Market Risk – Project report

Baptiste DUFOUR, Sacha CUIROT

Contents

Data cleaning and exploration

Before responding to any questions, an essential aspect of the project entailed analysing the dataset and creating fundamental functions that would be useful throughout the project.

We decided to create three functions, one that sorts an array, one that calculates the mean of an array and the last one that calculates the standard deviation of an array of returns.

Basic Functions

```
4]: def trie(tab):  #Function that sorts an array
        tab_trie=tab.copy()
        n = len(tab)
        for i in range(n - 1):
            for j in range(n - i - 1):
                if tab_trie[j] > tab_trie[j + 1]:
                    tab_trie[j], tab_trie[j + 1] = tab_trie[j + 1], tab_trie[j]
        return tab_trie
```

```
5]: def mean(tab): #Fonction that calculates the mean of an array
        n=len(tab)
        return 1/n*sum(tab)
```

```
6]: def sd(tab): #Fonction that calculates the standard deviation of an array
        n=len(tab)
        return np.sqrt((1/(n-1))*sum((tab-mean(tab))**2))
```

An important dataset for our analysis was the dataset of the Natixis stock daily prices from 2015 to 2018, as it played a crucial role in questions A, B, and C.

So, we needed to be sure it was a clean dataset by checking if there were null values or changing the format of the columns.

Data Cleaning

```
]: # Display the shape of the DataFrame
   print("Shape of natixis_Data:", dataNX.shape)

   # Display the header of the DataFrame
   print("\nHeader of natixis_Data:")
   print(dataNX.head())

   print("\nColumn names:")
   print(dataNX.columns)
```

```
: dataNX.columns = ['Date', 'Price'] #We change the name of the columns
  dataNX['Date'] = pd.to_datetime(dataNX['Date'], format='%d/%m/%Y') #We change the format of the Date
  dataNX['Price'] = dataNX['Price'].str.replace(',', '.').astype(float) #We convert the prices in float
  dataNX.head()
```

	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

```
missing_values =dataNX.isnull().sum()
# Display the number of missing values for each column
print("Missing values by column:")
print(missing_values)
```

```
Missing values by column:
Date     0
Price    0
dtype: int64
```

After completing this step, we were able to commence addressing the questions.

Question A (Ex2, part of Q1 and of Q2 of TD1)

a – From the time series of the daily prices of the stock Natixis between January 2015 and December 2016, provided with TD1, estimate a historical VaR on price returns at a one-day horizon for a given probability level (this probability is a parameter which must be changed easily).

You must base your VaR on a non-parametric distribution, biweight Kernel, that is

$$K(u) = \frac{15}{16}(1-u^2)^2 \mathbb{I}_{|u|\leq 1}).$$

The first step was to create an array of the returns between January 2015 and December 2016 and then to calculate the biweight Kernel VaR)

```
dataNX['returns'] = dataNX['Price'].pct_change().fillna(0) #We add a return column to the dataset
```

```
#Array of returns between January 2015 and December 2016
nx_r1=dataNX[(dataNX['Date'] >= '2015-01-01') & (dataNX['Date'] < '2017-01-01')]['returns'].iloc[1:].values
```

To calculate the VaR based on a non-parametric distribution, we needed to use the kernel density function with the appropriate K function. In our case, we used the biweight Kernel function.

A **kernel density**, where the Kernel K, a density with cdf $\mathcal{K}$ can be, for example, $K(x) = \frac{1}{\sqrt{2\pi}} \exp(-x^2/2)$, with a smoothing parameter $h > 0$ indicating the smoothness/robustness:

$$\hat{f}(x) = \frac{1}{nh}\sum_{i=1}^{n} K\left(\frac{x - X_i}{h}\right) \text{ and } \hat{F}(x) = \frac{1}{n}\sum_{i=1}^{n} \mathcal{K}\left(\frac{x - X_i}{h}\right).$$

Drawback: fixed-size resolution (h), thus spurious effects for tails and smoothing of essential elements in the main part of the distribution.

To do this, we have created four different python functions:

```python
def fkernel(x, returns, h):
    n = len(returns)
    f_x = (1/(n*h)) * sum(K((x - val)/h) for val in returns)
    return f_x
```

```python
def Fkernel(x,returns,h):
    n=len(returns)
    f_x=(1/n)*sum(K_cdf((x - val)/h) for val in returns)
    return f_x
```

```python
def K(x):
    if -1 <= x <= 1:
        return (15/16) * ((1 - x**2)**2)
    else:
        return 0
```

```python
def K_cdf(x):
    if -1 <= x <= 1:
        return 1/2+x*(15/16-5*x**2/8+3*x**4/16)
    else:
        return 0
```

The fkernel function computes the fkernel value for each x, and the K(x) function corresponds to the biweight function.

