# Python for Finance and Optimisation

Mohamed Mrad

**M2-IRFA & MMMEF**
2026-2027
Université Sorbonne Paris 1

# Table des matières


General Introduction . . . . . 5

**1 Foundations of Quantitative Finance & Data Engineering** **7**
Chapter Introduction . . . . . 7
1.1 Microstructural Concepts : Nature of Market Data . . . . . 8
1.1.1 Bid, Ask, Mid-Price and Bid-Ask Spread . . . . . 8
1.1.2 Execution Price and VWAP . . . . . 8
1.2 Mathematical Formalization of Returns . . . . . 8
1.2.1 Non-Additivity of Arithmetic Returns vs. Additivity of Logarithmic Returns 8
1.2.2 The Asset Aggregation Trap . . . . . 9
1.3 Data Engineering : Ingestion, Synchronization and Cleaning of Noisy Data . . . . 9
1.3.1 Geographical Desynchronization and the Epps Effect . . . . . 9
1.3.2 Anomaly Cleaning : Rolling Z-Score Filter . . . . . 9
1.3.3 Python Toolkit for Data Ingestion . . . . . 10
1.4 Modeling Time-Varying Volatility : EWMA . . . . . 14
1.4.1 Python Toolkit for EWMA . . . . . 14
1.5 Matrix Algebra and Co-Movement Spaces . . . . . 17
1.6 Case Study : Temporal Sampling Bias (Graded Assignment) . . . . . 17
1.6.1 Validation Algorithm . . . . . 18
1.6.2 Reflection Questions for Students . . . . . 18

**2 Risk Modeling, Non-Gaussian Estimation** **19**
Chapter Introduction . . . . . 19
2.1 The Limitations of Gaussian Finance . . . . . 20
2.1.1 The Stylized Facts of Mandelbrot and Fama . . . . . 20
2.1.2 Mathematical Formalization of Higher-Order Moments . . . . . 20
2.2 Statistical Analysis of the Distribution in Python . . . . . 20
2.2.1 Python Toolbox : The `scipy.stats` Submodule . . . . . 20
2.3 The Jarque-Bera normality test : . . . . . 25
2.4 The Shapiro–Wilk Normality Test . . . . . 26
2.5 Regulatory Risk Measures : VaR and CVaR . . . . . 31
2.5.1 Mathematical Interpretation of Cornish-Fisher CVaR . . . . . 35

**3 Modern Portfolio Theory & Optimization** **37**
3.0.1 Introduction to the Markowitz Framework . . . . . 37
3.0.2 Formulation 1 : Risk Minimization under a Target Return Constraint . . . 38
3.0.3 Formulation 2 : Sharpe Ratio Maximization . . . . . 39
3.0.4 Formulation 3 : The Sharpe Ratio along the Efficient Frontier . . . . . 41
3.0.5 Graphical Summary . . . . . 42
3.0.6 SLSQP Method — Sequential Least Squares Programming . . . . . 43

3.0.7 Some Remarks . . . 44
3.0.8 Complete Scripted Implementation . . . 45
3.1 Machine-Based Practical Work (Exercises) . . . 52
3.1.1 Solution . . . 52

**4 Asset Pricing Models (CAPM) and Factor Analysis 57**
4.1 Introduction . . . 57
4.2 The Mathematical Framework . . . 58
4.2.1 The CAPM (Capital Asset Pricing Model) . . . 58
4.2.2 Multi-Factor Models : The Fama-French Three-Factor Model . . . 59
4.2.3 Non-Parametric Approach : Principal Component Analysis (PCA) . . . 60
4.2.4 Some Remarks . . . 61
4.3 Python Implementation . . . 62
4.3.1 Key Functions . . . 62
4.3.2 Python Code . . . 66
4.4 Exercises . . . 73

**5 Financial Time Series and Volatility Forecasting 89**
5.1 Introduction . . . 89
5.2 Return and volatility : two different problems . . . 92
5.2.1 Augmented Dickey–Fuller Test . . . 94
5.3 First Family of Models . . . 96
5.3.1 AR Models . . . 96
5.3.2 MA and ARMA Models . . . 96
5.3.3 The ARCH Model . . . 97
5.3.4 From ARCH to GARCH . . . 98
5.3.5 Volatility Forecasting . . . 99
5.4 Python Implementation of a GARCH Model . . . 100
5.5 Some Comments . . . 101
5.6 Complete Python Example . . . 102
5.7 Interpretation of the Overall Approach . . . 104
5.8 Exercises . . . 104

**6 Portfolio Optimization and Dynamic Risk Management with Python 127**
6.1 Introduction . . . 127
6.2 From Historical Prices to Portfolio Returns . . . 127
6.3 Estimating Return and Risk . . . 129
6.4 Building a Benchmark Portfolio . . . 130
6.5 Constrained Optimization . . . 131
6.6 Evaluating the Portfolio Out of Sample . . . 133
6.7 From Static Risk to Conditional Risk . . . 134
6.7.1 Measuring Losses with VaR . . . 136
6.7.2 Stress Testing and Allocation Robustness . . . 137
6.8 Exercises . . . 139

# General Introduction

## Context and the Quantitative Revolution

Since the pioneering work of Harry Markowitz in 1952 on portfolio selection, the financial industry has undergone an unprecedented technological and mathematical transformation. Modern finance is no longer merely a discipline based on accounting valuation or macroeconomic intuition ; it has become quantitative, algorithmic, and data-driven.

Today, asset managers, investment banks, and *hedge funds* operate in an ecosystem where information spreads at the speed of light. Faced with the proliferation of asset classes (equities, bonds, derivatives, commodities, crypto-assets), the ability to model risk, capture market comovements, and allocate capital optimally has become a critical competitive advantage.

## Why Python for Quantitative Finance ?

For several decades, the financial computing landscape was divided :

- **C++** dominated the *Front Office* for high-performance computing (exotic option pricing, high-frequency trading) due to its execution speed.
- **MATLAB and R** were favored by researchers (*Quants*) for statistical prototyping and econometrics.

Python has established itself as the industry's de facto standard by enabling a unique convergence. Although it is an interpreted language (theoretically slower than C++), Python relies on low-level libraries written in C/C++ (`NumPy`, `SciPy`, `Pandas`). This allows financial engineers to benefit from an expressive, readable, and extremely fast-to-develop syntax, while maintaining matrix computation performance close to native code.

Furthermore, Python's native integration with the *Machine Learning* ecosystem (`Scikit-Learn`, `PyTorch`) and its ability to handle large-scale data make it an essential tool for modern financial engineering.

## Course Philosophy and Learning Objectives

The objective of this 18-hour course is to bridge the gap between the rigor of financial mathematical models and the reality of their computational implementation. Theoretical optimization often suffers from an "ivory tower" syndrome : textbook models assume perfect markets, Gaussian returns, and infinite liquidity.

This course is built around a pragmatic philosophy : **confront theory with the frictions of the real world**. The curriculum is structured around three fundamental pillars :

1. ***Data Engineering* and Market Infrastructure (Part I) :** Learn how to collect, clean, and synchronize heterogeneous financial data. We will examine how to handle market anomalies (*bad ticks*), deal with asynchronous market closures across global exchanges, and model the time-varying nature of volatility (EWMA).
2. **Advanced Risk Modeling (Part II) :** Move beyond the restrictive assumption of the Normal distribution. Real financial markets are characterized by frequent crashes and asymmetric behavior. We will learn how to model these "fat tails" and implement regulatory-grade risk measures (Basel III / Solvency II), such as *Value-at-Risk* (VaR) and *Expected Shortfall* (CVaR).
3. **Constrained Optimization Algorithms (Part III) :** Translate asset allocation strategies into numerical optimization problems. Beyond the classical Markowitz model, we

will code portfolios that are robust to market noise, including the Black-Litterman model, Equal Risk Contribution (*ERC*) portfolios, and approaches based on unsupervised *Machine Learning*.

At the end of this course, students will be able to design, code from end to end, and *backtest* (test against historical data) an automated and robust investment strategy suitable for a fund's production environment.

## Prerequisites

To get the most out of this material, students are expected to have :

— A solid understanding of the fundamentals of Python programming (data structures, loops, functions).

— Basic knowledge of matrix algebra (vector products, matrix inversion) and probability/statistics (expectation, variance, normal distribution).

# Chapitre 1

# Foundations of Quantitative Finance & Data Engineering

## Chapter Introduction

### Professional Context and Challenges

Data infrastructure is the nervous system of quantitative finance. Before applying complex optimization models or Machine Learning algorithms, the Quantitative Developer or Risk Analyst must face a fundamental reality : market data are asynchronous, incomplete, noisy, and subject to the microstructural rules of the order book.
Poor data acquisition management or a misunderstanding of the nature of returns systematically invalidates the results of downstream models, according to the fundamental computing principle : *"Garbage in, Garbage out"* (corrupted input data lead to abnormal output results).
This chapter establishes the technical foundations of the course. We will cover the mathematical modeling of financial time, time zone management, and the essential Data Engineering tools required to build a clean, robust, and production-ready data ingestion pipeline.

### Learning Objectives

By the end of this chapter, students will be able to :

1. Master the mathematical and economic distinction between arithmetic and logarithmic returns.
2. Identify and correct geographical desynchronization biases in global markets.
3. Design a complete Python pipeline to filter price anomalies (*bad ticks*) and align heterogeneous time series.
4. Model the unstable dynamics of short-term volatility using a recursive EWMA-type estimator.

### Keywords

Mid-Price, Epps Effect, Log-Returns, Forward Fill, EWMA, Frobenius Norm, Rolling Z-Score.

## 1.1 Microstructural Concepts : Nature of Market Data

Before executing optimization algorithms, the quantitative developer must understand the infrastructure of data originating from the order book. A single price does not exist in professional markets.

### 1.1.1 Bid, Ask, Mid-Price and Bid-Ask Spread

At any time $t$, the market presents two price streams :
— $P_t^{\text{bid}}$ (Bid Price) : The highest price at which a buyer is willing to purchase the asset.
— $P_t^{\text{ask}}$ (Ask Price) : The lowest price at which a seller is willing to sell the asset.
By definition, $P_t^{\text{ask}} > P_t^{\text{bid}}$. The difference between these two prices is called the **Bid-Ask Spread** :

$$S_t = P_t^{\text{ask}} - P_t^{\text{bid}} \tag{1.1}$$

The spread is a direct indicator of market illiquidity. For modeling purposes, the **Mid-Price** is frequently used :

$$P_t^{\text{mid}} = \frac{P_t^{\text{ask}} + P_t^{\text{bid}}}{2} \tag{1.2}$$

### 1.1.2 Execution Price and VWAP

Daily databases (such as Yahoo Finance) provide the transaction closing price (*Close*). For large order volumes, portfolio managers use the **VWAP** (*Volume Weighted Average Price*), which weights each execution price by the traded volume over the period :

$$P_{\text{VWAP}} = \frac{\sum_{i=1}^{M} P_i \times V_i}{\sum_{i=1}^{M} V_i} \tag{1.3}$$

## 1.2 Mathematical Formalization of Returns

The evolution of an asset price $P_t$ is modeled by a non-stationary stochastic process. Moving to returns makes it possible to work with stationary series (statistical properties that remain invariant over time).

### 1.2.1 Non-Additivity of Arithmetic Returns vs. Additivity of Logarithmic Returns

The arithmetic return over the period $[0, T]$, composed of $T$ sub-periods, is :

$$R_{0\to T} = \frac{P_T - P_0}{P_0} = \prod_{t=1}^{T}(1 + R_t) - 1 \tag{1.4}$$

The logarithmic return (or log-return) is defined as :

$$r_t = \ln\left(\frac{P_t}{P_{t-1}}\right) = \ln(P_t) - \ln(P_{t-1}) \tag{1.5}$$

Time additivity can be demonstrated by telescoping the sum :

$$r_{0\to T} = \ln\left(\frac{P_T}{P_0}\right) = \ln\left(\frac{P_1}{P_0} \times \frac{P_2}{P_1} \times \cdots \times \frac{P_T}{P_{T-1}}\right) = \sum_{t=1}^{T} r_t \tag{1.6}$$

### 1.2.2 The Asset Aggregation Trap

The arithmetic return of a portfolio ($R_p$) is strictly the linear combination of the arithmetic returns of its $N$ components, weighted by the weights $w_i$ :

$$R_p = \sum_{i=1}^{N} w_i R_i \tag{1.7}$$

Conversely, due to the non-linearity of the logarithmic function ($\ln(A+B) \neq \ln(A)+\ln(B)$), the logarithmic return of a portfolio is **not** equal to the weighted sum of log-returns :

$$r_p \neq \sum_{i=1}^{N} w_i r_i \tag{1.8}$$

## 1.3 Data Engineering : Ingestion, Synchronization and Cleaning of Noisy Data

### 1.3.1 Geographical Desynchronization and the Epps Effect

When optimizing assets traded across different time zones (e.g., Paris and New York), trading windows overlap only partially. If a raw daily correlation matrix is calculated without adjustment, the coefficients are subject to an artificial downward bias known as the **Epps Effect**.

### 1.3.2 Anomaly Cleaning : Rolling Z-Score Filter

Real-world data feeds contain *bad ticks* (data-entry errors or abnormal price jumps caused by API bugs (Software interface bugs)). We implement a filter based on a rolling Z-Score to remove returns that deviate by more than 3 standard deviations from the local mean.

## Z-Score

The **Z-Score** (or *standard score*) is a fundamental statistical tool. It measures **how many standard deviations an observation lies away from the mean of a group**.
It is an essential standardization method for comparing variables that come from completely different distributions or scales.

**Practical Use and Concrete Example** Imagine that you take two different exams :

— In **Mathematics**, you obtain **15/20** (the class average is 13).

— In **French**, you obtain **13/20** (the class average is 10).

To determine in which subject you actually performed better relative to the rest of the students, simply comparing the deviation from the mean is not sufficient. The dispersion of the grades (the standard deviation) must also be taken into account. This is precisely what the Z-Score allows us to do.

**Mathematical Formula** To calculate the Z-Score of an observation, the following formula is used :

$$Z = \frac{x - \mu}{\sigma} \tag{1.9}$$

Where :
- $x$ is the value of the observation being studied (e.g., your grade).
- $\mu$ represents the **mean** of the population or group.
- $\sigma$ represents the **standard deviation**, which measures the dispersion of the data around the mean.

**Interpretation of Results** The result is a real number (positive, negative, or zero) and can be interpreted as follows :
- $Z = 0$ : The value is exactly equal to the group mean.
- $Z > 0$ (e.g., $+1.5$) : The value is above the mean. A score of $+1.5$ means that the observation lies 1.5 standard deviations above the mean.
- $Z < 0$ (e.g., $-2$) : The value is below the mean.

**The Empirical Rule (Normal Distribution)** For a normal distribution (Gaussian curve) :
- Approximately **68%** of observations have a Z-Score between $-1$ and $+1$.
- Approximately **95%** of observations have a Z-Score between $-2$ and $+2$.
- A Z-Score greater than $+3$ or lower than $-3$ is extremely rare (less than 1% probability), making it very useful for detecting anomalies or outliers (*outliers*).

### 1.3.3 Python Toolkit for Data Ingestion

- `yf.download()` : Queries and downloads financial data.

#### Example

The following line of code makes it possible to download historical data for the different financial assets under study :

```
raw_prices = yf.download(tickers, start="2022-01-01", end="2026-01-01")["Adj Close"]
```

The function `yf.download()` belongs to the `yfinance` library. It retrieves historical market data available from Yahoo Finance.
The `tickers` variable contains the different assets being studied :

$$\texttt{tickers} = [\texttt{'AAPL'}, \texttt{'\^{}FCHI'}, \texttt{'\^{}N225'}, \texttt{'BTC-USD'}]$$

These codes correspond respectively to Apple, the CAC 40, the Nikkei 225, and Bitcoin expressed in U.S. dollars.
The argument

```
start="2022-01-01"
```

indicates the start date of the period under study, while

```
end="2026-01-01"
```

indicates the end date. Since the date specified in `end` is generally exclusive, the period considered is therefore :

$$\boxed{\text{2022-01-01} \leq t < \text{2026-01-01}}$$

The function `yf.download()` retrieves several pieces of information for each asset, including :
— `Open` : opening price ;
— `High` : highest price of the trading session ;
— `Low` : lowest price of the trading session ;
— `Close` : closing price ;
— `Adj Close` : adjusted closing price ;
— `Volume` : traded volume.
The expression

```
["Adj Close"]
```

then selects only the adjusted closing prices. The other information is therefore removed from the DataFrame.
The resulting structure is a dataset in which rows correspond to dates and columns correspond to the different assets :

| Date | AAPL | CAC 40 | Nikkei 225 | BTC-USD |
|---|---|---|---|---|
| $t_1$ | $P_{1,\text{AAPL}}$ | $P_{1,\text{CAC}}$ | $P_{1,\text{Nikkei}}$ | $P_{1,\text{BTC}}$ |
| $t_2$ | $P_{2,\text{AAPL}}$ | $P_{2,\text{CAC}}$ | $P_{2,\text{Nikkei}}$ | $P_{2,\text{BTC}}$ |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ |

The variable `raw_prices` therefore contains a `pandas DataFrame` grouping the adjusted closing prices of the different assets over the period under study.
The use of adjusted prices is particularly appropriate for return analysis, as they account for certain events affecting price series, notably dividends and stock splits (*splits*).
These data can then be used to calculate logarithmic returns :

$$\boxed{r_t = \ln\left(\frac{P_t}{P_{t-1}}\right)}$$

where $P_t$ denotes the adjusted price at date $t$.
— `df.rolling().mean()` and `df.rolling().std()` : Compute the local statistics required for the rolling Z-Score.
— `np.abs()` : Returns the absolute value to identify two-sided deviations.
— **Filling missing values with `ffill()` and `bfill()`**
The following expression, used with the `pandas` library,

```
raw_prices.ffill().bfill()
```

allows missing values (NaN, meaning “Not a Number”) in a data series to be filled. It combines two successive operations :

1. `ffill()` (*forward fill*) ;
2. `bfill()` (*backward fill*).

**1. `ffill()` : Forward filling.** The `ffill()` method replaces each missing value with

the last known value located before it.
For example, consider the following series :

$$[100,\ \text{NaN},\ \text{NaN},\ 105,\ \text{NaN},\ 110].$$

After applying `ffill()`, we obtain :

$$[100,\ 100,\ 100,\ 105,\ 105,\ 110].$$

Thus, for a position $i$ containing a missing value, `ffill()` uses the last available observation before $i$.

**2. `bfill()` : Backward filling.**
The `bfill()` method replaces a missing value with the first known value located after it.
For example :

$$[\text{NaN},\ \text{NaN},\ 100,\ 105,\ \text{NaN}]$$

becomes, after applying `ffill()` :

$$[\text{NaN},\ \text{NaN},\ 100,\ 105,\ 105].$$

The first two values remain missing because no observation is available before them. Applying `bfill()` then makes it possible to complete them :

$$[100,\ 100,\ 100,\ 105,\ 105].$$

**3. Combining the two methods**
The expression

$$\boxed{\texttt{raw_prices.ffill().bfill()}}$$

therefore performs the following operations :

$$\texttt{raw_prices} \xrightarrow{\texttt{ffill()}} \text{forward filling} \xrightarrow{\texttt{bfill()}} \text{filling of remaining missing values.}$$

Its behavior can be summarized as follows :

$$\text{NaN} \longrightarrow \begin{cases} \text{last known value before,} & \text{if it exists,} \\ \text{first known value after,} & \text{otherwise.} \end{cases}$$

This method is particularly useful for time series, for example when some price observations are missing. It therefore makes it possible to obtain a series containing no missing values, provided that at least one observation is known in the series.

```python
import yfinance as yf # Yahoo Finance
import pandas as pd # manipulate, clean, analyze and organize data
import numpy as np # mainly used for numerical computation

# Ingestion of an international and multi-asset portfolio
tickers = ['AAPL', '^FCHI', '^N225', 'BTC-USD']
raw_prices = yf.download(tickers, start="2022-01-01", end="2026-01-01")['Adj Close']

# AAPL (Apple stock), '^FCHI' (CAC 40 index),
# '^N225' (Japanese Nikkei 225 index),
```

```python
# BTC-USD (Bitcoin in US dollars)

# Filling liquidity gaps (specific public holidays)
prices_cleaned = raw_prices.ffill().bfill()

# Calculation of log returns
log_rets = np.log(prices_cleaned / prices_cleaned.shift(1)).dropna()

# BAD TICK FILTER: 20-day rolling Z-Score
rolling_mean = log_rets.rolling(window=20).mean()
rolling_std = log_rets.rolling(window=20).std()
z_scores = np.abs((log_rets - rolling_mean) / rolling_std)

# Replace extreme returns (Z-Score > 3) with 0 (price stagnation)
log_rets_filtered = log_rets.mask(z_scores > 3, 0)

print(f"Number of anomalies filtered and corrected: {(z_scores > 3).sum().sum()}
    ")
```

Listing 1.1 – Market Data Ingestion and Anomaly Filtering Pipeline

**Note : Counting Outliers with `sum()`**
The following Python expression is used to count the total number of values whose Z-Score is greater than 3 :

$$\boxed{\texttt{(z_scores > 3).sum().sum()}}$$

The presence of two calls to `sum()` is explained by the fact that `z_scores` is generally a `pandas DataFrame`, i.e., a two-dimensional data structure composed of rows and columns.
The condition

$$\texttt{z_scores > 3}$$

compares each value in the DataFrame with 3. It produces a matrix of Boolean values :

$$\begin{pmatrix} 2.1 & 4.5 & 1.2 \\ 3.8 & 1.5 & 5.2 \end{pmatrix} \longrightarrow \begin{pmatrix} \texttt{False} & \texttt{True} & \texttt{False} \\ \texttt{True} & \texttt{False} & \texttt{True} \end{pmatrix}.$$

In Python, Boolean values are interpreted as numbers when performing a sum :

$$\texttt{True} = 1, \qquad \texttt{False} = 0.$$

The previous matrix can therefore be represented as :

$$\begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}.$$

**First `sum()`**

The first call to `sum()` performs a column-wise sum :

$$\begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} \xrightarrow{\texttt{sum()}} \begin{pmatrix} 1 & 1 & 1 \end{pmatrix}.$$

This gives the number of values greater than 3 in each column.

**Second `sum()`**

The second call to `sum()` then adds the results obtained for each column :

$$1 + 1 + 1 = 3.$$

We therefore obtain the total number of values greater than 3 in the entire DataFrame.
Thus,

$$\boxed{\texttt{(z_scores > 3).sum().sum()} = \text{total number of values such that } z > 3}$$

The first `sum()` performs the sum across columns, while the second adds the results obtained for all columns.

**Note.** If `z_scores` is a `Series` rather than a `DataFrame`, a single call to `sum()` is generally sufficient :

$$\boxed{\texttt{(z_scores > 3).sum()}}$$

**Note.** `dropna()` is a pandas function used to remove rows or columns containing missing values (NaN).

## 1.4 Modeling Time-Varying Volatility : EWMA

Return volatility is not constant ; it exhibits clusters of high and low instability (the *Volatility Clustering* phenomenon). To capture this dynamic, we formalize the **EWMA** model (*Exponentially Weighted Moving Average*), which is notably used by *RiskMetrics*.
The variance at time $t$, denoted by $\sigma_t^2$, is an exponentially weighted average of the past variance and the most recent squared return :

$$\sigma_t^2 = \lambda \sigma_{t-1}^2 + (1 - \lambda) r_{t-1}^2 \tag{1.10}$$

where $\lambda$ is the persistence parameter (generally set to 0.94 for daily data).

### 1.4.1 Python Toolkit for EWMA

— `df.ewm(alpha, adjust=False)` : Directly implements the EWMA recurrence. The parameter `alpha` is related to $\lambda$ through the relationship $\alpha = 1-\lambda$. The argument `adjust=False` ensures the strict application of the recursive formula without normalization of the initial weights.

```
import matplotlib.pyplot as plt

# RiskMetrics lambda parameter
lambda_param = 0.94
alpha_param = 1 - lambda_param

# Daily EWMA variance calculation
ewma_variance = log_rets_filtered['AAPL'].pow(2).ewm(
    alpha=alpha_param,
    adjust=False
).mean()

# Annualization of EWMA volatility
```

```
ewma_vol_annualized = np.sqrt(ewma_variance * 252)

plt.figure(figsize=(12, 5))
plt.plot(
    ewma_vol_annualized,
    color='darkblue',
    label='EWMA Volatility (RiskMetrics $\lambda=0.94$)'
)
plt.title("Time-Varying Volatility Dynamics of Apple")
plt.xlabel("Date")
plt.ylabel("Annualized Volatility")
plt.legend()
plt.grid(True)
plt.show()
```

Listing 1.2 – EWMA Volatility Calculation and Visualization

**Explanations : Variance Estimation Using the EWMA Method**

The following line of code estimates the variance of the logarithmic returns of Apple stock (`AAPL`) using the EWMA (*Exponentially Weighted Moving Average*) method :

```
ewma_variance = log_rets_filtered['AAPL'].pow(2).ewm(
    alpha=alpha_param,
    adjust=False
).mean()
```

This expression can be broken down into several steps.

### Selecting AAPL Returns

The instruction `log_rets_filtered['AAPL']` selects the time series of logarithmic returns of Apple stock.
The logarithmic return at date $t$ is defined as :

$$r_t = \ln\left(\frac{P_t}{P_{t-1}}\right),$$

where $P_t$ represents the asset price at date $t$.

### Squaring the Returns

The instruction `.pow(2)` squares each return :

$$r_t \longrightarrow r_t^2.$$

For example :

$$[-0.02,\ 0.01,\ -0.03] \longrightarrow [0.0004,\ 0.0001,\ 0.0009].$$

Squaring returns makes it possible, in particular, to measure their dispersion around zero and constitutes the basis for variance estimation.

### Exponentially Weighted Moving Average

The instruction

```
.ewm(alpha=alpha_param, adjust=False)
```

sets up an exponentially weighted moving average (*Exponentially Weighted Moving Average*, EWMA).
This method assigns greater weight to recent observations than to older observations.
The EWMA variance can be represented by the recursive relationship :

$$\boxed{\widehat{\sigma}_t^2 = \alpha r_{t-1}^2 + (1-\alpha)\widehat{\sigma}_{t-1}^2}$$

where :

- $r_{t-1}$ represents the logarithmic return in the previous period ;
- $\widehat{\sigma}_{t-1}^2$ represents the estimated variance in the previous period ;
- $\alpha$ is the smoothing parameter ;
- $0 < \alpha < 1$.

When $\alpha$ is high, recent observations have a greater influence on the variance estimate. Conversely, when $\alpha$ is low, older observations retain greater influence.

### Calculation of the Mean

The instruction `.mean()` calculates the exponentially weighted mean of the squared returns. The EWMA variance can therefore be written as :

$$\boxed{\widehat{\sigma}_t^2 = \alpha r_{t-1}^2 + \alpha(1-\alpha) r_{t-2}^2 + \alpha(1-\alpha)^2 r_{t-3}^2 + \cdots}$$

The most recent observations therefore have the greatest weights.
Thus, the entire instruction

```
log_rets_filtered['AAPL'].pow(2).ewm(
    alpha=alpha_param,
    adjust=False
).mean()
```

can be summarized as :

| logarithmic returns → squared returns → exponentially weighted average → EWMA variance |
|---|

The variable `ewma_variance` therefore contains a time series representing the estimated conditional variance of AAPL returns.
Finally, to obtain EWMA volatility, it is sufficient to take the square root of the variance :

```
ewma_volatility = np.sqrt(ewma_variance)
```

which gives :

$$\boxed{\widehat{\sigma}_t = \sqrt{\widehat{\sigma}_t^2}}$$

EWMA volatility therefore provides a measure of volatility that assigns greater importance to recent market movements.

## 1.5 Matrix Algebra and Co-Movement Spaces

Let $\mathbf{r}_t = [r_{1,t}, r_{2,t}, \ldots, r_{N,t}]^T$ be the vector of returns at time $t$. The annualized variance-covariance matrix $\mathbf{\Sigma}$ (of dimension $N \times N$) is expressed in matrix form as :

$$\mathbf{\Sigma} = 252 \times \frac{1}{T-1} \sum_{t=1}^{T} (\mathbf{r}_t - \mu)(\mathbf{r}_t - \mu)^T \tag{1.11}$$

where $\mu$ is the vector of the average returns of each asset.
The Pearson correlation matrix is then given by :

$$\boxed{\mathbf{R} = \mathbf{D}^{-\frac{1}{2}} \mathbf{\Sigma} \mathbf{D}^{-\frac{1}{2}}}$$

where $\mathbf{\Sigma}$ denotes the covariance matrix and $\mathbf{D}$ the diagonal matrix of variances :

$$\mathbf{D} = \text{diag}\left(\sigma_1^2, \sigma_2^2, \ldots, \sigma_n^2\right).$$

Thus, each element of the correlation matrix $\mathbf{R}$ is given by :

$$\boxed{\rho_{ij} = \frac{\sigma_{ij}}{\sigma_i \sigma_j}}$$

where $\sigma_{ij} = \text{Cov}(r_i, r_j)$.

```python
import seaborn as sns # statistical data visualization

# Calculation of the annualized covariance matrix
Sigma = log_rets_filtered.cov() * 252

# Calculation of the Pearson correlation matrix
R = log_rets_filtered.corr()

# Professional visualization using Seaborn
plt.figure(figsize=(8, 6))
sns.heatmap(
    R,
    annot=True,
    cmap='bwr',
    vmin=-1,
    vmax=1,
    linewidths=0.5
)
plt.title("Co-Movement Matrix (Correlation $R$)")
plt.show()
```

Listing 1.3 – Matrix Calculation of Covariance and Correlation

## 1.6 Case Study : Temporal Sampling Bias (Graded Assignment)

**Problem :** An asset management model rebalances its portfolios every month. Is it preferable to estimate the risk parameters ($\mathbf{\Sigma}$) using daily or weekly returns ?

### 1.6.1 Validation Algorithm

The student must run the following script to measure the Frobenius norm of the difference matrix. The Frobenius norm measures the overall distance between two matrices :

$$\|\mathbf{\Sigma}_{\text{Daily}} - \mathbf{\Sigma}_{\text{Weekly}}\|_F = \sqrt{\sum_{i=1}^{N}\sum_{j=1}^{N}(\sigma_{ij,D} - \sigma_{ij,W})^2} \tag{1.12}$$

```python
# 1. Daily Estimation
Sigma_daily = log_rets_filtered.cov() * 252

# 2. Weekly Estimation using specific resampling
prices_weekly = prices_cleaned.resample('W').last()
log_rets_weekly = np.log(
    prices_weekly / prices_weekly.shift(1)
).dropna()

Sigma_weekly = log_rets_weekly.cov() * 52

# 3. Calculation of the Frobenius matrix distance
matrix_diff = Sigma_daily - Sigma_weekly
frobenius_norm = np.linalg.norm(matrix_diff, ord='fro')

print("--- Quantitative Audit Results ---")
print("Difference matrix :\n", matrix_diff)
print(f"\nGlobal Frobenius distance : {frobenius_norm:.6f}")
```

Listing 1.4 – Case Study Resolution Script

### 1.6.2 Reflection Questions for Students

1. Why is the Frobenius distance not strictly equal to 0 even though both methods measure risk over the same historical period ?
2. What is the impact of the Epps Effect on the weekly estimation compared with the daily estimation ?

# Chapitre 2

# Risk Modeling, Non-Gaussian Estimation

## Chapter Introduction

### Professional Context and Challenges

On October 19, 1987, during « Black Monday », the Dow Jones index collapsed by 22.6% in a single trading session. Under a classical normal distribution framework, such a movement represents an event more than 20 standard deviations ($20\sigma$) away from the mean, with a probability of occurrence so small that it should not happen even on the timescale of the age of the universe. Yet the crashes of 1998 (LTCM), 2008 (Lehman Brothers), 2020 (Covid-19), or liquidity crises in crypto-assets remind us of an unavoidable mathematical reality : **real financial markets are not Gaussian**.

For a portfolio manager or banking regulator, blindly using the normal distribution is a deadly danger. It systematically leads to an underestimation of extreme loss risks. The objective of this chapter is to learn how to measure risk where it actually hides : in asymmetries and fat tails. We will formalize and implement the key indicators of institutional risk management, required by the international Basel III and Solvency II frameworks.

### Learning Objectives

At the end of this chapter, students will be able to :

1. Mathematically diagnose and reject the normality assumption for an asset using the Jarque-Bera statistical test.
2. Model and interpret higher-order statistical moments (Skewness and Kurtosis) for highly volatile assets.
3. Implement in Python the algorithms for calculating *Value-at-Risk* (VaR) and *Expected Shortfall* (CVaR) using historical and parametric approaches.
4. Adjust risk calculations using the Cornish-Fisher expansion to incorporate non-Gaussian effects into capital requirements.

