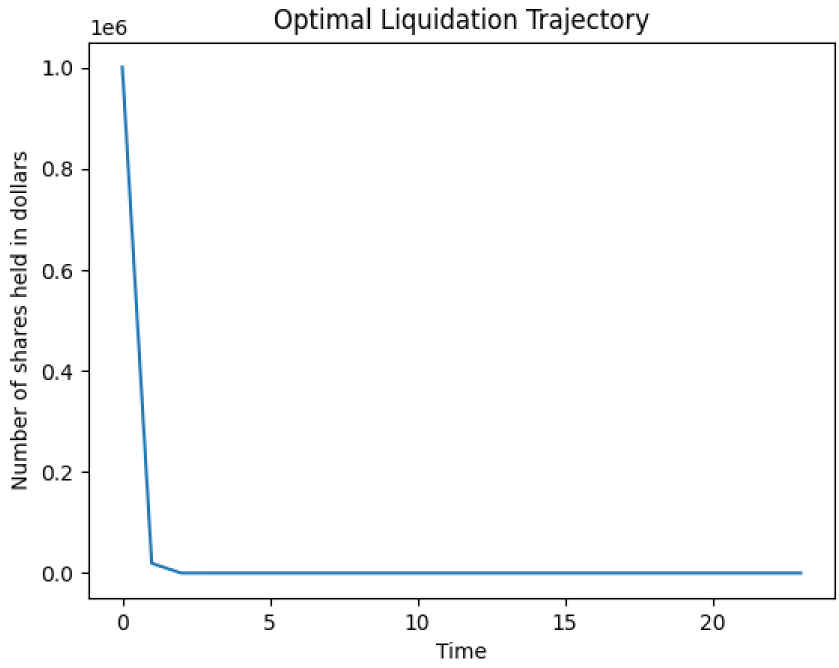


Figure 3: $X = 1E6$ and $\lambda = 2E - 4$

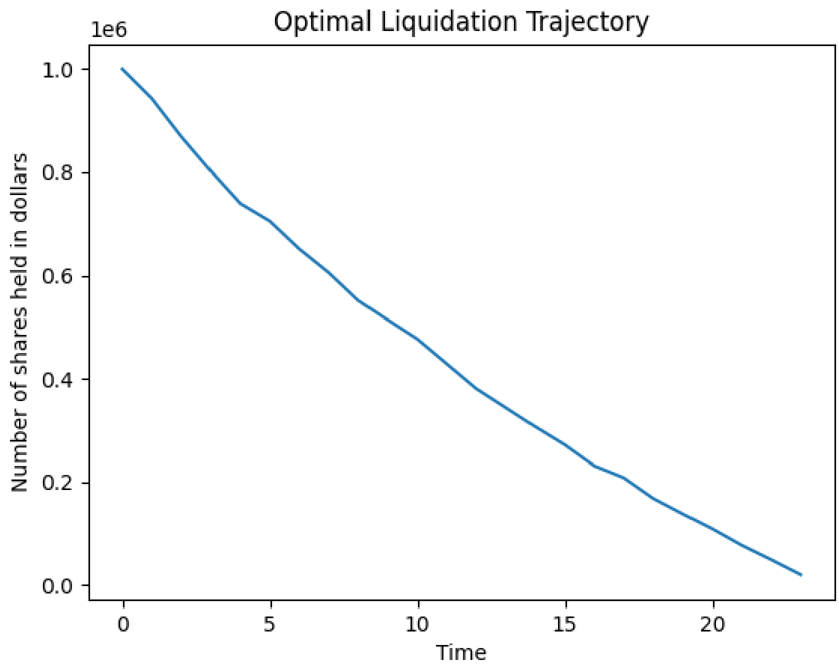


Figure 4: $X = 1E6$ and $\lambda = 2E - 15$

