Market Risk - Project

Valentin SENAUX - Robin SIODLAK - IF5

In this document, we will describe the main steps and methods used to complete the project as well as the results. A more technical version of this document, is included in the .ipynb file.

List of all the results

Here is a list of the results in order.

$$\boxed{\text{VaR}_{95\%} = -0.040190534624647445.}$$

$$\boxed{\text{Proportion above treshold } = 0.9647058823529412 > 0.95.}$$

$$\boxed{\hat{\mu}_w = -0.02487813407099429, \quad \hat{\sigma}_w = 0.10928976875063803.}$$

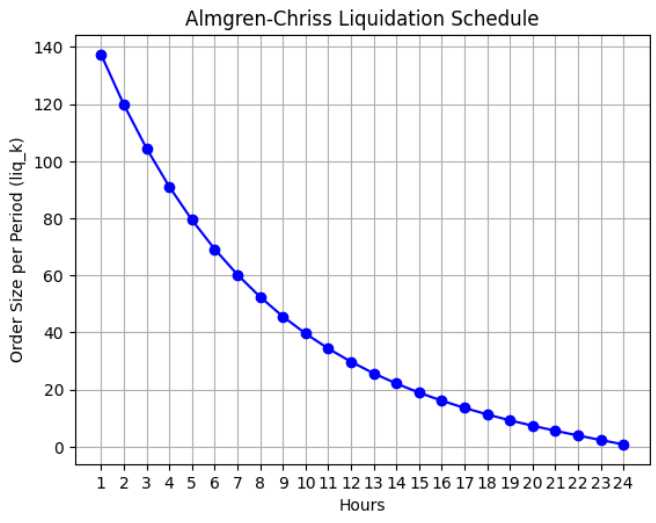
$$\boxed{\text{Call Premium 95\% } = 0.15382525956132653.}$$

$$\boxed{\xi^P_{losses} = -0.5263086875464077, \quad \xi^P_{gains} = 0.5772338569463368.}$$

$$\boxed{\hat{\gamma}_n = 2.1185612030126003 \cdot 10^{-5}.}$$

$$\boxed{\hat{\eta}_n = 2.9240658482954894 \cdot 10^{-6}.}$$

$$\boxed{R^2 = 0.84.}$$



Level	CADEUR-GBPEUR	CADEUR-SEKEUR	GBPEUR-SEKEUR
1	-0.226949	-0.135290	0.814780
2	-0.227190	-0.135506	0.814931
3	-0.227569	-0.135822	0.815184
4	-0.228495	-0.136924	0.815621
5	-0.229264	-0.137762	0.816206
6	-0.232206	-0.139938	0.817475
7	-0.241440	-0.151158	0.820864
8	-0.246501	-0.153196	0.825356
9	-0.274767	-0.183006	0.833338
10	-0.316037	-0.270196	0.854566
11	-0.493993	-0.398831	0.892320
12	-0.843762	-0.982279	0.929403

Table 1: Values for correlations of different levels of currency pairs.

$$\boxed{H_{\text{CADEUR}} \approx 0.66, \quad H_{\text{GBPEUR}} \approx 0.67, \quad H_{\text{SEKEUR}} \approx 0.65}$$

$$\boxed{\sigma_{Y,\text{CADEUR}} \approx 0.014, \quad \sigma_{Y,\text{GBPEUR}} \approx 0.018, \quad \sigma_{Y,\text{SEKEUR}} \approx 0.0089.}$$

1 Non-parametric VaR

In this section, we are asked to compute the non-parametric Value-at-Risk (VaR) of the Natixis stock based on kernel density estimation (kde). We use the data from 2015 to 2016 to compute the VaR, and then backtest it on the rest of the dataset.

1.1 Computation of the non-parametric VaR

Non-parametric VaR is defined using the non-parametric Kernel distribution estimator:

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

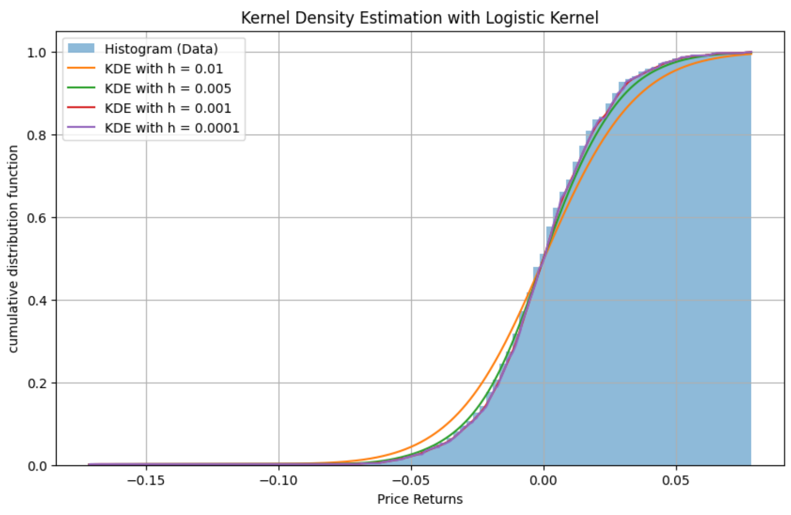
where:

- n : number of observations.
- h : bandwidth, smoothing parameter.
- K : the kernel *cumulative distribution function* (cdf). In our case, $K(x) := 1/(1+\exp(-x))$.

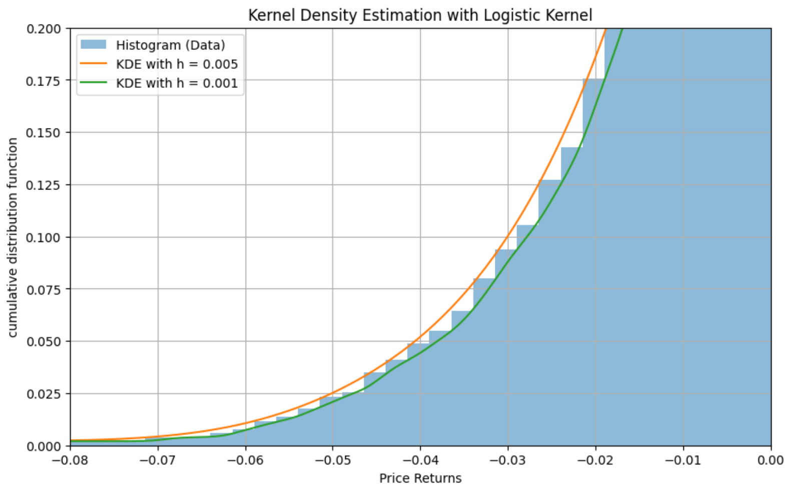
We chose to use the cdf instead of the density because it is easier to see the VaR when plotting the data. Here is the implementation of the kernel estimator:

```python
K = lambda x: 1/(1 + np.exp(-x)) # Logistic cdf.

def F_Kernel_Esti(data, h, x):
    n = len(data)
    return 1/n * sum([K((x - data[i])/h) for i in range(n)])

# Vectorized version of the function : Makes it quicker, the argument x is a vector.

def F_Kernel_Esti_Vec(data, h, X):
    n = len(data)
    return np.array([np.sum(1 / (1 + np.exp(-(x - data) / h))) / n for x in X])
```

The choice of bandwith is paramount in Kernel estimation. For the Gaussian Kernel, Silverman's rule of thumb gives the optimal parameter : $\hat{h}_{opt} = 1,06\hat{\sigma}n^{1/5}$. This doesn't apply in our case since we are using the logistic kernel. To chose the right bandwidth, we will use a graph :