The Fkernel function represents the cumulative distribution function (CDF) of the kernel density, while K_cdf is the cumulative distribution function of the biweight function.

These functions will be employed in the calculation of the non-parametric Value at Risk (VaR).

Now, let's describe our non-parametric VaR:

```python
def non_parametric_distribution(returns,alpha):
    h=pow((4*pow(sd(returns),5))/(3*len(returns)),1/5) #Thumb formula to calculate h
    returns_trie=trie(returns)
    #Calculation of the kernel density
    x_values = np.linspace(returns_trie.min(), returns_trie.max(),1000)
    f_kernel=np.array([fkernel(x,returns,h) for x in x_values])

    plt.hist(returns, bins=40, density=True,edgecolor='#333333',alpha=0.7, color='#FA8072')
    plt.plot(x_values, f_kernel, color='green', label='Kernel density', linewidth=2)
    plt.legend()
    plt.xlabel('Daily returns')
    plt.title("Distribution of the returns with the kernel density")
    plt.show()

    #Calculation of the cumulative distribution of the kernel density
    f_kernel_trie=trie(f_kernel)

    cumulative_kernel = np.cumsum(f_kernel) / np.sum(f_kernel_trie)
    #cumulative_kernel=np.array([Fkernel(x,returns,h) for x in x_values])
    cumulative_kernel=trie(cumulative_kernel)
    plt.plot(cumulative_kernel)
    plt.title("Cumulative Distribution of the kernel density")
    plt.xlabel("x")
    plt.ylabel("F(x)")
    plt.show()

    x_value = 0
    for i in range(len(cumulative_kernel)): #We take The smallest value of x such that F(x) >= alpha
        if cumulative_kernel[i] >= (1 - alpha):
            x_value = i
            break
    return x_values[i]
```

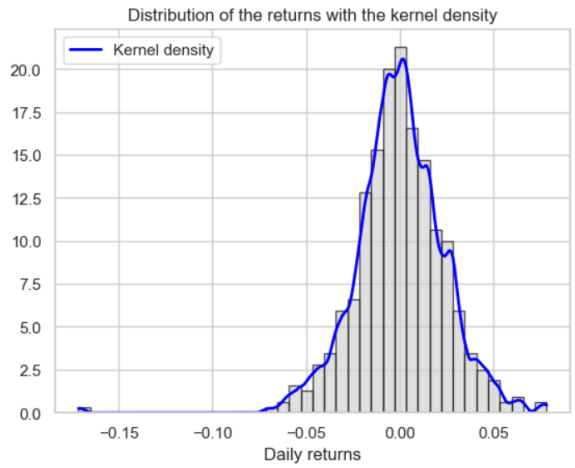
First of all, we calculate the value of h using the thumb formula with n the len of the returns and σ the standard deviation of the returns.

$$h = \left(\frac{4\,\sigma^5}{3n}\right)^{1/5}$$

```python
h=pow((4*pow(sd(returns),5))/(3*len(returns)),1/5) #Thumb formula to calculate h
```

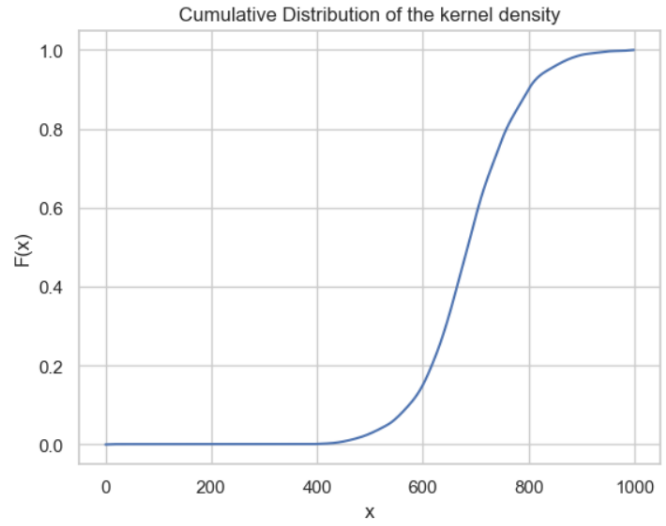
Then we create an array of 1000 values between the min of the returns the max of the returns and we applicate the kernel density function for each value of x.

```python
#Calculation of the kernel density
x_values = np.linspace(returns_trie.min(), returns_trie.max(),1000)
f_kernel=np.array([fkernel(x,returns,h) for x in x_values])
```



After we calculate the cumulative distribution function using cumsum and not FKernel because it provides more accurate results. We observe that the CDF reaches 1 after the maximum of the returns. This outcome differs from what we would obtain if we had used the FKernel and K_cdf functions.

```
cumulative_kernel = np.cumsum(f_kernel) / np.sum(f_kernel_trie)
```