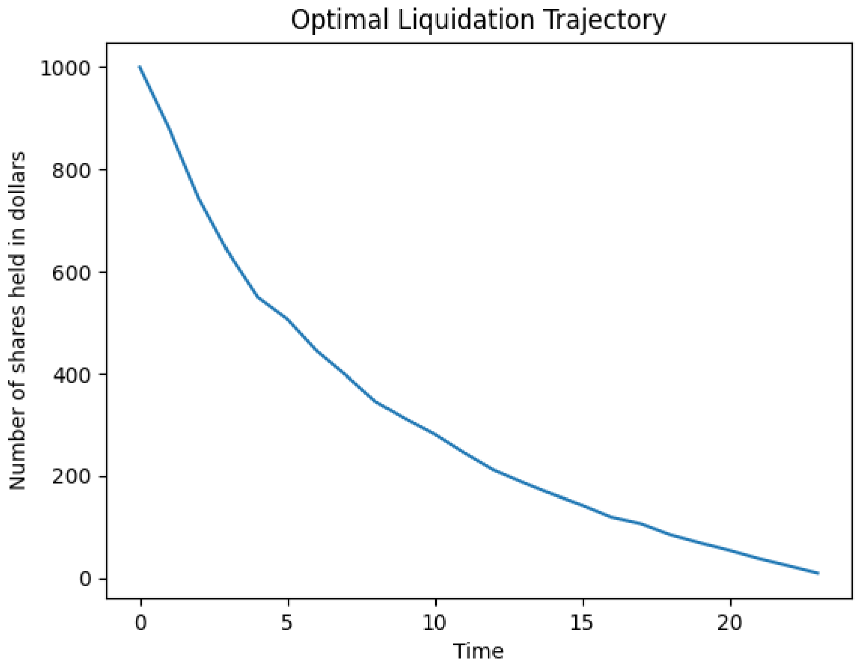


Figure 5: $X = 1000$ and $\lambda = 2E - 7$

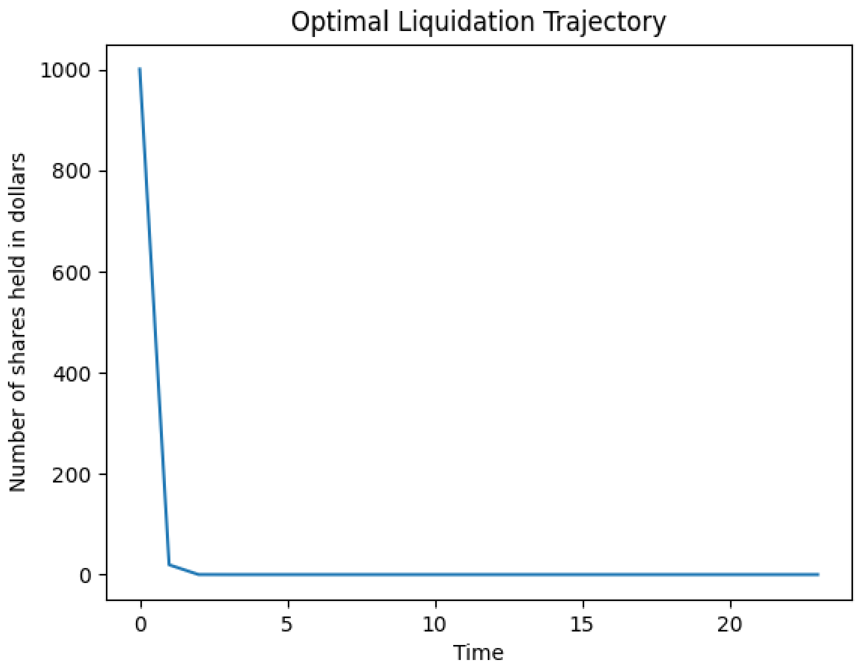


Figure 6: $X = 1000$ and $\lambda = 2E - 4$