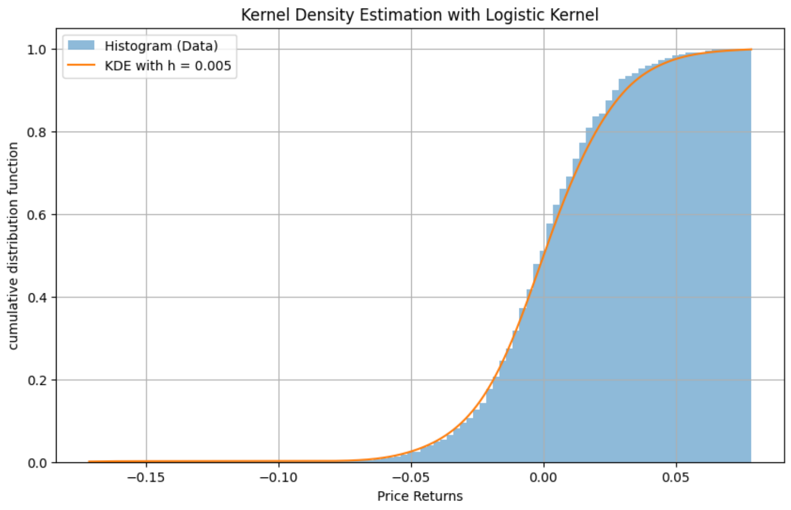


As we can see, the KDE with parameter $h = 0.01$ smooths the distribution too hard. On the other hand, the one with parameter $h = 0.0001$ over-learns the distribution. We can discard those values.
Next, we want to focus on the lower tail of the distribution. Since these are price returns, the VaR will be on the lower end of the graph. If we zoom in on that part of the graph :



Both of those parameters seem to estimate the distribution well. Since we are talking about risk, we better overestimate it. Hence we chose :

$$\boxed{h = 0.005.}$$



p.s : We do not really care about the underestimation of the upper tail of the distribution. We know that prices tend to have fatter lower tails and since the logistic distribution is symmetric, we cannot reproduce that characteristic.
Then, we proceed with the computation of the VaR based on the estimation we just did. To do so, we will simply go through the values we computed and stop whenever we hit the threshold :

```python
def Compute_VaR(data, alpha, h, grid_size = 500):
    x_values = np.linspace(data.min(), data.max(), grid_size)
    for x in x_values:
        if F_Kernel_Esti(data, h, x) >= 1 - alpha:
            return x
    return None  # If no value is found
```

This program gives the following VaR at 95% :

$$\boxed{\mathrm{VaR}_{95\%} = -0.0401905346246474445.}$$

Meaning that the VaR at 95% of the Natixis stock is a negative return of 4%.
In the html file, we also conducted tests based on easily identifiable values for the VaR, the program works.

1.2 Backtesting of the non-parametric VaR.

We proceed with the backtesting of the VaR, we need to check that 95% of the returns, at least, are above -4% on a different time period (year < 2017 vs year > 2016).
Here is the code to compute the exceedences :

```python
def validation(data, alpha):
    VaR = Compute_VaR(np.array(data), alpha, h)
    Count = 0
    for x in data:
        if x < VaR: # We trust our estimations of the distribution better in the lower tail.
            Count += 1
    return 1 - Count / len(data) # Proportion of values above the VaR.
```

On the backtest data, it returns :

$$\boxed{\text{Proportion above threshold } = 0.9647058823529412 > 0.95.}$$

Hence, we validate our computations of the VaR.

2 European Option Pricing

In this section, we want to compute the price of a Call Option, assuming the returns are iid of normal distribution, with exponential weighting of the data to put the emphasis on recent data.

The arithmetic variation of price is defined as:

$$\Delta S_t := S_t - S_{t-1}.$$

We assume that :

$$\Delta S_t \sim \mathcal{N}(\mu, \sigma^2),$$

Exponential weighting,

$$w_t = \lambda^{T-t},$$

with $\lambda \in (0, 1)$: exponential weighting parameter for the volatility. For our study we chose $\boxed{\lambda = 0.88}$ [1]. As with Silverman's rule of thumb, this value can highly vary based on the size of the windows considered and parameters. It will suffice in our study.

Then, the weighted parameters are given by :

$$\hat{\mu}_w = \frac{\sum_t w_t \Delta S_t}{\sum_t w_t}, \quad \hat{\sigma}_w^2 = \frac{\sum_t w_t \left(\Delta S_t - \hat{\mu}_w\right)^2}{\sum_t w_t}$$

The code used to compute those values is :

```python
mu_w = np.sum(weights * df_B['arithmetic_variations']) / np.sum(weights)
sigma_w = np.sqrt(np.sum(weights * (df_B['arithmetic_variations'] - mu_w) ** 2) / np.sum(weights)
    )
```

The values computed are :

$$\boxed{\hat{\mu}_w = -0.024878134070999429, \quad \hat{\sigma}_w = 0.10928976875063803.}$$

These values are coherent with the trend near the latest data. Let S_T be the price at the last date of 2018 (ie the last entry in the dataframe), the price at the next time is S_{T+1} and is given by :

$$\Delta S_{T+1} = \hat{\mu}_w + \hat{\sigma}_w \cdot B_{T+1}, \quad B_{T+1} \sim \mathcal{N}(0, 1),$$

with :

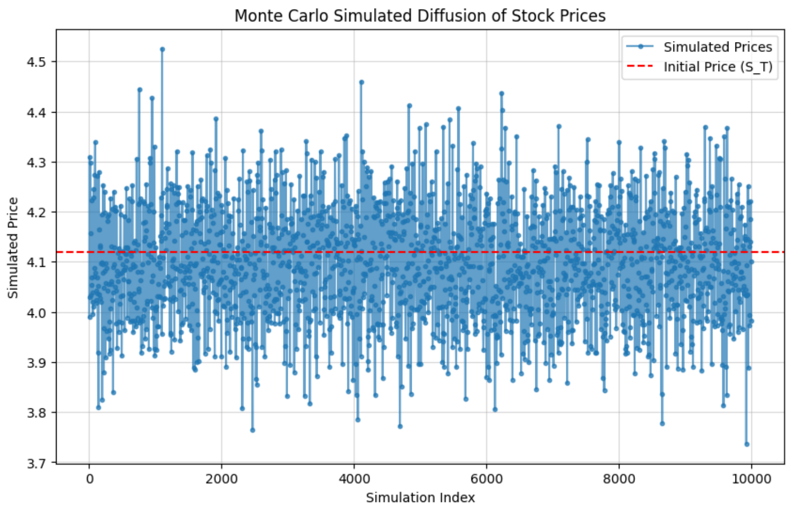
$$S_{T+1} = S_T + \Delta S_{T+1}.$$

Here is the code for the simulations :

```python
N = 1000  # Number simulations
S_T = df['price'].iloc[-1]  # Last price of 2018
B_T1 = np.random.normal(0, 1, N)  # Generate N standard normal samples

simulated_prices = S_T + mu_w + sigma_w * B_T1
```

We chose $N = 10000$: we are doing Monte Carlo simulations so it converges at a rate $n^{-1/2}$ and we do not want the computations to be too expansive either (The VaR is actually really close to the one with $N = 1000$). Here are the results of the simulations :

[1] Bernard. (2015). What should the value of lambda be in the exponentially weighted moving average volatility model?. Applied Economics. 47. 853-860. 10.1080/00036846.2014.982853.