### Keywords

Leptokurtic Distribution, Higher-Order Moments, Loss Quantile, Value-at-Risk, Expected Shortfall, Cornish-Fisher, Jarque-Bera Test.

## 2.1 The Limitations of Gaussian Finance

Classical financial theory (Bachelier, Markowitz, Black-Scholes) relies on the fundamental assumption that the logarithmic returns of assets follow a normal distribution (Gaussian distribution). Although this assumption greatly simplifies analytical calculations, it is systematically contradicted by real market data.

### 2.1.1 The Stylized Facts of Mandelbrot and Fama

As early as the 1960s, Benoît Mandelbrot and Eugene Fama highlighted the **stylized facts** of financial time series :

— **Fat Tails :** Extreme events (crashes, euphoric rallies) have a much higher probability of occurrence than predicted by the normal distribution.

— **Asymmetry (Negative Skewness) :** The return distributions of risky assets (such as stocks) often exhibit left skewness, indicating sharper downward movements than upward movements.

— **Volatility Clustering** : Volatility is not constant and propagates in clusters.

### 2.1.2 Mathematical Formalization of Higher-Order Moments

To quantify the deviation from normality, we analyze the centered moments of order 3 (**Skewness**) and order 4 (**Kurtosis**). Let $r$ denote the random variable representing returns, $\mu = E[r]$ the expected value, and $\sigma = \sqrt{E[(r-\mu)^2]}$ the standard deviation.
Fisher's skewness coefficient is defined as :

$$S = E\left[\left(\frac{r-\mu}{\sigma}\right)^3\right] \tag{2.1}$$

A perfect Gaussian distribution has $S = 0$.
The kurtosis coefficient is defined as :

$$K = E\left[\left(\frac{r-\mu}{\sigma}\right)^4\right] \tag{2.2}$$

A Gaussian distribution has $K = 3$. In quantitative finance, we use the **Excess Kurtosis** ($K_{\text{ex}} = K - 3$). If $K_{\text{ex}} > 0$, the distribution is said to be **leptokurtic** : it has fat tails and a higher central peak.

## 2.2 Statistical Analysis of the Distribution in Python

### 2.2.1 Python Toolbox : The `scipy.stats` Submodule

To audit the statistical shape of our time series, we use specialized functions :

— `stats.skew(array)` : Computes the empirical skewness coefficient. A negative value indicates a left-stretched tail (more extreme latent losses).

**Explanation :** In quantitative finance, this statistic is particularly useful for studying the asymmetry of return distributions.
Intuitively, skewness determines whether observations are distributed symmetrically around their mean or whether one of the two tails of the distribution is more significant than the other.

The important term in the formula is the cube $(r-\mu)^3$. Unlike the square used to calculate variance, the cube preserves the sign of the deviation from the mean.
Thus, an observation located above the mean contributes positively :

$$r-\mu>0 \quad \Longrightarrow \quad (r-\mu)^3>0$$

Whereas an observation located below the mean contributes negatively :

$$r-\mu<0 \quad \Longrightarrow \quad (r-\mu)^3<0$$

**Interpretation of the Sign** The sign of the skewness coefficient identifies the direction of the distribution's asymmetry.
— A positive skewness corresponds to right skewness :
— A zero skewness corresponds to an approximately symmetric distribution :
— A negative skewness corresponds to left skewness :
A negative skewness therefore means that the distribution has a **longer or heavier left tail** than its right tail.

**Example** Consider the following return series :

$$\{-2\%,-1\%,0\%,1\%,2\%,-10\%\} \tag{2.3}$$

Most returns are relatively close to zero, but one particularly large loss of $-10\%$ is observed. This observation contributes strongly and negatively to the skewness calculation because it appears in the term :

$$(-10\%)^3<0$$

The cube amplifies the influence of observations that are very far from the mean.
Consequently, a few strongly negative returns can lead to a significantly negative skewness.

**Why Is Skewness Important in Finance ?** Two assets can have similar means and volatilities while exhibiting very different return distributions.
For example, one asset may exhibit relatively symmetric gains and losses, while another may exhibit many small gains but also a few extremely large losses.
In the second case, the distribution may exhibit negative skewness : $\text{Skewness}(r)<0$.
This indicates that the distribution is asymmetric toward extreme negative returns.
Skewness therefore provides information complementary to volatility. Volatility mainly measures the dispersion of returns around their mean, while skewness provides information about the **asymmetry of the distribution**.

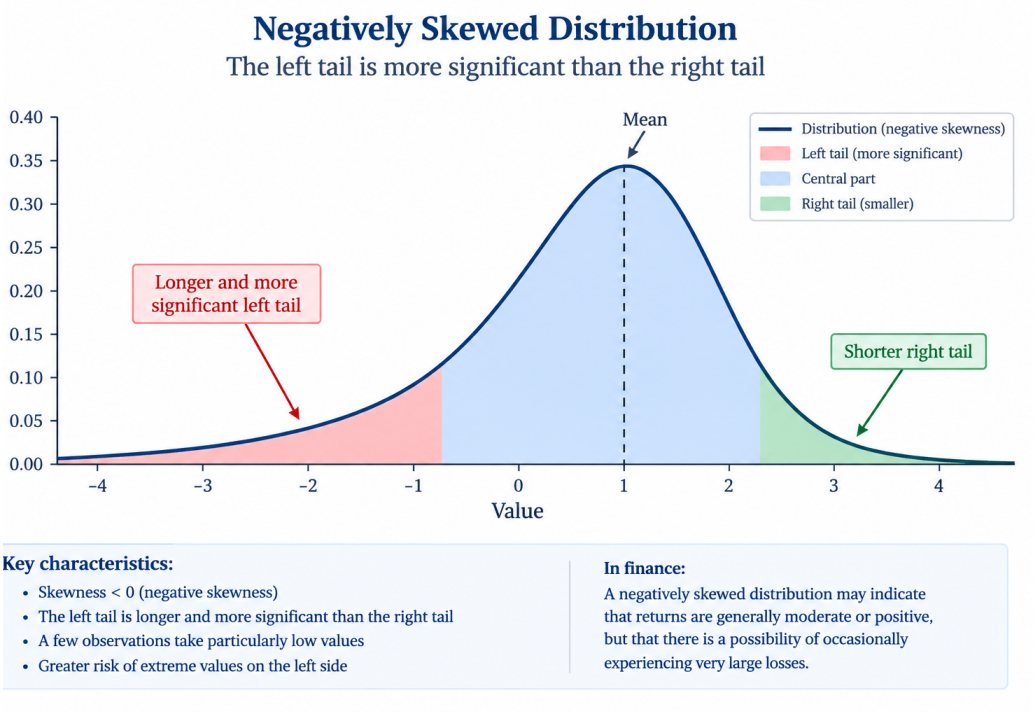



FIGURE 2.1 – Illustration of a negatively skewed distribution.

— `stats.kurtosis(array, fisher=True)` : allows the calculation of the kurtosis coefficient of a data series. In quantitative finance, kurtosis is particularly useful for studying the presence of extreme values in a return distribution. It can notably identify distributions exhibiting *fat tails*.

**Example :** The important term in the formula is :

$$(r - \mu)^4$$

The fourth power gives particularly strong importance to observations that are very far from the mean.
For example, consider two deviations from the mean :

$$r_1 - \mu = 1 \quad \text{and} \quad r_2 - \mu = 4.$$

With a power of two, the contributions would be :

$$1^2 = 1 \quad \text{and} \quad 4^2 = 16.$$

With a power of four, they become :

$$1^4 = 1 \quad \text{and} \quad 4^4 = 256 \tag{2.4}$$

The observation located four standard deviations from the mean therefore has a much greater influence on kurtosis. Kurtosis is thus particularly sensitive to extreme observations.

**The Fisher Convention** There are two conventions commonly used to define kurtosis. Pearson's kurtosis is given by :

$$K = \frac{\mathbb{E}\left[(r-\mu)^4\right]}{\sigma^4} \tag{2.5}$$

For a normal distribution, this quantity is $K = 3$. The Fisher convention consists of subtracting 3 from Pearson's kurtosis :

$$K_F = K_{ex} = \frac{\mathbb{E}\left[(r-\mu)^4\right]}{\sigma^4} - 3 \tag{2.6}$$

Thus, for a normal distribution :

$$K_F = 3 - 3 = 0 \tag{2.7}$$

This is the convention used when writing : `stats.kurtosis(array, fisher=True)`.
The parameter `fisher=True` therefore instructs the function to return the **excess kurtosis** rather than Pearson's kurtosis.

**Interpretation of the Result** When `fisher=True`, the reference value is therefore :

$$K_F = 0$$

The result can then be interpreted as follows :

$$K_F > 0 \quad \Longrightarrow \quad \text{fatter tails than those of a normal distribution}$$

$$K_F = 0 \quad \Longrightarrow \quad \text{kurtosis comparable to that of a normal distribution}$$

$$K_F < 0 \quad \Longrightarrow \quad \text{thinner tails than those of a normal distribution}$$

A distribution with positive Fisher kurtosis is generally described as **leptokurtic**.

**What Does "Fat Tails" Mean ?** A distribution has fat tails when it contains more observations very far from its mean than would be predicted by a normal distribution with the same mean and variance.
Consider, for example, a return series :

$$\{-10\%, -1\%, 0\%, 1\%, 12\%\}$$

The returns of $-10\%$ and $12\%$ are very far from the mean.
Because of the fourth power present in the kurtosis formula, these observations have a significant influence :

$$(-10\%)^4 \quad \text{and} \quad (12\%)^4.$$

Thus, the presence of a few extremely large in absolute value returns can strongly increase kurtosis.

**Financial Interpretation** This property is particularly important when analyzing financial returns. In many financial markets, returns do not perfectly follow a normal distribution. They may exhibit **fat tails**, meaning that extreme events occur more frequently than predicted by a normal distribution.
For example, a model based on a normal distribution may assign a very low probability to a return of $-8\%$ whereas such an event may be less rare in actual financial data.
Positive kurtosis therefore highlights this characteristic :

$$K_F > 0 \implies \text{fat tails} \implies \text{more extreme observations}$$

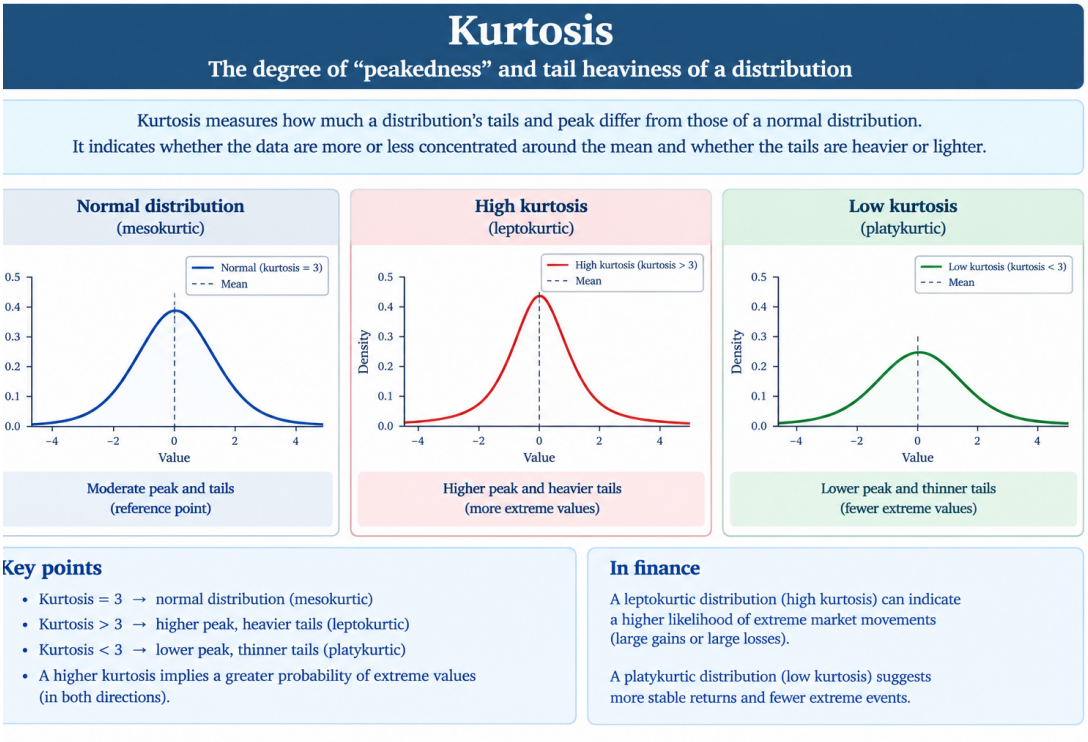



FIGURE 2.2 – Illustration of a negatively skewed distribution.

**Difference Between Skewness and Kurtosis** It is important to distinguish skewness from kurtosis.
**Skewness** primarily measures the asymmetry of the distribution :

$$\text{Skewness}(r) < 0 \implies \text{asymmetry toward negative returns} \tag{2.8}$$

**Kurtosis** primarily measures the importance of the distribution tails and the presence of extreme observations :

$$K_F > 0 \implies \text{fatter tails and more extreme values} \tag{2.9}$$

We can therefore remember :

$$\text{Skewness} \longrightarrow \text{distribution asymmetry} \tag{2.10}$$

and :

$$\text{Kurtosis} \quad \longrightarrow \quad \text{importance of extreme events} \tag{2.11}$$

An asset can therefore simultaneously exhibit negative skewness and positive kurtosis :

$$\text{Skewness}(r) < 0 \qquad \text{and} \qquad K_F > 0 \tag{2.12}$$

In this case, the return distribution exhibits both **asymmetry toward losses** and **fat tails**.
This combination is particularly important in risk management because it indicates that extreme losses can be both asymmetric and more frequent than suggested by a normal distribution.

## 2.3 The Jarque-Bera normality test :

Normality test based on the formula $JB = \frac{T}{6}\left(S^2 + \frac{K_{\text{ex}}^2}{4}\right)$, where :
— $T$ represents the number of observations ;
— $S$ represents skewness ;
— $K$ represents excess kurtosis.

The statistic therefore combines the deviation from zero skewness and the deviation from zero excess kurtosis.
The function `stats.jarque_bera(.)` returns two values : the test statistic and the *p-value*. If the *p-value* $< 0.05$, the normality hypothesis is rejected at the 95% confidence level.

**Explanation :** The **Jarque-Bera** test determines whether a data series is compatible with a **normal distribution**. It is particularly useful in finance for checking whether returns can reasonably be modeled using a normal distribution.
The test is based on two characteristics of the distribution : **skewness** and **kurtosis**.
Recall that for a normal distribution : Skewness $= 0$ and and, when using Fisher's excess kurtosis : Kurtosis $= 0$, so that $JB = 0$.

**Test Hypotheses** The null hypothesis is :

$$H_0 : \text{the distribution of the data is normal} \tag{2.13}$$

The alternative hypothesis is :

$$H_1 : \text{the distribution of the data is not normal} \tag{2.14}$$

**Interpretation of the p-value** A significance level of 5% is generally used :

$$\alpha = 0.05 \tag{2.15}$$

If :

$$p\text{-value} < 0.05 \tag{2.16}$$

then $H_0$ is rejected and we conclude that the data exhibit characteristics significantly different from those of a normal distribution.
Conversely, if :

$$p\text{-value} > 0.05 \tag{2.17}$$

we do not reject $H_0$. This means that the test does not provide sufficient evidence to reject the normality hypothesis.

**p-value of the Jarque–Bera test** Under the null hypothesis of normality, and asymptotically,

$$\boxed{JB \overset{a}{\sim} \chi^2(2)}$$

that is, the Jarque–Bera statistic asymptotically follows a $\chi^2$ distribution with two degrees of freedom. The p-value then corresponds to the probability, under the null hypothesis, of observing a test statistic at least as large as the one obtained from the sample :

$$P\left(\chi_2^2 \geq JB_{\text{obs}}\right)$$

or, equivalently, using the cumulative distribution function $F_{\chi_2^2}$ of the $\chi^2$ distribution with two degrees of freedom :

$$1 - F_{\chi_2^2}(JB_{\text{obs}})$$

In Python, the test can be performed using :

```
jb_stat, p_value = stats.jarque_bera(log_rets_filtered['BTC-USD'])
```

```python
import yfinance as yf
import pandas as pd
import numpy as np
import scipy.stats as stats

# Data ingestion for a highly volatile asset (e.g., Bitcoin)
ticker = 'BTC-USD'
prices = yf.download(ticker, start="2021-01-01", end="2026-01-01")['Adj Close'].ffill()
log_returns = np.log(prices / prices.shift(1)).dropna()

# Calculation of distribution moments via scipy.stats
skewness_val = stats.skew(log_returns)
kurtosis_ex = stats.kurtosis(log_returns, fisher=True)

# Execution of the Jarque-Bera test
jb_stat, p_value = stats.jarque_bera(log_returns)

print(f"--- Statistical Analysis of {ticker} ---")
print(f"Skewness : {skewness_val:.4f} (Expected for a normal distribution: 0)")
print(f"Excess Kurtosis    : {kurtosis_ex:.4f}  (Expected for a normal distribution: 0)")
print(f"Jarque-Bera Statistic : {jb_stat:.2f}")
print(f"Test p-value       : {p_value:.6f}")

if p_value < 0.05:
    print("Conclusion: We reject H0. The distribution is non-Gaussian.")
else:
    print("Conclusion: We cannot reject H0.")
```

Listing 2.1 – Normality Test and Higher-Order Moment Analysis

## 2.4 The Shapiro–Wilk Normality Test

The Shapiro–Wilk test is another statistical test used to determine whether a sample is compatible with a normal distribution.
In the financial context, this test can be used to assess whether the observed returns of an asset or portfolio can reasonably be considered normally distributed.

**Test hypotheses** Consider a sample of returns :

$$X_1, X_2, \ldots, X_n.$$

The Shapiro–Wilk test is based on the following hypotheses :

$$\boxed{\begin{array}{l} H_0 : X \sim \mathcal{N}(\mu, \sigma^2), \\ H_1 : X \text{ does not follow a normal distribution.} \end{array}}$$

The null hypothesis $H_0$ therefore means that the observations are compatible with a normal distribution. The objective of the test is to determine whether the data provide sufficient evidence to reject this hypothesis.

**General principle of the test** The idea behind the Shapiro–Wilk test is to compare the observations in the sample with what we would expect if the observations actually came from a normal distribution.
The first step consists of ordering the observations in ascending order :

$$X_{(1)} \leq X_{(2)} \leq \cdots \leq X_{(n)}.$$

The $X_{(i)}$ are called the **order statistics**. For example, for the sample :

$$\{0.03, -0.01, 0.02, 0.05, -0.02\},$$

we obtain :

$$X_{(1)} = -0.02, \quad X_{(2)} = -0.01, \quad X_{(3)} = 0.02, \quad X_{(4)} = 0.03, \quad X_{(5)} = 0.05.$$

The test then compares this ordered structure with what would be observed in a sample drawn from a normal distribution.

**The Shapiro–Wilk statistic** The test statistic is given by :

$$\boxed{W = \frac{\left(\sum_{i=1}^{n} a_i X_{(i)}\right)^2}{\sum_{i=1}^{n} (X_i - \bar{X})^2}}$$

where :

- $X_{(i)}$ denotes the $i$-th observation after ordering ;
- $\bar{X}$ is the sample mean ;
- $a_i$ are the coefficients of the Shapiro–Wilk test. They are chosen to efficiently measure the deviation between the observed distribution and a normal distribution.
- $n$ is the sample size.

The statistic $W$ generally lies between 0 and 1.

- A value of $W$ close to 1 indicates that the structure of the data is compatible with what is expected under a normal distribution.
- Conversely, a lower value of $W$ indicates a greater deviation from normality.
- However, the final decision of the test is generally based on the **p-value**, rather than simply on whether $W$ is close to 1.

**p-value of the Shapiro–Wilk test** Once the observed statistic $W_{\text{obs}}$ has been calculated, the p-value corresponds to the probability, under the null hypothesis, of obtaining a value of $W$ at least as small as the one observed :

$$P\left(W \leq W_{\text{obs}} \mid H_0\right)$$

The direction of the inequality is important. In the Shapiro–Wilk test, values of $W$ close to 1 are associated with data that are more consistent with a normal distribution, whereas low values of $W$ indicate a deviation from normality. Unlike the Jarque–Bera test, the statistic $W$ does not follow a simple standard distribution such as a normal distribution or a $\chi^2$ distribution. The distribution of $W$ under the null hypothesis depends notably on the sample size and is mathematically complex because it does not have a simple analytical form. Therefore, the p-value is obtained using numerical approximations implemented in statistical software.

**Where do the coefficients $a_i$ come from ?** The coefficients $a_i$ are the most technical part of the test. The idea is to construct coefficients that allow us to best compare the ordered observations in the sample with what we would expect to observe under a normal distribution. To understand their origin, consider a sample of independent random variables following a standard normal distribution :

$$Z_1, Z_2, \ldots, Z_n \sim \mathcal{N}(0,1).$$

These observations are also ordered :

$$Z_{(1)} \leq Z_{(2)} \leq \cdots \leq Z_{(n)}.$$

We then consider the expected values of the order statistics :

$$m_i = E[Z_{(i)}].$$

The vector of expected values is therefore :

$$m = \begin{pmatrix} m_1 \\ m_2 \\ \vdots \\ m_n \end{pmatrix}.$$

This vector represents the average positions that the ordered observations should have if the data came from a standard normal distribution. For example : $m_1 < 0,\ m_n > 0$. Moreover, due to the symmetry of the normal distribution : $m_i = -m_{n+1-i}$.

**The covariance matrix of the order statistics** The order statistics $Z_{(1)}, Z_{(2)}, \ldots, Z_{(n)}$ are not independent. Therefore, we define their covariance matrix as :

$$V = \text{Cov} \begin{pmatrix} Z_{(1)} \\ Z_{(2)} \\ \vdots \\ Z_{(n)} \end{pmatrix}.$$

In other words : $V_{ij} = \text{Cov}\left(Z_{(i)}, Z_{(j)}\right)$.
Based on the vector $m$ and the matrix $V$, the coefficients $a_i$ are constructed in order to assign appropriate weights to the different ordered observations. A theoretical expression, depending on the convention used for the vectors, is :

$$a = \frac{m^\top V^{-1}}{\sqrt{m^\top V^{-1} V^{-1} m}}.$$

This construction determines the weights that make the comparison between the observations and the structure expected under normality as effective as possible.

**Why are the $a_i$ not calculated by hand ?** In practice, the coefficients $a_i$ are generally not calculated directly by hand. Indeed, one would first need to determine $E[Z_{(i)}]$ as well as the covariances $\mathrm{Cov}\left(Z_{(i)}, Z_{(j)}\right)$. These calculations quickly become complex as $n$ increases.
Statistical software therefore uses numerical methods and specific approximations to obtain the coefficients required for the test.
For example, with Python and the `scipy` library, we can simply use : **scipy.stats.shapiro**().
The software automatically calculates the statistic $W$ as well as the p-value.
Therefore, it is mainly important to understand the role of the coefficients $a_i$, rather than trying to calculate them manually. We can summarize this as follows :
**The $a_i$ weight the ordered observations in order to compare their structure with that expected under a normal distribution.**
Intuitively :

$$W = \frac{\text{structure compatible with normality}}{\text{total variability}}.$$

— When the data are close to what is expected under normality, $W$ is close to 1.
— When the data exhibit a substantial deviation from normality, $W$ tends to decrease.

**Statistical decision** Once the statistic $W$ has been calculated, the test provides a p-value. For a significance level of : $\alpha = 5\% = 0.05$, we apply the following rule :

$$\boxed{\begin{cases} p\text{-value} < 0.05 & \Rightarrow \text{reject } H_0, \\ p\text{-value} \geq 0.05 & \Rightarrow \text{do not reject } H_0. \end{cases}}$$

Thus, $p < 0.05$ means that the data provide sufficient evidence to conclude that they are not compatible with a normal distribution at the 5% significance level.
Conversely, $p \geq 0.05$ means that there is not sufficient evidence to reject normality.
**Important :** failing to reject $H_0$ does not mean that we have proved that the data are normally distributed. The correct formulation is :

"We do not reject the normality hypothesis."

and not :

"The data are normally distributed."

**Python implementation** The `scipy.stats` library allows us to directly perform the Shapiro–Wilk test.

### Shapiro–Wilk test

```python
import numpy as np
from scipy.stats import shapiro

# Example of returns
returns = np.array([
    0.012, -0.008, 0.005, 0.003, -0.011,
    0.007, -0.004, 0.009, -0.002, 0.006
])

# Shapiro-Wilk test
W, p_value = shapiro(returns)

print("Shapiro-Wilk statistic:", W)
print("p-value:", p_value)

# Significance level
alpha = 0.05

if p_value < alpha:
    print("Reject H0: the returns are not normally distributed.")
else:
    print("Do not reject H0: the returns are consistent with normality.")
```

The result contains two main pieces of information :
— $W$ : the test statistic ;
— the p-value : used to make the statistical decision.

**Comparison between Shapiro–Wilk and Jarque–Bera** Both tests have the same null hypothesis :

$$H_0 : \quad X \sim \mathcal{N}(\mu, \sigma^2).$$

However, they do not test normality in exactly the same way.

| | **Jarque–Bera** | **Shapiro–Wilk** |
|---|---|---|
| Null hypothesis | The data follow a normal distribution | The data follow a normal distribution |
| Statistic | $JB$ | $W$ |
| Main idea | Uses skewness and kurtosis | Compares the structure of the ordered observations with that expected under normality |
| Characteristic value | $S = 0,\ K = 3$ under normality | $W$ close to 1 under normality |
| Decision | Based on the p-value | Based on the p-value |
| If $p < 0.05$ | Reject normality | Reject normality |

TABLE 2.1 – Comparison between the Jarque–Bera and Shapiro–Wilk tests.

**Joint interpretation of the two tests** It can be useful to apply both tests to the same returns. If $p_{\mathrm{SW}} > 0.05$ and $p_{\mathrm{JB}} > 0.05$, neither test provides sufficient evidence to reject the normality hypothesis at the 5% significance level.
We can therefore conclude :

| Normality is not rejected by either test at the 5% significance level. |
|---|

Conversely, if $p_{\mathrm{SW}} < 0.05$ and $p_{\mathrm{JB}} < 0.05$, both tests reject the normality hypothesis. It is then reasonable to conclude that the returns exhibit a statistically significant deviation from a normal distribution.

**When the two tests lead to different conclusions** It is possible that the two tests do not lead to the same conclusion. For example, $p_{\mathrm{SW}} < 0.05$ but $p_{\mathrm{JB}} > 0.05$. This means that Shapiro–Wilk rejects normality whereas Jarque–Bera does not reject it.
This difference is not necessarily contradictory because the two tests use different information :

| Shapiro–Wilk $\longrightarrow$ overall structure of the ordered observations |
|---|

whereas :

| Jarque–Bera $\longrightarrow$ skewness + kurtosis. |
|---|

A difference in conclusions can therefore occur when the deviation from normality comes from a particular characteristic of the distribution, for example :

— substantial skewness ;
— heavier tails than those of a normal distribution ;
— extreme observations ;
— another form of deviation from normality.

$\Rightarrow$ The two tests are therefore complementary rather than competing.

**Remark** For financial returns, normality is an important assumption in many classical models. However, financial returns frequently exhibit skewness, kurtosis greater than 3, and fat tails. It is therefore particularly useful to combine several tools in order to obtain a more complete view of the distribution of returns.

## 2.5 Regulatory Risk Measures : VaR and CVaR

In financial risk management, knowing the average return of a portfolio is not sufficient : it is also essential to quantify the magnitude of potential losses, particularly under adverse and extreme scenarios. **Value-at-Risk (VaR)** and **Conditional Value-at-Risk (CVaR)**, also known as *Expected Shortfall (ES)*, are two fundamental measures used to quantify this risk.
**VaR** answers the following question :

> *What level of loss should the portfolio exceed only with a given probability, over a given horizon ?*

**Concrete Example.** Suppose a portfolio has a **95% VaR over a one-day horizon of** $50,000$.
The term "1 day" means that we study the portfolio's loss over a single day. The 95% confidence level means that we consider the quantile corresponding to the worst 5% of scenarios.
Mathematically, if $L_{t,1j}$ denotes the portfolio loss over one day, then :

$$\boxed{\mathbb{P}(L_{t,1j} > 50,000) = 5\%} \tag{2.18}$$

or equivalently :

$$\boxed{\mathbb{P}(L_{t,1j} \leq 50,000) = 95\%} \tag{2.19}$$

In other words, for a given day, the model estimates a 5% probability that the loss exceeds $50,000$. It is important to distinguish the different elements of this definition :

$$\boxed{\underbrace{1 \text{ day}}_{\text{horizon}} \qquad \underbrace{95\%}_{\text{confidence level}} \qquad \underbrace{5\%}_{\text{exceedance probability}}} \tag{2.20}$$

VaR therefore only provides a **loss threshold**. It does not specify the magnitude of the loss once this threshold is exceeded. This is precisely the role of **CVaR**. CVaR measures the **average loss in scenarios where VaR is exceeded**. It therefore answers the following question :

*When the loss exceeds the VaR threshold, what is the average magnitude of that loss ?*

For example, if the one-day 95% VaR is $50,000$ and the one-day 95% CVaR is $75,000$, this means that among the scenarios belonging to the worst 5% loss tail, the average loss is $75,000$.
CVaR therefore focuses on the portion of the distribution beyond the VaR threshold. Thus, the two measures provide complementary information :

$$\boxed{\text{VaR} \longrightarrow \text{threshold separating ordinary scenarios from extreme scenarios}} \tag{2.21}$$

$$\boxed{\text{CVaR} \longrightarrow \text{average loss in extreme scenarios beyond VaR}} \tag{2.22}$$

This distinction is particularly important in quantitative finance. Two portfolios may have the same VaR while having very different CVaRs. The portfolio with the higher CVaR is then exposed to potentially larger extreme losses.

**Definition 2.5.1** *In general, consider a random variable $L$ representing the **loss** of a portfolio over a given time horizon. For a confidence level $\alpha \in (0,1)$,*

— *the **Value-at-Risk** is defined as the $\alpha$-quantile of the loss distribution :*

$$P(L > VaR_\alpha) = \alpha \tag{2.23}$$

— **Expected Shortfall (CVaR) :** *which measures the expected loss (absolute value), conditional on the loss exceeding the VaR threshold, is defined by*

$$CVaR_\alpha = E[L \mid L > VaR_\alpha] \tag{2.24}$$

**Python Toolbox for Risk Calculation :** To implement our three risk models (Historical, Parametric, Cornish-Fisher), we use the following key functions :

— `np.percentile(df, q)` : Extracts the $q$-th percentile of the distribution. For a historical VaR with a risk threshold $\alpha = 5\%$, we seek to isolate the breakpoint corresponding to the worst 5% of returns in the time series.

— `stats.norm.ppf(q)` : *Percent Point Function* (or quantile function). It is the inverse of the cumulative distribution function of the normal distribution. For $\alpha = 0.05$, it returns the standardized score $z_\alpha \approx -1.6448$.