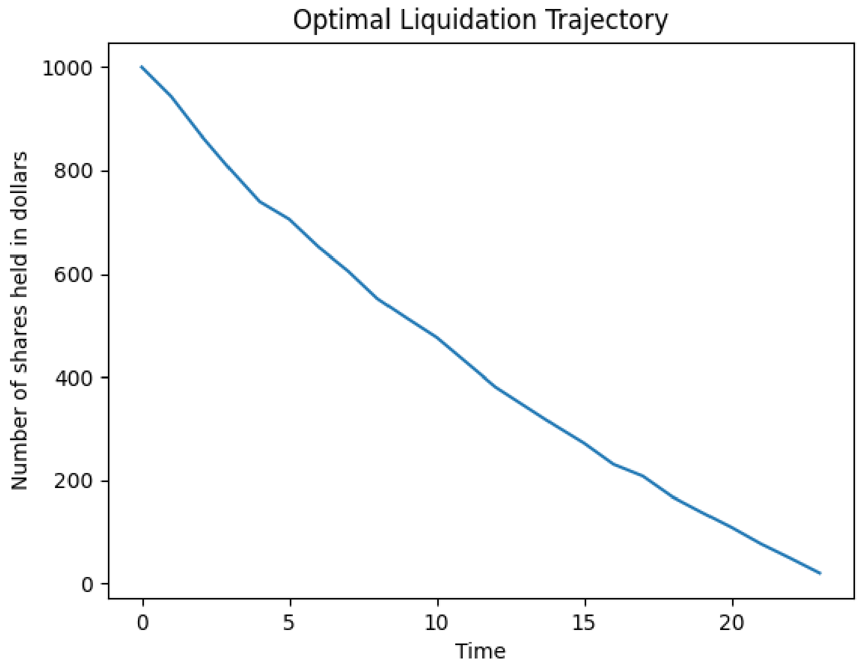
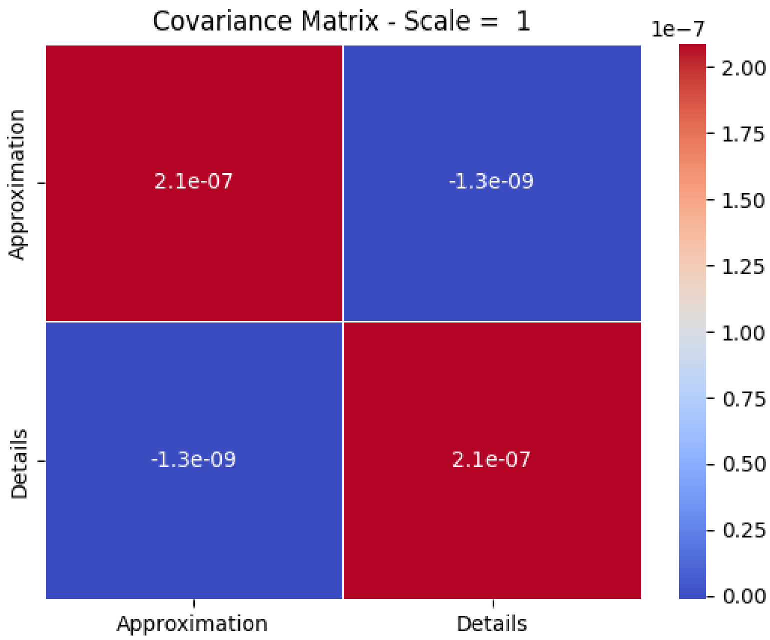
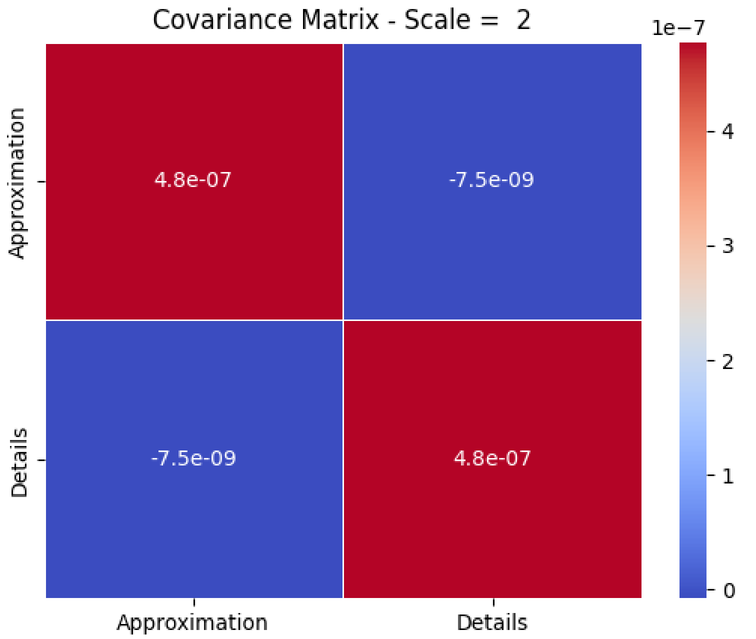
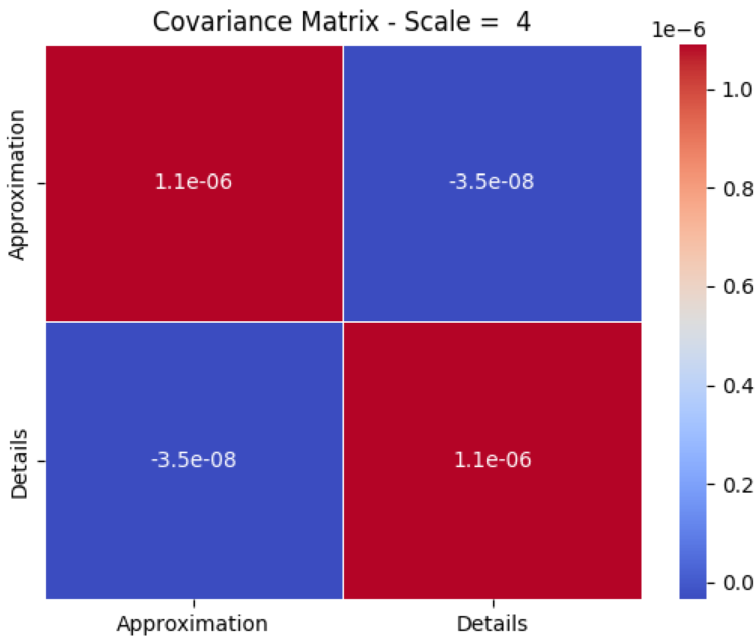