Which is coherent with our estimated parameters. The price of a European Call Option is given, according to the Black-Scholes-Merton formula, by :

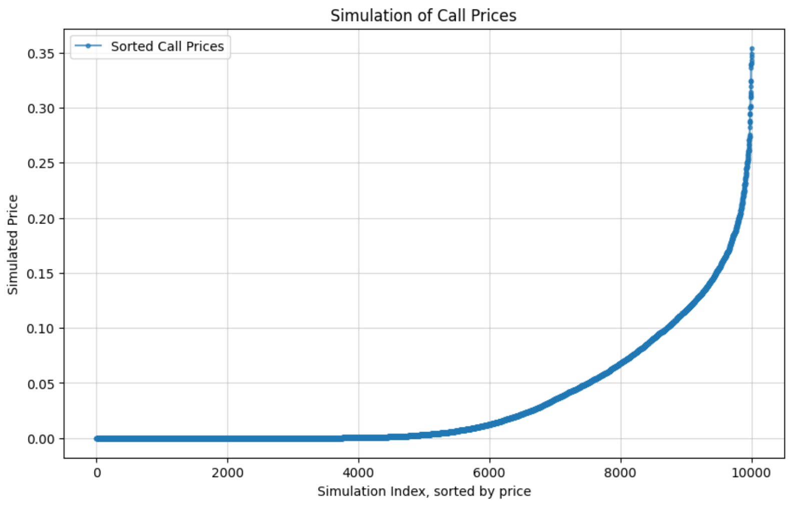
$$C = S_t \times \Phi(d_1) - Ke^{-rT} \times \Phi(d_2),$$

where:

- $K = S_T$: Strike price (at the money).
- $r = 0$: Risk-free rate.
- $T = 1/252$: Time to maturity (1 trading day).
- $\Phi(d)$: Cumulative normal distribution function.
- $d_1 = \frac{\ln(S_t/K)+\left(r+\frac{\sigma^2}{2}\right)T}{\sigma\sqrt{T}}$, $d_2 = d_1 - \sigma\sqrt{T}$.

```python
def black_scholes_call(S, K, T, r, sigma):
    d1 = (np.log(S / K) + (r + 0.5 * sigma**2) * T) / (sigma * np.sqrt(T))
    d2 = d1 - sigma * np.sqrt(T)
    return S * norm.cdf(d1) - K * np.exp(-r * T) * norm.cdf(d2)
```

Based on those simulations, here is a graph of the simulated call prices :



To price the call while accounting for 95% of the risk, we calculate the 95% quantile of the distribution of simulated call prices. With N the number of simulations and α the threshold, the VaR corresponds to the $N \times \alpha$ index of the sorted Call prices (we consider 95% of the lowest predicted prices at time $t+1$). The code simply is :

```python
VaR_Call = sorted(simulated_call_prices)[math.floor(alpha * N)]
```

It gives the following price for the Call option :

$$\boxed{\text{Call Premium 95\%} = 0.15382525956132653.}$$

This low price can be explained by the fact that the exponential weighting gave us a negative trend ($\mu < 0$) on the stock price and also the low variability of the same price at horizon one day. We also considered $r = 0$, which eliminates any growth in the discounted value of the strike price. This concludes this study.

3 Extreme Value Theory

In this section, we will compute the estimated value of the Max-Domain of attraction parameter as well as the extremal index of the same Natixis stock dataframe. The first step is the separation of positive and negative returns. The negative returns are also made positive (so we have a loss). To keep track of the dataframe names, here is the code :

```python
df_C = df["price_return"].copy()
df_C.dropna(inplace = True)
df_C_neg = abs(df_C[df_C < 0])
df_C_pos = df_C[df_C >= 0]
```

3.1 Max-Domain of attraction parameter.

Let (X_n) be a sequence of i.i.d. random variables, whose cdf F belongs to the max-domain of attraction of a GEV distribution of parameter $\xi \in \mathbb{R}$. Let k be a function $\mathbb{N} \to \mathbb{N}$. If

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

and

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

then, the Pickhands estimator of the GEV parameter ξ is given by :

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

where $X_{i:n}$ represent the order statistics of the sample $(X_1, X_2, \ldots, X_n)$, such that:

$$X_{1:n} \leq X_{2:n} \leq \cdots \leq X_{n:n}.$$

We define the following functions :

```python
k_1 = lambda n : n**(1/2)
k_2 = lambda n : n**(1/5)
k_3 = lambda n : np.log(n)
```

All of them satisfy the conditions for the Pickhands estimator to converge to ξ. The last one, is supposed to be the best (theoretically) since if :

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

then, $\xi^P_{k(n),n} \xrightarrow{a.s} \xi$ instead of just in probability.

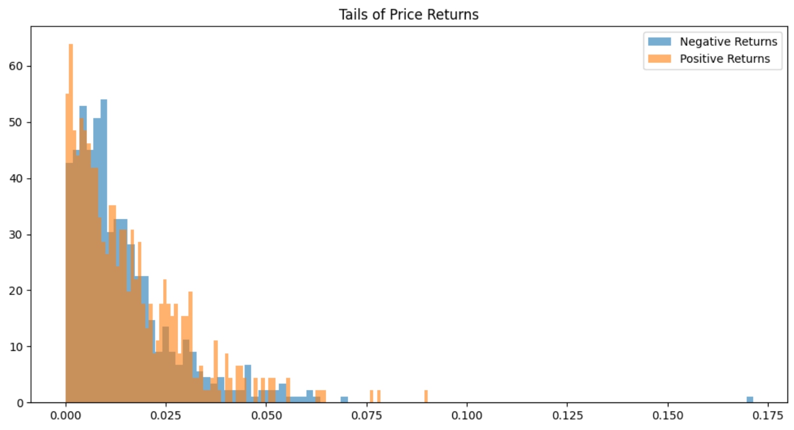
The code for the Pickhands estimator :

```python
def Pickands_estimator(data, k): # k is a function!
    n = len(data)
    data_sorted = np.sort(data)

    x_nk = data_sorted[math.floor(-k(n) + 1)]  # X_n-k(n)+1:n
    x_n2k = data_sorted[math.floor(-2*k(n) + 1)]  # X_n-2k(n)+1:n
    x_n4k = data_sorted[math.floor(-4*k(n) + 1)]  # X_n-4k(n)+1:n

    delta_num = x_nk - x_n2k
    delta_den = x_n2k - x_n4k
    delta = delta_num / delta_den

    return np.log(delta) / np.log(2)
```

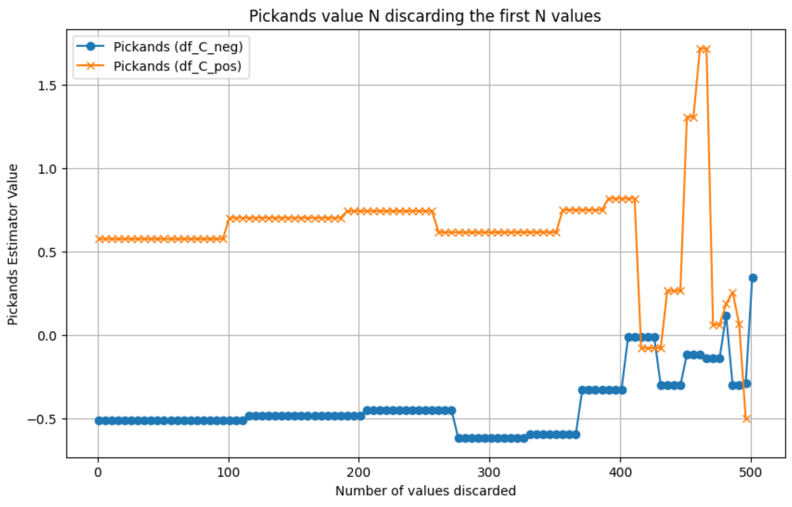
Using k_3, we obtain :

$$\boxed{\xi^P_{losses} = -0.5263086875464077, \quad \xi^P_{gains} = 0.5772338569463368.}$$

So $\xi < 0$ for the losses, the nature of the negative returns are of the Weibull kind. The tails of the negative returns are bounded. This isn't coherent with financial theory. Likewise for the gains. $\xi > 0$, so the nature of the positive returns are of the Fréchet kind. The tails of the positive returns are heavy. This also isn't coherent with financial theory.
Let's examine our data to confirm our results :



Our values of ξ seem coherent with the data. For the negative ones, probably because of the large hump at the beginning and the lack of very poor returns (only one, no clustering at the tail). Same for the positive returns, there is a hump near 0.025 making the distribution fatter, there are also more extreme returns.
As the following chart shows, no (reasonable) amount of discarded values (len(df_C_neg) = 519, len(df_C_pos) = 503)) around 0 (hence putting more weights in the tails), give the supposed values of ξ :



Hence we conclude that our empirical study is correct, even if it goes against financial theory (gains have heavy lower tails. Maybe Natixis has exceptional risk management ?).
Bonus : If we consider the returns iid, the VaR is given by the following formula :