To finish we find the smallest value of x such that the CDF >=1-alpha after this x using the definition of the VaR itself. It's the method of the empirical VaR.

$$VaR_\alpha(Y) = F_Y^{-1}(\alpha).$$

```
x_value = 0
for i in range(len(cumulative_kernel)): #We take The smallest value of x such that F(x) >= alpha
    if cumulative_kernel[i] >= (1 - alpha):
        x_value = i
        break
return x_values[i]
```

Thanks to this function we were able to calculate the non-parametric VaR using biweight function for different values of alpha. For alpha =0.95, we find a non-parametric VaR of-0.03782173385291748.

b – Which proportion of price returns between January 2017 and December 2018 exceed the VaR threshold defined in the previous question? Do you validate the choice of this non-parametric VaR?

To find the proportion of price returns that exceed the VaR, we have created a test_VaR function that returns the proportion of returns that exceed the VaR.

```
def test_VaR(returns,var):
    n=len(returns)
    n_inf=len(returns[returns<var])
    proportion=(n_inf/n)*100
    return proportion
```

In sample, we have found that 5.078125 % of price returns between January 2015 and December 2016 exceed the Non-Parametric VaR threshold for alpha = 0.95.

For alpha =0.9, 9.9609375 % of price returns between January 2015 and December 2016 exceed the Non-Parametric VaR threshold.

Out of sample, for the daily returns between January 2017 and December 2018, we found that 1.5717092337917484 % of price returns between January 2017 and December 2018 exceed the Non-Parametric VaR threshold for alpha = 0.95.

This can be clarified by noting that the VaR was computed based on historical data, and the returns observed between 2017 and 2018 exhibit a different distribution compared to those between 2015 and 2016. Specifically, there is a reduced occurrence of extreme losses in the 2017-2018 period as opposed to the 2015-2016 timeframe.

Question B (Ex2, Q5 of TD2)

Calculate the expected shortfall for the VaR calculated in question A. How is the result, compared to the VaR?

The Expected shortfall (ES) or average VaR (AVaR) represents the mean of the returns that exceed the VaR threshold.

the **average VaR** (AVaR), or **expected shortfall** (ES), or **conditional VaR**:
$ES_\alpha(X) = \frac{1}{1-\alpha}\int_\alpha^1 VaR_r(X)dr = \frac{1}{1-\alpha}\int_\alpha^1 F_{-X}^{-1}(r)dr.$

The ES is always greater than the VaR because it's a mean of values that exceed the VaR.

To calculate the ES, we have created a function according to the formula:

```
def Expected_Shortfall(returns, VaR):
    returns_sorted=trie(returns)
    return mean(returns_sorted[returns_sorted<=VaR]) #mean of the returns that exceed the VaR threshold
```

In this formula, we use the function mean that we have created before.

For the non-parametric VaR, we found an ES of -0.05336181815045533.

Question C (Ex2, Q1 and Q2 of TD3)

With the dataset provided for TD1 on Natixis prices, first calculate daily returns. You will then analyse these returns using a specific method in the field of the EVT.

a – Estimate the GEV parameters for the two tails of the distribution of returns, using the estimator of Pickands. What can you conclude about the nature of the extreme gains and losses?

First, we separate our returns in two different datasets: the gains and the losses to study their nature.

```
gains=nx_returns[nx_returns>0]
losses=-nx_returns[nx_returns<0]
losses_sorted=trie(losses)
gains_sorted=trie(gains)
```

After, using the estimator of Pickands formula, we create our own function:

Theorem

Soit (X_n) une suite de variables aléatoires indépendantes identiquement distribuées, de fonction de répartition F appartenant au max-domaine d'attraction d'une loi GEV de paramètre $\xi \in \mathbb{R}$. Soit k une fonction de $\mathbb{N}$ dans $\mathbb{N}$. Si

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

et

$$\lim_{n\to\infty} \frac{k(n)}{n} = 0,$$

alors, **l'estimateur de Pickands**, *défini par :*

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

converge **en probabilité** *vers ξ.*
De plus, si

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

alors la convergence de $\xi^P_{k(n),n}$ vers ξ se fait **presque sûrement** *et non pas seulement en probabilité.*

```
def pickands_estimator(data):
    data_sorted=trie(data)
    n = len(data)
    k_n = k(n)
    X_1 = data_sorted[n-math.floor(k_n)]
    X_2 = data_sorted[n-2*math.floor(k_n)]
    X_4 = data_sorted[n-4*math.floor(k_n)]
    pickands = (1/np.log(2)) * np.log((X_1-X_2)/(X_2-X_4))
    return pickands
```

$X_{n-k(n)+1:n}$ is the statistic order defined by $\min\{X_1, ..., X_n\} = X_{1:n} \leq X_{2:n} \leq ... \leq X_{n-1:n} \leq X_{n:n} = \max\{X_1, ..., X_n\}$;

To perform this computation, we simply need to sort our returns and extract the value at index n-k(n). Since arrays in Python start at 0, we subtract 1. To ensure we get the correct index, we utilize the math.floor function for rounding down.