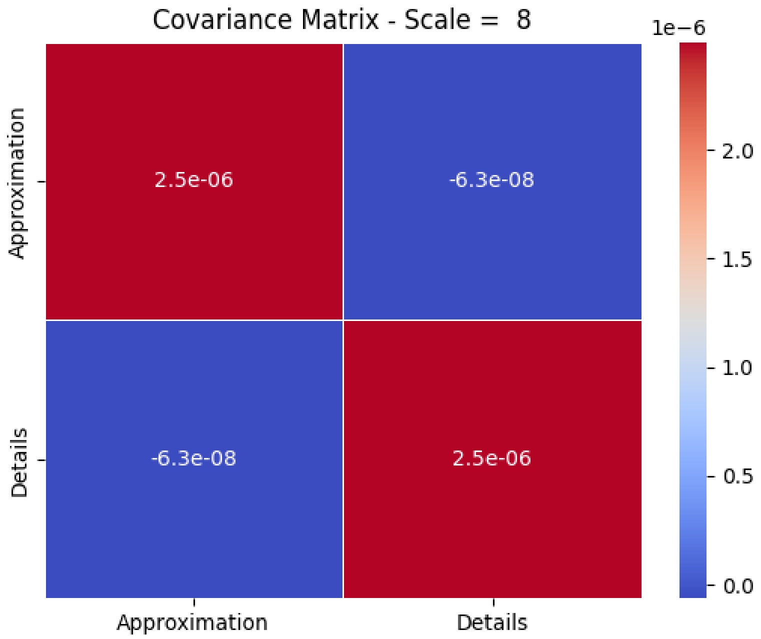
Figure 7: $X = 1000$ and $\lambda = 2E - 4$

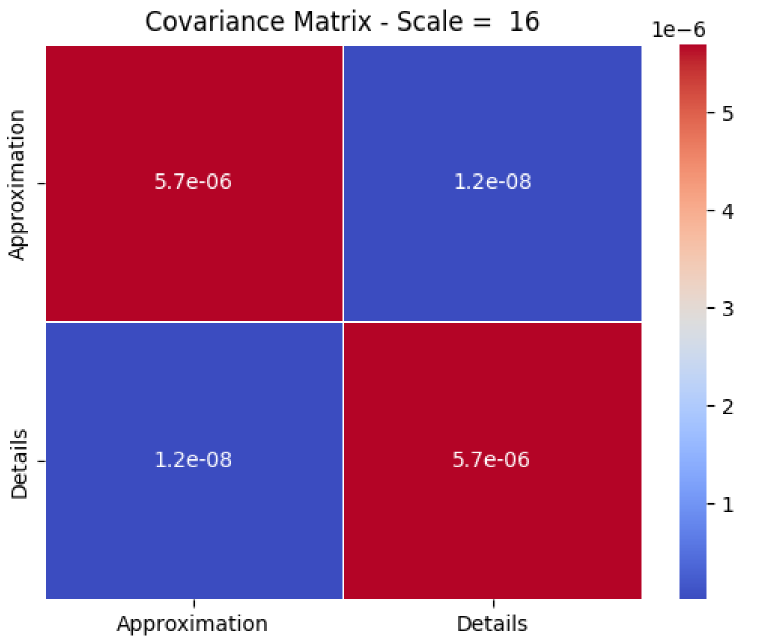
5.2 PART V : GRAPHICS FOR COVARIANCE MATRIX