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

for a certain k and n large. Here is the implementation :

```python
def VaR_pickhands(data, k, K, alpha):
    data_sorted = np.sort(data)
    xi = Pickands_estimator(data, K)
    print('xi = ', xi)
    n = len(data_sorted)

    X_1 = data_sorted[n - k] # X_{n - k + 1 : n}
    X_2 = data_sorted[n - 2 * k] # X_{n - 2k + 1 : n}

    coef = ((k / (n * (1 - alpha))) ** xi - 1) / (1 - 2 ** (-xi))

    return  coef * (X_1 - X_2) + X_1
```

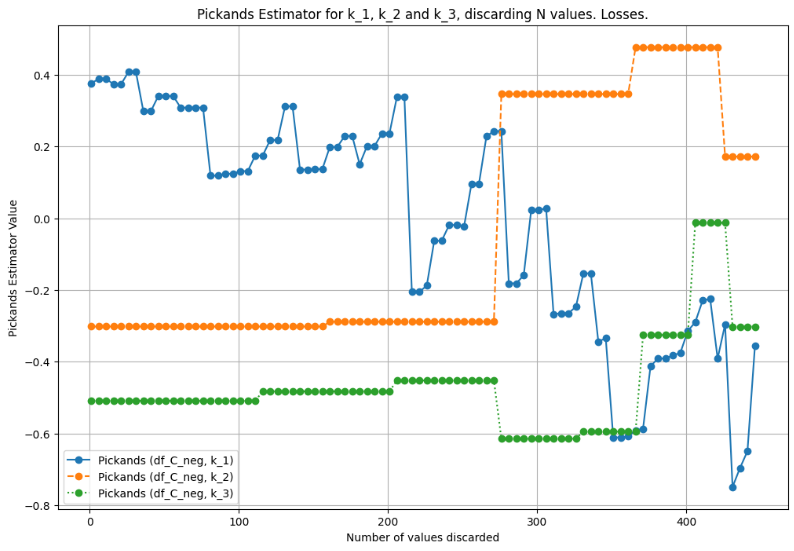
Using the Pickhands estimator, with k_3, we get :

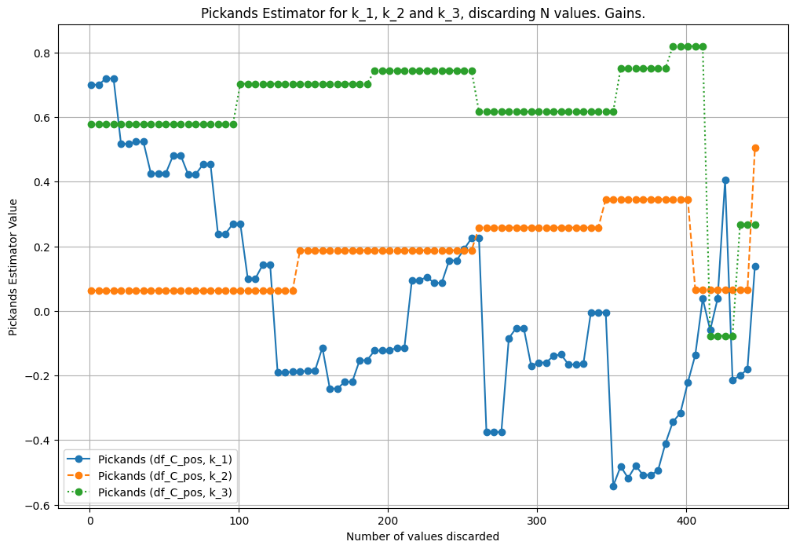
$$\boxed{\text{VaR Pickhands } = 0.04069861601611689.}$$

This confirms our results found in section 1, ie a maximum expected loss of 4% at 95% confidence level.

Remark : We are having extremely different results based on the picked function. We decided to pick ln because of the convergence but also because of those graphs :

$$k_1(n) := n^{1/2}, \quad k_2(n) := n^{1/5}, \quad k_3(n) := \ln(n).$$





Using ln gives way more stable values of ξ. In general (with our three functions...) the slower the divergence to $+\infty$, the more stable estimation of ξ we get.

3.2 Extremal Index

The extremal index is a quantity θ in $[0, 1]$. The lower θ is, the greater the dependance. In this subsection, we will compute θ using Blocks de-clustering (denoted $\hat{\theta}_n^B$ and Runs-declustering (denoted $\hat{\theta}_n^R$). The estimators are given by the following formulas :

$$\hat{\theta}_n^B(u; b) = \frac{\sum_{i=1}^{k} \mathbb{1}(M_{(i-1)b,ib} > u)}{\sum_{i=1}^{kb} \mathbb{1}(X_i > u)} \quad \text{and} \quad \hat{\theta}_n^R(u; r) = \frac{\sum_{i=1}^{n-r} \mathbb{1}(X_i > u, M_{i+1,i+r} \leq u)}{\sum_{i=1}^{n} \mathbb{1}(X_i > u)}.$$

Here is the code for Blocks de-clustering :

```python
def extremal_index_blocks(data, u, b):
    # Divide data into blocks, DO NOT SORT ! (order is important, we are testing independance of the extremes)
    n_blocks = len(data) // b
    blocks = [data[i * b: (i + 1) * b] for i in range(n_blocks)]

    # Count the number of blocks with at least one value over the threshold.
    cluster_count = sum(1 for block in blocks if np.max(block) > u)

    # Count the total number of observations over the threshold.
    over_thresh = np.sum(data > u)

    theta = cluster_count / over_thresh if over_thresh > 0 else 0
    return theta
```

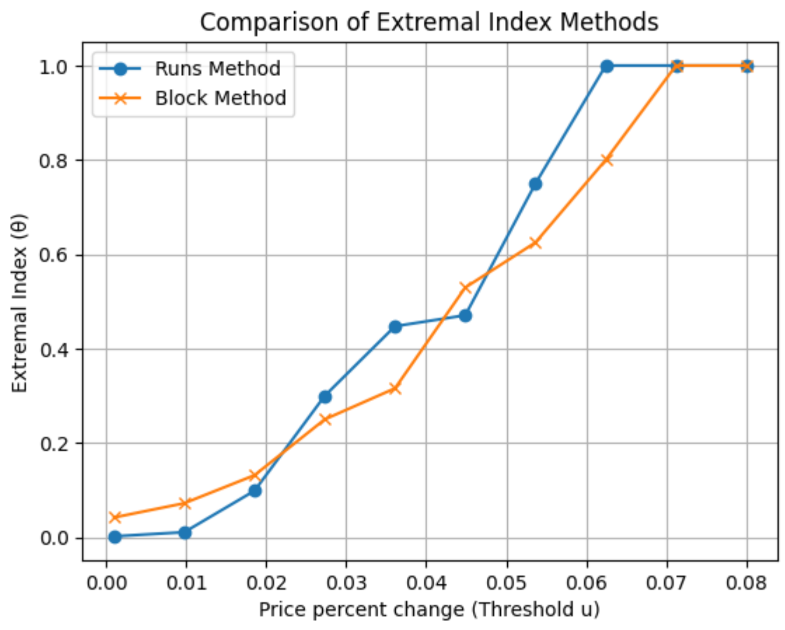
And for Runs-declustering :

```python
def extremal_index_runs(data, u, r):
    data = np.array(data)
    # Identify observations over the threshold.
    over_thresh = data > u
    n_over_thresh = over_thresh.sum()

    # De-cluster exceedances into clusters
    clusters = 0
    gap = 0  # Track gap between exceedances
    for i in range(len(data)):
        if over_thresh[i]:
            if gap >= r:  # Start of a new cluster after r gaps
                clusters += 1
            gap = 0  # Reset gap after an exceedance
        else:
            gap += 1  # Increment gap during non-exceedances

    theta = clusters / n_over_thresh if n_over_thresh > 0 else 0
    return theta
```

Using the whole Natixis dataset, we get the following values :

$$\hat{\theta}_n^B(u = 0.04, b = 10) = 0.7931034482758621, \quad \hat{\theta}_n^R(u = 0.04, r = 10) = 0.6206896551724138$$

It is normal that the values differ since the parameters b and r aren't equivalent. Here is a plot for the values of theta as a function of the threshold u, using $r = 15$ and $b = 50$:



This suggests that higher extremes are more independent compared to smaller extremes, which is expected. Both computation methods give the same result for the chosen parameters of runs and blocks.