In our case, we use the log function for k(n) which verifies:

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

$$\lim_{n\to\infty} \frac{k(n)}{n} = 0,$$

```
def k(n):
    return np.log(n)
```

We find:

```
The pickand estimator for the losses of the nataxis returns is -0.5089715779341932
The pickand estimator for the gains of the nataxis returns is 0.5772338569463368
```

The Pickands estimator is positive for gains, indicating that gains follow a Fréchet distribution, which is characterized by heavy tails.

Conversely, the Pickands estimator for losses is negative, suggesting that losses follow a Weibull distribution, which is a bounded distribution.

:

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

Remark

Alors :

- *si $\xi > 0$, la GEV est de type Fréchet ;*
- *si $\xi = 0$, la GEV est de type Gumbel ;*
- *si $\xi < 0$, la GEV est de type Weibull.*

b – Calculate the value at risk based on EVT for various confidence levels, with the assumption of iid returns.

To calculate the VaR, we used the formula of the lesson:

Estimateur de Pickands

$$VaR(p) = \frac{\left(\frac{k}{n(1-p)}\right)^{\xi^P} - 1}{1 - 2^{-\xi^P}} (X_{n-k+1:n} - X_{n-2k+1:n}) + X_{n-k+1:n},$$

où ξ^P est l'estimateur de Pickands du paramètre de la GEV.

The implementation of this formula is:

```python
def var_pickands_evt(data,alpha):
    data_sorted=trie(data) #so we are sure that the data is sorted
    n=len(data)
    e_pickands=np.abs(pickands_estimator(data_sorted)) #We always take the positive value of the pickands estimator
    K=1
    k_n=k(n)
    X_1=data_sorted[n-math.floor(k_n)]
    X_2=data_sorted[n-2*math.floor(k_n)]

    num=((K/(n*(1-alpha)))**e_pickands) - 1
    den=1-2**(-e_pickands)

    return (num/den)*(X_1-X_2) + X_1
```

The absolute value is consistently applied to the Pickands estimator to ensure a positive value representing losses. The utilisation of a negative value for the Pickands estimator of losses renders the function ineffective.

Using this approach, we find:

```
GEV Pickands VaR for losses with alpha=0.95: 0.04293722770534206
GEV Pickands VaR for gains with alpha=0.95: 0.032155624020667226


5.009633911368015 % of losses exceed the Pickands VaR for alpha = 0.95

9.236947791164658 % of gains exceed the Pickands VaR for alpha = 0.95
```

Question D (Ex2, Q3 and Q4 of TD4)

a – Estimate all the parameters of the model of Almgren and Chriss. Is this model well specified?

Almgren and Chriss model is a permanent impact model used to establish an optimal liquidation framework.

The price obtained by the liquidation is:

$$\sum_{k=1}^{N} n_k \tilde{S}_k = XS_0 + \sum_{k=1}^{N} \left[\left(\sigma \sqrt{\tau} \varepsilon_k - \tau g\left(\frac{n_k}{\tau}\right) \right) x_k - n_k h\left(\frac{n_k}{\tau}\right) \right]$$

Where

X : total Amount liquidated

S_0 : inital market Price before the beginning of the liquidation

$X * S_0$: liquidation price of X if we liquidate at once

$n_1, .., n_n = N$ Slices of liquidation $\sum_{k=1}^{N} n_k = X$

$t_1 = \frac{1}{n}, \ldots, t_n = \frac{N}{N}$ liquidation time

$x_k = X - \sum_{j=1}^{K} n_j$ remaining volume to be liquidated at time k

$\tau = t_k - t_{k-1} = \frac{1}{24}$

$\varepsilon_k \sim \mathcal{N}(0, 1)$

$\sigma =$ market volatility

$v_k = \frac{n_k}{\tau} =$ liquidation speed of the slice n_k

g is the **permanent impact function** $g(v_k) = \gamma v_k$

$\sigma\sqrt{\tau}\varepsilon_k =$ market risk related to the market movements

$\tau g(\frac{n_k}{\tau}) =$ **permanent impact** related to the liquidation of the (K-1) previous slices

where $h(\frac{n_k}{\tau}) = \xi sgn(n_k) + \eta\frac{n_k}{\tau}$

$$\text{sgn}(n_k) = \begin{cases} +1 & \text{if } n_k \text{ is a purchase} \\ -1 & \text{if } n_k \text{ is a sale} \end{cases}$$