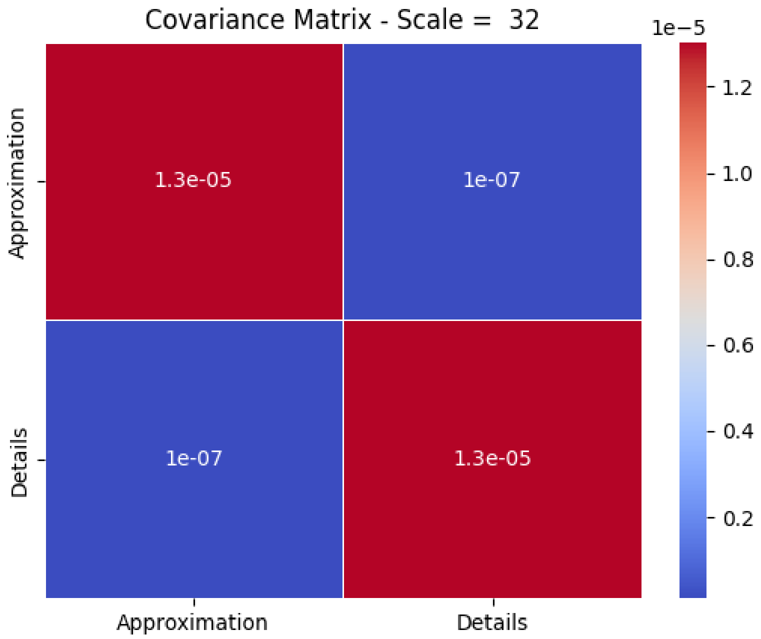


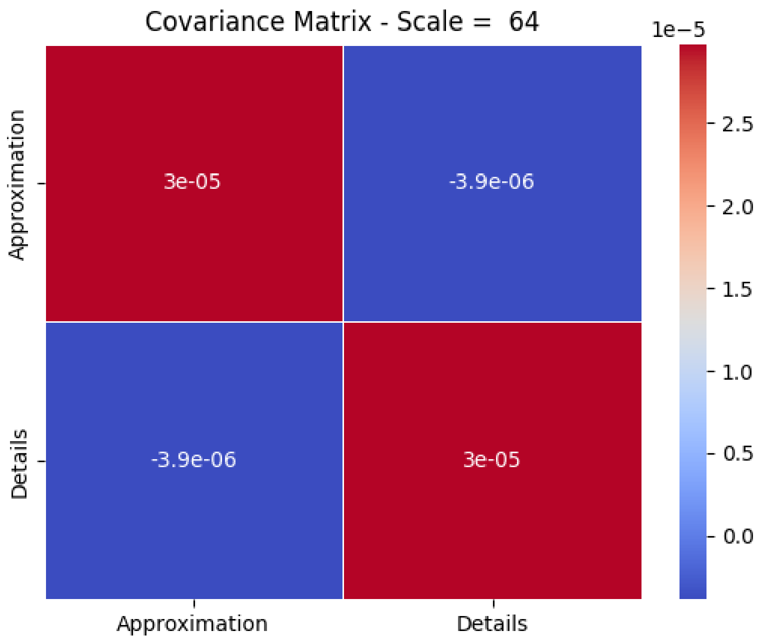

Covariance Matrix - Scale = 2
1e−7
4.8e-07
-7.5e-09
-7.5e-09
4.8e-07
Approximation
Details
Approximation
Details
4
3
2
1
0

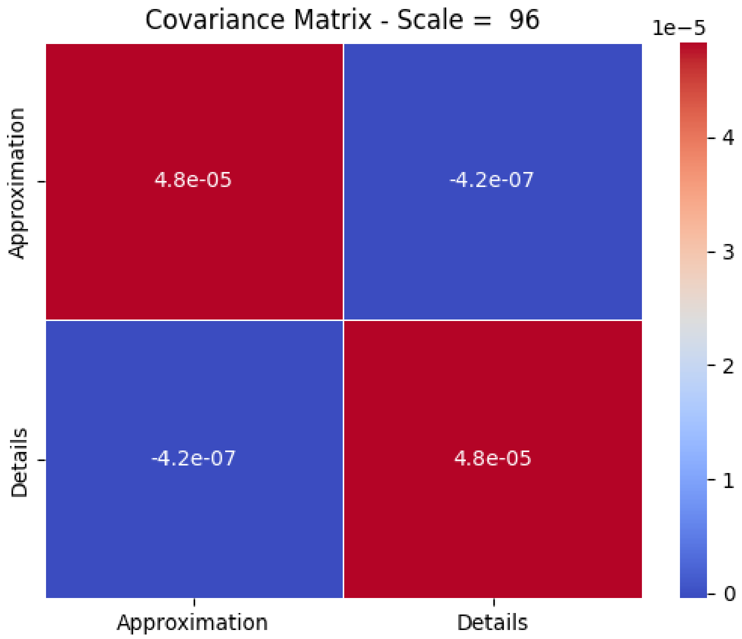

Covariance Matrix - Scale = 4
1e−6
1.1e-06
-3.5e-08
-3.5e-08
1.1e-06
Approximation
Details
Approximation
Details
1.0
0.8
0.6
0.4
0.2
0.0