4 Almgren & Chriss

In this section, we are trying to set up the optimal trading strategy to liquidate a large position in a single security. The data is from an unknown source and contains information such as time, bid-ask spread, volume, sign and price for each transaction (with a lot of NaNs for the volume). We begin by renaming columns to names that make more sense for the computations (to us at least...).

```python
df_D = pd.read_csv('../data/Dataset TD4.csv', sep = ";")
df_D.columns = ["t_k", "ba_spread", "n_k", "sgn", "S_prev"]
```

After conversions, the columns with their types are :

```
t_k          float64
ba_spread    float64
n_k          float64
sgn            int64
S_prev       float64
```

The first computation will be of τ the tick rate (time in between executions). We will simply take the average of the differences over all the dataset :

```python
tau = df_D["t_k"].diff().mean()
tau *= 24 # Convert to hours.
```

It gives a values of : $\boxed{\tau \approx 0.023803199999999997}$. Meaning a transaction every minute and a half approximately.

4.1 Estimation of the parameters

The price is supposed to follow the following model :

$$S_k = S_{k-1} + \sigma\sqrt{\tau}\varepsilon_k - \tau g\left(\frac{n_k}{\tau}\right), \quad (1)$$

With :

- S_k: Observed price at time k.
- S_{k-1} : Observed price at the previous time step.
- σ : Volatility.
- τ : Time step size.
- $\varepsilon_k \overset{iid}{\sim} \mathcal{N}(0,1)$: Noise.
- $g(v) := \gamma v$: Permanent price impact function.
- n_k : Trading volume at time k.

4.1.1 Estimation of gamma

From the price model :

$$\Delta S_k := S_k - S_{k-1} = \sigma\sqrt{\tau}\varepsilon_k - \gamma n_k \Leftrightarrow \frac{\Delta S_k}{n_k} = \frac{\sigma\sqrt{\tau}}{n_k}\varepsilon_k - \gamma.$$

By linearity of the gaussian distribution :

$$\frac{-\Delta S_k}{n_k} \sim \mathcal{N}\left(\gamma, \tau\left(\frac{\sigma}{n_k}\right)^2\right).$$

We define the first moment estimator of γ :

$$\hat{\gamma}_n = \frac{1}{n}\sum_{i=1}^{n}\frac{S_{k-1} - S_k}{n_k}$$

Since the ε_i are iid and of finite variance. By the law of large numbers (LLN), $\hat{\gamma}_n$ is a consistent estimator of γ :

$$\hat{\gamma}_n \xrightarrow[n\to+\infty]{d} \mathbb{E}[\hat{\gamma}_n] = \gamma.$$

The code to compute the estimation of γ is :

```python
# Create S_current by shifting S_prev backward
df_D["S_current"] = df_D["S_prev"].shift(-1)
df_D["delta_S_k"] = df_D["S_current"] - df_D["S_prev"]
df_D["delta_S_k_per_nk"] = df_D["delta_S_k"] / df_D["n_k"]
df_cleaned_gamma = df_D.dropna(subset = ["delta_S_k", "delta_S_k_per_nk"])
gamma_hat = - df_cleaned_gamma["delta_S_k_per_nk"].mean()
```

It gives :

$$\boxed{\hat{\gamma}_n = 2.1185612030126003 \cdot 10^{-5}.}$$

According to the original paper [2], this is coherent with a less liquid kind of security (10^{-5} to 10^{-4} range). We don't actually know the stock that the data comes from so we cannot confirm that.
With our value for γ, a buy of volume 1000 would impact the price in this way :

$$\text{Impact } = \gamma \times \text{ volume } = 2.1185612030126003 \times 1000 \approx 2.1\%.$$

The price would go up by 2.1%. Again, since we don't know the amount of shares traded, this result has to be put in perspective.

4.1.2 Estimation of eta

Then, we compute the volatility over a single time step :

```python
tau_seconds = df_D["t_k"].diff().mean() * 24 * 3600  # Convert tau to seconds
sigma = df_D["delta_S_k"].std() / np.sqrt(tau_seconds)
```

Giving :

$$\boxed{\sigma = 0.007959305124720978.}$$

We also define the column "xi" as half the bid-ask spread.

At time k, we observe a price that has been impacted by the transaction at $k-1$:

$$\overline{S}_k = S_k - h\left(\frac{n_k}{\tau}\right)$$

where:

- $h(x) = \xi \cdot \text{sgn}(n_k) + \eta\frac{n_k}{\tau}$: Temporary market impact function.
- ξ : Half the bid-ask spread.
- η : Coefficient.

Substituting, assuming $S_k \approx \overline{S}_k$:

$$\overline{S}_k - S_{k-1} = \sigma\sqrt{\tau}\varepsilon_k - \gamma n_k - \xi\text{sgn}(n_k) - \eta\frac{n_k}{\tau}$$

We rearrange the terms :

$$-\tau\frac{\Delta S_k + \gamma n_k + \xi\text{sgn}(n_k)}{n_k} = \eta - \frac{\sigma\tau^{3/2}\varepsilon_k}{n_k}, \qquad (2)$$

Since we have estimated the value of γ,

$$-\tau\frac{\Delta S_k + \hat{\gamma}_n n_k + \xi\text{sgn}(n_k)}{n_k} = -\tau\frac{\Delta S_k + (\hat{\gamma}_n - \gamma + \gamma)n_k + \xi\text{sgn}(n_k)}{n_k} = -\tau\frac{\Delta S_k + \gamma n_k + \xi\text{sgn}(n_k)}{n_k} - \tau(\hat{\gamma}_n - \gamma) =: X_k$$

Taking $\mathbb{E}(2)$:

$$\mathbb{E}\left[-\tau\frac{\Delta S_k + \gamma n_k + \xi\text{sgn}(n_k)}{n_k}\right] = \mathbb{E}\left[\eta - \frac{\sigma\tau^{3/2}\varepsilon_k}{n_k}\right] = \eta - 0 = \eta, \quad \text{since } \frac{\sigma\tau^{3/2}\varepsilon_k}{n_k} \sim \mathcal{N}(0, m > 0).$$

By the LLN,

$$\hat{\eta}_n := \frac{1}{n}\sum_{i=1}^{n} X_k \xrightarrow[n\to+\infty]{d} \mathbb{E}\left[-\tau\frac{\Delta S_k + \gamma n_k + \xi\text{sgn}(n_k)}{n_k}\right] + \tau(\gamma - \gamma) = \eta,$$

since $\hat{\gamma}_n$ is a consistent estimator of γ. Hence $\hat{\eta}_n$ is also a consistent estimator of η. Then, to compute η, we shift the values of the dataframe for them to match and apply the formula above :

```python
df_D_eta = df_D.copy()
df_D_eta['shifted_n'] = df_D_eta['n_k'].shift(1)
df_D_eta['shifted_sgn'] = df_D_eta['sgn'].shift(1)
df_D_eta['shifted_xi'] = df_D_eta['xi'].shift(1)
df_D_eta['tmp_for_eta'] = tau * (df_D_eta['delta_S_k'] + gamma_hat * df_D_eta['shifted_n'] +
    df_D_eta['shifted_xi'] * df_D_eta['shifted_sgn']) / df_D_eta['shifted_n']
eta_hat = -df_D_eta['tmp_for_eta'].mean()
```

[2] Almgren, Robert, and Neil Chriss. "Optimal execution of portfolio transactions." Journal of Risk 3 (2001): 5-40.