$\xi =$ Bid-Ask spread

Our goal is to establish all the parameters of the model. But before doing this, we are going to clean the dataset by deleting the rows with unknown volume and by changing the names of the columns of the dataset.

```
new_columns = {
    'transaction date (1=1day=24 hours)': 'Date',
    'bid-ask spread': 'bid_ask_spread',
    'volume of the transaction (if known)': 'Volume',
    'Sign of the transaction': 'Sign',
    'Price (before transaction)':'Price'
}
# Rename the columns
data.rename(columns=new_columns, inplace=True)
data['Volume'].fillna(0,inplace=True)
data=data[data['Volume']>0]
```

Now, we can start to estimate all the parameters.

We start by calculating the returns and by calculating the mean and the standard deviation of the returns.

```
data['returns'] = ((data['Price'] / data['Price'].shift(1)) - 1) * np.sqrt(1 / (data['Date'] - data['Date'].shift(1)))
```

```
mu=mean(returns)
sigma=sd(returns)
```

The first step is the estimation of γ.

$g(v_k) = \gamma * v_k$ with $v_k = \frac{n_k}{\tau}$

Permanent impact = $g(v_k) = P_{t+2} - P_t$

To find γ, we first calculate the permanent impact values and the liquidation speed to be able to perform a regression analysis to determine γ.

```
permanent_impact=[]
time=[]
for i in range(2,len(data)):
    permanent_impact.append(data['Price'].iloc[i]-data['Price'].iloc[i-2])
    time.append(data['Date'].iloc[i]-data['Date'].iloc[i-2])
permanent_impact=np.array(permanent_impact)
time=np.array(time)
```

```
nk=np.array(data['Volume'])
nk=nk[:-2]
vk=nk/time
```

Now we perform a regression analysis to determine γ. We use the linearRegression Package.

```
vk_reshaped = vk.reshape(-1, 1)
model_gamma = LinearRegression()
model_gamma.fit(vk_reshaped, permanent_impact)

X_new = np.array([min(vk), max(vk)]).reshape(-1, 1)
y_pred = model_gamma.predict(X_new)

# Coefficients
slope_model_gamma = model_gamma.coef_[0]
gamma=slope_model_gamma
intercept_model_gamma = model_gamma.intercept_
print(f"Estimated gamma : {slope_model_gamma}")
```

```
Estimated gamma : -1.1010675029136836e-08
```

The second step is the estimation of η.

$h(\frac{nk}{\tau}) = \xi sgn(n_k) + \eta(\frac{n_k}{\tau})$

$$\text{sgn}(n_k) = \begin{cases} +1 & \text{if } n_k \text{ is a purchase} \\ -1 & \text{if } n_k \text{ is a sale} \end{cases}$$

ξ = Bid-Ask spread

$h(\frac{n_k}{\tau}) = P_{t+1} - P_{t+2}$ = transient impact

We carry a linear regression of $h(\frac{n_k}{\tau}) - \xi sgn(n_k)$ in order to estimate η :

```
t=[]
Xi=np.array(data['bid_ask_spread']/2) #Bid Ask spread/2
sgn=np.array(data['Sign']) #Sign of the transaction
transient_impact=[]
for i in range(2,len(data)):
    t.append(data['Date'].iloc[i]-data['Date'].iloc[i-1])
    transient_impact.append(data['Price'].iloc[i-1]-data['Price'].iloc[i])  #Pt+1-Pt+2

t=np.array(t)
transient_impact=np.array(transient_impact)
Xi=Xi[:-2]
sgn=sgn[:-2]
Y=transient_impact-Xi*sgn
vk=nk/t
```

Xi corresponds to the bid-ask spread divided by 2 and sgn to the sign of the transaction.

As we did to estimate γ, we carry a linear Regression between the liquidation speed and the transient impact minus the bid-ask spread times the sign of the transaction.

```
vk_reshaped =vk.reshape(-1, 1)
model_nu = LinearRegression()
model_nu.fit(vk_reshaped, Y)

X_new = np.array([min(vk), max(vk)]).reshape(-1, 1)
y_pred = model_nu.predict(X_new)

# Coefficients
slope = model_nu.coef_[0]
nu=slope
intercept = model_nu.intercept_

print(f"Estimated η : {slope}")
```

```
Estimated η : -1.4168633631897522e-09
```

b – In the framework of Almgren and Chriss, what is your liquidation strategy (we recall that you can only make transactions once every hour)

Now that we have established all the parameters, we can start to establish the optimal liquidation strategy for different values of lambda (risk aversion).