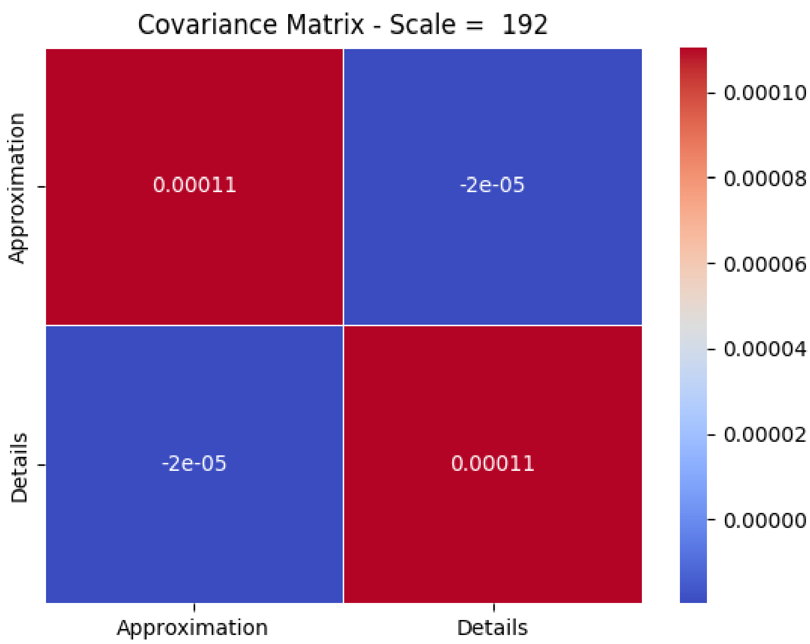

Covariance Matrix - Scale = 8
1e−6
2.5e-06
-6.3e-08
-6.3e-08
2.5e-06
Approximation
Details
Approximation
Details
2.0
1.5
1.0
0.5
0.0


Covariance Matrix - Scale = 16
1e−6
5.7e-06
1.2e-08
1.2e-08
5.7e-06
Approximation
Details
Approximation
Details
5
4
3
2
1

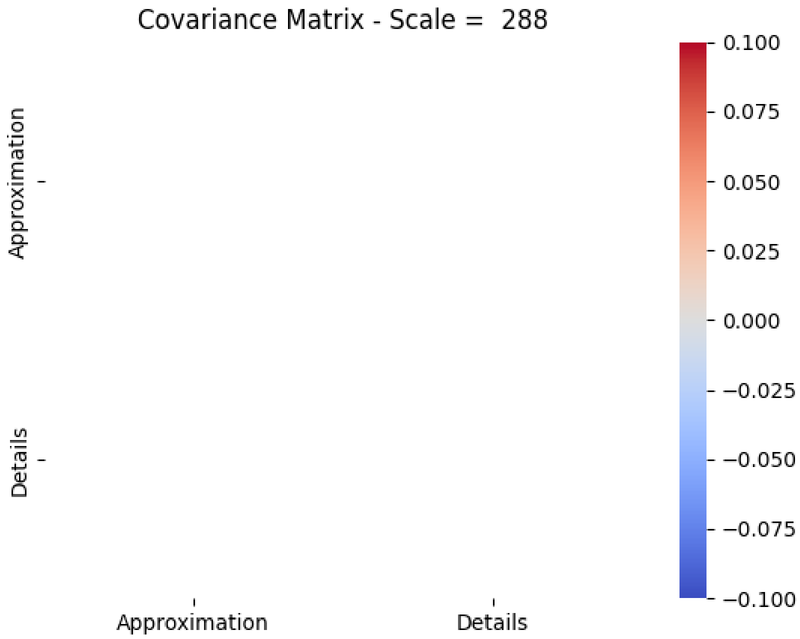

Covariance Matrix - Scale = 32
1e−5
1.2
1.0
0.8
0.6
0.4
0.2
Approximation
Details
1.3e-05
1e-07
1e-07
1.3e-05
Approximation
Details