It gives a value for η of :

$$\boxed{\hat{\eta}_n = 2.9240658482954894 \cdot 10^{-6}.}$$

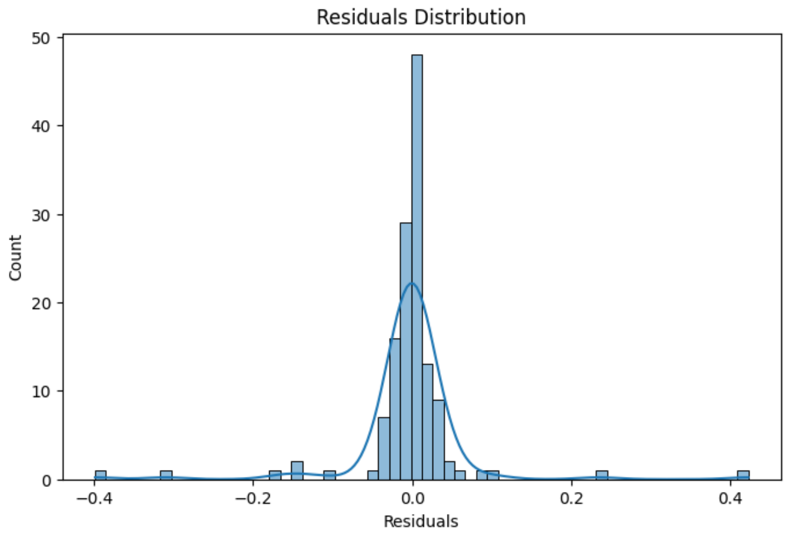
We now check that the model is well specified. We compute the residuals given by the following formula (derived from the model we used ...):

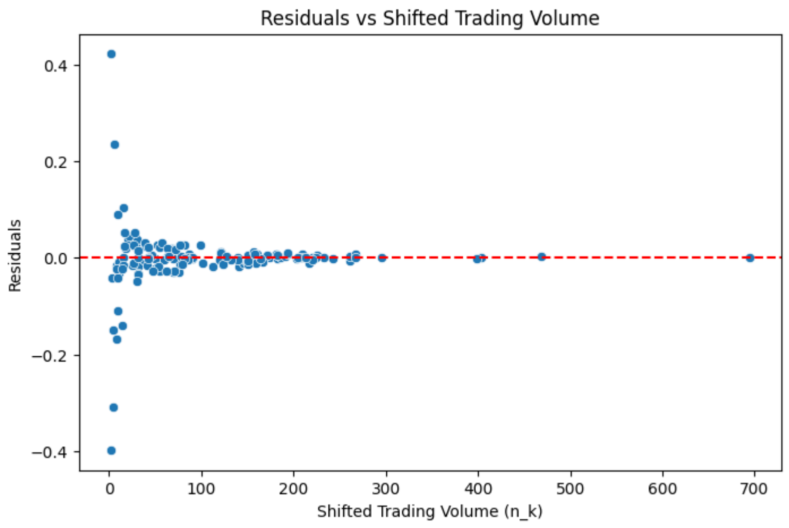
$$\varepsilon_k = \frac{\Delta S_k + \hat{\gamma} n_k + \xi \cdot \operatorname{sgn}(n_k)}{\sigma\sqrt{\tau} \cdot n_k} - \frac{\hat{\eta}}{\sigma\sqrt{\tau}} \sim \mathcal{N}(0, 0 < m < 1)$$

We supposed that the residuals follow a centered and reduced normal distribution. The criterion for our model to be correct is to have a mean of 0 and a low standard deviation. This means that the model isn't biased and that the linear approximation stays close to the observed data.

```python
df_D_eta['residuals'] = (df_D_eta['delta_S_k'] + gamma_hat * df_D_eta['shifted_n'] + df_D_eta['
    shifted_xi'] * df_D_eta['shifted_sgn']) / (sigma * np.sqrt(tau_seconds) * df_D_eta['shifted_n
    ']) - eta_hat / (sigma * np.sqrt(tau_seconds))
```

And plot the two graphs :





With :

$$\mu_{\varepsilon_k} = -0.003047865111213066, \text{ and } \sigma_{\varepsilon_k} = 0.06830616329835561.$$

The residuals are shaped like a Bell curve around 0 (approximately...), have a small deviation and they don't seem to have any particular distribution around 0 in the second graph. Now, we compute the R^2 coefficient :

$$\text{Sum Squares total} = \sum_{k=1}^{n} (\Delta S_k - \overline{\Delta S_k})^2.$$

$$\text{Sum Squares residual} = \sum_{k=1}^{n} (\Delta S_k - \Delta S_{k,\text{pred}})^2.$$

$$R^2 = 1 - \frac{\text{Sum Squares residual}}{\text{Sum Squares total}}.$$

The linear model is given by :

$$\Delta S_{k,\text{pred}} = -\left(\hat{\gamma}_n \cdot n_k + \xi \cdot \text{sgn}(n_k) + \hat{\eta}_n \cdot \frac{n_k}{\tau}\right),$$

The code to compute R^2 :

```
df_D_eta['predicted'] = - (gamma_hat * df_D_eta['shifted_n'] + df_D_eta['shifted_xi'] * df_D_eta[
    'shifted_sgn'] + eta_hat * df_D_eta['shifted_n'] / tau_seconds)
df_D_eta['residuals'] = df_D_eta['delta_S_k'] - df_D_eta['predicted']
Sum_Squares_residual = (df_D_eta['residuals'] ** 2).sum()
Sum_Squares_total = ((df_D_eta['delta_S_k'] - df_D_eta['delta_S_k'].mean()) ** 2).sum()
R_squared = 1 - (Sum_Squares_residual / Sum_Squares_total)
```

It gives a value of :

$$\boxed{R^2 = 0.84.}$$

Hence, we validate our model.

4.2 Liquidation strategy

Each tranche is computed such that :

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

Where :

- X : Total quantity to liquidate.
- $T = 1$ day : Time to do so.
- k : Trading period, one tick.
- $K \approx \sqrt{\frac{\lambda\sigma^2}{\eta}}$ with λ the risk aversion.

If we rewrite the equation like in the paper,

$$x_j = \frac{\sinh(K(T - (j - 1/2)\tau))}{\sinh(KT)} X.$$

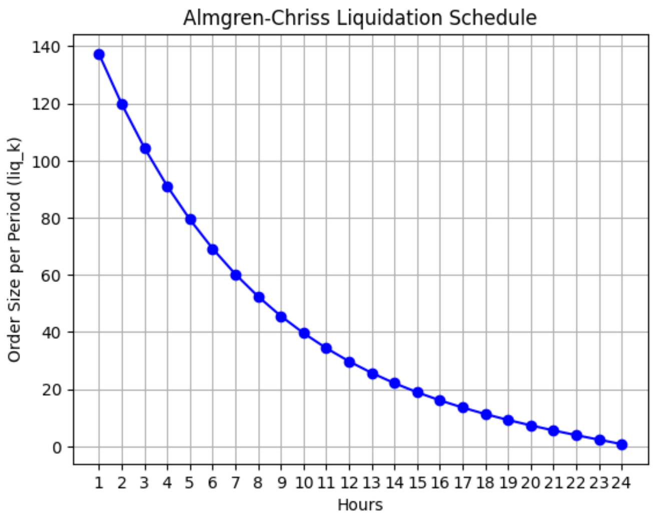
where :

- j : The index of the trading interval, ranging from 1 to N, where N is the total number of intervals.
- $\tau = \frac{T}{N}$: The duration of each trading interval.

Using the second formula with the index :

```
X = 1000
T = 1
N = 24 # Trading periods.
tau_liq = T / N
lambda__ = 0.88
K = np.sqrt(sigma**2 * lambda__/ eta_hat)

x_N = np.sinh(K * (T - (np.arange(1, N + 1) - 0.5) * tau_liq)) / np.sinh(K * T) * X # Amount of
    shares remaining at each time step.
liq_N = x_N / np.sum(x_N) * X  # Amount sold at each time step.
```

Giving the graph :



And liquidation schedule (manually adjusted to be integers):

```
Hour 1: 138 shares
Hour 2: 120 shares
Hour 3: 104 shares
Hour 4: 91 shares
Hour 5: 79 shares
Hour 6: 69 shares
Hour 7: 60 shares
Hour 8: 52 shares
Hour 9: 46 shares
Hour 10: 40 shares
Hour 11: 34 shares
Hour 12: 30 shares
Hour 13: 26 shares
Hour 14: 22 shares
Hour 15: 19 shares
Hour 16: 16 shares
Hour 17: 14 shares
Hour 18: 11 shares
Hour 19: 9 shares
Hour 20: 7 shares
Hour 21: 6 shares
Hour 22: 4 shares
Hour 23: 2 shares
Hour 24: 1 shares
```

We've successfully computed the optimal liquidation strategy according to Almgren and Chriss. Having access to a dark pool would be better though.