— `df[df <= threshold].mean()` : Uses Pandas Boolean masking to isolate only returns less than or equal to the VaR quantile, and then computes their average. This is the direct implementation of the conditional expectation of historical Expected Shortfall.

```
# Definition of risk parameters
alpha = 0.05 # 5% threshold (95% confidence)
capital = 1000000 # $1,000,000 portfolio

# Calculation of empirical parameters
mu = log_returns.mean()
sigma = log_returns.std()

# --- METHOD 1: HISTORICAL (NON-PARAMETRIC) ---
# Extraction of the 5th empirical percentile
var_hist_ret = np.percentile(log_returns, alpha * 100)
var_historical = -var_hist_ret * capital

# Historical CVaR: average of returns below the VaR
cvar_hist_ret = log_returns[log_returns <= var_hist_ret].mean()
cvar_historical = -cvar_hist_ret * capital

# --- METHOD 2: GAUSSIAN PARAMETRIC ---
# Retrieval of the classical z-score via stats.norm.ppf
z_alpha = stats.norm.ppf(alpha)
var_parametric = -(mu + z_alpha * sigma) * capital

# Gaussian CVaR:
# Theoretical average of returns located in the left tail
# For alpha = 0.05, phi(z_alpha) / alpha corresponds to the expectation
# of a standard normal variable conditioned on Z <= z_alpha
cvar_parametric_ret = -(mu - sigma * stats.norm.pdf(z_alpha) / alpha)
cvar_parametric = cvar_parametric_ret * capital

# --- METHOD 3: CORNISH-FISHER (NON-GAUSSIAN ADJUSTMENT) ---
S = stats.skew(log_returns)
K = stats.kurtosis(log_returns, fisher=True)

# Polynomial extension of the z_alpha
z_cf = (z_alpha +
        (z_alpha**2 - 1) * S / 6 +
        (z_alpha**3 - 3 * z_alpha) * K / 24 -
        (2 * z_alpha**3 - 5 * z_alpha) * (S**2) / 36)

var_cornish_fisher = -(mu + z_cf * sigma) * capital

# Cornish-Fisher CVaR:
# Cornish-Fisher directly provides an approximation of the quantile.
# To obtain an approximation of CVaR, we integrate the Cornish-Fisher
# quantiles over the entire left tail of the distribution.
u_values = np.linspace(0.0001, alpha, 1000)

z_values = stats.norm.ppf(u_values)

z_cf_values = (z_values +
               (z_values**2 - 1) * S / 6 +
               (z_values**3 - 3 * z_values) * K / 24 -
               (2 * z_values**3 - 5 * z_values) * (S**2) / 36)
```

```
54
55 q_cf_values = mu + sigma * z_cf_values
56
57 # Numerical integration of the left-tail quantiles
58 cvar_cf_ret = np.trapezoid(q_cf_values, u_values) / alpha
59 cvar_cornish_fisher = -cvar_cf_ret * capital
60
61 print("\n--- Risk Analysis (1-Day Horizon, $1M Capital) ---")
62 print(f"Historical VaR (95\%)          : {var_historical:.2f} $")
63 print(f"Historical CVaR (95\%)         : {cvar_historical:.2f} $")
64 print(f"Gaussian VaR (95\%)            : {var_parametric:.2f} $")
65 print(f"Gaussian CVaR (95\%)           : {cvar_parametric:.2f} $")
66 print(f"Cornish-Fisher VaR (95\%)      : {var_cornish_fisher:.2f} $")
67 print(f"Cornish-Fisher CVaR (95\%)     : {cvar_cornish_fisher:.2f} $")
```

Listing 2.2 – Comparative Implementation of VaR and CVaR

### Mathematical Origin of the Cornish-Fisher Expansion

The **Cornish-Fisher** formula is derived from the **Edgeworth expansion**, which approximates the cumulative distribution function of a non-normal distribution using the normal distribution. We seek the quantile $q_\alpha$ satisfying :

$$F(q_\alpha) = \alpha, \text{where F is the cumulative distribution function of a standard normal distribution}$$

We then express this quantile as a correction of the normal quantile $z_\alpha$ :

$$q_\alpha = z_\alpha + \delta,$$

where $\delta$ represents a correction related to the non-normal characteristics of the distribution.

By expanding the Edgeworth expansion around $z_\alpha$ and then approximately inverting the relationship, we obtain the Cornish-Fisher expansion :

$$\boxed{q_\alpha \approx z_\alpha + \frac{S}{6}(z_\alpha^2 - 1) + \frac{K}{24}(z_\alpha^3 - 3z_\alpha)\frac{S^2}{36}(2z_\alpha^3 - 5z_\alpha)} \tag{2.25}$$

where :

$$z_\alpha = \text{normal distribution quantile}, \qquad S = \text{skewness}, \qquad K = \text{excess kurtosis}. \tag{2.26}$$

The formula therefore corrects the normal quantile $z_\alpha$ to account for the **asymmetry** of the distribution, represented by $S$, and the **thickness of its tails**, represented by $K$.
When the distribution is normal :

$$S = 0 \qquad \text{and} \qquad K = 0. \tag{2.27}$$

The formula then reduces to :

$$\boxed{q_\alpha = z_\alpha} \tag{2.28}$$

The terms $z_\alpha^2 - 1$ and $z_\alpha^3 - 3z_\alpha$ naturally appear in the Edgeworth expansion and are related to the **Hermite polynomials** associated with the normal distribution.

### 2.5.1 Mathematical Interpretation of Cornish-Fisher CVaR

The important difference between VaR and CVaR is that the Cornish-Fisher formula directly provides an approximation of a **quantile**, and therefore of VaR, but not directly of CVaR.
The Cornish-Fisher VaR is approximately given by :

$$\boxed{\text{VaR}_\alpha^{CF} \approx -q_\alpha^{CF}} \tag{2.29}$$

where $q_\alpha^{CF}$ represents the return quantile obtained using the Cornish-Fisher expansion.
CVaR, however, requires considering all quantiles located in the left tail of the distribution. It corresponds to the average loss associated with the most unfavorable scenarios.
For a variable representing returns, CVaR can therefore be approximated by :

$$\boxed{\text{CVaR}_\alpha^{CF} \approx -\frac{1}{\alpha}\int_0^\alpha q_u^{CF}\,du} \tag{2.30}$$

where $q_u^{CF}$ denotes the Cornish-Fisher quantile corresponding to the probability level $u$.
In our example, with $\alpha = 0.05$, we therefore consider the quantiles between 0 and 5% :

$$\text{CVaR}_{0.05}^{CF} \approx -\frac{1}{0.05}\int_0^{0.05} q_u^{CF}\,du. \tag{2.31}$$

The idea therefore differs from VaR. VaR uses a single quantile :

$$\text{VaR}_\alpha^{CF} \approx -q_\alpha^{CF}, \tag{2.32}$$

whereas CVaR uses all the quantiles in the tail :

$$\text{CVaR}_\alpha^{CF} \approx -\frac{1}{\alpha}\int_0^\alpha q_u^{CF}\,du. \tag{2.33}$$

In Python, the integral is approximated numerically by constructing a grid of probabilities :

$$u_1, u_2, \ldots, u_M \in [0, \alpha]. \tag{2.34}$$

We then calculate the corresponding Cornish-Fisher quantile for each value $u_i$ :

$$q_{u_i}^{CF} = \mu + \sigma z_{CF}(u_i). \tag{2.35}$$

Finally, the integral is approximated numerically using the trapezoidal rule :

$$\int_0^\alpha q_u^{CF}\,du \approx \text{Trapz}\left(q_{u_1}^{CF}, \ldots, q_{u_M}^{CF}\right). \tag{2.36}$$

This provides an approximation of CVaR that accounts for the skewness and excess kurtosis of returns through the Cornish-Fisher expansion.
Thus :

$$\boxed{\text{Cornish-Fisher} \rightarrow \text{corrected quantiles} \rightarrow \text{tail integration} \rightarrow \text{CVaR}} \tag{2.37}$$

# Chapitre 3

# Modern Portfolio Theory & Optimization

### 3.0.1 Introduction to the Markowitz Framework

Published in 1952 by Harry Markowitz, *Modern Portfolio Theory* (MPT) laid the scientific foundations of diversification. Markowitz's fundamental paradigm is that a financial asset should not be analyzed in isolation, but according to its contribution to the overall risk-return trade-off of the portfolio to which it belongs.
The investor seeks to construct an optimal portfolio by combining assets whose imperfect correlation makes it possible to reduce overall risk (variance) without proportionally sacrificing expected return.

**The Mathematical Framework** Let us consider an investment universe composed of $N$ risky financial assets. We define the vector of weights $W$ allocated to each asset as :

$$W = \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_N \end{pmatrix}$$

For a portfolio with no short selling and fully invested, we impose the following structural constraints :

$$\sum_{i=1}^{N} w_i = 1 \quad \text{and} \quad w_i \geq 0 \quad \forall i \in \{1, \ldots, N\}$$

Let $\mu_i = E(R_i)$ be the expected return of asset $i$, and $\mu$ the column vector of these expected returns. The expected return of the overall portfolio, denoted $E(R_p)$, is the mathematical expectation of the weighted sum of returns :

$$E(R_p) = \sum_{i=1}^{N} w_i E(R_i) = W^T \mu$$

Let $\Sigma \in \mathbb{R}^{N \times N}$ be the variance-covariance matrix of asset returns. The generic element $\sigma_{ij}$ represents the covariance between asset $i$ and asset $j$ (if $i = j$, $\sigma_{ii} = \sigma_i^2$ is the variance of asset $i$). The overall portfolio variance $\sigma_p^2$ can be expressed as a quadratic form :

$$\sigma_p^2 = \sum_{i=1}^{N} \sum_{j=1}^{N} w_i w_j \sigma_{ij} = W^T \Sigma W$$

Thus, the portfolio standard deviation (or volatility) is

$$\sigma_p = \sqrt{W^T \Sigma W}$$

The objective of portfolio optimization is to determine how to allocate capital among several assets in order to obtain the best possible trade-off between return and risk.
Two classical formulations can be used to approach this problem. The first consists of finding, for a given level of return, the portfolio with the minimum risk. The second consists of directly finding the portfolio that maximizes the excess return obtained per unit of risk.
These two approaches are closely related and make it possible to understand the construction of the efficient frontier and the identification of the tangency portfolio.

### 3.0.2 Formulation 1 : Risk Minimization under a Target Return Constraint

A first way to construct the efficient frontier is to fix a target expected return level $\mu_0$ and search, among all portfolios capable of achieving this return, for the one with the lowest risk.

Let

$$W = \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{pmatrix}$$

be the vector of the weights of the different assets, $\mu$ the vector of expected returns, and $\Sigma$ the covariance matrix of returns.
The optimization problem therefore consists of minimizing this variance for a fixed target return $\mu_0$ :

$$(1) \quad \begin{aligned} \min_{W} \quad & \frac{1}{2} W^T \Sigma W \\ \text{subject to :} \quad & W^T \mu = \mu_0, \\ & W^T \mathbf{1} = 1, \\ & W \geq 0. \end{aligned}$$

The factor $\frac{1}{2}$ does not modify the optimal solution. It is traditionally introduced in order to simplify the derivatives of the objective function.

The different constraints have a simple economic interpretation :

— $W^T \mu = \mu_0$ imposes that the portfolio achieves exactly the target expected return ;

— $W^T \mathbf{1} = 1$ imposes that the entire capital is invested ;

— $W \geq 0$ prohibits short selling : each weight must be positive or zero.

For a given value of $\mu_0$, the optimization therefore provides the portfolio with **minimum variance** among all portfolios with the target return.
By progressively varying the target return $\mu_0$, we obtain a sequence of optimal portfolios. Each portfolio is characterized by a pair :

$$(\sigma_p, \mu_p).$$

The representation of these pairs in the return-risk plane then makes it possible to construct the **efficient frontier**.

**Graphical Interpretation** The following graph schematically illustrates the construction of the efficient frontier.
The region represents the set of feasible portfolios. Each portfolio is characterized by its risk $\sigma_p$ and expected return $\mu_p$. For a target return $\mu_0$, we search for the feasible portfolio with the lowest risk.
The resulting point belongs to the efficient frontier. By repeating this optimization for different target return levels, we progressively construct the efficient frontier.

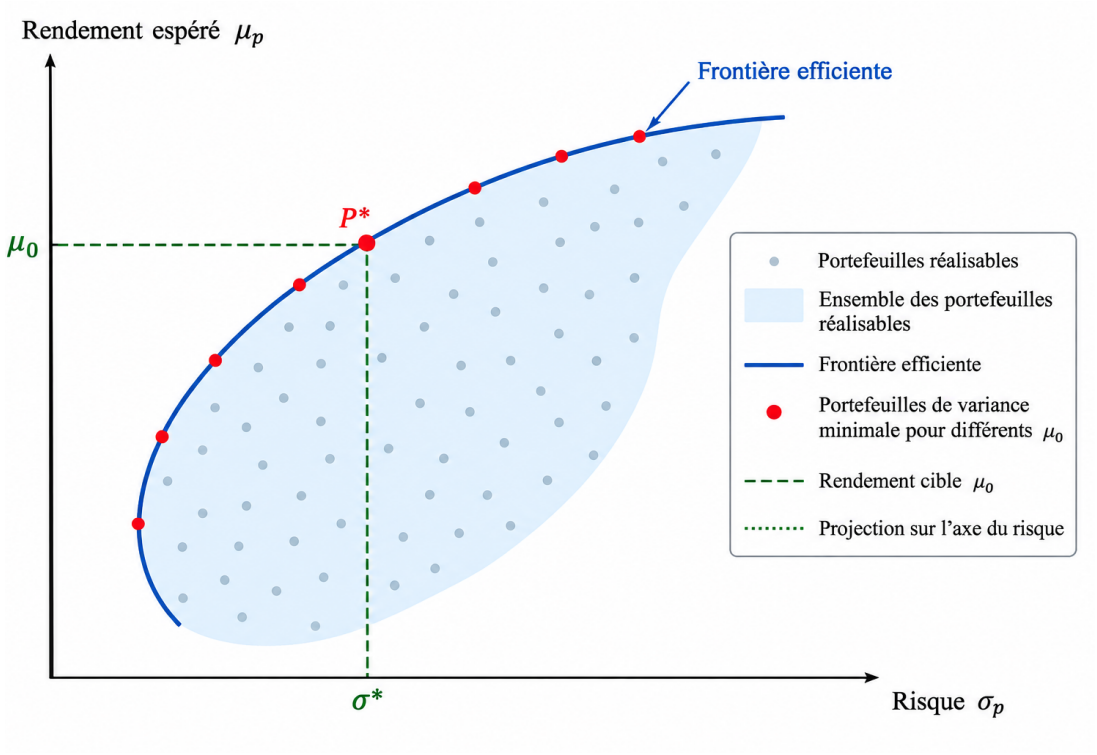



FIGURE 3.1 – Illustration of a negatively skewed distribution.

The principle is therefore as follows :

| Target return | $\longrightarrow$ | Variance minimization | $\longrightarrow$ | Optimal portfolio |
|---|---|---|---|---|

By repeating this optimization for different values of $\mu_0$, we obtain the efficient frontier.

### 3.0.3 Formulation 2 : Sharpe Ratio Maximization

When a risk-free asset offering a certain return $R_f$ is introduced, it becomes possible to compare risky portfolios while taking into account both their return and their level of risk.
The **Sharpe Ratio** measures the excess return obtained per unit of risk :

$$SR(W) = \frac{W^T\mu - R_f}{\sqrt{W^T\Sigma W}}$$

where :
— $W^T\mu$ represents the expected return of the risky portfolio ;
— $R_f$ represents the return of the risk-free asset ;
— $\sqrt{W^T\Sigma W} = \sigma_p$ represents the volatility of the portfolio.

The objective is then to find the risky portfolio that maximizes the Sharpe Ratio :

$$\boxed{\begin{array}{rl} \max\limits_{W} & \dfrac{W^T\mu - R_f}{\sqrt{W^T\Sigma W}} \\ \text{subject to :} & W^T\mathbf{1} = 1, \\ & W \geq 0. \end{array}}$$

In practice, many numerical optimization algorithms, including those available in `scipy.optimize`, are formulated as minimization problems.
It is therefore sufficient to change the sign of the objective function :

$$\boxed{\min_{W} \quad -SR(W) = -\frac{W^T\mu - R_f}{\sqrt{W^T\Sigma W}}}$$

subject to the same constraints :

$$W^T\mathbf{1} = 1, \qquad W \geq 0.$$

**The Tangency Portfolio :** The solution to this problem is called the **tangency portfolio**, or **maximum Sharpe portfolio**.
To understand its graphical importance, let us place the risk-free asset in the return-risk plane.
A risk-free asset has zero volatility. It is therefore represented by the point $(0, R_f)$.
Now consider a risky portfolio $P$, characterized by $(\sigma_P, \mu_P)$. The line connecting the risk-free asset to portfolio $P$ has slope :

$$\frac{\mu_P - R_f}{\sigma_P}.$$

But this expression corresponds exactly to the Sharpe Ratio of the portfolio :

$$\boxed{\text{slope of the line} = SR(P)}$$

Thus, maximizing the Sharpe Ratio graphically means searching, among all lines originating from the risk-free asset, for the one with the highest slope.
The portfolio corresponding to the point of tangency with the efficient frontier is called the **tangency portfolio**.
The line passing through the risk-free asset and the tangency portfolio is called the **Capital Market Line** (CML). Its slope is equal to the maximum Sharpe Ratio :

$$\boxed{\text{slope of the CML} = SR(W^*) = \max_{W} SR(W)}$$

where $W^*$ denotes the tangency portfolio.

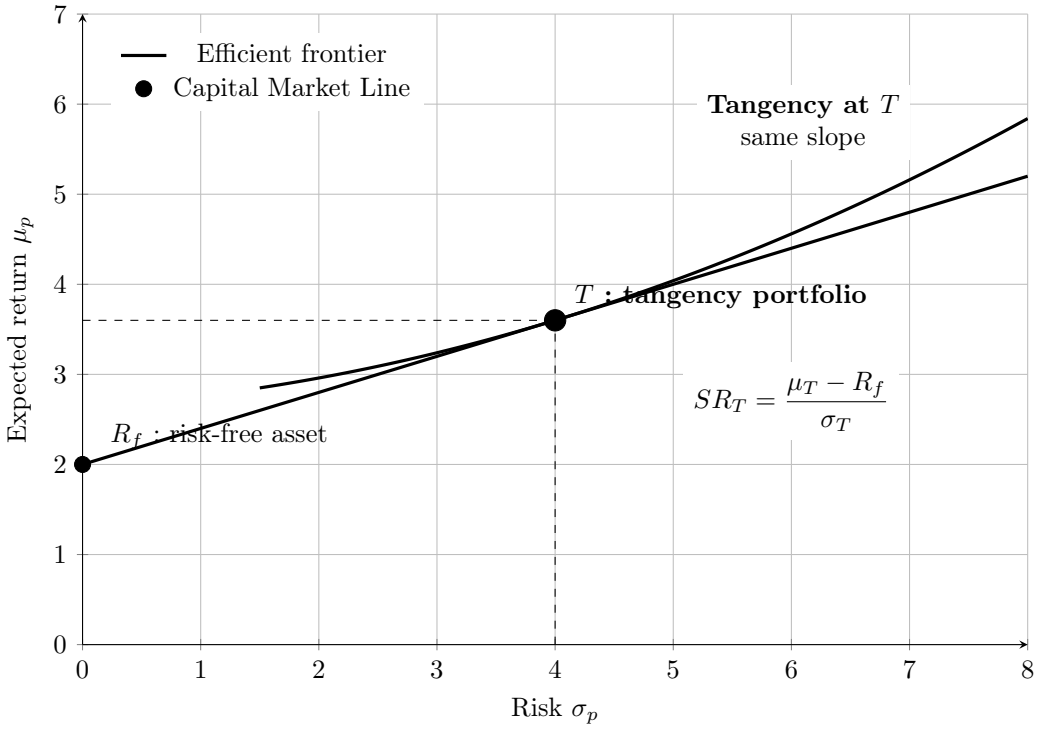



FIGURE 3.2 – The tangency portfolio $T$ is the point where the Capital Market Line is tangent to the efficient frontier. The slope of this line is equal to the Sharpe Ratio of the tangency portfolio. The data are purely illustrative.

#### Combination of the Risk-Free Asset and the Tangency Portfolio

Once the tangency portfolio $W^*$ has been determined, the investor can combine this portfolio with the risk-free asset. If $\alpha$ ($0 \leq \alpha \leq 1$) denotes the proportion invested in the tangency portfolio, then the proportion invested in the risk-free asset is $1 - \alpha$ and the expected return of the overall portfolio is :

$$\mu_C = (1 - \alpha)R_f + \alpha\mu_{W^*}.$$

We can also write :

$$\boxed{\mu_C = R_f + \alpha(\mu_{W^*} - R_f)}$$

and its risk is :

$$\boxed{\sigma_C = \alpha\sigma_{W^*}.}$$

These two equations show that the portfolios obtained by combining the risk-free asset and the tangency portfolio lie on a straight line in the return-risk plane : the **Capital Market Line**.

### 3.0.4 Formulation 3 : The Sharpe Ratio along the Efficient Frontier

The two previous formulations can be connected through a third representation that is particularly useful for understanding the role of the tangency portfolio.
For each target return $\mu_0$, the first optimization provides a minimum-variance portfolio $W(\mu_0)$. We can then calculate its Sharpe Ratio :

$$SR(\mu_0) = \frac{\mu_0 - R_f}{\sigma_p(\mu_0)}$$

with :

$$\sigma_p(\mu_0) = \sqrt{W(\mu_0)^T \Sigma W(\mu_0)}.$$

We therefore obtain a function that associates each target return with the Sharpe Ratio of the corresponding efficient portfolio :

$$\boxed{\mu_0 \quad \longmapsto \quad SR(\mu_0)}$$

The tangency portfolio then corresponds to the point at which this function reaches its maximum :

$$\boxed{\mu_0^* = \underset{\mu_0}{\text{argmax}}\, SR(\mu_0)}$$

In other words, the tangency portfolio is the efficient portfolio that offers the best excess return per unit of risk.

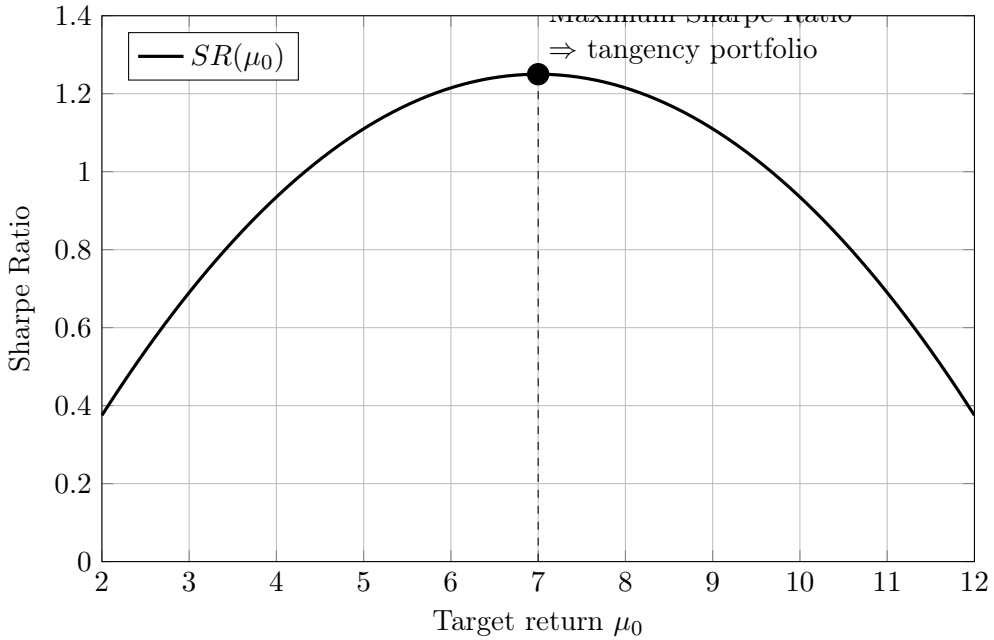


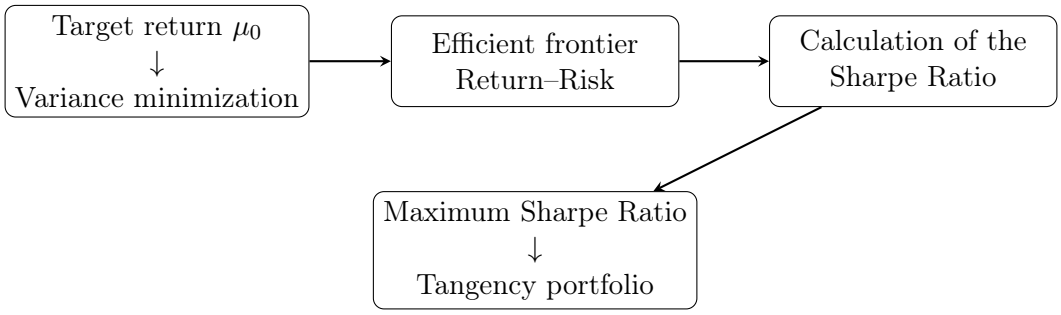

FIGURE 3.3 – Evolution of the Sharpe Ratio along the efficient frontier. The maximum corresponds to the tangency portfolio. The values shown are purely illustrative.

This third graph directly highlights the link between the two optimization problems.

For each value of $\mu_0$, we have an efficient portfolio. We then calculate its Sharpe Ratio. The maximum of the curve corresponds to the tangency portfolio.

### 3.0.5 Graphical Summary

The three representations make it possible to progressively understand the reasoning behind portfolio optimization.




The fundamental idea to remember is therefore :

| |
|---|
| Minimize risk for each target return<br>$\Longrightarrow$ Construct the efficient frontier<br><br>Maximize the Sharpe Ratio<br>$\Longrightarrow$ Identify the tangency portfolio |

The first formulation therefore makes it possible to construct the efficient frontier, while the second makes it possible to select, on this frontier, the portfolio that offers the best excess return per unit of risk when a risk-free asset is available.

### 3.0.6 SLSQP Method — Sequential Least Squares Programming

SLSQP is a constrained optimization method based on the principle of *Sequential Quadratic Programming* (SQP).
We seek to solve the problem

$$\min_x f(x) \qquad \text{subject to} \qquad g(x) = 0, \quad h(x) \geq 0. \tag{3.1}$$

**Principle** We start from an **initial point** $x_0$, chosen by the user. At iteration $k$, rather than directly searching for the new $x$, we seek a **displacement** $d$ from the current point :

$$\boxed{x = x_k + d}. \tag{3.2}$$

We then locally approximate the objective using a second-order Taylor expansion :

$$f(x_k + d) \simeq f(x_k) + \nabla f(x_k)^T d + \frac{1}{2} d^T H_k d. \tag{3.3}$$

Since $f(x_k)$ is constant with respect to $d$, the optimal displacement is obtained by solving :

$$\begin{aligned} \min_d \quad & \nabla f(x_k)^T d + \frac{1}{2} d^T H_k d \\ \text{subject to} \quad & g(x_k) + \nabla g(x_k)^T d = 0, \\ & h(x_k) + \nabla h(x_k)^T d \geq 0. \end{aligned} \tag{3.4}$$

The constraints are here **linearized** around $x_k$, while the objective is approximated by a quadratic function. The resulting problem is therefore a **quadratic programming (QP) problem**. The solution $d_k$ provides a direction of displacement. The new iterate is then :

$$\boxed{x_{k+1} = x_k + \alpha_k d_k}, \tag{3.5}$$

where $\alpha_k$ is determined by a line search in order to control the length of the displacement.

**Role of $H_k$ :** $H_k$ represents an **approximation of the Hessian of the Lagrangian**

$$\mathcal{L}(x, \lambda, \mu) = f(x) + \lambda^T g(x) - \mu^T h(x). \tag{3.6}$$

It therefore contains information about the **local curvature** of the problem. In practice, $H_0$ is often initialized with the identity matrix :

$$H_0 = I, \tag{3.7}$$

and then updated during the iterations using a quasi-Newton method, typically **BFGS**.

**Link with Gradient Descent :** Gradient descent uses a first-order approximation :

$$f(x_k + d) \simeq f(x_k) + \nabla f(x_k)^T d, \tag{3.8}$$

which leads to the direction :

$$d_k = -\nabla f(x_k). \tag{3.9}$$

SLSQP generalizes this idea to constrained problems : it searches for the **best local displacement** $d$ while taking into account both the **curvature** of the objective through $H_k$ and the **constraints** through their linearization.
Thus :

$$\boxed{\text{Gradient descent : } d_k = -\nabla f(x_k)} \tag{3.10}$$

whereas SLSQP searches for :

$$\boxed{d_k = \arg\min_d \left[ \nabla f(x_k)^T d + \frac{1}{2} d^T H_k d \right] \quad \text{subject to linearized constraints.}} \tag{3.11}$$

### 3.0.7 Some Remarks

In financial *Data Engineering* practice, raw data extracted from markets (via yFinance, for example) are captured at a **daily** frequency. However, performance, management mandates, and asset comparisons are universally expressed over an **annual** horizon. It is therefore essential to annualize the calculated statistics.

A standard trading year contains approximately **252 trading days** (excluding weekends and public holidays).

— Let $R_t$ be the daily log-return of an asset. Under the assumption of stationarity, the expected daily return is $\mu_d = E(R_t)$. The annual log-return is obtained by summing the daily log-returns over the 252 trading days, so the expectation follows by linearity :

$$\mu_{\text{annuel}} = E\left( \sum_{t=1}^{252} R_t \right) = 252 \times E(R_t) = \mu_d \times 252$$

— Under the standard assumption of independent and identically distributed (i.i.d.) daily returns (absence of temporal autocorrelation, a property related to weak-form market efficiency), the variance of a sum of random variables is equal to the sum of their variances :

$$\sigma^2_{\text{annuel}} = \text{Var}\left( \sum_{t=1}^{252} R_t \right) = \sum_{t=1}^{252} \text{Var}(R_t) = 252 \times \sigma_d^2$$

Therefore, for volatility (standard deviation), the relationship follows a **square-root-of-time** rule :

$$\sigma_{\text{annuel}} = \sqrt{252 \times \sigma_d^2} = \sigma_d \times \sqrt{252}$$