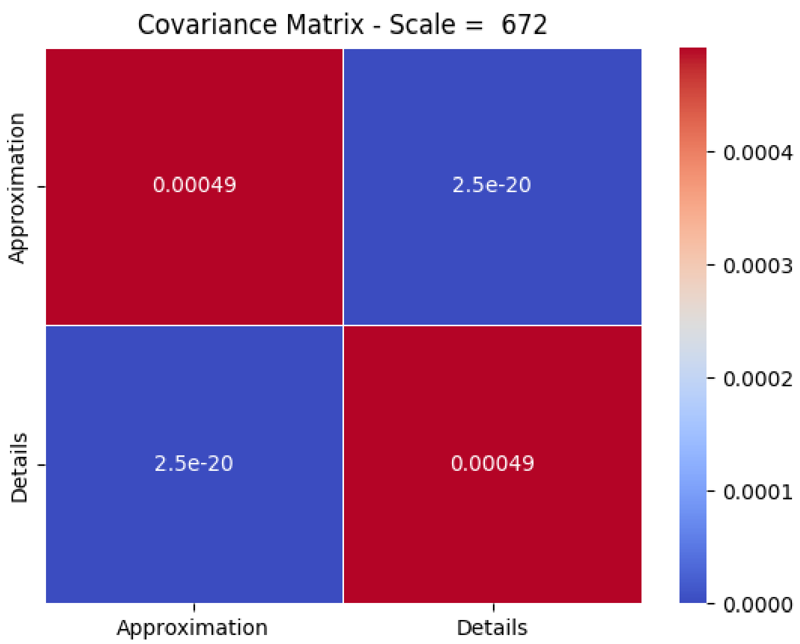

Covariance Matrix - Scale = 64
1e−5
2.5
2.0
1.5
1.0
0.5
0.0
Approximation
Details
3e-05
-3.9e-06
-3.9e-06
3e-05
Approximation
Details





in the following figure we can observe a bug due to np.nan value, we don't really kow how to explain maybe it's due to a zero division.





And now if we want to observe only numerical value :

```
Covariance Matrix - Scale = 1:
[[ 2.08921389e-07 -1.34754245e-09]
 [-1.34754245e-09  2.08921389e-07]]
Covariance Matrix - Scale = 2:
[[ 4.77592136e-07 -7.52579987e-09]
 [-7.52579987e-09  4.77592136e-07]]
Covariance Matrix - Scale = 4:
[[ 1.09177069e-06 -3.48708318e-08]
 [-3.48708318e-08  1.09177069e-06]]
Covariance Matrix - Scale = 8:
[[ 2.49577650e-06 -6.34663584e-08]
 [-6.34663584e-08  2.49577650e-06]]
Covariance Matrix - Scale = 16:
[[5.70531930e-06 1.24850787e-08]
 [1.24850787e-08 5.70531930e-06]]
Covariance Matrix - Scale = 32:
[[1.30423009e-05 1.04702538e-07]
 [1.04702538e-07 1.30423009e-05]]
Covariance Matrix - Scale = 64:
[[ 2.98145652e-05 -3.85538079e-06]
 [-3.85538079e-06  2.98145652e-05]]
Covariance Matrix - Scale = 96:
[[ 4.83585709e-05 -4.24590738e-07]
 [-4.24590738e-07  4.83585709e-05]]
Covariance Matrix - Scale = 192:
[[ 1.10547193e-04 -1.96732684e-05]
 [-1.96732684e-05  1.10547193e-04]]
```

```
Covariance Matrix - Scale = 288:
[[nan nan]
 [nan nan]]
Covariance Matrix - Scale = 672:
[[4.92631702e-04 2.52134744e-20]
 [2.52134744e-20 4.92631702e-04]]
```