5 Wavelets and Hurst exponent

In this section, we take interest in the wavelet transform and it's application to computing correlations. The datasets are three exchange rates, the Canadian dollar, Great British Pound and Swedish Krona against the Euro. We named each of the datasets : "df_CADEUR", "df_GBPEUR" and "df_SEKEUR". The column of the exchange rate is considered to be the average of the high and low price. In each dataset, it is named "HL_avg".

5.1 Haar transform for correlations

5.1.1 Haar transform implementation

We begin with a bit of theory. The Haar mother wavelet is given by :

$$\psi^{Haar} : t \in \mathbb{R} \mapsto \mathbb{1}_{[0,1/2)}(t) - \mathbf{1}_{[1/2,2)}(t).$$

And the scaling function :

$$\phi : t \mapsto \mathbb{1}_{[0,1]}(t).$$

We know that the scaling function satisfies the recursive relation :

$$\phi(x) = \sum_{k=0}^{2N-1} a_k \phi(2x-k)$$

In our case, there is one vanishing moment so $N = 1$, giving :

$$\phi(x) = a_0\phi(2x) + a_1\phi(2x-1)$$

Since the scaled components are othogonal :

$$\|\phi(x)\|_{L^2}^2 = \|a_0\phi(2x) + a_1\phi(2x-1)\|_{L^2}^2 = \|a_1\phi(2x)\|_{L^2}^2 + \|a_2\phi(2x-1)\|_{L^2}^2 = |a_0|^2\|\phi(2x)\|_{L^2}^2 + |a_1|^2\|\phi(2x-1)\|_{L^2}^2$$

Assuming $a_0 = a_1$, the normalization condition $\|\phi(x)\|_{L^2} = \|\phi(2x)\|_{L^2} = \|\phi(2x-1)\|_{L^2} = 1$, gives :

$$a_0 = a_1 = \frac{1}{\sqrt{2}}.$$

For the one level Haar transform :
Consider an observation $X = (x_1, \dots, x_n)$ where n is even. The approximation coefficients are given by :

$$a_{k-1} = \frac{x_{2k} + x_{2k+1}}{\sqrt{2}}$$

The detail coefficients are :

$$d_k = \frac{x_{2k} - x_{2k+1}}{\sqrt{2}}.$$

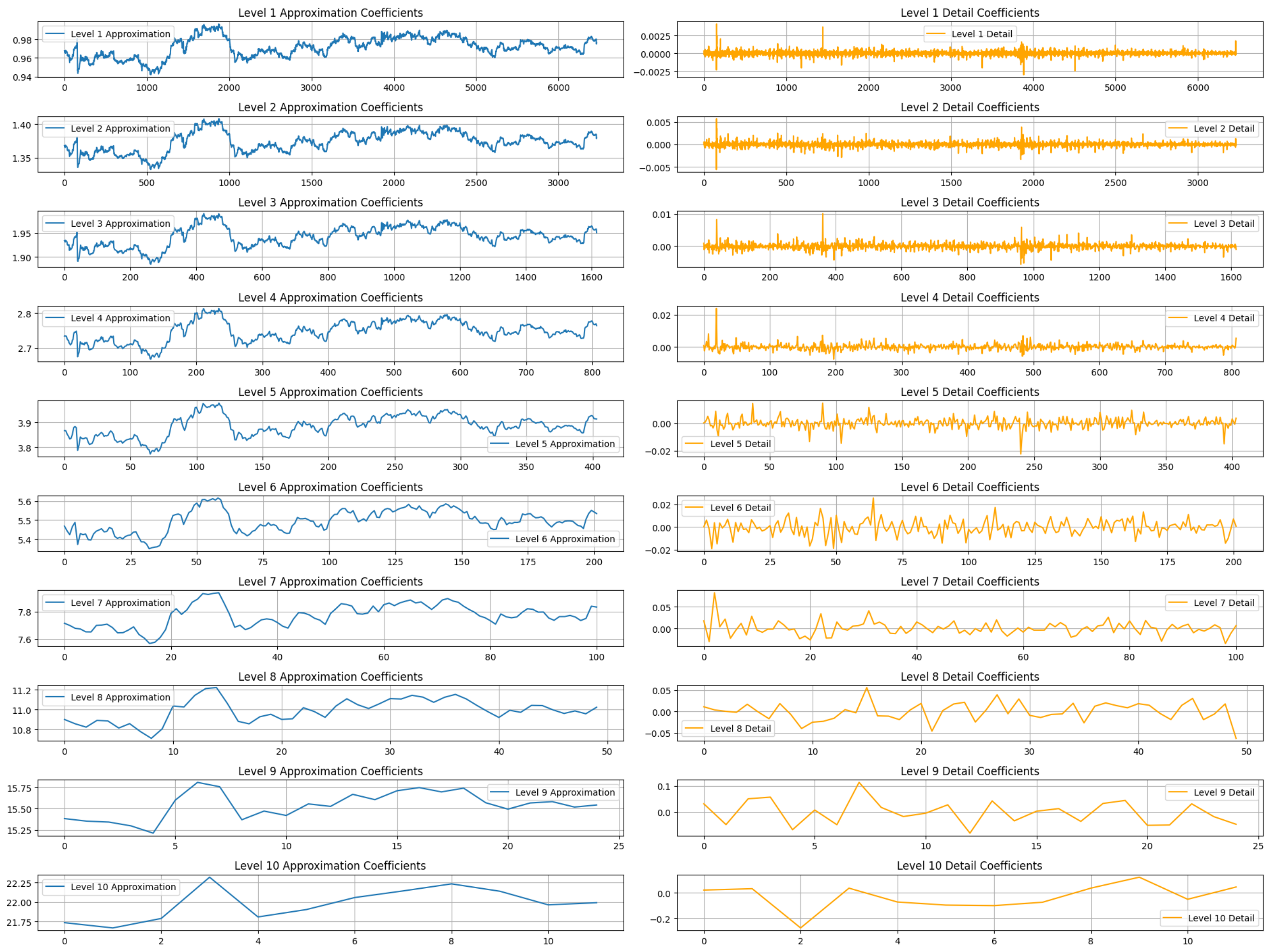
Hence, the code for a singular transform is :

```python
def haar_transform_single(signal):
    n = len(signal)
    if n % 2 != 0:
        raise ValueError("Signal length must be even for Haar transform.") # See the loop below
    ...
    approximation = []
    detail = []
    for i in range(0, n, 2):
        a = (signal[i] + signal[i + 1]) / np.sqrt(2)
        d = (signal[i] - signal[i + 1]) / np.sqrt(2)
        approximation.append(a)
        detail.append(d)

    return np.array(approximation), np.array(detail)
```