In Python code, directly multiplying the daily variance-covariance matrix `returns.cov()` by 252 makes it possible to annualize all variances (diagonal elements) and covariances (off-diagonal elements) at once according to this statistical rule :

$$\Sigma_{\text{annuel}} = 252 \times \Sigma_{\text{journalière}}.$$

### 3.0.8 Complete Scripted Implementation

Here is the clean and commented implementation of the complete script in Python :

```
# ============================================================
# 0. IMPORTING LIBRARIES
# ============================================================
import numpy as np
import pandas as pd
import yfinance as yf
import matplotlib.pyplot as plt
import scipy.optimize as sco

# ============================================================
# 1. DATA EXTRACTION AND CLEANING
# ============================================================

# List of assets composing our investment universe
assets = ['AAPL', 'MSFT', 'AMZN', 'NVDA']

# Retrieval of adjusted closing prices from Yahoo Finance
# for the period from 01/01/2021 to 01/01/2026.
# Adjusted prices account in particular for splits and dividends.
data = yf.download(
    assets,
    start='2021-01-01',
    end='2026-01-01'
)['Close']

# ------------------------------------------------------------
# Calculation of daily logarithmic returns
# ------------------------------------------------------------
# data.shift(1) shifts prices by one day:
#
#             P_t
# r_t = ln( -------- )
#             P_{t-1}
#
# dropna() removes the first observation because there is no
# P_{t-1} for the first day.
returns = np.log(data / data.shift(1)).dropna()

# ============================================================
# 2. ANNUALIZATION OF STATISTICS
# ============================================================
# We consider approximately 252 trading days per year.
TRADING_DAYS = 252

# ------------------------------------------------------------
# Annualized average return
```

```
# ------------------------------------------------------------

# returns.mean() calculates the average daily logarithmic return
# for each asset.
# We annualize by multiplying by 252:
# mu_i = 252 * average(r_i)
mean_returns = returns.mean() * TRADING_DAYS

# ------------------------------------------------------------
# Annualized covariance matrix
# ------------------------------------------------------------
# returns.cov() calculates the daily covariance matrix:
#
#                        Cov(r_1,r_1)   Cov(r_1,r_2) ...
# Sigma_daily = ...
# To annualize the covariance:
# Sigma = 252 * Sigma_daily
#
cov_matrix = returns.cov() * TRADING_DAYS


# ============================================================
# 3. MONTE CARLO SIMULATION
# ============================================================

# Number of random portfolios we will generate
num_portfolios = 10000

# We create a matrix of dimension 3 x 10,000.
# For each portfolio, we will store:
# results[0,i] = portfolio volatility
# results[1,i] = portfolio return
# results[2,i] = Sharpe Ratio
#
results = np.zeros((3, num_portfolios))

# This list will store the weights
# of each of the 10,000 portfolios.
weights_record = []

# Fixing the random seed to obtain
# the same results at each execution.
np.random.seed(42)

# ------------------------------------------------------------
# Generation of random portfolios
# ------------------------------------------------------------

for i in range(num_portfolios):

    # Generation of 4 positive random numbers,
    # one for each of the 4 assets.
    weights = np.random.random(len(assets))

    # ------------------------------------------------------
    # Weight normalization
    # ------------------------------------------------------
    # We impose:
    #
    # w_1 + w_2 + ... + w_N = 1
    # that is, 100% of the capital is invested.
```

```
    weights /= np.sum(weights)

    # We store the portfolio weights
    weights_record.append(weights)

    # -------------------------------------------------------
    # Portfolio return
    # -------------------------------------------------------
    # Mathematical formula:
    # mu_p = w^T mu
    #
    # that is:
    # mu_p = sum(w_i * mu_i)
    #
    p_ret = np.dot(weights, mean_returns)

    # -------------------------------------------------------
    # Portfolio volatility
    # -------------------------------------------------------
    # Portfolio variance:
    # sigma_p^2 = w^T Sigma w
    #
    # Volatility:
    # sigma_p = sqrt(w^T Sigma w)
    #
    p_vol = np.sqrt(
        np.dot(
            weights.T,
            np.dot(cov_matrix, weights)
        )
    )

    # -------------------------------------------------------
    # Sharpe Ratio
    # -------------------------------------------------------
    # General formula:
    #
    # Sharpe = (mu_p - R_f) / sigma_p
    #
    # Here, we assume R_f = 0.
    # Therefore:
    # Sharpe = mu_p / sigma_p
    #
    p_sharpe = p_ret / p_vol

    # Storing the results
    results[0, i] = p_vol
    results[1, i] = p_ret
    results[2, i] = p_sharpe

# ============================================================
# 4. NUMERICAL OPTIMIZATION
# ============================================================
# We define a function that calculates the three statistics
# of a portfolio from its weights.
def portfolio_stats(weights):

    # Transformation into a NumPy array
    weights = np.array(weights)

    # -------------------------------------------------------
```

```
    # Portfolio return
    # ------------------------------------------------------
    # mu_p = w^T mu
    #
    p_ret = np.dot(weights, mean_returns)

    # ------------------------------------------------------
    # Volatility
    # ------------------------------------------------------
    # sigma_p = sqrt(w^T Sigma w)
    #
    p_vol = np.sqrt(
        np.dot(
            weights.T,
            np.dot(cov_matrix, weights)
        )
    )

    # ------------------------------------------------------
    # Sharpe Ratio
    # ------------------------------------------------------
    # R_f = 0
    #
    # Sharpe = mu_p / sigma_p
    #
    p_sharpe = p_ret / p_vol

    # We return:
    # [return, volatility, Sharpe]
    return np.array([p_ret, p_vol, p_sharpe])

# ------------------------------------------------------------
# Optimization objective function
# ------------------------------------------------------------

def min_func_sharpe(weights):

    # We want to MAXIMIZE the Sharpe Ratio.
    # But scipy.optimize.minimize() seeks to MINIMIZE
    # a function.
    # Therefore:
    # maximize(Sharpe)
    # is equivalent to:
    # minimize(-Sharpe)
    #
    return -portfolio_stats(weights)[2]

# ============================================================
# OPTIMIZATION CONSTRAINTS
# ============================================================

# ------------------------------------------------------------
# Constraint: the sum of weights must equal 1
# ------------------------------------------------------------
# scipy requires a constraint of the form:
# f(x) = 0
# We therefore define:
# sum(w_i) - 1 = 0
# which is equivalent to:
# sum(w_i) = 1
#
```

```
230 constraints = (
231     {
232         'type': 'eq',
233         'fun': lambda x: np.sum(x) - 1
234     }
235 )
236 
237 # ------------------------------------------------------------
238 # Weight constraints
239 # ------------------------------------------------------------
240 # Each weight must satisfy:
241 # 0 <= w_i <= 1
242 # This means:
243 # - no short selling
244 # - no asset can have a weight greater than 100%
245 bounds = tuple(
246     (0, 1) for asset in range(len(assets))
247 )
248 
249 # ------------------------------------------------------------
250 # Algorithm starting point
251 # ------------------------------------------------------------
252 # We start with an equally weighted portfolio:
253 #
254 # w = (25%, 25%, 25%, 25%)
255 #
256 initial_guess = len(assets) * [1. / len(assets)]
257 
258 # ============================================================
259 # SOLVING THE OPTIMIZATION PROBLEM
260 # ============================================================
261 
262 # We seek the portfolio that minimizes:
263 #
264 #          - mu_p
265 # min =  -----------
266 #         sigma_p
267 #
268 # which is equivalent to maximizing the Sharpe Ratio:
269 #
270 #              mu_p
271 # max=  -----------
272 #            sigma_p
273 #
274 # subject to:
275 # sum(w_i) = 1
276 # and:
277 # 0 <= w_i <= 1
278 #
279 opt_results = sco.minimize(
280     min_func_sharpe,
281     initial_guess,
282     method='SLSQP',
283     bounds=bounds,
284     constraints=constraints
285 )
286 
287 # Retrieval of the optimal weights
288 optimal_weights = opt_results.x
289 
290 # ============================================================
```

```
# 5. PORTFOLIO VISUALIZATION
# ============================================================

plt.figure(figsize=(10, 6))

# ------------------------------------------------------------
# Scatter plot of the 10,000 simulated portfolios
# ------------------------------------------------------------
# X-axis:
# portfolio volatility = risk
# Y-axis:
# expected portfolio return
# Color:
# Sharpe Ratio
#
plt.scatter(
    results[0, :],        # Volatility
    results[1, :],        # Return
    c=results[2, :],      # Sharpe Ratio
    cmap='viridis',
    marker='o',
    s=10,
    alpha=0.3
)

# Color bar indicating the Sharpe Ratio value
plt.colorbar(
    label='Annualized Sharpe Ratio'
)

# ------------------------------------------------------------
# Addition of the optimal portfolio
# ------------------------------------------------------------

# portfolio_stats(optimal_weights) returns:
#
# [return, volatility, Sharpe]
# Therefore:
# [1] = volatility
# [0] = return
#
plt.scatter(
    portfolio_stats(optimal_weights)[1],
    portfolio_stats(optimal_weights)[0],
    color='red',
    marker='*',
    s=200,
    label='Max Sharpe (Optimum)'
)

# Titles and axes of the graph
plt.title(
    "Markowitz Portfolio \& Optimal Portfolio",
    fontsize=12
)

plt.xlabel(
    "Annualized Volatility (Risk)"
)

plt.ylabel(
```

```
    "Annualized Expected Return"
)

# Legend
plt.legend()

# Grid
plt.grid(
    True,
    linestyle='--',
    alpha=0.5
)

# Saving the graph
plt.savefig(
    'frontiere_efficiente.png'
)

# Closing the figure
plt.close()

# ============================================================
# 6. DISPLAYING THE RESULTS
# ============================================================

print("-" * 40)
print("OPTIMAL ALLOCATION (MAX SHARPE):")
print("-" * 40)

# We associate each asset with its optimal weight.
for asset, weight in zip(assets, optimal_weights):

    # :.2\% displays the weight as a
    # percentage with two decimal places.
    print(
        f"Optimal weight for {asset} : {weight:.2\%}"
    )

print("-" * 40)
```

## 3.1 Machine-Based Practical Work (Exercises)

### Exercise 1 : The Minimum Variance Portfolio (Global Minimum Variance - GMV)

1. Write a Python function `min_func_variance(weights)` that returns only the calculated portfolio volatility.
2. Use `scipy.optimize.minimize` with the `SLSQP` algorithm, the same bounds and constraints, to extract the weights of the GMV portfolio.
3. Add this point to the previous graph (use a blue square marker, for example) to distinguish it from the Max Sharpe portfolio.

### 3.1.1 Solution

1. Place after `min_func_sharpe()` :

```
# ------------------------------------------------------------
# Objective function: Global Minimum Variance Portfolio (GMV)
# ------------------------------------------------------------

def min_func_variance(weights):

    # We seek to MINIMIZE the portfolio volatility.
    #
    # portfolio_stats(weights) returns:
    # [return, volatility, Sharpe]
    #
    # We therefore retrieve only the volatility:
    return portfolio_stats(weights)[1]

```

2. Just after the optimization of the Max Sharpe portfolio :

```
# Retrieval of the optimal weights
optimal_weights = opt_results.x


# ============================================================
# OPTIMIZATION OF THE GMV PORTFOLIO
# ============================================================

# We now seek to directly minimize
# the portfolio volatility.
#
# Constraints:
#
# sum(w_i) = 1
#
# and:
#
# 0 <= w_i <= 1

gmv_results = sco.minimize(
    min_func_variance,
    initial_guess,
    method='SLSQP',
    bounds=bounds,
    constraints=constraints
)
```

```
27
28
29 # Retrieval of the optimal weights of the GMV portfolio
30 gmv_weights = gmv_results.x
```

`gmv_weights` contains the weights of the Global Minimum Variance Portfolio.

3. After the Max Sharpe portfolio, add the following code :

```
1  # ------------------------------------------------------------
2  # Addition of the GMV portfolio
3  # ------------------------------------------------------------
4
5  # portfolio_stats(gmv_weights) returns:
6  #
7  # [return, volatility, Sharpe]
8  #
9  # Therefore:
10 #
11 # [1] = volatility
12 # [0] = return
13
14 plt.scatter(
15     portfolio_stats(gmv_weights)[1],
16     portfolio_stats(gmv_weights)[0],
17     color='blue',
18     marker='s',
19     s=100,
20     label='GMV (Minimum Variance)'
21 )
```

### Exercise 2 : Integration of Allocation Limits (Regulatory Weight Constraints)

1. Modify the `bounds` argument of the optimizer so that no individual asset can represent more than 40% of the total portfolio, and none can fall below the minimum threshold of 5%.
2. Solve the Max Sharpe problem again under this geometric constraint.
3. Verify with a `print` that all calculated weights strictly satisfy :

$$0.05 \leq w_i \leq 0.40 \quad \forall i$$

### Solution

1. Replace,

```
1 bounds = tuple(
2     (0, 1) for asset in range(len(assets))
3 )
```

with

```
1 bounds = tuple(
2     (0.05, 0.40) for asset in range(len(assets))
3 )
```

2. Run the Max Sharpe optimization again

```
1 # ============================================================
2 # SOLVING THE MAX SHARPE PROBLEM UNDER CONSTRAINTS
3 # ============================================================
```

```
4
5 opt_results = sco.minimize(
6     min_func_sharpe,
7     initial_guess,
8     method='SLSQP',
9     bounds=bounds,
10    constraints=constraints
11 )
12
13 optimal_weights = opt_results.x
```

3. Add the following code :

```
1 # ============================================================
2 # CONSTRAINT VERIFICATION
3 # ============================================================
4
5 print("-" * 40)
6 print("CONSTRAINT VERIFICATION:")
7 print("-" * 40)
8
9 for asset, weight in zip(assets, optimal_weights):
10
11     print(
12         f"{asset} : {weight:.2\%} "
13         f"| Satisfies bounds : {0.05 <= weight <= 0.40}"
14     )
15
16 print("-" * 40)
17
18 print(
19     "All weights satisfy the constraints:",
20     np.all((optimal_weights >= 0.05) &
21            (optimal_weights <= 0.40))
22 )
```

### Exercise 3 – How Stable Is a Markowitz Portfolio ?

Consider the following six assets :

$$\text{AAPL}, \quad \text{MSFT}, \quad \text{AMZN}, \quad \text{NVDA}, \quad \text{JPM}, \quad \text{XOM}.$$

Use daily adjusted closing prices over the last five years and Assume :

$$w_i \geq 0, \qquad \sum_{i=1}^{6} w_i = 1, \qquad R_f = 0.$$

#### A – Maximum Sharpe Ratio Portfolio

1. Compute daily logarithmic returns :

$$r_{i,t} = \ln\left(\frac{P_{i,t}}{P_{i,t-1}}\right).$$

2. Annualize the expected returns and covariance matrix :

$$\mu_{\text{annuel}} = 252\,\mu_{\text{daily}},$$

$$\Sigma_{\text{annuel}} = 252\,\Sigma_{\text{daily}}.$$

3. Using the SLSQP method, determine the portfolio maximizing :

$$\max_{w} \frac{w^\top \mu}{\sqrt{w^\top \Sigma w}}$$

subject to :

$$\sum_{i=1}^{6} w_i = 1, \qquad w_i \geq 0.$$

4. Report :
   — the optimal weights ;
   — the expected return ;
   — the volatility ;
   — the Sharpe ratio.

**B – Does the Estimation Period Matter ?**

1. Repeat the optimization using three different estimation windows :

| 12 months | 36 months | 60 months |
|---|---|---|

   For each window, compute the maximum-Sharpe portfolio.
2. Present the weights in the following table :

| | *AAPL* | *MSFT* | *AMZN* | *NVDA* | *JPM* | *XOM* |
|---|---|---|---|---|---|---|
| 12 months | | | | | | |
| 36 months | | | | | | |
| 60 months | | | | | | |

   Then create a graph comparing the three allocations.

**C – Interpretation**

Answer the following questions.

1. Question 1.
   — Are the three optimal portfolios similar or different ?
   — Which assets have the most stable weights ?
   — Which assets have the largest changes ?
2. Question 2.
   — Why can changing the estimation period lead to different optimal portfolios ?
   — Discuss the role of :

$$\mu \qquad \text{and} \qquad \Sigma.$$

3. Question 3.
   Suppose that two estimation periods produce very different portfolios. Does this necessarily mean that the Markowitz model is wrong ? Explain your answer.

4. Question 4.
   In practice, why might an investor prefer to impose additional allocation constraints such as

$$0.05 \leq w_i \leq 0.30$$

   instead of using the unconstrained optimal portfolio ?

# Chapitre 4

# Asset Pricing Models (CAPM) and Factor Analysis

## 4.1 Introduction

In the previous chapter, we saw how to construct a portfolio by optimizing the return-risk trade-off using the Markowitz approach. However, a fundamental question remains : **how can we assess whether the return of an individual asset or portfolio is justified given the risk taken ?**

This chapter formalizes the transition from a purely mathematical approach (Markowitz) to a structural approach to performance. We will first study the Capital Asset Pricing Model (CAPM) to decompose risk into systematic and idiosyncratic components. We will then extend this framework to multi-factor models (Fama-French) before seeing how to move away from rigid economic theories through a purely statistical method : Principal Component Analysis (PCA).

**The market portfolio :** It is a theoretical portfolio that represents the entire financial market. It contains all assets available in the market, with each asset weighted according to its relative market value.
For a publicly traded company, the **market capitalization** is defined as :

$$\text{Capitalisation}_i = P_i \times N_i,$$

where :

— $P_i$ represents the price of one share of company $i$ ;
— $N_i$ represents the total number of shares outstanding.

Market capitalization therefore represents the total value of the company in the market. It should not be confused with the price of a single share. The weight of asset $i$ in the market portfolio is then given by :

$$w_i^M = \frac{\text{Capitalisation}_i}{\sum_{j=1}^{n} \text{Capitalisation}_j}.$$

We can therefore verify that :

$$\sum_{i=1}^{n} w_i^M = 1.$$

For example, if the total market capitalization is 10 billion euros and a company has a capitalization of 1 billion euros, its weight in the market portfolio is :

$$w_i^M = \frac{1}{10} = 10\%.$$

Thus, the market portfolio does not assign greater weight to assets with the highest expected return. The weights are determined by the relative size of companies in the market.

**Link with the tangency portfolio :** The tangency portfolio is obtained within the framework of mean-variance optimization. When a risk-free asset is available, it corresponds to the risky portfolio that maximizes the Sharpe ratio :

$$\max_w \frac{w^\top \mu - R_f}{\sqrt{w^\top \Sigma w}}.$$

The tangency portfolio therefore depends on $\mu, \Sigma$ and $R_f$.
The market portfolio and the tangency portfolio are therefore, in general, two different concepts :

$$\boxed{\text{Market portfolio} \quad \neq \quad \text{Tangency portfolio}}$$

The market portfolio is constructed from market capitalizations, whereas the tangency portfolio is obtained through an optimization procedure.
However, within the theoretical framework of the **Capital Asset Pricing Model (CAPM)**, under its assumptions, these two portfolios coincide :

$$\boxed{w^M = w^T}$$

where $w^M$ represents the market portfolio and $w^T$ the tangency portfolio.
This equality is a central result of the CAPM. In this theoretical framework, all investors hold the same risky portfolio, namely the market portfolio. They differ only in the proportion invested in the risk-free asset.
The relationship between the two concepts can therefore be summarized as follows :

| Market portfolio capitalization-weighted | $\overset{\text{CAPM Assumptions}}{=}$ | Tangency portfolio maximizing the Sharpe ratio |
|---|---|---|

In practice, the tangency portfolio calculated from historical data does not necessarily coincide with the observed market portfolio. The equality between the two is a theoretical result of the CAPM and relies on its assumptions.

## 4.2 The Mathematical Framework

### 4.2.1 The CAPM (Capital Asset Pricing Model)

The CAPM (or MEDAF) states that the expected return of a risky asset $i$, denoted $E(R_i)$, is determined solely by its exposure to systematic market risk. The fundamental formula is :

$$E(R_i) = R_f + \beta_i \left(E(R_m) - R_f\right).$$

Where :

— $R_f$ is the risk-free rate.

— $E(R_m)$ is the expected return of the market portfolio (e.g., S&P 500 or CAC 40).
— $(E(R_m) - R_f)$ represents the market risk premium.
— $\beta_i$ measures the sensitivity and contribution of the asset to market risk.

The coefficient $\beta_i$ is mathematically defined as the ratio of the covariance between the asset's return and the market return, divided by the market variance :

$$\beta_i = \frac{\text{Cov}(R_i, R_m)}{\text{Var}(R_m)}.$$

**Linear Regression and Performance Decomposition ($\alpha$ and $\beta$)** In financial practice, we estimate this model through a standard linear regression (*Ordinary Least Squares - OLS*) on historical data :

$$R_{i,t} - R_{f,t} = \alpha_i + \beta_i(R_{m,t} - R_{f,t}) + \epsilon_{i,t}.$$

Where :

— $\alpha_i$ (Jensen's Alpha) represents the abnormal excess return of the asset. A positive and statistically significant alpha indicates a skilled portfolio manager or an exploitable market anomaly.
— $\epsilon_{i,t}$ is the error term, representing the risk specific to the company (e.g., a scandal or a technological success specific to the company) ; in this model, it is generally assumed that $E(\epsilon_{i,t}) = 0$. This risk can be eliminated through diversification, unlike systematic risk ($\beta_i$).

### 4.2.2 Multi-Factor Models : The Fama-French Three-Factor Model

Given the empirical limitations of the single-factor CAPM, Eugene Fama and Kenneth French (1993) introduced two additional macroeconomic risk factors to explain returns :

$$R_{i,t} - R_{f,t} = \alpha_i + \beta_i(R_{m,t} - R_{f,t}) + s_i\mathbf{SMB}_t + h_i\mathbf{HML}_t + \epsilon_{i,t}.$$

Where :

— **SMB** (*Small Minus Big*) : the SMB factor represents the **size premium**. It measures the difference in return between a portfolio composed of small-capitalization companies (*Small*) and a portfolio composed of large-capitalization companies (*Big*).
In simplified form, the SMB factor can be represented as :

$$SMB = R_{\text{Small}} - R_{\text{Big}}.$$

A positive value of $SMB$ means that, over the period considered, small-cap stocks achieved a higher return than large-cap stocks. Conversely, a negative value indicates that large-cap stocks achieved a higher return.
The existence of a difference in returns between small and large companies constitutes an important empirical observation in the study of financial markets. This difference is often referred to as the **size premium**. It may notably be interpreted as compensation associated with certain characteristics and risks specific to small companies, such as lower liquidity, greater sensitivity to financing conditions, or greater vulnerability to economic conditions.
It is nevertheless important to distinguish this empirical observation from its economic interpretation : a positive size premium does not necessarily imply that the outperformance of small-cap stocks is exclusively compensation for illiquidity risk.

— **HML** (*High Minus Low*) : the HML factor represents the **value premium**. It measures the difference in return between companies with a high *Book-to-Market* ratio (*High*) and those with a low *Book-to-Market* ratio (*Low*).
In simplified form :

$$HML = R_{\text{High B/M}} - R_{\text{Low B/M}}.$$

The *Book-to-Market* ratio compares a company's book value of equity with its market value :

$$\text{Book-to-Market} = \frac{\text{Book value of shareholders' equity (Assets-debts)}}{\text{Market capitalization}}.$$

Companies with a high *Book-to-Market* ratio are generally classified as *Value* companies. Their book value is relatively large compared with their market valuation.
Conversely, companies with a low *Book-to-Market* ratio are generally associated with the *Growth* style. Their market valuation is relatively high compared with their book value.
A positive value of $HML$ means that, over the period considered, *Value* companies achieved a higher return than *Growth* companies. A negative value corresponds to the opposite situation.
The HML factor therefore makes it possible to measure a portfolio's exposure to the value factor. Positive exposure to the HML factor corresponds to a more *Value*-oriented position, whereas negative exposure corresponds more closely to a *Growth* orientation. "'

### 4.2.3 Non-Parametric Approach : Principal Component Analysis (PCA)

Unlike factor approaches that impose factors defined a priori by economic theory (for example, Market, Size, or Style), PCA is a descriptive statistical method used to extract the main sources of common variation from a large set of assets.
The first step is to start from asset prices and calculate their returns :

$$R_{i,t} = \frac{P_{i,t} - P_{i,t-1}}{P_{i,t-1}},$$

where $P_{i,t}$ denotes the price of asset $i$ at date $t$.
We then obtain a return matrix containing the time-series observations for all assets. From these returns, we construct the covariance matrix :

$$\Sigma_{ij} = \text{Cov}(R_i, R_j).$$

The matrix $\Sigma$ therefore contains information about how the returns of different assets move together. It is precisely this covariance structure that PCA seeks to summarize.
Mathematically, PCA consists of diagonalizing the covariance matrix by finding its eigenvalues $\lambda_k$ and associated eigenvectors $V_k$ :

$$\Sigma V_k = \lambda_k V_k.$$

Each eigenvector defines a particular direction in return space. This direction makes it possible to construct a linear combination of returns, called a **principal component** :

$$PC_k = V_k^\top R,$$

where $R$ denotes the vector of asset returns.

Thus, principal components are themselves factors constructed from the observed returns. For example, the first principal component can be written as :

$$PC_1 = v_{11}R_1 + v_{21}R_2 + \cdots + v_{N1}R_N.$$

The coefficients $v_{i1}$ are called *loadings* and indicate the contribution of each asset to the construction of the factor.
The eigenvalues $\lambda_k$ measure the amount of variance explained by each component. The proportion of variance explained by component $k$ is given by :

$$\frac{\lambda_k}{\sum_{j=1}^{N} \lambda_j}.$$

When the first principal component has loadings with the same sign and comparable magnitude across many assets, it can be interpreted empirically as a factor representing a common market movement.
However, it is important to distinguish this statistical factor from the theoretical market portfolio. PCA does not know the economic weights of the market and does not directly construct the market portfolio. It simply seeks the directions that explain the greatest amount of variance in the observed returns.
The approach can therefore be summarized as follows :

$$\boxed{\text{Prices} \rightarrow \text{Returns} \rightarrow \text{Covariances} \rightarrow \text{PCA} \rightarrow \text{Statistical Factors}}$$

### 4.2.4 Some Remarks

The main objective of PCA is not to directly predict asset returns, but to **summarize the information contained in a large number of returns through a reduced number of statistical factors**.
Suppose that we observe the returns of $N$ assets :

$$R_t = \begin{pmatrix} R_{1,t} \\ R_{2,t} \\ \vdots \\ R_{N,t} \end{pmatrix}.$$

When assets move in a similar manner, their returns exhibit significant correlations and covariances. A large part of the variation observed in the $N$ returns can then be explained by a few common movements.
PCA seeks precisely to identify these common movements. It therefore makes it possible to move from a large number of observed variables to a reduced number of factors :

$$\boxed{R_1, R_2, \ldots, R_N \quad \longrightarrow \quad PC_1, PC_2, \ldots, PC_K} \qquad K \ll N.$$

For example, instead of analyzing separately the returns of 100 assets, PCA may make it possible to summarize a large part of their variation through only a few principal components.
The first principal component is the linear combination of returns that explains the largest possible share of the variance. The second explains the largest remaining share of variance, subject to being orthogonal to the first, and so on.
The first components can therefore be interpreted as the main **statistical factors** present in the data. For example, if the coefficients of $PC_1$ are positive and relatively similar for a large number of stocks, this component may represent a common movement of the entire market.

However, it is important to distinguish this approach from that of the CAPM or the Fama-French model. In these models, the factors are defined **a priori** based on an economic theory :

$$\begin{aligned} \text{CAPM} &\rightarrow \text{Market Factor,} \\ \text{Fama-French} &\rightarrow \text{Market, SMB, HML.} \end{aligned}$$

With PCA, by contrast, the factors are not imposed in advance. They are **extracted directly from the data** based on the covariance structure of returns :

| Returns → Covariance Matrix → PCA → Statistical Factors |
|---|

PCA therefore constitutes a complementary approach to economic factor models. It makes it possible to empirically search for the main sources of common variation without assuming their economic interpretation in advance.
Once these statistical factors have been identified, it is also possible to use them in a linear regression model. For example, with the first three principal components :

$$R_{i,t} = \alpha_i + \beta_{i1} PC_{1,t} + \beta_{i2} PC_{2,t} + \beta_{i3} PC_{3,t} + \varepsilon_{i,t}.$$

In this approach, PCA is therefore first used to **construct the factors**, while regression is then used to **measure each asset's exposure to these factors** through the coefficients $\beta_{ik}$.
Thus, the two steps answer two different questions :

| | |
|---|---|
| PCA : | What are the main common factors present in the data ? |
| Regression : | How is each asset exposed to these factors ? |

## 4.3 Python Implementation

### 4.3.1 Key Functions

**statsmodels.api.add_constant(X) :** By default, the linear regression function of the **statsmodels** library fits a line that necessarily passes through the origin $(0, 0)$. Mathematically, this would amount to forcing $\alpha$ to be zero from a statistical perspective ($\alpha = 0$).

To correctly estimate the equation $Y = \alpha + \beta X$, a column filled with 1s must be added to our matrix of independent variables $X$. This is the role of `add_constant`. Without this function, the calculated $\beta$ coefficient would be biased because all residual excess performance of the asset would be artificially incorporated into the slope of the line.

**sm.OLS(Y, X).fit() :** This method implements the Ordinary Least Squares (OLS) estimator. The objective is to estimate the model coefficients by minimizing the sum of squared residuals. For each observation, the residual is the difference between the observed value and the value predicted by the model :

$$\epsilon_t = Y_t - \hat{Y}_t$$

OLS therefore seeks the coefficients that minimize :

$$\min \sum_{t=1}^{T} \epsilon_t^2$$

Graphically, this corresponds to finding the regression line that minimizes the sum of the squared vertical distances between the observations and the estimated line.

In the CAPM framework, we can write :

$$R_{i,t} - R_{f,t} = \alpha_i + \beta_i(R_{M,t} - R_{f,t}) + \epsilon_t$$

The `.fit()` method then estimates $\alpha$ and $\beta$ ($\alpha = model.params[0]$ and $\beta = model.params[1]$) and returns various statistics used to evaluate the regression, including *t-statistics*, *p-values*, and $R^2$.

**The t-statistic (*t-stat*) :** The t-statistic measures how far an estimated coefficient is from zero relative to the uncertainty of the estimate :

$$t = \frac{\hat{\theta}}{SE(\hat{\theta})}$$

For example, if :

$$\hat{\beta} = 1.20 \qquad \text{and} \qquad SE(\hat{\beta}) = 0.10$$

then :

$$t = \frac{1.20}{0.10} = 12$$

A large absolute value of $t$ indicates that the coefficient is relatively far from zero compared with its estimation uncertainty. The associated *p-value* can then be used to test the hypothesis $H_0 : \beta = 0$.

**The coefficient of determination $R^2$ :** The $R^2$ measures the proportion of the variability in $Y$ explained by the model :

$$R^2 = 1 - \frac{\sum_{t=1}^{T}(Y_t - \hat{Y}_t)^2}{\sum_{t=1}^{T}(Y_t - \bar{Y})^2}$$

For example, if $R^2 = 0.64$, the model explains 64% of the observed variability in $Y$ in the sample, while the remaining 36% is not explained by the variables included in the model.

Thus, in a CAPM regression :

- $\hat{\alpha}$ and $\hat{\beta}$ are the estimated coefficients ;
- the *t-stat* measures their distance from zero relative to their uncertainty ;
- the *p-value* is used to test whether they are statistically different from zero ;
- $R^2$ indicates the proportion of return variability explained by the model.

**PCA(n_components=k)** From the `sklearn.decomposition` package, this class configures the dimensionality reduction algorithm. The `n_components` argument defines the number of eigenvectors that we wish to retain.

The execution takes place in three phases automated by the `.fit()` function :

1. Internal calculation of the covariance matrix of the asset basket.
2. Singular Value Decomposition (*Singular Value Decomposition-SVD*) to extract the orthogonal axes maximizing residual variance.
3. Ordering of the components in decreasing order of importance (from the highest eigenvalue to the lowest).

**pca.components** This attribute stores a matrix containing the **factor loadings**. Each row corresponds to a factor (PC1, PC2...) and each column contains the asset's attribution coefficient within that factor.

If all the values in the first row (`pca.components_[0]`) have the same sign and similar magnitude, this validates the financial interpretation : the first factor acts as an **implicit market index** that drives all stocks in the same direction.

**Example of Interpreting `pca.components_` :** To better understand the interpretation of the factors obtained through Principal Component Analysis (PCA), consider a portfolio composed of three stocks : Apple, Microsoft, and Tesla.
Suppose that we have daily returns for these three stocks and apply PCA while requesting two principal components :

```
pca = PCA(n_components=2)
pca.fit(returns)

print(pca.components_)
```

Suppose that Python returns the following matrix :

$$\texttt{pca.components_} = \begin{pmatrix} 0.60 & 0.55 & 0.58 \\ -0.70 & 0.10 & 0.70 \end{pmatrix}$$

The columns correspond to the different assets, while the rows correspond to the two principal components :

| | Apple | Microsoft | Tesla |
|---|---|---|---|
| $PC_1$ | 0.60 | 0.55 | 0.58 |
| $PC_2$ | $-0.70$ | 0.10 | 0.70 |

**Interpretation of the first component :** The first row corresponds to $PC_1$. The three coefficients are positive and relatively close to one another. This means that the three stocks contribute in the same direction to this component.
This component can be represented, in a simplified way, as :

$$PC_{1,t} = 0.60R_{\text{Apple},t} + 0.55R_{\text{Microsoft},t} + 0.58R_{\text{Tesla},t}$$

The subscript $t$ is important : it indicates that the stock returns change over time. Thus, the coefficients 0.60, 0.55, and 0.58 are determined by the PCA, while the returns $R_{\text{Apple},t}$, $R_{\text{Microsoft},t}$, and $R_{\text{Tesla},t}$ vary from one day to another.
Consequently, the value of the factor $PC_1$ also changes over time.

**Example of the factor's evolution over time :** Suppose that the observed returns over three days are :

| | Apple | Microsoft | Tesla |
|---|---|---|---|
| Day 1 | 1.0% | 0.8% | 1.2% |
| Day 2 | $-0.5\%$ | $-0.3\%$ | $-0.7\%$ |
| Day 3 | 2.0% | 1.5% | 2.3% |

For each date, the value of the factor $PC_1$ is obtained by combining the observed returns with the PCA coefficients.
For example, for the first day :

$$PC_{1,1} = 0.60(1.0\%) + 0.55(0.8\%) + 0.58(1.2\%)$$

For the second day :

$$PC_{1,2} = 0.60(-0.5\%) + 0.55(-0.3\%) + 0.58(-0.7\%)$$

And for the third day :

$$PC_{1,3} = 0.60(2.0\%) + 0.55(1.5\%) + 0.58(2.3\%)$$

We therefore obtain a time series for the factor $PC_1$ :

| Date | $PC_1$ |
|---|---|
| Day 1 | factor value on Day 1 |
| Day 2 | factor value on Day 2 |
| Day 3 | factor value on Day 3 |

Therefore, even though the coefficients in `pca.components_` remain fixed, the value of the factor $PC_1$ changes over time because the returns of the underlying assets change over time.

**Interpretation of the second component** The second row corresponds to $PC_2$ :

$$PC_{2,t} = -0.70R_{\text{Apple},t} + 0.10R_{\text{Microsoft},t} + 0.70R_{\text{Tesla},t}$$

Here, the coefficients associated with Apple and Tesla have opposite signs. Therefore, the component $PC_2$ can represent a contrast between the movements of these two assets.
Since the coefficient associated with Microsoft is close to zero, its contribution to this component is relatively small.
The following intuitive rule can be useful :

- coefficients with the same sign indicate that the assets contribute in the same direction to the factor ;
- coefficients with opposite signs indicate that the factor contrasts the movements of certain assets ;
- a coefficient close to zero indicates that the asset contributes relatively little to the component.