We use the formula:

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

Where xk represents the remaining amount of money to liquidate at time k.

K is obtained by:

$$K \overset{\tau \to 0}{\sim} \sqrt{\frac{\lambda \sigma^2}{\eta}} + \mathcal{O}(\tau).$$

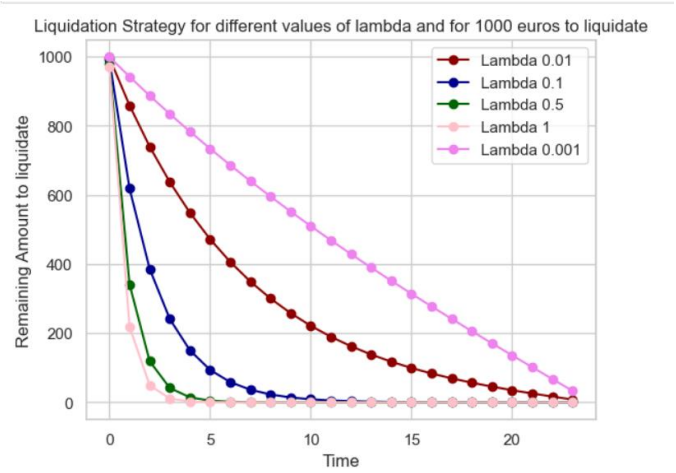
We implemented the formula in python:

```python
def xk_values(sigma, mu,l,X,T,to):
    xk = np.zeros(24)
    K = np.sqrt(l * (sigma**2) / abs(mu))   #Calculate K
    for i in range(24):
        xk[i] = (np.sinh(K*(T- (i + 1/2*to)))/np.sinh(K*T))*X
    return xk
```

X represents the amount of money to liquidate.

We recall that we can only make one transaction per hour, so T equals to 24 and τ equals to 1/24.

The results are:



The results appear coherent as a higher level of risk aversion corresponds to a faster liquidation process.

Question E (Q4 of TD5)

As we did for the other datasets, the first step was the cleaning of the dataset. We calculated the mid-price of each exchange rate for each day.

```python
data_TD5= pd.read_excel("Dataset TD5.xlsx",header=2)
data_TD5.set_index('Date', inplace=True)
data_TD5.index = pd.to_datetime(data_TD5.index)


GBPEUR = data_TD5.iloc[:, 0:2]
GBPEUR['Mid Price'] = (GBPEUR['HIGH'] + GBPEUR['LOW']) / 2

SEKEUR = data_TD5.iloc[:, 4:6]
SEKEUR['Mid Price'] = (SEKEUR['HIGH.1'] + SEKEUR['LOW.1']) / 2

CADEUR = data_TD5.iloc[:, 8:11]
CADEUR['Mid Price'] = (CADEUR['HIGH.2'] + CADEUR['LOW.2']) / 2
```

a – Determine the correlation matrix at each scale using wavelets. Determine the volatility vector with a scaling thanks to the Hurst exponents. From the correlation matrix and the volatility vector, calculate the covariance matrix and conclude about the volatility of the portfolio

We divided this question in two parts. First, we determined the correlation matrix at each scale using wavelets and in the second parts, we determined the volatility with a scaling thanks to the Hust exponents.

We used the Haar wavelets for the first part and especially the Haar father and Haar son wavelets.

We recall that:

$\varphi(t) = \begin{cases} 1 \; if \; t \in [0,1] \\ 0 \; else \end{cases}$ for the Haar Father wavelet.

$\varphi_{j,k}(t) = 2^{\frac{j}{2}}\varphi(2^j t - k)$ for the Haar son wavelets.

We implemented the formulas in python.

```python
def haar_father_wavelet(t):
    return np.where((0 <= t) & (t <= 1), 1.0, 0.0)
```

```python
def haar_son_wavelet(j, k, t):
    return 2**(j/2) * haar_father_wavelet(2**j * t - k)
```

The first step of the calculation of the correlation matrix was the calculation of the geometrical returns of the portfolio. We decided to take the geometrical returns and not the log returns because we used the Haar father wavelet and not the Haar mother wavelet.