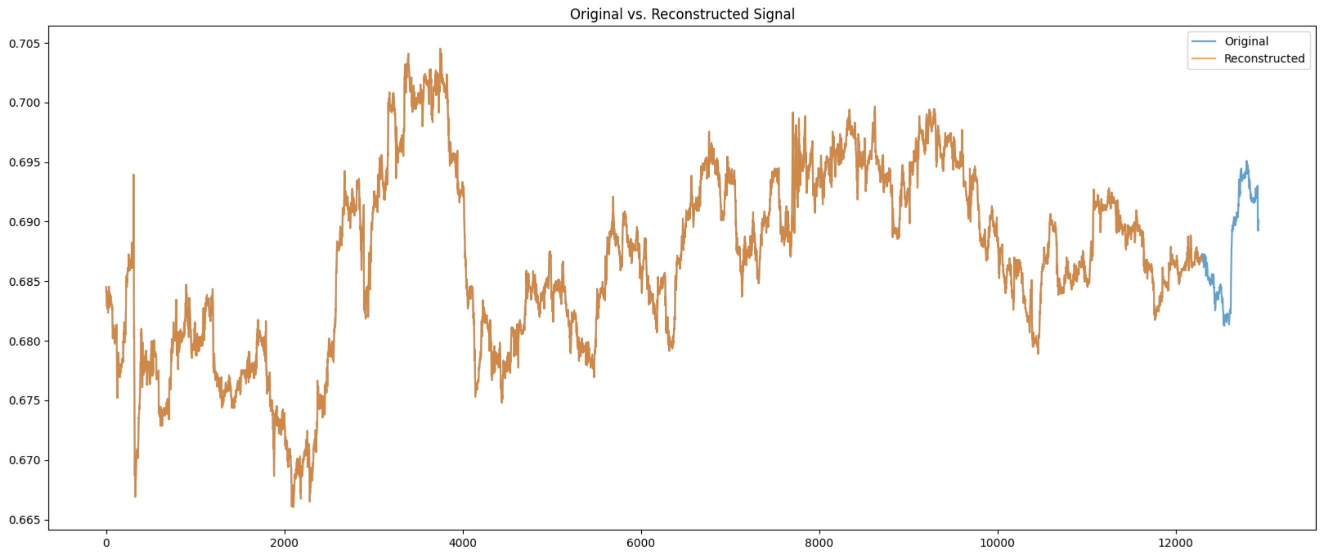
Then we apply the recursive algorithm :

```python
def haar_transform_recursive(signal, max_levels = None):
    coefficients = []
    current_signal = signal
    level = 0

    while len(current_signal) >= 2:
        if max_levels is not None and level >= max_levels:
            break  # Stop if the desired number of levels is reached

        # Ensure the signal length is even by truncating if necessary
        if len(current_signal) % 2 != 0:
            current_signal = current_signal[:-1]  # Truncate the last value

        # Perform one level of the Haar transform
        approximation, detail = haar_transform_single(current_signal)
        coefficients.append((approximation, detail))

        # Continue with the approximation coefficients
        current_signal = approximation
        level += 1  # Increment the level counter

    return coefficients # Returns tupples of approximation AND detail coefficients
```

Plotting the different approximations levels :



Using the inverse transform, we confirm our results :



Although we've seemed to have reconstructed the signal correctly, we do not understand why the length of the reconstructed signal gets lower and lower as we increase the approximation levels and coefficients computed.

5.1.2 Haar transform correlations

Now that we have a working Haar transform algorithm, we proceed with the correlation analysis. The following code computes the different levels of Haar transforms for each exchange rate. Then, it computes the correlations, two by two :

```
coefs_cadeur = haar_transform_recursive(df_CADEUR["HL_avg"].iloc[1:].values)
```

```python
coefs_gbpeur = haar_transform_recursive(df_GBPEUR["HL_avg"].iloc[1:].values)
coefs_sekeur = haar_transform_recursive(df_SEKEUR["HL_avg"].iloc[1:].values)

correlations = []
for level, (coef_cadeur, coef_gbpeur, coef_sekeur) in enumerate(zip(coefs_cadeur, coefs_gbpeur,
    coefs_sekeur)):
    # We get the wavelet approximation coefficients :
    approx_cadeur, _ = coef_cadeur
    approx_gbpeur, _ = coef_gbpeur
    approx_sekeur, _ = coef_sekeur

    # Compute the correlations between the approximations :
    corr_cadeur_gbpeur = np.corrcoef(approx_cadeur, approx_gbpeur)[0, 1]
    corr_cadeur_sekeur = np.corrcoef(approx_cadeur, approx_sekeur)[0, 1]
    corr_gbpeur_sekeur = np.corrcoef(approx_gbpeur, approx_sekeur)[0, 1]

    correlations.append({"Level": level + 1, "CADEUR-GBPEUR": corr_cadeur_gbpeur, "CADEUR-SEKEUR"
    : corr_cadeur_sekeur, "GBPEUR-SEKEUR": corr_gbpeur_sekeur})
```

It gives the following values for correlations :

Level	CADEUR-GBPEUR	CADEUR-SEKEUR	GBPEUR-SEKEUR
1	-0.226949	-0.135290	0.814780
2	-0.227190	-0.135506	0.814931
3	-0.227569	-0.135822	0.815184
4	-0.228495	-0.136924	0.815621
5	-0.229264	-0.137762	0.816206
6	-0.232206	-0.139938	0.817475
7	-0.241440	-0.151158	0.820864
8	-0.246501	-0.153196	0.825356
9	-0.274767	-0.183006	0.833338
10	-0.316037	-0.270196	0.854566
11	-0.493993	-0.398831	0.892320
12	-0.843762	-0.982279	0.929403

Table 2: Values for different levels of currency pairs.

The values of CADEUR-GBPEUR and CADEUR-SEKEUR become increasingly negative as the level increases, this is the Epps effect as we observe a decrease in correlation as the sampling frequency increases (i.e., shorter time intervals). GBPEUR-SEKEUR does not display that tendency. We are not aware of any macro-economic factors explaining that fact.

5.2 Hurst exponents and annualized volatility

In this section, we will estimate the Hurst exponent of each time series. An estimation of the exponent can be achieved by computing the following quantities :

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

And :

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

An estimator of the Hurst exponent H is :

$$\hat{H} = \frac{1}{2}\ln\left(\frac{M_2'}{M_1}\right)$$

In python :

```python
def hurst_exponent_estimator(data, N = None):
    if N is None:
        N = len(data) - 1  # Default to all available increments

    M2_sum, M2_prime_sum  = 0, 0

    # M_2
    for i in range(1, len(data)):
        increment = data[i] - data[i - 1]
        M2_sum += abs(increment)**2
    M2 = M2_sum / N

    # M_2'
    for i in range(2, len(data), 2):
        increment2 = data[i] - data[i - 2]
        M2_prime_sum += abs(increment2)**2
    M2_prime = M2_prime_sum / (N // 2)

    H = 0.5 * np.log2(M2_prime / M2)
    return H
```

We get the following Hurst exponents :

$$\boxed{H_{\text{CADEUR}} \approx 0.66, \quad H_{\text{GBPEUR}} \approx 0.67, \quad H_{\text{SEKEUR}} \approx 0.65}$$

Then, we compute the annualized volatility :

$$\sigma_Y^2 = \sigma_{data}^2 \times \text{Annualization factor}^H.$$

In our case, there are 4 data points each hour for a whole trading year, hence :

$$\text{Annualization factor} = 4 \times 24 \times 252.$$

Here is the code used to compute it :

```python
df_CADEUR["price_return"] = df_CADEUR["HL_avg"].pct_change()
df_GBPEUR["price_return"] = df_GBPEUR["HL_avg"].pct_change()
df_SEKEUR["price_return"] = df_SEKEUR["HL_avg"].pct_change()
annualized_volatility = lambda y,x : np.std(x) * np.sqrt(4 * 24 * 252)**hurst_exponent_estimator(
    y)
```

It gives :

$$\boxed{\sigma_{Y,\text{CADEUR}} \approx 0.014, \quad \sigma_{Y,\text{GBPEUR}} \approx 0.018, \quad \sigma_{Y,\text{SEKEUR}} \approx 0.0089.}$$

SEKEUR has the lowest volatility, suggesting it is the most stable, possibly due to less frequent trading or fewer impactful events affecting SEK relative to EUR. GBPEUR has the highest annualized volatility, indicating it is the most volatile among the three pairs. This could reflect higher sensitivity to market dynamics, political events, or economic data. CADEUR falls in between.

6 Final remarks

As was previously said, the whole code can be found in the .ipynb file. We will try our best to make it run out of the box if you have the correct folder structure (we somewhat changed the format of some datasets so they'll be included). The code will also be a bit modified compared to our version (mostly file structure). Thank you for your patience, we hope our analysis fulfilled your expectations.