**Distinguishing between factors and coefficients :** It is important to distinguish between the *loadings* and the factors themselves.
In the previous example, the values 0.60, 0.55, and 0.58 are the coefficients (or *loadings*) associated with $PC_1$.
These coefficients indicate **how each asset contributes to the factor**. They do not represent the performance of the asset and do not change from one date to another when the PCA is estimated only once over the entire sample.
The factor $PC_1$, on the other hand, is a time series :

$$PC_1 = (PC_{1,1}, PC_{1,2}, PC_{1,3}, \ldots, PC_{1,T})$$

Each value corresponds to the value of the factor at a given date.
In Python, the factor values can be obtained using :

```
factors = pca.transform(returns)
```

They can then be organized into a table :

```
factor_df = pd.DataFrame(
    factors,
    index=returns.index,
    columns=["PC1", "PC2"]
)
```

This produces a structure such as :

| Date | $PC_1$ | $PC_2$ |
|---|---|---|
| 01/01 | 0.021 | −0.013 |
| 02/01 | −0.014 | 0.008 |
| 03/01 | 0.035 | 0.019 |
| 04/01 | −0.006 | −0.022 |

This table therefore represents the time evolution of the factors extracted by PCA. The distinction can be summarized as follows :

`pca.components_` $\longrightarrow$ composition of the factors

`pca.transform(returns)` $\longrightarrow$ factor values over time

In other words, `pca.components_` answers the question :

*"How are the assets combined to construct the factor ?"*

whereas `pca.transform(returns)` answers the question :

*"What is the value of this factor at each date ?"*

**Evolution of the coefficients over time** Finally, it is important to distinguish between two situations.
If PCA is estimated only once over the entire sample, the coefficients in `pca.components_` remain fixed. It is the value of the factors that changes over time.
However, if PCA is re-estimated regularly, for example every year or using a rolling window, the coefficients themselves may also change :

$$\text{PCA}_{t_1} \rightarrow \text{coefficients}_{t_1}$$

$$\text{PCA}_{t_2} \rightarrow \text{coefficients}_{t_2}$$

This makes it possible to study whether the factor structure of the portfolio changes over time.

### 4.3.2 Python Code

The following script retrieves data for an asset (`AAPL`) and its market index (`^GSPC` for the S&P 500) to estimate the CAPM using `statsmodels`, and then applies PCA using `scikit-learn` to a subset of assets.

```
import numpy as np
import pandas as pd
import yfinance as yf
import statsmodels.api as sm
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt


# ======================================================================
# 1. CAPM ESTIMATION VIA STATSMODELS
# ======================================================================

```

```
13 # ---------------------------------------------------------------------
14 # 1.1 Data retrieval
15 # ---------------------------------------------------------------------
16 # We select:
17 #   - AAPL: the asset we want to analyze
18 #   - ^GSPC: the S&P 500, used here as a proxy for the market portfolio
19 #
20 # The theoretical market portfolio contains all assets
21 # in the economy, weighted by their market capitalization.
22 #
23 # In practice, it is not directly observable.
24 # An equity index is therefore often used as an approximation.
25 #
26 # Here:
27 #
28 #       R_M = R_S&P500
29 #
30 # where R_M denotes the return of the market portfolio.
31
32 tickers = ['AAPL', '^GSPC']
33
34 data = yf.download(tickers, start='2021-01-01', end='2026-01-01')['Close']
35
36
37 # ---------------------------------------------------------------------
38 # 1.2 Calculation of daily returns
39 # ---------------------------------------------------------------------
40 # pct_change() calculates:
41 #
42 #       R_t = (P_t - P_{t-1}) / P_{t-1}
43 #
44 # and dropna() removes the first observation, for which a return
45 # cannot be calculated.
46
47 returns = data.pct_change().dropna()
48
49
50 # ---------------------------------------------------------------------
51 # 1.3 Definition of the CAPM variables
52 # ---------------------------------------------------------------------
53 # We seek to explain the return of AAPL stock
54 # using the market return.
55 #
56 # Dependent variable:
57 #
58 #       Y = R_AAPL
59 #
60 # Explanatory variable:
61 #
62 #       X = R_S&P500 = R_M
63 #
64 # Simplifying assumption:
65 #
66 #       R_f = 0
67 #
68 # The CAPM model is therefore written:
69 #
70 #       R_AAPL,t = alpha + beta * R_M,t + epsilon_t
71 #
72 # In a complete version with a risk-free rate:
73 #
```

```
#       R_AAPL,t - R_f,t = alpha + beta (R_M,t - R_f,t) + epsilon_t

Y = returns['AAPL']
X = returns['^GSPC']


# ----------------------------------------------------------------------
# 1.4 Adding the constant
# ----------------------------------------------------------------------
# statsmodels does not automatically add a constant.
#
# The constant corresponds here to the alpha of the model.
#
# We therefore obtain:
#
#       X_with_const = [1, R_M]
#
# which makes it possible to estimate:
# R_AAPL = alpha + beta * R_M + epsilon

X_with_const = sm.add_constant(X)


# ----------------------------------------------------------------------
# 1.5 Estimation using OLS
# ----------------------------------------------------------------------
# OLS = Ordinary Least Squares
#
# The model estimates the alpha and beta coefficients that minimize
# the sum of squared errors.

model = sm.OLS(Y, X_with_const).fit()

# Display the complete statistical summary
print(model.summary())


# ----------------------------------------------------------------------
# 1.6 Extraction of the main results
# ----------------------------------------------------------------------

# Daily alpha:
# constant estimated by the regression

alpha_journalier = model.params['const']

# Beta: sensitivity of AAPL's return to market variations.
#
# For example: beta = 1.2 means that, on average, a 1 \% change in
# the market is associated with a 1.2 \% change in AAPL,
# all else being equal.

beta_estime = model.params['^GSPC']


# ----------------------------------------------------------------------
# 1.7 Annualization of alpha
# ----------------------------------------------------------------------
# Alpha is estimated at a daily frequency.
#
# Here, we use an approximate linear annualization:
```

```
135
136 alpha_annuel = alpha_journalier * 252
137
138
139 # ---------------------------------------------------------------------
140 # 1.8 Display of the results
141 # ---------------------------------------------------------------------
142
143 print("\n--- CAPM RESULTS ---")
144
145 print(f"Beta (Systematic Risk) : {beta_estime:.4f}")
146
147 print(f"Annual Alpha (Performance) : {alpha_annuel:.2\%}")
148
149 print(
150     f"R-squared (Variance explained by the market) : "
151     f"{model.rsquared:.2\%}"
152 )
153
154
155 # =====================================================================
156 # 2. EXTRACTION OF RISK FACTORS VIA PCA
157 # =====================================================================
158
159 # ---------------------------------------------------------------------
160 # 2.1 Construction of a diversified basket of assets
161 # ---------------------------------------------------------------------
162 # Unlike the CAPM, we do not define a market portfolio
163 # in advance here.
164 #
165 # We simply provide PCA with a set of assets:
166 #
167 #       AAPL, MSFT, AMZN, NVDA, JPM, V, XOM, PG
168 #
169 # PCA will search for the main sources of common variation
170 # present in their returns.
171
172 basket = ['AAPL', 'MSFT','AMZN','NVDA','JPM','V','XOM', 'PG']
173
174
175 # ---------------------------------------------------------------------
176 # 2.2 Price retrieval
177 # ---------------------------------------------------------------------
178
179 basket_data = yf.download(
180     basket,
181     start='2021-01-01',
182     end='2026-01-01'
183 )['Close']
184
185
186 # ---------------------------------------------------------------------
187 # 2.3 Return calculation
188 # ---------------------------------------------------------------------
189
190 basket_returns = basket_data.pct_change().dropna()
191
192
193 # ---------------------------------------------------------------------
194 # 2.4 PCA
195 # ---------------------------------------------------------------------
```

```
# Here we ask PCA to extract the first 3 principal components.
#
# Each component is a linear combination of the returns
# of the different assets.
#
# For example, the first factor can be written:
#
#       PC1 = w1 R_AAPL + w2 R_MSFT + ... + w8 R_PG
#
# The coefficients w_i are called "loadings".
#
# IMPORTANT:
# sklearn's PCA centers the variables by default,
# meaning that it subtracts their mean.
#
# It does NOT automatically standardize the variables
# to unit variance.

pca = PCA(n_components=3)

# Estimation of the principal components

pca.fit(basket_returns)


# ------------------------------------------------------------------------
# 2.5 Explained variance
# ------------------------------------------------------------------------
# explained_variance_ratio_ indicates the proportion of total variance
# explained by each component.
#
# Example: PC1: 50 \%, PC2: 20 \% and PC3: 10 \%
# means that the first three factors together explain
# 80 \% of the total return variance.

explained_variance = pca.explained_variance_ratio_


# ------------------------------------------------------------------------
# 2.6 Display of explained variance
# ------------------------------------------------------------------------

print("\n--- PCA RESULTS ---")

for i, variance in enumerate(explained_variance):

    print(
        f"Principal Factor {i+1} "
        f"(PC{i+1}) explains : "
        f"{variance:.2\%} of total variance"
    )


# Total variance explained by the first 3 factors

print(
    f"Total variance explained by the first 3 factors : "
    f"{np.sum(explained_variance):.2\%}"
)


```

```
# ======================================================================
# 3. INTERPRETATION OF THE FIRST FACTOR: PC1
# ======================================================================

# ----------------------------------------------------------------------
# 3.1 Extraction of PC1 loadings
# ----------------------------------------------------------------------
# components_[0] contains the coefficients of the first principal component.
#
# Each coefficient indicates the contribution of the asset
# to the construction of PC1.

loadings_pc1 = pd.Series(
    pca.components_[0],
    index=basket)


# ----------------------------------------------------------------------
# 3.2 Display of the loadings
# ----------------------------------------------------------------------

print("\n--- PC1 LOADINGS ---")

print(loadings_pc1)


# ----------------------------------------------------------------------
# 3.3 Visualization of the loadings
# ----------------------------------------------------------------------
# The graph shows which assets contribute positively or negatively to PC1.

plt.figure(figsize=(8, 4))

loadings_pc1.plot(
    kind='bar',
    color='darkblue'
)

plt.title(
    "Factor Loadings of the First Principal Factor (PC1)"
)

plt.ylabel("Weight in the factor")

plt.axhline(
    0,
    color='black',
    linestyle='--',
    linewidth=0.8
)

plt.grid(
    axis='y',
    alpha=0.3
)

plt.tight_layout()

plt.savefig('pca_loadings.png')

plt.close()
```

```
318
319
320 # ======================================================================
321 # 4. LINK BETWEEN THE CAPM, THE MARKET PORTFOLIO AND PCA
322 # ======================================================================
323
324 # ----------------------------------------------------------------------
325 # CAPM
326 # ----------------------------------------------------------------------
327 #
328 # In the CAPM, the risk factor is defined in advance:
329 #
330 #       R_M = return of the market portfolio
331 #
332 # Since the true market portfolio is not directly observable,
333 # we use here: R_M = R_S&P500
334 #
335 # This is precisely the variable that appears in:
336 # X = returns['^GSPC']
337 #
338 # The model then estimates:
339 # R_AAPL = alpha + beta R_M + epsilon
340 #
341 # ----------------------------------------------------------------------
342 # PCA
343 # ----------------------------------------------------------------------
344 #
345 # In PCA, the factor is NOT defined in advance.
346 # The factors are constructed directly from the data.
347 # PCA searches for the directions that explain the most variance
348 # in asset returns.
349 #
350 # We obtain: PC1, PC2, PC3, ... which are statistical factors.
351 #
352 # PC1 may sometimes resemble a market factor if the different
353 # assets have loadings with the same sign and comparable magnitude.
354 #
355 # But: PC1 is automatically different from the market portfolio
356 # and the market portfolio is not explicitly used
357 # in this part of the code.
358
359
360 # ======================================================================
361 # 5. CONCEPTUAL SUMMARY
362 # ======================================================================
363
364 print("\n--- CONCEPTUAL SUMMARY ---")
365
366 print("""
367 CAPM:
368     Factor = market
369     Proxy used here = S&P 500 (^GSPC)
370     Objective = measure AAPL's exposure to market risk
371
372 PCA:
373     Factors = statistically constructed from returns
374     Factors extracted here = PC1, PC2 and PC3
375     Objective = identify the main common sources
376                 of asset variation
377
378 Market portfolio:
```

```
    It is NOT explicitly constructed as a vector of weights
    in this code.

    It intervenes indirectly in the CAPM through:
        R_M = R_S&P500

    On the other hand, it is not directly used in the PCA.
""")
```

## 4.4 Exercises

**Exercise 1 : Linear Regression and Estimation of CAPM Parameters** The objective of this exercise is to empirically estimate Jensen's $\alpha$ and the $\beta$ of an investment fund relative to its benchmark index.

You must write a Python script that fulfills the following conditions :

1. Stochastically generates a historical series of 60 months of returns for the market portfolio and for the asset being studied (using a fixed random seed for reproducibility).
2. Calculates the excess returns of the asset and the market relative to a fixed risk-free rate.
3. Uses the `statsmodels` library to perform a linear regression using Ordinary Least Squares (OLS). *Warning : make sure to explicitly include a constant for the estimation of the intercept.*
4. Extracts and displays the estimated coefficients as well as the statistical significance (*p-value*) of $\alpha$.

### Python Solution Code

```python
import numpy as np
import pandas as pd
import statsmodels.api as sm

# 1. Generation of historical data (60 months)
np.random.seed(42)
rendements_marche = np.random.normal(0.006, 0.04, 60) # Average market return = 0.6\%/month
taux_sans_risque = 0.001                               # Rf = 0.1\% per month constant

# Asset generation (True beta = 1.3, Target monthly alpha = 0.16\%)
bruit = np.random.normal(0, 0.02, 60)
rendements_actif = taux_sans_risque + 1.3 * (rendements_marche - taux_sans_risque) + 0.0016 + bruit

# 2. Calculation of excess returns
exces_marche = rendements_marche - taux_sans_risque
exces_actif = rendements_actif - taux_sans_risque

# 3. Modeling and OLS Regression via statsmodels
# A constant must be added to estimate Jensen's Alpha
X = sm.add_constant(exces_marche)
modele = sm.OLS(exces_actif, X).fit()

# 4. Display of the report and extraction of coefficients
print(modele.summary())
print(f"\n--- Estimation Results ---")
```

```
print(f"Estimated Beta (slope) : {modele.params[1]:.4f}")
print(f"Estimated monthly Alpha (intercept) : {modele.params[0]:.6f}")
print(f"Alpha P-value : {modele.pvalues[0]:.4f}")
```

**Exercise 2 : Factor Extraction through Principal Component Analysis (PCA)** Within factor analysis, we often seek to identify unobservable (statistical) factors that drive the behavior of a basket of stocks.

Write a Python script that performs the following tasks :

1. Creates a DataFrame of historical returns for 4 distinct financial assets, constructed so that they share a strong correlation induced by an invisible common factor.
2. Performs standardization (centering and scaling) of the return data, a mathematically mandatory step before any PCA.
3. Applies the Principal Component Analysis algorithm from `scikit-learn` to extract the first two components.
4. Displays the percentage of total variance explained by the first principal component.
5. Extracts the *loadings* (weights of the first component) for each asset and interprets these results with respect to factor sensitivities.

### Python Solution Code

```
import numpy as np
import pandas as pd
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

# 1. Generation of correlated returns for a portfolio of 4 assets
np.random.seed(10)
facteur_commun = np.random.normal(0, 0.02, 100)
rendements = pd.DataFrame({
    'Actif_A': 0.8 * facteur_commun + np.random.normal(0, 0.01, 100),
    'Actif_B': 1.2 * facteur_commun + np.random.normal(0, 0.01, 100),
    'Actif_C': 0.5 * facteur_commun + np.random.normal(0, 0.01, 100),
    'Actif_D': 1.0 * facteur_commun + np.random.normal(0, 0.01, 100)
})

# 2. Standardization of return data
scaler = StandardScaler()
rendements_scales = scaler.fit_transform(rendements)

# 3. Initialization and fitting of PCA
pca = PCA(n_components=2)
pca.fit(rendements_scales)

# 4. Variance explained by the first implicit market factor
variance_expliquee = pca.explained_variance_ratio_[0] * 100
print(f"Variance explained by Principal Component 1 : {variance_expliquee:.2f
    }\%\n")

# 5. Extraction of statistical betas (Loadings)
print("Factor sensitivities (PC1 Loadings) :")
for actif, loading in zip(rendements.columns, pca.components_[0]):
    print(f"  Sensitivity of {actif} : {loading:.4f}")
```

**Exercise 3 — CAPM Estimation and Interpretation** We want to study the relationship between the return of a stock and the market return using the CAPM market model.

We consider the stocks `AAPL`, `MSFT`, and `JPM`, as well as the S&P500 index `GSPC` as an approximation of the market portfolio.

1. Download the adjusted prices of the four assets over the period from January 1, 2021 to January 1, 2026.
2. Calculate the arithmetic daily returns.
3. For each of the three stocks, estimate the model :

$$R_{i,t} = \alpha_i + \beta_i R_{M,t} + \varepsilon_{i,t},$$

   where $R_{M,t}$ is the return of the S&P 500.
4. Use `statsmodels` and `OLS` to estimate $\alpha_i$ and $\beta_i$.
5. Economically interpret the coefficient $\beta_i$, $\alpha_i$, and the $R^2$ of each regression.
6. Compare the three stocks. Which stock appears to have the greatest exposure to systematic market risk ?
7. Graphically verify the relationship between the returns of each stock and those of the S&P 500 using a scatter plot and the regression line.

**Solution :** We consider the stocks `AAPL`, `MSFT`, and `JPM`, as well as the S&P 500 index `^GSPC` as an approximation of the market portfolio.

**1. Download the data** We begin by downloading the adjusted prices :

```
import numpy as np
import pandas as pd
import yfinance as yf
import statsmodels.api as sm
import matplotlib.pyplot as plt

tickers = ['AAPL', 'MSFT', 'JPM', '^GSPC']

data = yf.download(
tickers,
start='2021-01-01',
end='2026-01-01'
)['Adj Close']

print(data.head())
```

“‘latex We then have a time series containing the adjusted prices of the three stocks and the S&P 500. The symbol `^GSPC` corresponds to the S&P 500. In this exercise, the S&P 500 is used as an empirical approximation of the market portfolio.

**2. Calculating daily returns** Daily arithmetic returns are calculated using :

$$R_t = \frac{P_t - P_{t-1}}{P_{t-1}}.$$

In Python :

```
returns = data.pct_change().dropna()
print(returns.head())
```

The function `pct_change()` directly calculates $R_t$
The `dropna()` method then removes the first observation, for which no return can be calculated.

**3. Estimating the CAPM models** For each of the three stocks, we estimate the model :

$$R_{i,t} = \alpha_i + \beta_i R_{M,t} + \varepsilon_{i,t}.$$

In our case $R_{M,t} \approx R_{\text{S\&P500},t}$.
We can therefore estimate the three regressions separately.

```
assets = ['AAPL', 'MSFT', 'JPM']
results = {}
for asset in assets:
Y = returns[asset]
X = returns['^GSPC']
X_with_const = sm.add_constant(X)
model = sm.OLS(Y, X_with_const).fit()
results[asset] = model
print("\n==============================")
print(asset)
print("==============================")
print(model.summary())
```

The command `sm.add_constant(X)` adds a column of 1 to the explanatory variable in order to allow the estimation of alpha :

$$R_{i,t} = \alpha_i \times 1 + \beta_i R_{M,t} + \varepsilon_{i,t}.$$

The coefficient associated with the constant is therefore $\alpha_i$, while the coefficient associated with the S&P 500 is $\beta_i$.

**4. Interpreting $\beta$, $\alpha$ and $R^2$ :** The main results can be extracted as follows :

```
for asset in assets:
model = results[asset]
alpha = model.params['const']
beta = model.params['^GSPC']
r_squared = model.rsquared
print(f"\n{asset}")
print(f"Daily alpha       : {alpha:.6f}")
print(f"Beta              : {beta:.4f}")
print(f"R-squared         : {r_squared:.2\%}")
```

**Beta interpretation.** The coefficient $\beta_i$ measures the sensitivity of the stock's return to market movements :

$$\beta_i = \frac{\text{Cov}(R_i, R_M)}{\text{Var}(R_M)}.$$

For example, if a stock has $\beta_i = 1.2$, then, all else being equal, a 1% market movement is associated with an average movement of approximately 1.2% in the stock according to the model.
A $\beta$ greater than 1 indicates a higher sensitivity to the market than the market itself.
A $\beta$ lower than 1 indicates lower sensitivity.

**Alpha interpretation.** Alpha corresponds to the constant of the regression :

$$R_{i,t} = \alpha_i + \beta_i R_{M,t} + \varepsilon_{i,t}.$$

It represents the average component of the return that is not explained by the market factor in this regression.
The estimated alpha should be interpreted with caution : a non-zero alpha in the sample does not necessarily mean that it is statistically different from zero.

**Interpretation of $R^2$.** The coefficient of determination $R^2$ measures the proportion of the variance of the stock returns explained by the regression model.
For example, if $R^2 = 0.60$, this means that 60% of the observed variance of the stock returns is explained by the linear relationship with the S&P 500 return in the sample considered. The remaining 40% corresponds to the part not explained by this regression.

**5. Comparing the three stocks** A summary table can be constructed :

```
summary = []
for asset in assets:
model = results[asset]
summary.append({
    'Asset': asset,
    'Alpha': model.params['const'],
    'Beta': model.params['^GSPC'],
    'R2': model.rsquared
})
summary_df = pd.DataFrame(summary)
print(summary_df)
```

The table makes it possible to directly compare the three stocks.
The stock with the highest $\beta$ is the one whose return is, in the sample studied, the most sensitive to market movements.
Thus, to identify the stock with the greatest exposure to systematic risk, we compare the estimated values of :

$$\beta_{\text{AAPL}}, \qquad \beta_{\text{MSFT}}, \qquad \beta_{\text{JPM}}.$$

However, a high beta should not be confused with a necessarily higher return : beta measures sensitivity to the market factor, not future performance.

**6. Graphical representation** The relationship between the returns of a stock and those of the market can be represented graphically.

For example, for AAPL :

```
asset = 'AAPL'
Y = returns[asset]
X = returns['^GSPC']
model = results[asset]
plt.figure(figsize=(8, 5))
plt.scatter(X, Y, alpha=0.4)
x_line = np.linspace(X.min(), X.max(), 100)
y_line = model.params['const'] + model.params['^GSPC'] * x_line
plt.plot(x_line, y_line)
plt.xlabel('S&P 500 return')
plt.ylabel('AAPL return')
plt.title('AAPL and market return')
plt.grid(alpha=0.3)
plt.show()
```

Each point represents a daily observation :

$$(R_{M,t}, R_{i,t})\,.$$

The line represents the estimated relationship :

$$\widehat{R}_{i,t} = \widehat{\alpha}_i + \widehat{\beta}_i R_{M,t}.$$

The slope of this line corresponds to the estimated beta $\widehat{\beta}_i$ and its intercept corresponds to the estimated alpha $\widehat{\alpha}_i$.

```
# ================================================================
# EXERCISE 5--CAPM ESTIMATION AND INTERPRETATION
# ================================================================

import numpy as np
import pandas as pd
import yfinance as yf
import statsmodels.api as sm
import matplotlib.pyplot as plt


# ================================================================
# QUESTION 1-- DOWNLOAD THE DATA
# ================================================================

tickers = ['AAPL', 'MSFT', 'JPM', '^GSPC']

data = yf.download(
    tickers,
    start='2021-01-01',
    end='2026-01-01',
    auto_adjust=False
)

# Retrieve adjusted prices
data = data['Adj Close']

print("\n==============================")
print("ADJUSTED PRICES")
print("==============================")
print(data.head())

print("\nMissing values:")
print(data.isna().sum())


# ================================================================
# QUESTION 2--CALCULATE DAILY RETURNS
# ================================================================

returns = data.pct_change().dropna()

print("\n==============================")
print("DAILY RETURNS")
print("==============================")
print(returns.head())


# ================================================================
# QUESTION 3 -- ESTIMATE THE CAPM MODELS
# ================================================================

# List of stocks studied
assets = ['AAPL', 'MSFT', 'JPM']

# Dictionary to store the regression results
results = {}

for asset in assets:

    # Dependent variable:
```

```
    # stock return
    Y = returns[asset]

    # Explanatory variable:
    # market return, approximated here by the S&P 500
    X = returns['^GSPC']

    # Add a constant to estimate alpha
    X_with_const = sm.add_constant(X)

    # OLS regression
    model = sm.OLS(Y, X_with_const).fit()

    # Store the model
    results[asset] = model

    # Display the results
    print("\n=======================================")
    print(f"CAPM REGRESSION : {asset}")
    print("=======================================")

    print(model.summary())


# ================================================================
# QUESTION 4 -- INTERPRET ALPHA, BETA AND R^{2}
# ================================================================

summary = []

for asset in assets:

    model = results[asset]

    # Daily alpha
    alpha_daily = model.params['const']

    # Beta
    beta = model.params['^GSPC']

    # R^2
    r_squared = model.rsquared

    # Approximate annualization of alpha
    alpha_annual = alpha_daily * 252

    summary.append({
        'Asset': asset,
        'Daily alpha': alpha_daily,
        'Annualized alpha': alpha_annual,
        'Beta': beta,
        'R2': r_squared
    })


# Create the summary table
summary_df = pd.DataFrame(summary)

print("\n=======================================")
print("CAPM RESULTS")
print("=======================================")
```

```
123
124 print(summary_df.to_string(index=False))
125
126 # More readable display of the main coefficients
127
128 for asset in assets:
129
130     model = results[asset]
131
132     alpha = model.params['const']
133     beta = model.params['^GSPC']
134     r_squared = model.rsquared
135
136     print(f"\n{asset}")
137     print(f"Daily alpha       : {alpha:.6f}")
138     print(f"Annualized alpha  : {alpha * 252:.2\%}")
139     print(f"Beta              : {beta:.4f}")
140     print(f"R-squared         : {r_squared:.2\%}")
141
142
143 # ================================================================
144 # QUESTION 5 -- COMPARE THE THREE STOCKS
145 # ================================================================
146
147 # Identify the stock with the highest beta
148
149 asset_max_beta = summary_df.loc[
150     summary_df['Beta'].idxmax(),
151     'Asset'
152 ]
153
154 max_beta = summary_df['Beta'].max()
155
156 print("\n========================================")
157 print("BETA COMPARISON")
158 print("========================================")
159
160 print(
161     f"The stock with the highest beta is "
162     f"{asset_max_beta} with a beta of {max_beta:.4f}."
163 )
164
165 # Rank the stocks according to their beta
166
167 beta_comparison = summary_df[
168     ['Asset', 'Beta']
169 ].sort_values(
170     by='Beta',
171     ascending=False
172 )
173
174 print("\nBetas ranked from highest to lowest:")
175 print(beta_comparison.to_string(index=False))
176
177
178 # ================================================================
179 # QUESTION 6 -- GRAPHICAL REPRESENTATION
180 # ================================================================
181
182 for asset in assets:
183
```

```
    # Market return
    X = returns['^GSPC']

    # Stock return
    Y = returns[asset]

    # Model corresponding to the stock
    model = results[asset]

    # X values used to plot the regression line
    x_line = np.linspace(
        X.min(),
        X.max(),
        100
    )

    # Regression line:
    #
    # Y_hat = alpha + beta * X

    y_line = (
        model.params['const']
        + model.params['^GSPC'] * x_line
    )

    # Create the graph
    plt.figure(figsize=(8, 5))

    # Scatter plot
    plt.scatter(
        X,
        Y,
        alpha=0.4
    )

    # Regression line
    plt.plot(
        x_line,
        y_line,
        linewidth=2,
        label='Regression line'
    )

    # Axes
    plt.xlabel('S&P 500 return')
    plt.ylabel(f'{asset} return')

    # Title
    plt.title(
        f'{asset} and market return'
    )

    plt.legend()
    plt.grid(alpha=0.3)

    plt.show()
```

**Exercise 4 :**

**1. Retrieving Fama-French data** For this exercise, the factors $Mkt-Rf$, $SMB$, $HML$ and $RF$ are available in the Kenneth R. French database. You can directly download the monthly data with Python using the following code :

```
import pandas as pd

url = "https://mba.tuck.dartmouth.edu/pages/faculty/\
ken.french/ftp/F-F_Research_Data_Factors_CSV.zip"

ff = pd.read_csv(url, skiprows=3)
```

The variable `ff` then contains the Fama-French data. The main columns of interest are :

```
Mkt-RF    SMB    HML    RF
```

**Warning :** the values provided by Kenneth French are expressed as percentages. For example : $2.50\% = 0.025$. The factors must therefore be converted to decimal values before being used in the regression :

```
ff[['Mkt-RF', 'SMB', 'HML', 'RF']] /= 100
```

You can then display the first observations with :

```
print(ff.head())
```

**2. Retrieving AAPL stock data** Then use `yfinance` to retrieve AAPL stock prices over the period January 2021–December 2025.
Using the adjusted prices, calculate the monthly returns of AAPL.

**3. Preparing the data** You must now bring the two data sources to the same monthly frequency and align them on the same dates :

$$\text{AAPL Price} \longrightarrow \text{AAPL Returns}$$

“‘latex then :

$$\text{AAPL Returns} + \text{Fama-French Factors.}$$

Check that there are no missing values before performing the regressions.

**4. CAPM estimation** Estimate the model :

$$R_{AAPL,t} - R_{f,t} = \alpha + \beta_{MKT}(Mkt - Rf)_t + \varepsilon_t.$$

Identify $\alpha$, $\beta_{MKT}$, and $R^2$. Interpret the results.

**5. Estimation of the Fama-French model** Then estimate :

$$R_{AAPL,t} - R_{f,t} = \alpha + \beta_{MKT}(Mkt - Rf)_t + \beta_{SMB}SMB_t + \beta_{HML}HML_t + \varepsilon_t.$$

Identify the coefficients $\alpha$, $\beta_{MKT}$, $\beta_{SMB}$ and $\beta_{HML}$, as well as $R^2$.
Interpret the sign and significance of the coefficients $\beta_{SMB}$ and $\beta_{HML}$.