J represents the Time Scale that we decide to use. It can be changed.

```python
# Calculer les rendements géométriques pour chaque taux de change
GBPEUR['geo_returns'] = (GBPEUR['Mid Price'] - GBPEUR['Mid Price'].shift(freq=j)) / GBPEUR['Mid Price'].shift(freq=j)
SEKEUR['geo_returns'] = (SEKEUR['Mid Price'] - SEKEUR['Mid Price'].shift(freq=j)) / SEKEUR['Mid Price'].shift(freq=j)
CADEUR['geo_returns'] = (CADEUR['Mid Price'] - CADEUR['Mid Price'].shift(freq=j)) / CADEUR['Mid Price'].shift(freq=j)
```

Next, we computed the scaling coefficients cj,k.

$$c_{j_0,k} = \int_{\mathbb{R}} z(t)\phi_{j_0,k}(t).$$

```python
def scaling_coefficient(geo_return, j,k, dt=0.01):
    # Conversion of min and max in days
    min_date = geo_return.index.min()
    max_date = geo_return.index.max()

    # Calculate the difference in months
    min_date_days = (min_date - min_date).days
    max_date_days = (max_date - min_date).days

    t_values = np.arange(min_date_days, max_date_days, dt).astype(int)

    coefficients = geo_return.iloc[t_values] * haar_son_wavelet(j,k, t_values) * dt

    # Sum of the coefficients
    c_jk = np.sum(coefficients)

    return c_jk
```

In our case, we decided to use the Riemann integration to calculate the scaling coefficient.

Riemann integration approximates the integral of a function (represented by geo_return) by summing the products of function values and a specific wavelet function (haar_son_wavelet) over discrete time intervals.

$$\int_a^b f(x)dx \approx \sum_i f(xi) * \Delta xi$$

We also created a function that calculates the mean of the scaling coefficients to calculate the covariance later.

```
def scaling_coefficient_mean(geo_returns, j):
    # Calculation of T, the difference between the first and last day of the index
    T = geo_returns.index[-1] - geo_returns.index[0]
    T_days = T.days

    k_values = np.arange(1, int(T.days / j))

    scaling_coef_values = np.array([scaling_coefficient(geo_returns, j, k) for k in k_values])

    c_j = (j / T_days) * np.sum(scaling_coef_values)

    return c_j
```

After completing the previous steps, it was time to compute the correlation matrix. To achieve this, we needed to calculate the covariance, variance, and correlation of the scaled values of the returns. For creating the necessary functions, we employed the following formulas:

$$\hat{\sigma}^2 = \frac{1}{n-1}\sum_{t=1}^{n}(x_t - \hat{\mu})^2$$

$$\operatorname{Cov}[X,Y] = \mathbb{E}[(X - \mathbb{E}[X])(Y - \mathbb{E}[Y])] = \mathbb{E}[XY] - \mathbb{E}[X]\mathbb{E}[Y]$$

$$\operatorname{Corr}[X,Y] = \operatorname{Cov}[X,Y] / \sqrt{\operatorname{Var}[X]\operatorname{Var}[Y]}$$

```
def calculate_covariance(geo_returns1, geo_returns2, j):
    scaling_coef_mean_1 = scaling_coefficient_mean(geo_returns1, j)
    scaling_coef_mean_2 = scaling_coefficient_mean(geo_returns2, j)

    T_days = (geo_returns1.index[-1] - geo_returns1.index[0]).days
    k_values = np.arange(1, int(T_days / j))

    c_j1 = np.array([scaling_coefficient(geo_returns1, j, k) for k in k_values]) - scaling_coef_mean_1
    c_j2 = np.array([scaling_coefficient(geo_returns2, j, k) for k in k_values]) - scaling_coef_mean_2

    covariance = (j / T_days) * np.sum(c_j1 * c_j2)
    return covariance
```

```
def calculate_variance(geo_returns, j):
    scaling_coef_mean = scaling_coefficient_mean(geo_returns, j)
    T_days = (geo_returns.index[-1] - geo_returns.index[0]).days
    k_values = np.arange(1, int(T_days / j))

    c_j = np.array([scaling_coefficient(geo_returns, j, k) for k in k_values])

    variance = (j / T_days) * np.sum((c_j - scaling_coef_mean) ** 2)
    return variance
```

```
def calculate_correlation(geo_returns1, geo_returns2, j):
    covariance = calculate_covariance(geo_returns1, geo_returns2, j)
    variance1 = calculate_variance(geo_returns1, j)
    variance2 = calculate_variance(geo_returns2, j)

    correlation = covariance / (np.sqrt(variance1) * np.sqrt(variance2))
    return correlation
```

The final step involved calculating the correlation matrix, a matrix representing the correlations between various geo_returns, accomplished by applying the correlation function to the dataset.

```
correlation_matrix = pd.DataFrame(index=['GBPEUR', 'SEKEUR', 'CADEUR'], columns=['GBPEUR', 'SEKEUR', 'CADEUR'])

geo_returns_list = [
    GBPEUR['geo_returns'],
    SEKEUR['geo_returns'],
    CADEUR['geo_returns']
]

j = j.days

# Calculation of the correlation matrix
for i in range(len(geo_returns_list)):
    for l in range(len(geo_returns_list)):
        if i == l:
            correlation_matrix.iloc[i, l] = 1
        else:
            correlation_value = calculate_correlation(geo_returns_list[i], geo_returns_list[l], j)
            correlation_matrix.iloc[i, l] = correlation_value
```

The second step of the question was the determination of the volatility vector with a scaling thanks to the Hurst exponents.