Now for the volatility vector with our second methods :

```
Here is my volatility for 64 points steps considered with returns sharing 5 points for each intervals
 : [0.001257402282691442, 0.003122740326080706, 0.0009804829824466917, 0.0018200466501994642, 0.0118
97649970878686, 0.002553583147229226, 0.003887904389741938, 0.0027861103689652785, 0.0016065298940142
062, 0.0024093812947514463, 0.0012216204321311044, 0.0025123701197196533, 0.002891505554951855, 0.001
198656800263487, 0.0012257928424311952, 0.0020542216848208105, 0.00167678548439103, 0.002329149119908
087, 0.002892043614852962, 0.001329277885751402, 0.001961453808999486, 0.0012844811835973373, 0.00049
51181341645334, 0.0015743458265055916, 0.002339697717183839, 0.0009690446250435722, 0.002221948174597
569, 0.0020322878357699713, 0.0021319520730558234, 0.0026799610909469624, 0.001536139378070152, 0.001
639538328960625, 0.002309723333326093, 0.0017827757882502597, 0.0022285952054072653, 0.00560823712800
6361, 0.001843214818942703, 0.0025508020611566804, 0.0038764115408702524, 0.002014272800092385, 0.001
5240757378062425, 0.0030569533721722065, 0.001151026047214523, 0.0023120507364491477, 0.0044054136530
63203, 0.0032193092459496557, 0.0026494962798478397, 0.002445858143350698, 0.0019208389542794539, 0.0
032725978237644793, 0.0030275559796094664, 0.0005388840619793891, 0.0035853933650340374, 0.0030468601
772411547, 0.0010180605908748496, 0.001833208896666156, 0.001980928174672048, 0.0014020570097885816,
0.004242284389536092, 0.001378679530598233, 0.0018404258603797977, 0.0036863802948789623, 0.002803519
3090901146, 0.0019120070593726536, 0.0021297012509228037, 0.0020796855798201784, 0.001392336874386041
, 0.00245411786079203, 0.0026888570758568436, 0.0025362961799408044, 0.0013311845476734231, 0.0023927
382422246745, 0.0013741316627983926, 0.002169730551104578, 0.0006853874987688105, 0.00243212712058714
62, 0.0017130952368862328, 0.0029141007318004555, 0.0036332478708193494, 0.0033021940989617655, 0.000
7527038159674251, 0.002701313514181724, 0.0017501740466047031, 0.0015530805934826672, 0.0041665103391
27541, 0.002852140814413369, 0.0013907718689554938, 0.0031481637144287616, 0.002558011142823661, 0.00
08222356699915606, 0.0020426767528772425, 0.002109873067365097, 0.0020449976034180866, 0.002985479227
5798674, 0.0028062137853766423, 0.0015365674272727447, 0.002416094585035, 0.0010380777941811146, 0.00
3424299329110428, 0.002315224352358672, 0.002412818853024138, 0.0031370101236857344, 0.00290796566820
32, 0.002150683575867992, 0.0021313477342224676, 0.003019489687711474, 0.0020909322435292994, 0.00294
1042840080113, 0.00293797145650463, 0.0012537063863988847, 0.0020902243862977485, 0.00274229497864598
43, 0.0017264608945657374, 0.0027696675888110667, 0.0026918171513944686, 0.003007599853503133, 0.003
1045472120091035, 0.0024106214721028316, 0.0021677108785205846, 0.0031853334996534122, 0.017991477426
71075, 0.0048844092254433135, 0.0032011158748156297, 0.0027054530000330504, 0.003969126165093797, 0.0
033033536825140012, 0.002272676993208136, 0.00524445531448911, 0.0019472137042311553, 0.0016025994447
643804, 0.0022860534253578276, 0.004492249149980387, 0.004396598461563056, 0.003396816784934544, 0.00
4204933823892505, 0.001969189626951761, 0.000664389042983146, 0.006411598187302683, 0.001518457724851
8495, 0.003970530887472588, 0.0025080681075235357, 0.0022615634605283167, 0.0020682165230984316, 0.00
34775247919223745, 0.0016050834184628386, 0.0024170613050556976, 0.0019204554904475172, 0.00230795963
32591973, 0.0032241027736379458, 0.0037900334180903054, 0.001855079516436561, 0.005179465013208334, 0
.0010732128783669966, 0.001901957343814371, 0.00227878910742709, 0.0016357712339130856, 0.00208706684
62996556, 0.0017904694658638243, 0.0022113189348104043, 0.004498088328146036, 0.0016904756068112997,
0.0033727747643843087, 0.0017735878010000788, 0.001966839826108725, 0.0014866083858142594, 0.00383781
90160515773, 0.0023859124285329175, 0.0009090180268357401, 0.0016963387995975337, 0.00154833456652808
3, 0.0016972876110547533, 0.0017424474559695616, 0.0016087489102750996, 0.0016986665295664258, 0.0011
536360695522405, 0.0018794653352549448, 0.0010783879448454587, 0.002875516684373083, 0.00168106372373
09079, 0.0008078888182726485, 0.003867213156013646, 0.0029809015043620614, 0.0014080559136337367, 0.0
027229772899439184, 0.0011039183061605858, 0.001565670878615533, 0.0023492615907702557, 0.00100866005
87766567, 0.0007036976076967833, 0.001568471124183834, 0.0008442480635584103, 0.0005740174761917512,
0.0017900008085755988, 0.0016796113067108621, 0.0020664539725860982, 0.004146070555155448, 0.00105212
0311935784, 0.0029244517019989262, 0.002825496989869198, 0.0014031427602837314, 0.0012318260799948338
, 0.0015404207890429926]
```