**6. Comparison of the two models** Construct a comparative table of the results obtained. Answer the following questions :

1. Which model has the higher $R^2$ ?
2. What does this difference in $R^2$ mean ?
3. Are the $SMB$ and $HML$ factors statistically significant ?
4. What do the coefficients $\beta_{SMB}$ and $\beta_{HML}$ tell us about the profile of the AAPL stock ?
5. Is alpha statistically different from zero ?

```

# ============================================================
# EXERCISE 4 - CAPM AND FAMA-FRENCH ON AAPL
# ============================================================

import pandas as pd
import numpy as np
import yfinance as yf
import statsmodels.api as sm


# ============================================================
# 1. RETRIEVING FAMA-FRENCH DATA
# ============================================================

url = (
    "https://mba.tuck.dartmouth.edu/pages/faculty/"
    "ken.french/ftp/F-F_Research_Data_Factors_CSV.zip"
)

ff = pd.read_csv(url, skiprows=3)

# Rename the first column
ff = ff.rename(columns={ff.columns[0]: "Date"})

# Keep only numerical observations
ff = ff[
    pd.to_numeric(ff["Date"], errors="coerce").notna()
].copy()

# Convert Date to integer
ff["Date"] = ff["Date"].astype(int)

# Keep only the 2021-2025 period
ff = ff[
    (ff["Date"] >= 202101) &
    (ff["Date"] <= 202512)
].copy()

# Convert YYYYMM -> date
ff["Date"] = pd.to_datetime(
    ff["Date"].astype(str),
    format="%Y%m"
)

# Convert percentages to decimals
ff[["Mkt-RF", "SMB", "HML", "RF"]] /= 100

# Set dates to the end of the month
ff["Date"] = ff["Date"] + pd.offsets.MonthEnd(0)
```

```
51
52 # Set Date as the index
53 ff = ff.set_index("Date")
54
55 print("\n=== FAMA-FRENCH FACTORS ===")
56 print(ff.head())
57
58
59 # ============================================================
60 # 2. RETRIEVING AAPL DATA
61 # ============================================================
62
63 aapl = yf.download(
64     "AAPL",
65     start="2021-01-01",
66     end="2026-01-01",
67     auto_adjust=False
68 )
69
70 # Retrieve adjusted price
71 prices = aapl["Adj Close"]
72
73 # End-of-month prices
74 aapl_monthly = prices.resample("ME").last()
75
76 # Monthly returns
77 aapl_returns = aapl_monthly.pct_change()
78
79 # Convert to DataFrame
80 aapl_returns = aapl_returns.to_frame(name="AAPL")
81
82 # Harmonize dates
83 aapl_returns.index = (
84     aapl_returns.index
85     .to_period("M")
86     .to_timestamp("M")
87 )
88
89 print("\n=== AAPL MONTHLY RETURNS ===")
90 print(aapl_returns.head())
91
92
93 # ============================================================
94 # 3. MERGING THE DATA
95 # ============================================================
96
97 data = aapl_returns.join(
98     ff,
99     how="inner"
100 )
101
102 # Remove any missing values
103 data = data.dropna()
104
105 print("\n=== FINAL DATA ===")
106 print(data.head())
107
108 print("\nNumber of observations:", len(data))
109
110 print("\nMissing values:")
111 print(data.isna().sum())
```

```
112
113
114 # ============================================================
115 # 4. AAPL EXCESS RETURN
116 # ============================================================
117
118 data["AAPL_excess"] = data["AAPL"] - data["RF"]
119
120 print("\n=== EXCESS RETURN ===")
121 print(data[["AAPL", "RF", "AAPL_excess"]].head())
122
123
124 # ============================================================
125 # 5. CAPM ESTIMATION
126 # ============================================================
127
128 # Dependent variable
129 y = data["AAPL_excess"]
130
131 # Explanatory variable
132 X_capm = data[["Mkt-RF"]]
133
134 # Add the constant
135 X_capm = sm.add_constant(X_capm)
136
137 # Regression
138 capm = sm.OLS(y, X_capm).fit()
139
140 print("\n")
141 print("=" * 70)
142 print("CAPM RESULTS")
143 print("=" * 70)
144
145 print(capm.summary())
146
147
148 # Retrieve results
149 alpha_capm = capm.params["const"]
150 beta_mkt_capm = capm.params["Mkt-RF"]
151 r2_capm = capm.rsquared
152
153 p_alpha_capm = capm.pvalues["const"]
154
155
156 # ============================================================
157 # 6. ESTIMATION OF THE FAMA-FRENCH MODEL
158 # ============================================================
159
160 # Explanatory variables
161 X_ff = data[[
162     "Mkt-RF",
163     "SMB",
164     "HML"
165 ]]
166
167 # Add the constant
168 X_ff = sm.add_constant(X_ff)
169
170 # Regression
171 ff_model = sm.OLS(y, X_ff).fit()
172
```

```
print("\n")
print("=" * 70)
print("FAMA-FRENCH MODEL RESULTS")
print("=" * 70)

print(ff_model.summary())


# Retrieve results
alpha_ff = ff_model.params["const"]
beta_mkt_ff = ff_model.params["Mkt-RF"]
beta_smb_ff = ff_model.params["SMB"]
beta_hml_ff = ff_model.params["HML"]

r2_ff = ff_model.rsquared

p_alpha_ff = ff_model.pvalues["const"]
p_beta_smb_ff = ff_model.pvalues["SMB"]
p_beta_hml_ff = ff_model.pvalues["HML"]


# ============================================================
# 7. COMPARATIVE TABLE
# ============================================================

results = pd.DataFrame({
    "CAPM": [
        alpha_capm,
        beta_mkt_capm,
        np.nan,
        np.nan,
        r2_capm
    ],

    "Fama-French": [
        alpha_ff,
        beta_mkt_ff,
        beta_smb_ff,
        beta_hml_ff,
        r2_ff
    ]

}, index=[
    "Alpha",
    "Beta MKT",
    "Beta SMB",
    "Beta HML",
    "R^2"
])

print("\n")
print("=" * 70)
print("COMPARATIVE TABLE")
print("=" * 70)

print(results.round(4))


# ============================================================
# 8. P-VALUES TABLE
# ============================================================
```

```
234
235 pvalues = pd.DataFrame({
236     "CAPM": [
237         p_alpha_capm,
238         capm.pvalues["Mkt-RF"],
239         np.nan,
240         np.nan
241     ],
242
243     "Fama-French": [
244         p_alpha_ff,
245         ff_model.pvalues["Mkt-RF"],
246         p_beta_smb_ff,
247         p_beta_hml_ff
248     ]
249
250 }, index=[
251     "Alpha",
252     "Beta MKT",
253     "Beta SMB",
254     "Beta HML"
255 ])
256
257 print("\n")
258 print("=" * 70)
259 print("P-VALUES")
260 print("=" * 70)
261
262 print(pvalues.round(4))
263
264
265 # ============================================================
266 # 9. SIGNIFICANCE TEST
267 # ============================================================
268
269 print("\n")
270 print("=" * 70)
271 print("SIGNIFICANCE TESTS")
272 print("=" * 70)
273
274 # Alpha
275 if p_alpha_ff < 0.05:
276     print("Alpha: significant at the 5% level.")
277 else:
278     print("Alpha: not significant at the 5% level.")
279
280 # SMB
281 if p_beta_smb_ff < 0.05:
282     print("SMB: significant at the 5% level.")
283 else:
284     print("SMB: not significant at the 5% level.")
285
286 # HML
287 if p_beta_hml_ff < 0.05:
288     print("HML: significant at the 5\% level.")
289 else:
290     print("HML: not significant at the 5\% level.")
291
292
293 # ============================================================
294 # 10. COMPARISON OF R^2
```

```
# ============================================================

print("\n")
print("=" * 70)
print("COMPARISON OF R^2")
print("=" * 70)

print("R^2 CAPM         :", round(r2_capm, 4))
print("R^2 Fama-French  :", round(r2_ff, 4))
print("Difference       :", round(r2_ff - r2_capm, 4))
```

### Exercise 5 — PCA and Statistical Factor Model

Consider a portfolio composed of the following eight stocks :

`AAPL`, `MSFT`, `AMZN`, `NVDA`, `JPM`, `V`, `XOM`, `PG`.

The objective is to identify the main statistical factors common to the returns of these stocks.

1. Download the adjusted prices of the eight stocks over the period from January 1, 2021 to January 1, 2026.
2. Calculate the daily return matrix :

$$R_t = \begin{pmatrix} R_{AAPL,t} \\ R_{MSFT,t} \\ \vdots \\ R_{PG,t} \end{pmatrix}.$$

3. Apply PCA to the returns and retain the first three principal components.
4. For each component, calculate the proportion of explained variance :

$$\frac{\lambda_k}{\sum_{j=1}^{N} \lambda_j}.$$

5. Plot the variance explained by $PC_1$, $PC_2$ and $PC_3$.
6. Extract the factor loadings (*loadings*) of $PC_1$ and plot them.
7. Examine the signs and magnitudes of the loadings of $PC_1$. Can this component be interpreted as a common movement of the stocks ?
8. Then construct the time series of the three factors :

$$PC_{1,t}, \qquad PC_{2,t}, \qquad PC_{3,t}.$$

9. For the stock `AAPL`, estimate the factor model :

$$R_{AAPL,t} = \alpha + \beta_1 PC_{1,t} + \beta_2 PC_{2,t} + \beta_3 PC_{3,t} + \varepsilon_t.$$

10. Interpret the coefficients $\beta_1$, $\beta_2$ and $\beta_3$.

# Chapitre 5

# Financial Time Series and Volatility Forecasting

## 5.1 Introduction

In previous chapters, we studied how to measure risk, construct an optimal portfolio, and analyze the determinants of returns using the CAPM and factor models.
However, we have mainly considered observations as a set of data to which statistical methods could be applied. An important question therefore remains open : **what happens when the temporal ordering of observations becomes essential ?**
In finance, this question is fundamental. Is a return observed today related to the return observed yesterday ? Is a period of high volatility likely to be followed by another period of high volatility ? Can we use the information available today to forecast tomorrow's risk ?
These questions naturally lead us to **financial time series**.
A time series is simply a variable observed successively over time :

$$X_1, X_2, \ldots, X_t, \ldots, X_T.$$

The essential difference from a conventional dataset is therefore that the ordering of observations has meaning. The return observed in January 2025 occurs before the return observed in February 2025, and this information may contain an exploitable statistical structure.
The objective of this chapter is not, however, to assume that markets are easily predictable. On the contrary, an important empirical observation is that financial returns often exhibit **low linear autocorrelation**, particularly in liquid markets and at daily frequency.
But this apparent lack of return predictability does not mean that everything is unpredictable. A much more robust property appears when we focus on the **magnitude of movements** rather than their direction : calm periods tend to be followed by calm periods, while highly volatile periods tend to be followed by other volatile periods.
This property is called **volatility clustering**.
This observation will progressively motivate the ARCH and GARCH models.

**Why treat financial data as time series ?** Suppose that we have daily prices of a stock :

$$P_1, P_2, \ldots, P_T.$$

One might be tempted to directly apply standard statistical methods to this series. However, prices present an important first problem : their level depends strongly on the past.
Indeed, a financial price can be represented in simplified form as :

$$P_t = P_{t-1}(1 + R_t),$$

where $R_t$ represents the return over period $t$.
The price therefore accumulates past movements. A shock that occurred several years ago continues to be incorporated into the current price level.
This property makes prices difficult to handle directly with certain classical statistical methods. In practice, we therefore very often work with **returns**.
In quantitative finance, the choice between simple and logarithmic returns depends on the problem being studied. In what follows, we will mainly use $R_t$ to denote the return, without systematically distinguishing between the two conventions.

**The first question : do returns depend on the past ?** Suppose that we observe an unusually high return today. A natural question arises :

If today's return is known, does it tell us anything about tomorrow's return ?

In other words, is there a statistical relationship between $R_t$ and $R_{t-1}$ ?
We can generalize this question by comparing the current return with the return observed $k$ periods earlier : $R_t$ and $R_{t-k}$.
This notion leads to **autocorrelation**.

**Autocovariance and autocorrelation** Before defining autocorrelation, we need to understand what we are trying to measure.
Covariance measures how two variables tend to move together. For a time series, we can therefore ask whether $X_t$ and $X_{t-k}$ tend to be simultaneously high or low. We define the autocovariance at lag $k$ by :

$$\gamma_k = \mathrm{Cov}(X_t, X_{t-k}).$$

To compare different series, however, it is more convenient to normalize this quantity by the variance :

$$\rho_k = \frac{\gamma_k}{\gamma_0}, \text{ where } \gamma_0 = \mathrm{Var}(X_t).$$

The quantity $\rho_k$ is called the **autocorrelation function**, or *Autocorrelation Function*, ACF.
It therefore measures the linear dependence between an observation and an observation located $k$ periods earlier.

**What does the lag $k$ mean ?** The lag simply indicates how many periods separate the two observations.

For $k = 1$ :

$$\rho_1 = \mathrm{Corr}(X_t, X_{t-1}).$$

We therefore compare each observation with the one immediately preceding it.
For $k = 2$ :

$$\rho_2 = \mathrm{Corr}(X_t, X_{t-2}).$$

This time, we compare each observation with the one observed two periods earlier.
If the data are daily, then $k = 1$ corresponds to one day, $k = 5$ corresponds approximately to one trading week, and so on.

**Why does autocorrelation not directly depend on time ?** This question is particularly important. One might think that we should calculate

$$\mathrm{Corr}(X_{2020}, X_{2019})$$

and then separately :

$$\mathrm{Corr}(X_{2025}, X_{2024}).$$

Why then do we speak of a single autocorrelation at lag 1 ?
The answer comes from the assumption of **stationarity**. For a weakly stationary time series, we assume that :

$$E[X_t] = \mu, \text{ and } \mathrm{Var}(X_t) = \sigma^2,$$

and, most importantly, that the covariance between two observations depends only on the time distance between them, and not on their precise calendar dates :

$$\mathrm{Cov}(X_t, X_{t-k}) = \gamma_k.$$

The key point is therefore :

$$\boxed{\text{dependence depends on the lag } k, \text{ not on calendar time } t.}$$

Thus, under the stationarity assumption, the relationship between two observations separated by one period is assumed to be comparable regardless of the date.
This means that

$$\mathrm{Cov}(X_{2020}, X_{2019}) \text{ and } \mathrm{Cov}(X_{2025}, X_{2024})$$

are considered realizations of the same statistical structure corresponding to lag $k = 1$.
It is important to understand that this is not a fundamental law of financial markets. **Stationarity is a modeling assumption.**
Markets can change regime. A crisis period may exhibit dynamics very different from a calm period.
When we calculate an ACF over an entire sample, we are therefore implicitly assuming that the average relationship observed over this period is sufficiently stable to be modeled.

**Empirical autocorrelation** In practice, we obviously do not know the true population autocorrelation $\rho_k$. We only have a sample :

$$X_1, X_2, \ldots, X_T.$$

We therefore calculate an empirical estimate :

$$\hat{\rho}_k = \frac{\sum_{t=k+1}^{T} (X_t - \bar{X})(X_{t-k} - \bar{X})}{\sum_{t=1}^{T} (X_t - \bar{X})^2}, \text{ where } \bar{X} = \frac{1}{T}\sum_{t=1}^{T} X_t.$$

This quantity tells us whether observations separated by a lag $k$ tend to move together.

**Interpretation :** A positive autocorrelation means that observations tend to move in the same direction. For example, $\rho_1 > 0$ means that a value above the mean tends to be followed by another value above the mean, and conversely.

A negative autocorrelation indicates, on the contrary, a tendency to alternate :

$$X_t > \bar{X} \quad \Rightarrow \quad X_{t+1} < \bar{X}$$

on average.
Finally, an autocorrelation close to zero means that there is little detectable linear dependence at the lag considered.

**A first surprising observation about financial returns** One might think that a financial time series must necessarily exhibit strong autocorrelation because markets evolve over time. Yet this is generally not what we observe for daily returns of major markets.
The ACF of returns is often relatively close to zero, particularly at the first few lags. This means that it is difficult to simply predict tomorrow's return from today's return using an elementary linear model.
We can summarize this idea, in simplified form, as :

$$E[R_{t+1} \mid \mathcal{F}_t] \approx \mu,$$

where $\mathcal{F}_t$ represents all information available at date $t$.
In a very simple model, knowing past returns therefore provides little information for directly predicting the next return. But this does not mean that markets are completely random.
There remains another, much more interesting question :

> Even if we cannot easily predict the direction of the next movement, can we predict its magnitude ?

This question leads us to volatility.

## 5.2 Return and volatility : two different problems

Suppose that we successively observe the following returns :

$$+0.2\%, \quad -0.1\%, \quad +0.3\%, \quad -0.2\%.$$

Then consider a period such as :

$$+4\%, \quad -5\%, \quad +3\%, \quad -4\%.$$

In both cases, the signs of the returns may be difficult to predict. But their magnitude is clearly different. In the first case, the market is calm, while in the second, it is highly agitated. This distinction is fundamental.
The direction of the return may be difficult to predict while the **uncertainty surrounding the return** may be predictable.
Mathematically, we can distinguish :

$$E[R_{t+1} \mid \mathcal{F}_t] \text{ from } E[R_{t+1}^2 \mid \mathcal{F}_t].$$

The first quantity concerns the conditional mean, while the second is related to the conditional variance.
This is where a key idea in quantitative finance appears :

the mean may be difficult to predict while volatility may be predictable.

**Volatility and the clustering of turbulent periods** Volatility measures the magnitude of fluctuations in returns.
In the classical framework, we can use the standard deviation

$$\sigma = \sqrt{\mathrm{Var}(R_t)}.$$

But in financial markets, it is often more relevant to consider volatility that changes over time, $\sigma_t$.
Why ?
Because risk is clearly not constant. A financial crisis, a central bank announcement, or an important economic release can trigger several consecutive days of high volatility. Conversely, some periods exhibit very small fluctuations.
We therefore often observe sequences such as :

$$\underbrace{\text{calm, calm, calm}}_{\text{low volatility}} \longrightarrow \underbrace{\text{large movement, large movement, large movement}}_{\text{high volatility}}.$$

This property is called **volatility clustering**.

**How can this phenomenon be detected ?** A simple approach is to study the squared returns $R_t^2$.
Why squared returns ?
Because $R_t^2$ ignores the sign of the return while essentially retaining its magnitude.
A return of $+5\%$ and a return of $-5\%$ both produce

$$R_t^2 = 0.0025.$$

We can therefore study the autocorrelation of $R_t^2$. Indeed, it is common to observe much stronger autocorrelation for squared returns than for returns themselves.
This provides a very intuitive picture of financial dynamics :

the sign may be difficult to predict, but the magnitude may exhibit temporal memory.

**Visualizing autocorrelation with Python** The `statsmodels` library makes it easy to calculate and visualize an autocorrelation function.

```
import matplotlib.pyplot as plt
from statsmodels.graphics.tsaplots import plot_acf

plot_acf(returns, lags=30)
plt.title("ACF of returns")
plt.show()
```

We can then apply exactly the same idea to squared returns :

```
squared_returns = returns ** 2

plot_acf(squared_returns, lags=30)
plt.title("ACF of squared returns")
plt.show()
```

The comparison between the two graphs is particularly informative. If the ACF of returns is weak while that of squared returns remains positive over several lags, we observe precisely the phenomenon we are trying to model.
The return series appears to be weakly predictable in the mean, but its level of risk varies over time.

**Stationarity** Time-series models often rely on an assumption of stationarity. It is therefore important to understand this concept before introducing AR, MA, or GARCH models.
A series is said to be weakly stationary when its main statistical properties do not depend on time. In particular, we require :

$$E[X_t] = \mu, \quad \mathrm{Var}(X_t) = \sigma^2, \text{ and } \mathrm{Cov}(X_t, X_{t-k}) = \gamma_k.$$

The mean and variance do not depend on $t$, while the covariance depends only on the lag $k$.
This assumption should not be confused with the claim that "markets never change". A financial series may be only **approximately stationary** over a given period.
This is a simplification that allows us to construct a statistical model.

### 5.2.1 Augmented Dickey–Fuller Test

Before using time-series models such as AR, MA, or ARMA models, it is important to determine whether the series being studied is *stationary*. Indeed, many modeling methods assume that the statistical properties of the series do not change over time. We have previously seen that a weakly stationary series satisfies, in particular,

$$\mathbb{E}[X_t] = \mu, \qquad \mathrm{Var}(X_t) = \sigma^2, \tag{5.1}$$

and that its covariance between two dates depends essentially on the distance between those dates, rather than on their position in time.
The question is therefore the following : *how can we determine, from observed data, whether a series can reasonably be considered stationary ?*
A classical approach is to use the ADF test (*Augmented Dickey–Fuller*). This test is widely used in econometrics and finance to investigate the presence of a *unit root*, which is a typical source of non-stationarity.

**Intuition :** **(i)** To understand the ADF test, first consider a very simple autoregressive model :

$$X_t = \phi X_{t-1} + \varepsilon_t. \tag{5.2}$$

The behavior of the series then depends strongly on the value of $\phi$. If $|\phi| < 1$, a past shock gradually disappears. For example, if the series receives a positive shock today, its effect will still be present tomorrow, but it will gradually decrease over time. The series therefore has some form of memory, while remaining compatible with stationary behavior.
In contrast, when $\phi = 1$, we obtain

$$X_t = X_{t-1} + \varepsilon_t. \tag{5.3}$$

This is a random walk. In this case, a shock does not gradually disappear : it accumulates in the level of the series. If a positive shock occurs today, it permanently changes the future level of the series.
This is precisely the situation referred to as a *unit root*. It is an important source of non-stationarity.

The idea behind the Dickey–Fuller test is therefore to find evidence allowing us to determine whether the autoregressive coefficient is sufficiently below 1 or whether, on the contrary, the data are compatible with the hypothesis $\phi = 1$.

**(ii)** In financial data, the dynamics of a series are rarely described solely by its first lag. The errors themselves may exhibit some temporal dependence. To account for this possibility, the ADF test enriches the basic model by adding several lags of the differences of the series.
A common formulation of the test is

$$\Delta X_t = \alpha + \gamma X_{t-1} + \sum_{i=1}^{p} \delta_i \Delta X_{t-i} + \varepsilon_t, \tag{5.4}$$

where

$$\Delta X_t = X_t - X_{t-1}. \tag{5.5}$$

The term $\alpha$ can account for a constant, while the terms

$$\Delta X_{t-1}, \Delta X_{t-2}, \ldots, \Delta X_{t-p} \tag{5.6}$$

capture part of the short-term dynamics of the series.
This enriched version gives the test its name, *Augmented Dickey–Fuller* : the Dickey–Fuller test is "augmented" by adding lags of the differences.
The key parameter is nevertheless $\gamma$. The test seeks to determine whether this coefficient is compatible with zero.
The hypotheses are generally written as

$$H_0 : \gamma = 0 \qquad \text{against} \qquad H_1 : \gamma < 0. \tag{5.7}$$

The null hypothesis corresponds to the presence of a unit root, and therefore to a non-stationary situation. Conversely, rejecting $H_0$ provides evidence in favor of stationarity.
It is important to understand the meaning of this formulation. The test does not directly attempt to "prove" that the series is stationary. It starts from the hypothesis that a unit root is present and determines whether the data provide sufficient evidence to reject this hypothesis.

**(iii)** In practice, Python returns, among other quantities, the ADF statistic and its $p$-value :

```
from statsmodels.tsa.stattools import adfuller

result = adfuller(returns)

print("ADF statistic :", result[0])
print("p-value       :", result[1])
```

The quantity that is generally easiest to interpret is the $p$-value.
**1.** If the $p$-value is small, the data are difficult to reconcile with the null hypothesis of a unit root. We therefore reject $H_0$ and consider the data to provide evidence in favor of stationarity.
**2.** Conversely, a large $p$-value means that we do not have sufficient evidence to reject the unit-root hypothesis. We therefore cannot conclude that the series is stationary.

**Remark 5.2.1** *Note that stationarity of returns does not mean that returns are independent or identically distributed. A return series can have a relatively stable mean while still exhibiting dependence in its volatility. This is precisely what we observed with the autocorrelation of squared returns and the phenomenon of volatility clustering.*

## 5.3 First Family of Models

### 5.3.1 AR Models

We can now return to our initial question :

Can the past of a series be used to predict its future value ?

The autoregressive model, or **AR**, provides a very simple answer to this question. In an AR(1) model, we write :

$$X_t = c + \phi X_{t-1} + \varepsilon_t.$$

The idea is intuitive : the current value depends on a constant, the past value, and a random shock. The term $\phi X_{t-1}$ represents precisely the information provided by the past.
If $\phi$ is positive, a high value of $X_{t-1}$ tends to be followed by a high value of $X_t$.
If $\phi$ is negative, we instead obtain an alternating relationship.
If $\phi$ is close to zero, the past value provides little linear information for predicting the current return.

**Stationarity condition** For an AR(1) model, a standard stationarity condition is :

$$|\phi| < 1.$$

If $|\phi|$ is below 1, the influence of a past shock gradually decreases.
Indeed, when tracing the model backward in time, we encounter terms proportional to :

$$\phi, \quad \phi^2, \quad \phi^3, \dots$$

and these terms converge to zero. The system therefore has memory, but this memory eventually disappears.
These properties explain why AR models can be interesting to test on returns, but also why they are often insufficient to capture the dynamics of financial risk.

### 5.3.2 MA and ARMA Models

Another way to represent a time series is to directly model the influence of past shocks. An MA($q$) model, or *Moving Average*, is written :

$$X_t = \mu + \varepsilon_t + \theta_1 \varepsilon_{t-1} + \cdots + \theta_q \varepsilon_{t-q}.$$

The idea is different from the AR model. The AR model uses past values of the series, whereas the MA model uses past shocks.
We can then combine the two mechanisms using an ARMA($p, q$) model :

$$X_t = c + \sum_{i=1}^{p} \phi_i X_{t-i} + \varepsilon_t + \sum_{j=1}^{q} \theta_j \varepsilon_{t-j}.$$

These models are particularly useful when the main objective is to model the dynamics of the **conditional mean**. But we have seen that the most interesting financial problem may lie elsewhere.
Even if $E[R_t \mid \mathcal{F}_{t-1}]$ is relatively stable, the conditional variance may change substantially :

$$\mathrm{Var}(R_t \mid \mathcal{F}_{t-1}) = \sigma_t^2.$$

We therefore need a model specifically devoted to this variance.

**Modeling risk rather than returns** Until now, we have sought to explain $R_t$. We will now seek to explain $\sigma_t^2$.
A very useful representation is to write

$$R_t = \mu_t + \varepsilon_t,$$

where $\mu_t$ represents the conditional mean and $\varepsilon_t$ represents the innovation or shock.
We can then decompose the shock as :

$$\varepsilon_t = \sigma_t z_t, \text{ with } E[z_t] = 0, \qquad \text{Var}(z_t) = 1.$$

The conditional variance then becomes :

$$\text{Var}(\varepsilon_t \mid \mathcal{F}_{t-1}) = \sigma_t^2.$$

The role of the volatility model is therefore to determine how $\sigma_t^2$ evolves over time.

**Remark 5.3.1** *If we use a constant variance*

$$\text{Var}(R_t) = \sigma^2,$$

*we assume that risk is identical every day. This means that a calm day and a crisis day have exactly the same theoretical volatility.*
*This assumption is unrealistic. Instead, we want to write :*

$$\text{Var}(R_t \mid \mathcal{F}_{t-1}) = \sigma_t^2.$$

*The quantity $\sigma_t^2$ can then increase following a large shock and gradually decrease when the market becomes calm again.*

### 5.3.3 The ARCH Model

The ARCH model, introduced by Engle, proposes an extremely intuitive idea :

Future volatility depends on the magnitude of recently observed errors.

In an ARCH($q$) model, we write :

$$\sigma_t^2 = \omega + \sum_{i=1}^{q} \alpha_i \varepsilon_{t-i}^2 \text{ with } \varepsilon_k | F_{k-1} = \sigma_k z_k \text{ where } z_k \text{ is a standardized random variable.}$$

For an ARCH(1) :

$$\sigma_t^2 = \omega + \alpha \varepsilon_{t-1}^2.$$

This equation is particularly easy to interpret. If the previous shock is small, $\varepsilon_{t-1}^2 \approx 0$, then $\sigma_t^2 \approx \omega$.
Conversely, if we observe a large shock, $|\varepsilon_{t-1}| \gg 0$, then $\varepsilon_{t-1}^2$ becomes large, which causes $\sigma_t^2$ to increase. The ARCH model therefore naturally reproduces volatility clustering.
The square plays an essential role here. A $+5\%$ shock and a $-5\%$ shock should both signal strong market activity. Since

$$(+5\%)^2 = (-5\%)^2,$$

this shows that the model is not trying to determine whether the market goes up or down. It is trying to measure the **magnitude of the shock**.
This property corresponds exactly to our earlier empirical observation : the sign of the return may be difficult to predict while its magnitude exhibits temporal dependence.

### 5.3.4 From ARCH to GARCH

The ARCH model nevertheless has an important limitation. To capture volatility that remains high for a long time, we need to include several lags of shocks :

$$\varepsilon_{t-1}^2, \varepsilon_{t-2}^2, \varepsilon_{t-3}^2, \dots$$

This can lead to a large number of parameters. The GARCH model then introduces an additional idea :

> Current volatility depends not only on recent shocks, but also on previously estimated volatility.

The conditional variance follows a GARCH$(p, q)$ model :

$$\boxed{\sigma_t^2 = \omega + \sum_{i=1}^{q} \alpha_i \varepsilon_{t-i}^2 + \sum_{j=1}^{p} \beta_j \sigma_{t-j}^2} \tag{5.8}$$

with

$$\omega > 0, \qquad \alpha_i \geq 0, \qquad \beta_j \geq 0. \tag{5.9}$$

Thus, conditional on the information available up to date $t-1$,

$$\boxed{\varepsilon_t \mid \mathcal{F}_{t-1} \sim \text{a centered distribution with variance } \sigma_t^2.} \tag{5.10}$$

The GARCH(1,1) model is :

$$\boxed{\sigma_t^2 = \omega + \alpha \varepsilon_{t-1}^2 + \beta \sigma_{t-1}^2}$$

This equation is one of the most widely used models for representing the dynamics of financial volatility.
The parameter $\omega$ represents a baseline level of variance. The parameter $\alpha$ measures the response of volatility to a new shock. When $\alpha$ is large, a large shock produces a large increase in conditional volatility. Finally, $\beta$ measures volatility persistence. When $\beta$ is large, high volatility today tends to remain high tomorrow.

**Remark 5.3.2 (i)** *Suppose that a calm period is characterized by low variance : $\sigma_t^2$ low. Then a large shock occurs : $|\varepsilon_t|$ high. The term $\alpha\varepsilon_t^2$ then increases the variance forecast for the following period. We therefore obtain :*

$$\sigma_{t+1}^2 = \omega + \alpha\varepsilon_t^2 + \beta\sigma_t^2.$$

*Volatility immediately increases. But the phenomenon does not stop there. In the following period, the term $\beta\sigma_{t+1}^2$ transmits part of this volatility into the future.*
*Thus, a single large shock can produce a succession of periods of high volatility. This is exactly the phenomenon observed in markets :*

$$\boxed{\textit{large shock} \to \textit{increase in volatility} \to \textit{persistence} \to \textit{gradual return to calm}}$$

*The GARCH model was therefore not introduced arbitrarily. It constitutes a direct response to an empirical property of financial markets.*
**(ii)** *In a GARCH(1,1), the quantity $\alpha + \beta$ is particularly important. When it is close to 1, volatility is highly persistent. A large shock therefore continues to influence forecasts for a long time.*
*Indeed, a standard second-order stationarity condition is*

$$\alpha \geq 0, \qquad \beta \geq 0, \qquad \alpha + \beta < 1.$$

*In this case, the unconditional variance is given by :*

$$\mathrm{Var}(\varepsilon_t) = \frac{\omega}{1 - \alpha - \beta} \text{ (see Appendix 1),}$$

*which helps us understand the role of $\alpha + \beta$, especially when this quantity is close to 1.*

- *Suppose $\alpha + \beta = 0.95$. This means that a large part of the effect of a shock is carried over from one period to the next.*
- *Conversely, if $\alpha + \beta = 0.50$, the effect of the shock disappears much more quickly.*

*This property is particularly important in risk management : a model that estimates strong persistence can forecast a period of high risk long after the initial shock.*

### 5.3.5 Volatility Forecasting

**One-step ahead :** Once the GARCH model has been estimated, we can use it to forecast future risk. The one-step-ahead forecast is :

$$\widehat{\sigma}^2_{t+1} = \omega + \alpha\varepsilon_t^2 + \beta\sigma_t^2.$$

At date $t$, we know $\varepsilon_t$ and $\sigma_t^2$. We can therefore produce an estimate of the variance of the next return. The corresponding volatility is simply :

$$\widehat{\sigma}_{t+1} = \sqrt{\widehat{\sigma}^2_{t+1}}.$$

This forecast can then be used in many financial applications : portfolio risk estimation, risk measures such as VaR, dynamic allocation, comparison of risk across different assets, and so on.
**Multiple horizons :** The model can also be used to forecast volatility over several periods. We define the unconditional variance :

$$\bar{\sigma}^2 = \frac{\omega}{1 - \alpha - \beta}.$$

The forecast at horizon $h$ can then be written :

$$E_t[\sigma^2_{t+h}] = \bar{\sigma}^2 + (\alpha + \beta)^{h-1} \left(\widehat{\sigma}^2_{t+1} - \bar{\sigma}^2\right).$$

This expression is particularly interesting. When $\alpha + \beta < 1$, the term $(\alpha + \beta)^{h-1}$ decreases as $h$ increases. The volatility forecast therefore gradually returns toward the unconditional variance :

$$E_t[\sigma^2_{t+h}] \longrightarrow \bar{\sigma}^2.$$

This captures the economic idea that an exceptional shock can have lasting consequences, but its influence eventually decreases.

**Remark 5.3.3** *The classical GARCH model uses $\varepsilon_{t-1}^2$. This means that a positive and a negative shock of the same magnitude have exactly the same effect on variance :*

$$(+5\%)^2 = (-5\%)^2.$$

*However, in some equity markets, bad news can cause a larger increase in volatility than good news. This is referred to as the* ***leverage effect*** *or volatility asymmetry.*
*The standard GARCH model does not directly capture this asymmetry, unlike certain extensions, notably : EGARCH and GJR-GARCH.*
*This observation once again shows that statistical modeling should start from properties observed in the data.*

## 5.4 Python Implementation of a GARCH Model

The `arch` library provides an implementation directly suited to financial volatility models. We can start by calculating returns :

```
import numpy as np
import pandas as pd

returns = prices.pct_change().dropna()
```

To estimate a GARCH(1,1) model with a constant mean and Student-t innovations :

```
from arch import arch_model

model = arch_model(
returns * 100,
mean="Constant",
vol="GARCH",
p=1,
q=1,
dist="t"
)

results = model.fit(disp="off")

print(results.summary())
```

The factor 100 simply expresses returns in percentage units rather than decimals.
**Reading the results**

The output of `results.summary()` provides estimates for $\omega$, $\alpha$ and $\beta$. We can then examine $\alpha + \beta$.
A high value indicates strong volatility persistence.
Standard errors, test statistics, and *p-values* can also be used to assess the statistical significance of the parameters.
We therefore recover a logic already encountered in the OLS regressions of the previous chapter :

$$\text{estimation} \rightarrow \text{uncertainty} \rightarrow \text{statistical test} \rightarrow \text{economic interpretation}.$$

**Volatility forecasting with Python :** Once the model has been estimated, we can produce forecasts :

```
forecast = results.forecast(
horizon=5,
reindex=False
)
```

```
5
6  variance_forecast = forecast.variance.iloc[-1]
7
8  volatility_forecast = np.sqrt(
9  variance_forecast
10 )
11
12 print(volatility_forecast)
```

The parameter **horizon=5** requests a forecast over the next five periods.
The variable **variance_forecast** contains the forecast variances. We then calculate $\sigma = \sqrt{\sigma^2}$ to return to a volatility expressed in the same units as the returns.