To estimate the hurst exponent, we used the formula:

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

With

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

```
def estimate_hurst_exponent(mid_price, j):
  geo_returns_j = (mid_price - mid_price.shift(freq=j)) / mid_price.shift(freq=j)
  geo_returns_2j = (mid_price - mid_price.shift(freq= (2 * j))) / mid_price.shift(freq=(2 * j))

  geo_returns_j = geo_returns_j.dropna()
  geo_returns_2j = geo_returns_2j.dropna()

  T = geo_returns_j.index[-1] - geo_returns_j.index[0]
  T_days = T.days
  j_days=j.days

  N=T_days/j_days

  sum_M2 = np.sum([geo_returns_j[i]**2 for i in range(0, len(geo_returns_j), j_days)])
  M2 = sum_M2 / N

  sum_M2_prime = np.sum([geo_returns_2j[i]**2 for i in range(0, len(geo_returns_2j), 2*j_days)])
  M2_prime = sum_M2_prime / (N // 2)

  hurst_exponent = 0.5 * np.log2(M2_prime / M2)

  return hurst_exponent
```

For j=3 days, we found:

```
Hurst Exponent for GBPEUR: -0.7092193640565762
Hurst Exponent for GBPEUR: -1.1242259797404615
Hurst Exponent for GBPEUR: -1.2981687879027703
```

Now that we have done this, we can calculate the adjusted volatility for each return.

$$Vol(j) = (\frac{j}{256})^H Vol(yearly)$$

```
def adjusted_volatility(hurst_exponent, geo_returns,j):
    j_days=j.days
    ecart_type = geo_returns.std()

    adjusted_vol_j = (j_days / 256) ** hurst_exponent * (256) ** hurst_exponent * ecart_type

    return adjusted_vol_j
```

The last step is the calculation of the volatility of the portfolio.

$Vol(j) = \sqrt{w^T cov(j) w}$ with cov(j) the covariance matrix and $w = (\frac{1}{3}\ \frac{1}{3}\ \frac{1}{3})^T$

```
# Initialisation of the covariance matrix
covariance_matrix = pd.DataFrame(index=['GBPEUR', 'SEKEUR', 'CADEUR'], columns=['GBPEUR', 'SEKEUR', 'CADEUR'])

geo_returns_list = [
    GBPEUR['geo_returns'],
    SEKEUR['geo_returns'],
    CADEUR['geo_returns']
]

j = j.days

# Calculation of the correlation matrix
for i in range(len(geo_returns_list)):
    for l in range(len(geo_returns_list)):
        if i == l:
            covariance_matrix.iloc[i, l] = calculate_variance(geo_returns_list[l],j)
        else:
            covariance_value = calculate_covariance(geo_returns_list[i], geo_returns_list[l], j)
            covariance_matrix.iloc[i, l] = covariance_value

print(covariance_matrix)
```

```
w=np.array([1/3,1/3,1/3])
```

```
Vol_ptf=np.sqrt(w@covariance_matrix@w.T)
print(f"The volatility of the portfolio for j = {j} days is {Vol_ptf}")
```

```
The volatility of the portfolio for j = 3 days is 0.0055017614660233475
```

b – Determine the volatility directly from the series of prices of the portfolio and justify your methodological choices (for example with overlapping returns or not).

As we did for question a, we began by calculating the returns.

Once we had the returns, we created a new dataset with all the returns and added a new column representing the average of the returns.

```
ptf = pd.DataFrame(index=GBPEUR.index)

ptf['GBPEUR_geo_returns'] = GBPEUR['geo_returns']
ptf['SEKEUR_geo_returns'] = SEKEUR['geo_returns']
ptf['CADEUR_geo_returns'] = CADEUR['geo_returns']

ptf['Average_geo_returns'] = ptf.mean(axis=1)
```

Now we can just compute the empirical variance with different value of j.

```
ptf['Squared_Average_geo_returns'] = ptf['Average_geo_returns'] ** 2
T = GBPEUR['geo_returns'].index[-1] - GBPEUR['geo_returns'].index[0]
N = T/j
M2_j = ptf['Squared_Average_geo_returns'].mean()

print(f"The empirical variance M^2(j) is: {M2_j}")

vol = np.sqrt(M2_j)

print(f"The volatility (vol) for j = {j} is: {vol}")
```

```
The empirical variance M^2(j) is: 8.832094161317969e-05
The volatility (vol) for j = 7 days 00:00:00 is: 0.0093979221965911
```