**A complete economic interpretation of the model :** Suppose that markets experience an exceptional movement today, i.e., we observe $|\varepsilon_t| \gg 0$. The model then increases the variance forecast through $\alpha\varepsilon_t^2$, so tomorrow's forecast volatility increases. Then the term $\beta\sigma_t^2$ transmits part of this volatility to subsequent periods and finally, if $\alpha + \beta < 1$, the effect of the shock gradually disappears.
We therefore obtain the mechanism :

| shock → increase in risk → persistence → return toward the normal level |
|---|

## 5.5 Some Comments

**(i)** Why is all this modeling useful to a quantitative analyst ?
Because portfolio risk depends directly on volatility and on dependencies between assets. In previous chapters, we used a covariance matrix for portfolio optimization

$$\sigma_P^2 = w^T \Sigma w.$$

We assumed this covariance matrix to be constant and therefore stable, which is not realistic. With conditional volatility models, we can consider a time-varying covariance matrix $\Sigma_t$, and portfolio variance then becomes

$$\sigma_{P,t}^2 = w_t^T \Sigma_t w_t.$$

**(ii)** Now imagine that a portfolio consists of a single stock. At date $t$, the GARCH model forecasts $\widehat{\sigma}_{t+1} = 2.5\%$, and the following day a crisis occurs and the model estimates $\widehat{\sigma}_{t+2} = 4.8\%$. Yet the portfolio has exactly the same composition. What has changed is the **risk state of the market**. A dynamic management strategy can then adapt its exposure

$$w_{t+1} = f(\widehat{\sigma}_{t+1}),$$

in order to reduce its exposure when forecast volatility increases.
In other words, the model is not necessarily used to predict future returns, but to control conditional risk.
**(iii)** It should be noted that estimating a model is not the final step. A fundamental question remains :

Has the model actually captured the temporal dependence present in the data ?

After estimation, we can study the standardized residuals

$$z_t = \frac{\varepsilon_t}{\sigma_t}.$$

If the model is correctly specified, these residuals should be much closer to an independent series. We can then examine $ACF(z_t)$ but also $ACF(z_t^2)$.
If the ACF of squared residuals remains strongly positive, this may indicate that the model has not completely captured the volatility dynamics.

## 5.6 Complete Python Example

We can now combine the different steps of the chapter into a single procedure.

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

from statsmodels.graphics.tsaplots import plot_acf
from statsmodels.tsa.stattools import adfuller
from arch import arch_model

# ============================================================

# 1. CALCULATION OF RETURNS

# ============================================================

returns = prices.pct_change().dropna()

# ============================================================

# 2. VISUALIZATION OF RETURNS

# ============================================================

plt.figure(figsize=(10, 4))

plt.plot(returns)

plt.title("Financial returns")
plt.xlabel("Date")
plt.ylabel("Return")

plt.show()

# ============================================================

# 3. STATIONARITY TEST

# ============================================================

adf_result = adfuller(returns)

print("ADF statistic :", adf_result[0])
print("p-value       :", adf_result[1])

# ============================================================

# 4. AUTOCORRELATION OF RETURNS

# ============================================================

plot_acf(
```

```
returns,
lags=30
)

plt.title("ACF of returns")
plt.show()

# ============================================================

# 5. AUTOCORRELATION OF SQUARED RETURNS

# ============================================================

squared_returns = returns ** 2

plot_acf(
squared_returns,
lags=30
)

plt.title("ACF of squared returns")
plt.show()

# ============================================================

# 6. GARCH(1,1) MODEL

# ============================================================

model = arch_model(
returns * 100,
mean="Constant",
vol="GARCH",
p=1,
q=1,
dist="t"
)

results = model.fit(
disp="off"
)

# ============================================================

# 7. RESULTS

# ============================================================

print(results.summary())

# ============================================================

# 8. VOLATILITY FORECAST

# ============================================================

forecast = results.forecast(
horizon=5,
reindex=False
)

```

```
variance_forecast = forecast.variance.iloc[-1]

volatility_forecast = np.sqrt(
variance_forecast
)

print("\nVolatility forecast:")
print(volatility_forecast)
```

## 5.7 Interpretation of the Overall Approach

It is useful to step back from the previous code.
We began with a simple financial question :

> Do returns exhibit temporal dependence ?

We then studied the autocorrelation of returns. If this autocorrelation is weak, we observed that this does not mean that all temporal structure has disappeared. We therefore studied $R_t^2$, which makes it possible to highlight dependence in the magnitude of movements. This led us to introduce a conditional variance $\sigma_t^2$.
The ARCH model then proposed making this variance depend on recent shocks :

$$\sigma_t^2 = \omega + \alpha\varepsilon_{t-1}^2.$$

The GARCH model added variance persistence :

$$\sigma_t^2 = \omega + \alpha\varepsilon_{t-1}^2 + \beta\sigma_{t-1}^2.$$

Finally, the model can be used to produce a forecast of future risk.

## 5.8 Exercises

### Exercise 1 — Returns and Stationarity

1. Download historical prices for a stock and calculate its daily returns.
2. Plot the prices and then the returns.
3. Apply the ADF test to both series.
4. Compare the results and explain why returns are generally more suitable for time-series modeling than prices.

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

from statsmodels.tsa.stattools import adfuller

# ============================================================

# 1. DOWNLOAD THE DATA

# ============================================================

ticker = "AAPL"

```

```
prices = yf.download(
ticker,
start="2021-01-01",
end="2026-01-01",
auto_adjust=True
)["Close"]

prices = prices.dropna()

# ============================================================

# 2. CALCULATION OF RETURNS

# ============================================================

returns = prices.pct_change().dropna()

# ============================================================

# 3. PRICE PLOT

# ============================================================

plt.figure(figsize=(10, 5))

plt.plot(prices)

plt.title(f"Price of {ticker}")
plt.xlabel("Date")
plt.ylabel("Price")

plt.grid(alpha=0.3)
plt.show()

# ============================================================

# 4. RETURN PLOT

# ============================================================

plt.figure(figsize=(10, 5))

plt.plot(returns)

plt.title(f"Daily returns of {ticker}")
plt.xlabel("Date")
plt.ylabel("Return")

plt.grid(alpha=0.3)
plt.show()

# ============================================================

# 5. ADF TEST ON PRICES

# ============================================================

adf_prices = adfuller(prices)

print("========================================")
print("ADF TEST - PRICES")
```

```
print("========================================")

print(f"ADF statistic : {adf_prices[0]:.4f}")
print(f"p-value       : {adf_prices[1]:.4f}")

# ============================================================

# 6. ADF TEST ON RETURNS

# ============================================================

adf_returns = adfuller(returns)

print("\n========================================")
print("ADF TEST - RETURNS")
print("========================================")

print(f"ADF statistic : {adf_returns[0]:.4f}")
print(f"p-value       : {adf_returns[1]:.4f}")

# ============================================================

# 7. AUTOMATIC INTERPRETATION

# ============================================================

if adf_prices[1] < 0.05:
print("\nPrices are stationary at the 5\% level.")
else:
print("\nWe do not reject the non-stationarity of prices.")

if adf_returns[1] < 0.05:
print("Returns are stationary at the 5\% level.")
else:
print("We do not reject the non-stationarity of returns.")
```

### Exercise 2 — Autocorrelation

1. Calculate and plot the ACF of returns for the first 30 lags.
2. Repeat the analysis for squared returns $R_t^2$.
3. Compare the two graphs.
4. If the ACF of returns is weak but the ACF of squared returns remains positive over several lags, explain what this means economically.

```

# ============================================================

# EXERCISE 2 -- Autocorrelation of returns

# and squared returns

# ============================================================

import numpy as np
import matplotlib.pyplot as plt

from statsmodels.graphics.tsaplots import plot_acf
from statsmodels.tsa.stattools import acf
```

```
15
16 # ------------------------------------------------------------
17
18 # 1. Calculation of logarithmic returns
19
20 # ------------------------------------------------------------
21
22 # We calculate the logarithmic return:
23
24 #
25
26 # R_t = ln(P_t / P_{t-1})
27
28 #
29
30 # The first observation is removed because there is no
31
32 # previous price P_{t-1} for the first date.
33
34 returns = np.log(prices / prices.shift(1)).dropna()
35
36 # ------------------------------------------------------------
37
38 # 2. Calculation of squared returns
39
40 # ------------------------------------------------------------
41
42 # Squared returns measure the magnitude of the movement
43
44 # without taking its direction into account.
45
46 #
47
48 # For example:
49
50 # (+3\%)^2 = (-3\%)^2
51
52 #
53
54 # We therefore use R_t^2 as a simple measure of the
55
56 # intensity of the market movement.
57
58 returns_squared = returns ** 2
59
60 # ------------------------------------------------------------
61
62 # 3. Autocorrelation of returns
63
64 # ------------------------------------------------------------
65
66 # Autocorrelation at lag k measures the correlation between
67
68 # return R_t and past return R_{t-k}:
69
70 #
71
72 # rho(k) = Corr(R_t, R_{t-k})
73
74 #
75
```

```
# If autocorrelations are close to zero, this means that

# past returns contain little information about future returns.

fig, ax = plt.subplots(figsize=(10, 4))

plot_acf(
returns,
lags=30,        # We examine the first 30 lags
ax=ax,
alpha=0.05      # 95\% confidence interval
)

ax.set_title("Autocorrelation of returns")
ax.set_xlabel("Lag")
ax.set_ylabel("Autocorrelation")

plt.tight_layout()
plt.show()

# ------------------------------------------------------------

# 4. Autocorrelation of squared returns

# ------------------------------------------------------------

# We now perform exactly the same analysis

# on squared returns.

#

# This step is particularly important in finance:

# returns may be weakly autocorrelated while

# their squares exhibit substantial autocorrelation.

#

# This reveals the phenomenon of

# "volatility clustering".

fig, ax = plt.subplots(figsize=(10, 4))

plot_acf(
returns_squared,
lags=30,
ax=ax,
alpha=0.05
)

ax.set_title("Autocorrelation of squared returns")
ax.set_xlabel("Lag")
ax.set_ylabel("Autocorrelation")

plt.tight_layout()
plt.show()

# ------------------------------------------------------------
```

```
137
138 # 5. Numerical calculation of autocorrelations
139
140 # ------------------------------------------------------------
141
142 # The acf() function directly returns
143
144 # the autocorrelation coefficients.
145
146 acf_returns = acf(
147 returns,
148 nlags=10
149 )
150
151 acf_squared = acf(
152 returns_squared,
153 nlags=10
154 )
155
156 # ------------------------------------------------------------
157
158 # 6. Display of return autocorrelations
159
160 # ------------------------------------------------------------
161
162 print("Autocorrelation of returns:")
163
164 for lag, value in enumerate(acf_returns):
165 print(f"Lag {lag}: {value:.4f}")
166
167 # ------------------------------------------------------------
168
169 # 7. Display of squared-return autocorrelations
170
171 # ------------------------------------------------------------
172
173 print("\nAutocorrelation of squared returns:")
174
175 for lag, value in enumerate(acf_squared):
176 print(f"Lag {lag}: {value:.4f}")
177
178 # ------------------------------------------------------------
179
180 # 8. Interpretation
181
182 # ------------------------------------------------------------
183
184 # Lag 0 is always equal to 1 because a variable is perfectly
185
186 # correlated with itself.
187
188 #
189
190 # We are therefore mainly interested in the values at
191
192 # lags 1, 2, 3, ...
193
194 #
195
196 # We generally look for:
197
```

```
#

# - weak autocorrelation of returns;

# - stronger autocorrelation of squared returns.

#

# This second property means that periods of high volatility

# tend to be followed by other periods of high volatility,

# while calm periods tend to be followed by other calm periods.

#

# This is the phenomenon of "volatility clustering".

#

# This observation provides an essential motivation

# for ARCH and GARCH models.
```

### Exercise 3 — Estimation of a GARCH(1,1)

1. Estimate a model :

$$R_t = \mu + \varepsilon_t \text{ with } \varepsilon_t = \sigma_t z_t$$

and

$$\sigma_t^2 = \omega + \alpha \varepsilon_{t-1}^2 + \beta \sigma_{t-1}^2.$$

2. Identify the parameters $\omega$, $\alpha$ and $\beta$.
3. Calculate $\alpha + \beta$ and interpret this quantity in terms of volatility persistence.
4. Produce a volatility forecast at :

$$h = 1, \quad h = 5, \quad h = 20 \text{ days.}$$

5. Plot the forecasts.
6. Analyze how forecast volatility gradually returns toward its long-run level.

```
# ============================================================
# SOLUTION - GARCH(1,1) AND VOLATILITY FORECASTING
# ============================================================
#
# The model considered is:
#
#     R_t = mu + epsilon_t
#
# with
#
#     epsilon_t = sigma_t * z_t
#
```

```
# and
#
#     sigma_t^2 =
#         omega
#         + alpha * epsilon_{t-1}^2
#         + beta * sigma_{t-1}^2
#
# The objective is to:
#   1. Estimate the GARCH(1,1) model,
#   2. Identify omega, alpha and beta,
#   3. Calculate alpha + beta,
#   4. Interpret volatility persistence,
#   5. Forecast volatility at 1, 5 and 20 days,
#   6. Plot the forecasts,
#   7. Observe the gradual return toward the long-run volatility level.
#
# ============================================================


# ------------------------------------------------------------
# 0. IMPORT THE REQUIRED LIBRARIES
# ------------------------------------------------------------

import numpy as np
import matplotlib.pyplot as plt
from arch import arch_model


# ------------------------------------------------------------
# 1. CALCULATE RETURNS
# ------------------------------------------------------------

# We assume that we already have a price series called "prices".
#
# We calculate logarithmic returns:
#
#     R_t = log(P_t / P_{t-1})
#
# We then express returns in percentage terms.
#
# For example:
#
#     0.01  ->  1 %
#
# Scaling returns by 100 generally makes the numerical
# estimation of the GARCH model more convenient.

returns = np.log(prices / prices.shift(1)).dropna()

returns_pct = returns * 100


# ------------------------------------------------------------
# 2. ESTIMATE THE GARCH(1,1) MODEL
# ------------------------------------------------------------

# The model we want to estimate is:
#
#     R_t = mu + epsilon_t
#
#     epsilon_t = sigma_t * z_t
```

```
#
#     sigma_t^2 =
#         omega
#         + alpha * epsilon_{t-1}^2
#         + beta * sigma_{t-1}^2
#
# In arch_model:
#
#     mean="Constant"
#         -> the conditional mean is a constant mu
#
#     vol="GARCH"
#         -> we use a GARCH volatility model
#
#     p=1
#         -> one lag of past conditional variance
#
#     q=1
#         -> one lag of squared innovations
#
#     dist="normal"
#         -> z_t follows a standard normal distribution.
#
# Important:
# In the "arch" library, p refers to lags of the variance
# and q refers to lags of squared innovations.

model = arch_model(
    returns_pct,
    mean="Constant",
    vol="GARCH",
    p=1,
    q=1,
    dist="normal"
)

# The fit() method estimates the model parameters.
#
# The parameters are not obtained from a simple algebraic
# formula. They are estimated by maximum likelihood.

results = model.fit()

# Display the complete estimation results:
print(results.summary())


# ------------------------------------------------------------
# 3. IDENTIFY THE PARAMETERS
# ------------------------------------------------------------

# We extract the estimated parameters from the results object.

mu = results.params["mu"]

omega = results.params["omega"]

alpha = results.params["alpha[1]"]

beta = results.params["beta[1]"]

```

```
135
136 print("\n========================================")
137 print("ESTIMATED PARAMETERS")
138 print("========================================")
139
140 print(f"mu    = {mu:.6f}")
141 print(f"omega = {omega:.6f}")
142 print(f"alpha = {alpha:.6f}")
143 print(f"beta  = {beta:.6f}")
144
145
146 # The estimated GARCH(1,1) model is therefore:
147 #
148 #     sigma_t^2 =
149 #         omega_hat
150 #         + alpha_hat * epsilon_{t-1}^2
151 #         + beta_hat * sigma_{t-1}^2
152 #
153 # Interpretation:
154 #
155 # omega:
156 #     represents the baseline level of the variance.
157 #
158 # alpha:
159 #     measures how strongly volatility reacts to a new shock.
160 #     If alpha is large, a large shock epsilon_{t-1}
161 #     produces a strong increase in current volatility.
162 #
163 # beta:
164 #     measures the persistence of past volatility.
165 #     If beta is large, a period of high volatility tends
166 #     to be followed by another period of high volatility.
167
168
169 # ------------------------------------------------------------
170 # 4. CALCULATE alpha + beta
171 # ------------------------------------------------------------
172
173 # The quantity alpha + beta is fundamental in a GARCH(1,1).
174 #
175 # It measures the overall persistence of volatility:
176 #
177 #     alpha + beta
178 #
179 # If alpha + beta is close to 1, volatility is highly persistent.
180 #
181 # If alpha + beta is smaller, the effect of shocks disappears
182 # more quickly.
183
184 persistence = alpha + beta
185
186 print("\n========================================")
187 print("VOLATILITY PERSISTENCE")
188 print("========================================")
189
190 print(f"alpha + beta = {persistence:.6f}")
191
192
193 # For a stationary variance, we require:
194 #
195 #     alpha + beta < 1
```

```
#
# In this case, volatility gradually returns toward
# its long-run level.

if persistence < 1:

    print("The condition alpha + beta < 1 is satisfied.")
    print("Volatility is persistent but gradually returns")
    print("toward its long-run level.")

elif np.isclose(persistence, 1):

    print("alpha + beta is close to 1.")
    print("Volatility is extremely persistent.")

else:

    print("alpha + beta >= 1.")
    print("A finite unconditional variance is not guaranteed.")


# ------------------------------------------------------------
# 5. CALCULATE THE LONG-RUN VOLATILITY
# ------------------------------------------------------------

# For a GARCH(1,1), if alpha + beta < 1,
# the long-run variance is:
#
#     sigma_bar^2 = omega / (1 - alpha - beta)
#
# This is the level toward which the variance forecasts
# converge as the forecast horizon becomes large.

long_run_variance = omega / (1 - alpha - beta)

# Since volatility is the square root of variance:
#
#     sigma_bar = sqrt(sigma_bar^2)

long_run_volatility = np.sqrt(long_run_variance)

print("\n========================================")
print("LONG-RUN LEVEL")
print("========================================")

print(f"Long-run variance     = {long_run_variance:.6f}")
print(f"Long-run volatility   = {long_run_volatility:.4f}%")


# ------------------------------------------------------------
# 6. FORECAST THE VARIANCE FOR THE NEXT 20 DAYS
# ------------------------------------------------------------

# The GARCH model can now be used to forecast future variance.
#
# The one-step-ahead forecast is:
#
#     sigma_{t+1|t}^2
#       = omega
#       + alpha * epsilon_t^2
#       + beta * sigma_t^2
```

```
#
# For subsequent horizons:
#
#     sigma_{t+h|t}^2
#       = omega
#       + (alpha + beta)
#         * sigma_{t+h-1|t}^2
#
# This recursive equation explains why volatility gradually
# returns toward its long-run level.

forecast = results.forecast(
    horizon=20,
    reindex=False
)


# The library directly provides forecasts of the conditional
# variance.

variance_forecast = forecast.variance.iloc[-1]


# We want volatility rather than variance.
#
# Therefore:
#
#     volatility = sqrt(variance)

volatility_forecast = np.sqrt(variance_forecast)


# ------------------------------------------------------------
# 7. FORECASTS AT h = 1, 5 AND 20 DAYS
# ------------------------------------------------------------

# The first observation corresponds to h = 1.
# The fifth observation corresponds to h = 5.
# The twentieth observation corresponds to h = 20.

vol_h1 = volatility_forecast.iloc[0]

vol_h5 = volatility_forecast.iloc[4]

vol_h20 = volatility_forecast.iloc[19]


print("\n========================================")
print("VOLATILITY FORECASTS")
print("========================================")

print(f"Volatility at h = 1 day  : {vol_h1:.4f}%")
print(f"Volatility at h = 5 days : {vol_h5:.4f}%")
print(f"Volatility at h = 20 days: {vol_h20:.4f}%")


# ------------------------------------------------------------
# 8. PLOT THE VOLATILITY FORECASTS
# ------------------------------------------------------------

# We create the forecast horizons:
```

```
#
#     1, 2, ..., 20 days

horizons = np.arange(1, 21)


plt.figure(figsize=(10, 5))


# Plot the predicted volatility.

plt.plot(
    horizons,
    volatility_forecast.values,
    marker="o",
    label="Forecast volatility"
)


# Add a horizontal line corresponding to the long-run
# volatility.
#
# This allows us to visualize the gradual convergence
# toward the long-run level.

plt.axhline(
    long_run_volatility,
    linestyle="--",
    label="Long-run volatility"
)


plt.xlabel("Forecast horizon (days)")

plt.ylabel("Forecast volatility (%)")

plt.title("GARCH(1,1) Volatility Forecast")

plt.legend()

plt.grid(True)

plt.tight_layout()

plt.show()


# ------------------------------------------------------------
# 9. INTERPRETATION
# ------------------------------------------------------------

# The graph illustrates an important property of GARCH models.
#
# Suppose that current volatility is above its long-run level.
#
# The short-term forecast will also be relatively high:
#
#     sigma_{t+1|t}^2
#
# However, as the forecast horizon increases, the influence
# of the current shock gradually decreases.
```

```
#
# Indeed:
#
#     sigma_{t+h|t}^2
#       = omega
#       + (alpha + beta)
#         * sigma_{t+h-1|t}^2
#
# If:
#
#     alpha + beta < 1
#
# the influence of the current volatility is progressively
# reduced because it is multiplied by a number smaller than 1.
#
# Consequently, the variance forecasts converge toward:
#
#     omega / (1 - alpha - beta)
#
# and the volatility forecasts converge toward:
#
#     sqrt(omega / (1 - alpha - beta)).
#
#
# Therefore:
#
#     alpha + beta close to 1
#         -> very persistent volatility
#         -> slow return toward the long-run level
#
#     smaller alpha + beta
#         -> less persistent volatility
#         -> faster return toward the long-run level
#
#
# ============================================================
# KEY TAKEAWAY
# ============================================================
#
# The GARCH(1,1) model allows us to go from historical returns
# to a dynamic forecast of future volatility:
#
#     Observed returns
#             |
#             v
#     Estimate GARCH(1,1)
#             |
#             v
#     omega, alpha, beta
#             |
#             v
#     Calculate alpha + beta
#             |
#             v
#     Measure volatility persistence
#             |
#             v
#     Forecast future volatility
#             |
#             v
#     Gradual return toward
```

```
#     the long-run volatility level
#
# The two key results to remember are:
#
#     alpha + beta < 1
#
# and
#
#     long-run variance
#       = omega / (1 - alpha - beta)
#
# ============================================================
```

## Appendix 1 – Conditional Expectation and Conditional Variance in a GARCH Model

In a GARCH model, we write the innovation as

$$\varepsilon_t = \sigma_t z_t,$$

where $z_t$ is a standardized noise satisfying

$$\mathbb{E}[z_t] = 0, \qquad \mathbb{E}[z_t^2] = 1$$

and where $\sigma_t^2$ represents the conditional variance at time $t$.
Conditioning on the information available up to $t-1$, denoted by $\mathcal{F}_{t-1}$, we obtain

$$\mathbb{E}[\varepsilon_t^2 \mid \mathcal{F}_{t-1}] = \mathbb{E}[\sigma_t^2 z_t^2 \mid \mathcal{F}_{t-1}].$$

Since $\sigma_t^2$ is determined by past information, it is known conditional on $\mathcal{F}_{t-1}$. It can therefore be taken outside the expectation :

$$\mathbb{E}[\varepsilon_t^2 \mid \mathcal{F}_{t-1}] = \sigma_t^2 \mathbb{E}[z_t^2 \mid \mathcal{F}_{t-1}].$$

Furthermore, since $z_t$ is a new shock, independent of past information, and since $\mathbb{E}[z_t^2] = 1$, we obtain

$$\boxed{\mathbb{E}[\varepsilon_t^2 \mid \mathcal{F}_{t-1}] = \sigma_t^2}$$

In other words, the conditional variance of the innovation is precisely $\sigma_t^2$ :

$$\boxed{\mathrm{Var}(\varepsilon_t \mid \mathcal{F}_{t-1}) = \sigma_t^2}$$

Now consider a GARCH$(p, q)$ model :

$$\sigma_t^2 = \omega + \sum_{i=1}^{q} \alpha_i \varepsilon_{t-i}^2 + \sum_{j=1}^{p} \beta_j \sigma_{t-j}^2.$$

Taking expectations on both sides of this identity, we can easily identify the unconditional variance :

$$\boxed{\mathbb{E}[\sigma_t^2] = \frac{\omega}{1 - \sum_{i=1}^{q} \alpha_i - \sum_{j=1}^{p} \beta_j}}$$

which is finite if and only if

$$\boxed{\sum_{i=1}^{q} \alpha_i + \sum_{j=1}^{p} \beta_j < 1.}$$

In the particular case of GARCH$(1, 1)$,

$$\sigma_t^2 = \omega + \alpha \varepsilon_{t-1}^2 + \beta \sigma_{t-1}^2,$$

the condition becomes

$$\boxed{\alpha + \beta < 1}$$

and the unconditional variance is

$$\boxed{\mathbb{E}[\sigma_t^2] = \frac{\omega}{1-\alpha-\beta}.}$$

## Appendix 2 – Parameter Estimation and Forecasting in a GARCH(1,1) Model

Once the GARCH(1,1) model has been specified, an essential question arises : how do we determine the values of the parameters $\omega$, $\alpha$ and $\beta$ from the data ?
Consider the model

$$R_t = \mu + \varepsilon_t \text{ with } \varepsilon_t = \sigma_t z_t \text{ and } \mathbb{E}[z_t] = 0,\ \mathbb{E}[z_t^2] = 1,$$

and

$$\boxed{\sigma_t^2 = \omega + \alpha\varepsilon_{t-1}^2 + \beta\sigma_{t-1}^2}.$$

The question is how to estimate the parameters $\omega, \alpha$, and $\beta$.
Suppose that we have a series of observed returns

$$R_1, R_2, \ldots, R_T.$$

For given values of $\omega$, $\alpha$ and $\beta$, the GARCH model allows us to recursively calculate the conditional variances :

$$\sigma_t^2 = \omega + \alpha\varepsilon_{t-1}^2 + \beta\sigma_{t-1}^2. \tag{5.11}$$

Thus, for any combination of parameters $(\omega, \alpha, \beta)$, we can calculate a complete conditional-variance path :

$$\sigma_1^2, \sigma_2^2, \ldots, \sigma_T^2.$$

The objective is then to find the values of $\omega$, $\alpha$ and $\beta$ that allow the model to reproduce as closely as possible the observed dynamics of returns.

**Maximum likelihood :** To understand how the parameters are actually obtained, suppose that the standardized innovations follow a normal distribution :

$$z_t \overset{iid}{\sim} \mathcal{N}(0,1). \tag{5.12}$$

Since $\varepsilon_t = \sigma_t z_t$, conditional on the information available up to $t-1$,

$$\varepsilon_t \mid \mathcal{F}_{t-1} \sim \mathcal{N}(0, \sigma_t^2).$$

The conditional density of $\varepsilon_t$ is therefore

$$f(\varepsilon_t \mid \mathcal{F}_{t-1}) = \frac{1}{\sqrt{2\pi\sigma_t^2}} \exp\left(-\frac{\varepsilon_t^2}{2\sigma_t^2}\right). \tag{5.13}$$

For all observations, the likelihood is given by

$$L(\omega, \alpha, \beta) = \prod_{t=1}^{T} f(\varepsilon_t \mid \mathcal{F}_{t-1}). \tag{5.14}$$

It is more convenient to work with the logarithm of the likelihood. We then obtain the log-likelihood :

$$\boxed{\ell(\omega, \alpha, \beta) = -\frac{1}{2} \sum_{t=1}^{T} \left[ \log(2\pi) + \log(\sigma_t^2) + \frac{\varepsilon_t^2}{\sigma_t^2} \right]}. \tag{5.15}$$

Parameter estimation therefore consists of solving the following optimization problem :

$$\boxed{(\hat{\omega}, \hat{\alpha}, \hat{\beta}) = \underset{\omega,\alpha,\beta}{\arg, \max;} \ell(\omega, \alpha, \beta)}. \tag{5.16}$$

In other words, the algorithm searches for the values of $\omega$, $\alpha$ and $\beta$ that maximize the likelihood of the returns actually observed.
It is important to understand that $\alpha$ and $\beta$ are therefore not obtained from a direct formula. They are obtained through a numerical optimization procedure.

**Parameter estimation with Python :** In practice, we do not perform the likelihood optimization manually. The Python library `arch` allows us to estimate the GARCH parameters directly.
The model can be specified as follows :

```
from arch import arch_model

# Construction of the GARCH(1,1) model

model = arch_model(
returns * 100,
mean="Constant",
vol="GARCH",
p=1,
q=1,
dist="normal"
)

# Parameter estimation by maximum likelihood

results = model.fit()

# Display the results

print(results.summary())
```

The options `p=1, q=1` correspond here to a GARCH(1,1) model, according to the convention used by the `arch` library. The option `dist="normal"` assumes that the standardized innovations $z_t$ follow a normal distribution.
The estimated parameters can then be retrieved directly :

```

# Retrieve the estimated parameters

omega = results.params["omega"]
alpha = results.params["alpha[1]"]
beta = results.params["beta[1]"]
```

```
7
8  print("omega =", omega)
9  print("alpha =", alpha)
10 print("beta  =", beta)
```

For example, we may obtain

$$\hat{\omega} = 0.020, \qquad \hat{\alpha} = 0.100, \qquad \hat{\beta} = 0.850.$$

We can also calculate volatility persistence :

```
1
2  # Volatility persistence
3
4  persistence = alpha + beta
5
6  print("Persistence =", persistence)
```

In our example,

$$\hat{\alpha} + \hat{\beta} = 0.100 + 0.850 = 0.950. \tag{5.17}$$

Since this value is close to 1, volatility is highly persistent.

**Variance forecasting** Once the parameters have been estimated, they can be used to forecast future variance. The one-step-ahead variance forecast is :

$$\boxed{\hat{\sigma}^2_{t+1|t} = \hat{\omega} + \hat{\alpha}\varepsilon_t^2 + \hat{\beta}\sigma_t^2}. \tag{5.18}$$

To forecast variance two periods ahead, we must account for the fact that $\varepsilon_{t+1}$ has not yet been observed. We therefore use the conditional expectation :

$$\mathbb{E} * t[\varepsilon * t + 1^2] = \sigma^2_{t+1}. \tag{5.19}$$

We therefore obtain

$$\hat{\sigma}^2_{t+2|t} = \hat{\omega} + (\hat{\alpha} + \hat{\beta})\hat{\sigma}^2_{t+1|t}. \tag{5.20}$$

More generally, for a horizon $h \geq 2$,

$$\boxed{\hat{\sigma}^2_{t+h|t} = \hat{\omega} + (\hat{\alpha} + \hat{\beta})\hat{\sigma}^2_{t+h-1|t}}. \tag{5.21}$$

This relationship directly shows the role of $\alpha + \beta$ in the persistence of volatility forecasts.

**Forecasting with Python** The `arch` library can directly calculate variance forecasts :

```
1
2  # Forecast variance over the next 20 periods
3
4  forecast = results.forecast(horizon=20)
5
6  # Retrieve the forecast variances
7
8  variance_forecast = forecast.variance.iloc[-1]
9
10 print(variance_forecast)
```

The volatility forecasts are then obtained by taking the square root of the forecast variances :

```
# Convert variance into volatility

volatility_forecast = variance_forecast ** 0.5

print(volatility_forecast)
```

The interest of the GARCH model is therefore that it allows us both to estimate the temporal dynamics of volatility and to use this dynamics to forecast future risk.

## Appendix 3 – From ARMA to GARCH : Modeling the Mean and Volatility

The objective of an ARMA-GARCH model is to distinguish two sources of dynamics in a financial time series. On the one hand, returns may exhibit some temporal dependence in their **mean**. On the other hand, even when this dependence is weak, their **variance** may evolve over time and exhibit periods of high and low volatility. The idea is therefore to model these two components separately.

**Step 1 : Modeling the mean with an ARMA model** We start with a series of observed returns

$$R_1, R_2, \ldots, R_T.$$

If returns exhibit temporal dependence, we can use an ARMA model to describe their conditional mean. For example, an ARMA$(1,1)$ model is :

$$R_t = c + \phi R_{t-1} + \theta \varepsilon_{t-1} + \varepsilon_t, \tag{5.22}$$

where $\varepsilon_t$ represents the unpredictable innovation at date $t$.
Under the classical assumption,

$$\varepsilon_t \overset{iid}{\sim} \mathcal{N}(0, \sigma^2). \tag{5.23}$$

The parameters $c$, $\phi$ and $\theta$ are unknown and must be estimated from the observed returns, generally by maximum likelihood. We therefore obtain the estimators

$$\hat{c}, \qquad \hat{\phi}, \qquad \hat{\theta},$$

as well as a series of estimated residuals

$$\hat{\varepsilon}_1, \hat{\varepsilon}_2, \ldots, \hat{\varepsilon}_T.$$

The purpose of this first step is to remove from the returns any dynamics in their mean that may be explained by the ARMA model. The residuals then represent the part of the returns that is not explained by the ARMA model.
In Python, this estimation can be performed using the `ARIMA` class from `statsmodels`. An ARMA$(1,1)$ corresponds to an ARIMA$(1,0,1)$ :

```
from statsmodels.tsa.arima.model import ARIMA

# Estimation of an ARMA(1,1)

# ARMA(p,q) = ARIMA(p,0,q)

```

```
model_arma = ARIMA(
returns,
order=(1, 0, 1)
)

results_arma = model_arma.fit()

# Display the results

print(results_arma.summary())
```

The estimated parameters can be retrieved from the `results_arma` object. The model residuals are then obtained with

```

# Residuals of the ARMA model

residuals = results_arma.resid
```

**Step 2 : Using the residuals to estimate the GARCH model** In an ARMA-GARCH approach, the innovations used in this second step are the residuals obtained from the mean model.

The `arch` library can be used to estimate the volatility model. For example, for a GARCH$(1,1)$ :

```
from arch import arch_model

# GARCH(1,1) model applied to the ARMA residuals

model_garch = arch_model(
residuals * 100,
mean="Zero",
vol="GARCH",
p=1,
q=1,
dist="normal"
)

# Maximum likelihood estimation

results_garch = model_garch.fit()

# Display the results

print(results_garch.summary())
```

We then retrieve the estimated parameters :

```

# Estimated parameters of the GARCH(1,1)

omega = results_garch.params["omega"]
alpha = results_garch.params["alpha[1]"]
beta = results_garch.params["beta[1]"]

print("omega =", omega)
print("alpha =", alpha)
print("beta  =", beta)
```

We can also calculate volatility persistence :

```

# Volatility persistence
```

```
3
4 persistence = alpha + beta
5
6 print("alpha + beta =", persistence)
```

For a GARCH$(1,1)$ model, the condition

$$\boxed{\alpha + \beta < 1} \tag{5.24}$$

ensures the existence of a finite unconditional variance. When $\alpha + \beta$ is close to 1, volatility is highly persistent.
From this point onward, we can forecast volatility as explained previously.

# Chapitre 6

# Portfolio Optimization and Dynamic Risk Management with Python

## 6.1 Introduction

In the previous chapter, we studied financial time series and how ARCH and GARCH models represent volatility that changes over time. In particular, we saw that risk does not necessarily remain constant : after a major shock, volatility may increase and remain elevated for several periods.
A natural question then arises : **how can this information about risk be used to construct and manage a financial portfolio ?**
In the chapters devoted to Markowitz, we constructed portfolios using expected returns and the covariance matrix. In this approach, portfolio variance is written as

$$\sigma_P^2 = w^\top \Sigma w,$$

where $w$ denotes the vector of portfolio weights and $\Sigma$ the return covariance matrix. This formulation remains fundamental, but it often relies on an estimate of $\Sigma$ that is assumed to remain fixed throughout the investment period.
However, the previous chapter led us to a more realistic representation : volatility and, more generally, the dependence structure between assets can change. We will therefore extend the mean–variance approach by distinguishing parameters estimated from data, weights selected by the optimizer, and the risk actually observed after the portfolio has been constructed.
The aim is not to assume that an algorithm can predict markets perfectly. Rather, it is to build a coherent workflow.
We will use `yfinance` to retrieve market data, `pandas` and `numpy` to manipulate them, and then `scipy.optimize` to solve a constrained optimization problem. The risk measures introduced in previous chapters, particularly VaR, will then be used to evaluate the portfolio out of sample.

## 6.2 From Historical Prices to Portfolio Returns

Consider a portfolio consisting of several assets. For each asset $i$, we have an adjusted price $P_{i,t}$ at date $t$. Adjusted prices are preferable to raw prices for return analysis because, depending on the adjustments provided by the source, they account for events such as dividends and stock splits.
We will illustrate the process using a small universe of liquid stocks. The securities selected here are not an investment recommendation ; they are used solely to illustrate the calculation steps.

```python
import yfinance as yf
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# List of assets under study.
# Each symbol corresponds to a listed security available from Yahoo Finance.
tickers = ["AAPL", "MSFT", "JPM", "XOM"]

# Download adjusted prices over a historical period.
# auto_adjust=True asks yfinance to adjust prices
# to account for supported corporate actions.
prices = yf.download(
    tickers,
    start="2018-01-01",
    end="2025-01-01",
    auto_adjust=True,
    progress=False
)["Close"]

# Remove dates for which all prices are not available.
# This step ensures that each retained row has an observation
# for every asset in the universe.
prices = prices.dropna(how="any")

print(prices.head())
print(prices.tail())
```

Listing 6.1 – Downloading prices with yfinance

The `download` method returns a date-indexed table. Each column corresponds to an asset. The structure of the result should always be checked : financial data may contain missing dates, and the formats returned by a library may vary depending on the parameters or version used.

**Remark 6.2.1** *Downloading historical data does not guarantee that the data are perfectly clean. In empirical work, missing values, common dates, changes in universe composition, and source quality must be checked. An optimization can be numerically correct yet economically misleading if the input data are poorly prepared.*

**Calculating returns $R_t$** As in the previous chapter, we denote the return observed at date $t$ by $R_t$. For an asset whose price moves from $P_{t-1}$ to $P_t$, the simple return is defined as

$$R_t = \frac{P_t - P_{t-1}}{P_{t-1}} = \frac{P_t}{P_{t-1}} - 1.$$

The logarithmic return can also be used :

$$r_t = \ln\left(\frac{P_t}{P_{t-1}}\right).$$

In what follows, we retain the notation $R_t$ for simple returns, to remain consistent with the portfolio notation used in previous chapters. For small daily returns, the two conventions often produce similar values, but they are not identical.
For several assets, $R_t$ denotes the vector of returns at date $t$ :

$$R_t = \begin{pmatrix} R_{1,t} \\ R_{2,t} \\ \vdots \\ R_{n,t} \end{pmatrix}.$$

```python
# The simple return is the ratio between two consecutive prices,
# minus 1. pct_change() applies this calculation column by column.
# The first observation becomes missing because no previous price
# is available to calculate the first return.
R = prices.pct_change().dropna()

# Quick check of dimensions and descriptive statistics.
print("Dimensions de R :", R.shape)
print(R.describe())

# Plot the returns: variations are more visible
# than on the price chart because they are centered around zero.
R.plot(figsize=(12, 5), title="Daily asset returns")
plt.ylabel("Simple return")
plt.grid(alpha=0.3)
plt.tight_layout()
plt.show()
```

Listing 6.2 – Calculation and Visualization of Returns

The matrix $R$ contains historical returns. Each row corresponds to a date and each column to an asset. We will use part of these observations to estimate the parameters, then reserve a later period to evaluate the strategy. This separation is essential : if the same data are used both to construct and evaluate the portfolio, the resulting performance may be overly optimistic.

**Separating estimation and evaluation** We split the observations chronologically : the earlier data are used to estimate the parameters, while the later data are used to test the portfolio. Dates should not be shuffled randomly, as this would indirectly use future information to make decisions that should have been made in the past.

```python
# Use the first 80% of dates to estimate the parameters.
# Reserve the end of the sample for out-of-sample evaluation.
split = int(0.80 * len(R))

R_train = R.iloc[:split].copy()
R_test = R.iloc[split:].copy()

print("Start of estimation sample:", R_train.index.min())
print("End of estimation sample:", R_train.index.max())
print("Start of test sample:", R_test.index.min())
print("End of test sample:", R_test.index.max())
```

Listing 6.3 – Chronological Train-Test Split

This method does not eliminate the risk of overfitting, but it allows us to answer a more realistic question : do the weights obtained from information available in the past behave reasonably over a period that was not used to calculate them ?

## 6.3 Estimating Return and Risk

**The vector of mean returns** Using the estimation sample, we calculate the mean return of each asset. If $T$ observations are available, the empirical mean return of asset $i$ is

$$\widehat{\mu}_i = \frac{1}{T} \sum_{t=1}^{T} R_{i,t}.$$

The vector of estimated means is denoted by $\widehat{\mu}$.

Daily mean returns are generally small and estimated with substantial uncertainty. An optimizer that gives them excessive weight can therefore produce unstable allocations. Thus, $\widehat{\mu}$ should not be interpreted as exact knowledge of future returns.

**The covariance matrix** The empirical covariance matrix is denoted by $\widehat{\Sigma}$. Its $(i, j)$ element measures the historical co-movement between the returns of assets $i$ and $j$. The diagonal elements are variances, while the off-diagonal elements are covariances :

$$\widehat{\Sigma} = \begin{pmatrix} \widehat{\mathrm{Var}}(R_1) & \widehat{\mathrm{Cov}}(R_1, R_2) & \cdots \\ \widehat{\mathrm{Cov}}(R_2, R_1) & \widehat{\mathrm{Var}}(R_2) & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix}.$$

For a weight vector $w$, the estimated portfolio variance is written as

$$\widehat{\sigma}_P^2 = w^\top \widehat{\Sigma} w, \qquad \widehat{\sigma}_P = \sqrt{w^\top \widehat{\Sigma} w}.$$

Diversification therefore depends not only on each asset's individual risk, but also on their dependence on one another.

```
# Daily mean of each column in R_train.
mu_hat = R_train.mean()

# Daily covariance matrix estimated over the same period.
Sigma_hat = R_train.cov()

# Indicative annualization for daily data:
# approximately 252 trading sessions per year.
# The mean is multiplied by 252 and the covariance by 252.
mu_ann = 252 * mu_hat
Sigma_ann = 252 * Sigma_hat

print("Annualized mean returns:")
print(mu_ann)
print("\nAnnualized covariance matrix:")
print(Sigma_ann)
```

Listing 6.4 – Parameter Estimation on the Training Set

Annualization is a common calculation convention for daily returns. It relies on scaling assumptions and does not mean that future annual returns are known. In particular, temporal dependence and regime changes can make this approximation imperfect.

## 6.4 Building a Benchmark Portfolio

Before optimizing the weights, it is useful to construct a simple benchmark portfolio. An equally weighted portfolio assigns the same weight to every asset :

$$w_i = \frac{1}{n}, \qquad i = 1, \ldots, n.$$

This strategy does not require estimating a future return for each asset. It is not necessarily optimal, but it provides a transparent benchmark.

```
n = R_train.shape[1]

# An identical weight is assigned to each of the n assets.
```

```
w_equal = np.repeat(1 / n, n)

# Daily portfolio return over the test period.
# The matrix product calculates the sum w_i * R_{i,t} at each date.
R_equal_test = R_test.to_numpy() @ w_equal

# Volatility is calculated from returns actually observed
# during the test period, not on the estimation sample.
mean_daily = np.mean(R_equal_test)
vol_daily = np.std(R_equal_test, ddof=1)

# Descriptive annualization of the mean and volatility.
mean_annual = 252 * mean_daily
vol_annual = np.sqrt(252) * vol_daily

print(f"Annualized return : {mean_annual:.2%}")
print(f"Annualized volatility : {vol_annual:.2%}")
```

Listing 6.5 – Equally Weighted Portfolio and Out-of-Sample Performance

The parameter `ddof=1` tells NumPy to use the denominator $T-1$ when calculating the empirical standard deviation. The equally weighted portfolio's performance will be compared with that of the optimized portfolio, bearing in mind that results depend on the period studied.

## 6.5 Constrained Optimization

In the chapter devoted to Markowitz, we saw that diversification can be formulated as an optimization problem. Here we consider the minimum-variance portfolio under the usual budget and non-negative-weight constraints :

$$\min_{w} \quad w^{\top}\widehat{\Sigma}w$$

under the constraints

$$\sum_{i=1}^{n} w_i = 1, \qquad w_i \geq 0.$$

The first constraint means that all capital is invested. The second prohibits short selling. This is a modeling choice : other strategies may allow negative weights, but they then require additional controls on leverage and financing.
We also add a concentration constraint :

$$0 \leq w_i \leq 0.60.$$

No asset may therefore account for more than 60 % of the portfolio. This bound is illustrative ; in practice, it should be selected according to the investment guidelines and mandate.

**Solving the problem with `scipy.optimize`** The `minimize` function from `scipy.optimize` solves an optimization problem numerically. We provide the objective function, an initial allocation, the constraints, and the variable bounds. The SLSQP method is suitable for many continuous problems with equality and inequality constraints.

```
from scipy.optimize import minimize

# The objective function returns the annualized portfolio variance.
# For a weight vector w, the formula is w' Sigma_hat w.
def portfolio_variance(w, covariance):
    return float(w.T @ covariance @ w)
```

```

# Initial allocation: equally weighted portfolio.
# It already satisfies the budget constraint and selected bounds.
w0 = np.repeat(1 / n, n)

# Equality constraint: the weights must sum to 1.
constraints = [
    {"type": "eq", "fun": lambda w: np.sum(w) - 1}
]

# Each weight must remain between 0% and 60%.
bounds = [(0.0, 0.60) for _ in range(n)]

# SLSQP searches for weights that minimize estimated variance
# while satisfying the constraints.
result = minimize(
    fun=portfolio_variance,
    x0=w0,
    args=(Sigma_ann.to_numpy(),),
    method="SLSQP",
    bounds=bounds,
    constraints=constraints
)

# Check the solver status before using the weights.
if not result.success:
    raise RuntimeError(f"Optimization Failure : {result.message}")

w_min_var = result.x

print("Optimized weights:")
for ticker, weight in zip(R_train.columns, w_min_var):
    print(f"{ticker:>6s} : {weight:.2%}")

print("Sum of weights:", w_min_var.sum())
print("Annualized variance:", result.fun)
```

Listing 6.6 – Variance Optimization under Constraints

Checking `result.success` is important. A numerical solver may fail to converge or encounter a constraint problem. It is therefore not enough to call an optimization function : its status must be examined and the solution checked against the problem's economic constraints.

**Adding a target-return constraint** It is also possible to minimize variance while requiring the estimated return to be at least a target level $\mu^\star$ :

$$\min_{w} \quad w^\top \widehat{\Sigma} w$$

sous

$$w^\top \widehat{\mu} \geq \mu^\star, \qquad \sum_i w_i = 1, \qquad 0 \leq w_i \leq 0.60.$$

The target return must be chosen carefully. If it is too high, the problem may become infeasible. Even when a solution exists, the estimated return remains uncertain and is not a promise of realized return.

```
# Example of an annualized target: 7%.
# This value should be adapted to the universe and period under study.
target_return = 0.07

```

```
# The constraint function must be non-negative:
# w' mu_ann - target_return >= 0.
constraints_target = [
    {"type": "eq", "fun": lambda w: np.sum(w) - 1},
    {
        "type": "ineq",
        "fun": lambda w: float(w @ mu_ann.to_numpy() - target_return)
    }
]

result_target = minimize(
    fun=portfolio_variance,
    x0=w0,
    args=(Sigma_ann.to_numpy(),),
    method="SLSQP",
    bounds=bounds,
    constraints=constraints_target
)

if result_target.success:
    w_target = result_target.x
    print("Weights with target return:")
    print(pd.Series(w_target, index=R_train.columns))
    print("Estimated annualized return:",
          float(w_target @ mu_ann.to_numpy()))
else:
    print("No solution obtained:", result_target.message)
```

Listing 6.7 – Minimum-Variance Portfolio with a Target Return

This extension is related to the Markowitz efficient frontier. It makes it possible to compare different target-return levels, but it remains sensitive to errors in estimated means. An optimization that produces extreme weights should therefore be interpreted as a possible sign of fragility, not automatically as an economic discovery.

## 6.6 Evaluating the Portfolio Out of Sample

The weights calculated on the training sample are now applied to returns in the test period. The weights remain fixed in this first experiment. This assumption makes interpretation easier and allows strategies to be compared without immediately introducing rebalancing rules.

```
# Optimized portfolio returns on the test dates.
R_opt_test = R_test.to_numpy() @ w_min_var

def performance_summary(portfolio_returns):
    """Calculate a few annualized descriptive statistics."""
    # Remove any missing observations.
    x = pd.Series(portfolio_returns).dropna()

    # Daily mean and volatility, followed by annualization.
    annual_return = x.mean() * 252
    annual_volatility = x.std(ddof=1) * np.sqrt(252)

    # Mean return divided by volatility.
    # This ratio is not a full Sharpe ratio if the risk-free rate
    # has not been subtracted from the return.
    ratio = annual_return / annual_volatility if annual_volatility > 0 else np.
    nan

```

```
    return pd.Series({
        "Annualized return": annual_return,
        "Annualized volatility": annual_volatility,
        "Return/volatility ratio": ratio
    })

comparison = pd.DataFrame({
    "Equally weighted": performance_summary(R_equal_test),
    "Minimum variance": performance_summary(R_opt_test)
})

print(comparison)
```

Listing 6.8 – Compare the equally weighted portfolio with the optimized portfolio.

The ratio calculated here is a simple descriptive measure. To calculate the Sharpe ratio in the strict sense, the risk-free rate must be subtracted, expressed over a period consistent with that of the returns. Caution is also needed when annualizing returns : annualizing the arithmetic mean is not equivalent to the compounded return achieved over a year.

## 6.7 From Static Risk to Conditional Risk

In previous chapters, the matrix $\Sigma$ was generally estimated from a historical sample and used as a fixed matrix. The previous chapter showed that conditional volatility can change over time. It is therefore natural to consider a conditional covariance matrix $\Sigma_t$ :

$$\sigma_{P,t}^2 = w_t^\top \Sigma_t w_t.$$

This expression distinguishes two sources of risk variation :

— changes in the weights $w_t$ when the portfolio is rebalanced ;

— changes in the risk matrix $\Sigma_t$ when volatilities or dependencies between assets change.

As a first approximation, covariance can be estimated over a rolling window. This method does not replace a full multivariate model, but it illustrates that estimated risk depends on the period selected.

```
# Keep the optimized weights fixed and calculate
# the portfolio return at each date.
R_portfolio = pd.Series(
    R.to_numpy() @ w_min_var,
    index=R.index,
    name="R_P"
)

# A 60-trading-day window provides a local volatility estimate.
# min_periods=40 avoids reporting an estimate too early,
# based on very few observations.
rolling_vol = R_portfolio.rolling(
    window=60,
    min_periods=40
).std(ddof=1) * np.sqrt(252)

rolling_vol.plot(figsize=(12, 4),
                 title="Annualized volatility glissante du portefeuille")
plt.ylabel("Annualized volatility")
plt.grid(alpha=0.3)
plt.tight_layout()
plt.show()
```

Listing 6.9 – Rolling Portfolio Volatility

A rolling window gives greater emphasis to recent observations, but the window length is itself a modeling choice. A short window reacts quickly to market changes but produces noisier estimates ; a long window is more stable but may react slowly to regime shifts.

**Using a GARCH forecast in the workflow** The previous chapter introduced the GARCH(1,1) model :

$$R_t = \mu + \varepsilon_t, \qquad \varepsilon_t = \sigma_t z_t,$$

et

$$\sigma_t^2 = \omega + \alpha\varepsilon_{t-1}^2 + \beta\sigma_{t-1}^2.$$

The model forecasts the conditional variance of an asset. For a portfolio of several assets, constructing the full $\Sigma_t$ also requires modeling conditional covariances or correlations between assets. Estimating a separate GARCH model for each asset is therefore not sufficient on its own to obtain every component of $\Sigma_t$.

A first educational application is to fit a GARCH model to the portfolio return series itself. This approach forecasts aggregate volatility without claiming to provide a multivariate conditional covariance matrix.

```
from arch import arch_model

# Fit the GARCH model using the training period only.
# Multiplying by 100 expresses returns in percentage points, which
# often improves the numerical stability of estimation.
R_port_train = pd.Series(
    R_train.to_numpy() @ w_min_var,
    index=R_train.index,
    name="R_P"
) * 100

garch_model = arch_model(
    R_port_train,
    mean="Constant",
    vol="GARCH",
    p=1,
    q=1,
    dist="t"
)

# disp="off" hides solver iterations while retaining the results.
garch_result = garch_model.fit(disp="off")
print(garch_result.summary())

# Forecast variance over the next five trading sessions.
garch_forecast = garch_result.forecast(horizon=5, reindex=False)
variance_forecast_pct = garch_forecast.variance.iloc[-1]

# Variances are expressed in squared percentage points.
# Taking the square root gives volatility in percentage points.
vol_forecast_pct = np.sqrt(variance_forecast_pct)

print("Forecast volatility over the next five trading sessions (\%):")
print(vol_forecast_pct)
```

Listing 6.10 – GARCH Forecast of Portfolio Returns

This forecast should be interpreted as a conditional estimate based on information available at the end of the estimation sample. To evaluate a truly dynamic strategy, estimation would need

to be repeated at every decision date, using only the data available at that time. This procedure is more computationally expensive, but avoids using future information.

### 6.7.1 Measuring Losses with VaR

**Portfolio return and loss** We denote the portfolio return at date $t$ by $R_{P,t}$. The corresponding loss can be defined as $L_t = -R_{P,t}$ when returns are expressed as a proportion of portfolio value. A positive loss then corresponds to a negative return.

At confidence level $1 - \alpha$, Value-at-Risk (VaR) is a quantile of the loss distribution :

$$\mathrm{VaR}_{1-\alpha}(L) = q_{1-\alpha}(L).$$

For example, the 95 % VaR is the 95th percentile of losses. Depending on the sign convention, VaR can also be presented as a negative-return threshold. It is therefore important to state whether the measure applies to loss $L_t$ or return $R_{P,t}$.

**Historical VaR estimation** The historical approach does not assume that returns follow a normal distribution. It directly uses empirical quantiles of losses observed in the estimation sample. This method is simple, but assumes that the selected history is a relevant reference for future risk.

```
# Portfolio returns over the estimation period.
R_P_train = pd.Series(
    R_train.to_numpy() @ w_min_var,
    index=R_train.index
)

# Define loss as the negative of return.
L_train = -R_P_train

# 95th percentile of losses: 95% VaR.
var_95 = L_train.quantile(0.95)

# Apply this threshold to the test period, which was not used
# to estimate the weights or the VaR threshold.
R_P_test_series = pd.Series(
    R_test.to_numpy() @ w_min_var,
    index=R_test.index
)
L_test = -R_P_test_series

# An exceedance occurs when the loss exceeds the estimated VaR.
exceptions = L_test > var_95
exception_rate = exceptions.mean()

print(f"Historical 95% VaR: {var_95:.2%}")
print(f"Test-period exceedance frequency: {exception_rate:.2%}")
print(f"Target theoretical frequency: {0.05:.2%}")
```

Listing 6.11 – Historical VaR and Breach Frequency

Over a sufficiently long period, and if the model is well calibrated, the exceedance frequency of a 95 % VaR should be close to 5 %. In a finite sample, it will not be exactly equal to this value. Too many exceedances may indicate that risk is underestimated ; too few may indicate an excessively conservative threshold.

**CVaR : Accounting for Loss Severity** VaR gives a threshold but does not describe the magnitude of losses beyond it. Conditional Value-at-Risk (CVaR), also called Expected Shortfall in many conventions, focuses on average loss in the tail beyond VaR. For a loss variable $L$, under regularity conditions, it can be written as :

$$\mathrm{CVaR}_{1-\alpha}(L) = E\left[L \mid L \geq \mathrm{VaR}_{1-\alpha}(L)\right].$$

This empirical definition should be interpreted cautiously when observations equal the quantile or when the sample is very short. Nevertheless, it is useful for distinguishing a loss threshold from a measure of the severity of extreme losses.

```python
# Calculate the mean of losses above VaR.
tail_losses = L_train[L_train >= var_95]
cvar_95 = tail_losses.mean()

print(f"Historical 95% VaR: {var_95:.2%}")
print(f"Empirical 95% CVaR: {cvar_95:.2%}")
```

Listing 6.12 – Empirical CVaR Calculation

CVaR is generally greater than or equal to VaR when working with losses and a consistent quantile definition. It therefore complements the information provided by VaR, particularly when the return distribution has heavy tails.

### 6.7.2 Stress Testing and Allocation Robustness

An optimization based on historical covariance describes risk using fluctuations observed in a sample. However, a future crisis may differ from past crises. Stress tests therefore complement statistical measures by specifying explicit scenarios, such as a simultaneous market decline, a sector shock, or an increase in correlations.
Suppose a scenario assigns a hypothetical return $s_i$ to each asset. The portfolio return under that scenario is :

$$R_P^{\text{stress}} = w^\top s.$$

This is not a probabilistic forecast : the scenario is used to measure the portfolio's exposure to a selected adverse situation.

```python
# Hypothetical scenario: each asset experiences a different decline.
# The figures are stress assumptions, not forecasts.
stress_returns = pd.Series(
    {
        "AAPL": -0.20,
        "MSFT": -0.18,
        "JPM":  -0.25,
        "XOM":  -0.15
    }
)

# Reorder the shocks to match the order of the optimized weights.
stress_vector = stress_returns.reindex(R_train.columns)

# If an asset is missing from the scenario, stop the calculation rather
# than silently assigning an incorrect shock.
if stress_vector.isna().any():
    raise ValueError("The scenario does not cover every portfolio asset.")

# The dot product gives the portfolio return under the scenario.
stress_portfolio_return = float(w_min_var @ stress_vector.to_numpy())

```

```python
print(f"Portfolio return under stress: {stress_portfolio_return:.2%}")
```

Listing 6.13 – Scenario of a Simultaneous Decline in Asset Prices

The shocks must match exactly the assets included in the portfolio. A sound stress test examines several scenarios, documents its assumptions, and compares results with historical episodes. The scenario above is deliberately simple : it illustrates the calculation but does not, by itself, constitute a full crisis analysis.

**Sensitivity of weights to estimates :** Optimized weights can vary substantially when the sample or estimated parameters change slightly. This sensitivity is particularly important when mean returns are used as optimization inputs.
A basic check is to compare the weights obtained from two estimation windows. Very different allocations indicate that the solution depends strongly on the selected period.

```python
def minimum_variance_weights(returns_window, upper_bound=0.60):
    """Calculate minimum-variance weights over a given window."""
    # Covariance is estimated only over the supplied window.
    cov = returns_window.cov().to_numpy()
    n_assets = returns_window.shape[1]

    # Equally weighted initial allocation.
    initial_weights = np.repeat(1 / n_assets, n_assets)

    # Weights sum to 1: all capital is invested.
    budget_constraint = {
        "type": "eq",
        "fun": lambda w: np.sum(w) - 1
    }

    # No short selling and concentration is capped.
    weight_bounds = [(0.0, upper_bound)] * n_assets

    # Minimize variance using the same solver as before.
    opt = minimize(
        portfolio_variance,
        x0=initial_weights,
        args=(cov,),
        method="SLSQP",
        bounds=weight_bounds,
        constraints=[budget_constraint]
    )

    if not opt.success:
        raise RuntimeError(opt.message)

    return pd.Series(opt.x, index=returns_window.columns)

# Compare two historical windows of the same type.
midpoint = len(R_train) // 2
weights_first_half = minimum_variance_weights(R_train.iloc[:midpoint])
weights_second_half = minimum_variance_weights(R_train.iloc[midpoint:])

weight_comparison = pd.DataFrame({
    "First window": weights_first_half,
    "Second window": weights_second_half
})

```

```
print(weight_comparison)
```

Listing 6.14 – Compare weights across two estimation windows

This comparison alone does not tell us which allocation is better. Rather, it shows that optimization is conditional on the available estimates. To limit unstable allocations, one can consider more conservative bounds, covariance regularization, sector constraints, or a penalty on changes in weights.

## 6.8 Exercises

**Exercise 1 — Building an Asset Universe**

1. Download adjusted prices for four to six assets using `yfinance`.
2. Calculate simple returns $R_t$ and check for missing values values.
3. Calculate historical means, volatilities, and correlations.
4. Explain why correlation between two assets matters for diversification.

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

# 1. Download adjusted prices for five assets
actifs = ["AAPL", "MSFT", "NVDA", "JNJ", "XOM"]

prix = yf.download(
    actifs,
    start="2018-01-01",
    end="2025-01-01",
    auto_adjust=True,
    progress=False
)["Close"]

# Check for missing data
print("Valeurs manquantes :\n", prix.isna().sum())

# Remove incomplete dates
prix = prix.dropna()

# 2. Calculate simple returns
R = prix.pct_change().dropna()

# Check for missing values
print("Valeurs manquantes dans les rendements :", R.isna().sum().sum())

# 3. Calculate means, volatilities, and correlations
moyennes = R.mean() * 252
volatilites = R.std() * np.sqrt(252)
correlations = R.corr()

resultats = pd.DataFrame({
    "Average annual return (\%)": moyennes * 100,
    "Annual Volatility  (\%)": volatilites * 100
})

print("\nStatistiques annuelles :\n", resultats.round(2))
```

```
print("\nCorrelation matrix :\n", correlations.round(2))

# Plot the correlations
plt.figure(figsize=(7, 5))
plt.imshow(correlations, cmap="coolwarm", vmin=-1, vmax=1)
plt.colorbar(label="Correlation")
plt.xticks(range(len(actifs)), actifs, rotation=45)
plt.yticks(range(len(actifs)), actifs)
plt.title("Correlation matrix")
plt.tight_layout()
plt.show()
```

**Remark 6.8.1** *A low or negative correlation between two assets supports portfolio diversification and can reduce its risk.*

### Exercise 2 — Equally Weighted and Minimum-Variance Portfolios

1. Construct an equally weighted portfolio.
2. Solve the minimum-variance problem subject to the constraints $\sum_i w_i = 1$ et $0 \leq w_i \leq 0.60$.
3. Compare the weights and estimated volatilities.
4. Evaluate both portfolios over a chronological test period.
5. Explain why a lower estimated variance does not guarantee better future performance.

```
import yfinance as yf
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import minimize

# 1. Download prices and calculate returns
actifs = ["AAPL", "MSFT", "NVDA", "JNJ", "XOM"]

prix = yf.download(
    actifs, start="2018-01-01", end="2025-01-01",
    auto_adjust=True, progress=False
)["Close"].dropna()

R = prix.pct_change().dropna()

# Split the data chronologically: 80% training, 20% test
n = int(len(R) * 0.8)
R_train = R.iloc[:n]
R_test = R.iloc[n:]

# 2. Equally weighted portfolio
N = len(actifs)
w_equi = np.ones(N) / N

# 3. Portefeuille de variance minimale
cov = R_train.cov().values

def variance(w):
    return w @ cov @ w

contraintes = {"type": "eq", "fun": lambda w: np.sum(w) - 1}
```

```
bornes = [(0, 0.60)] * N

res = minimize(
    variance, w_equi, method="SLSQP",
    bounds=bornes, constraints=contraintes
)

if not res.success:
    raise RuntimeError(res.message)

w_min = res.x

# Compare weights and estimated volatilities
poids = pd.DataFrame({
    "Equipondere": w_equi,
    "Minimum variance": w_min
}, index=actifs)

vol_equi = np.sqrt(w_equi @ cov @ w_equi) * np.sqrt(252)
vol_min = np.sqrt(w_min @ cov @ w_min) * np.sqrt(252)

print("Poids (\%) :\n", (poids * 100).round(2))
print(f"\nEstimated volatility, equally weighted: {vol_equi:.2\%}")
print(f"Estimated minimum volatility: {vol_min:.2\%}")

# 4. Evaluate the portfolios over the test period
rend_equi = R_test @ w_equi
rend_min = R_test @ w_min

def performance(r):
    volatilite = r.std() * np.sqrt(252)
    rendement = (1 + r).prod() ** (252 / len(r)) - 1
    return pd.Series({
        "Annualized return (\%)": rendement * 100,
        "Annualized volatility (\%)": volatilite * 100
    })

evaluation = pd.DataFrame({
    "Equally weighted": performance(rend_equi),
    "Minimum variance": performance(rend_min)
})

print("\nResults over the test period:\n", evaluation.round(2))

# Compare portfolio value paths
valeurs = pd.DataFrame({
    "Equally weighted": (1 + rend_equi).cumprod(),
    "Minimum variance": (1 + rend_min).cumprod()
})

valeurs.plot(figsize=(9, 5), title="Out-of-sample performance")
plt.ylabel("Value of an initial investment of 1")
plt.show()
```

**Remark 6.8.2** *Minimum-variance optimization uses historical data to reduce estimated risk. This does not guarantee better future performance because returns, volatilities, and correlations can change.*

### Exercise 3 — Target Return and Sensitivity

1. Add a target-return constraint to the optimization.
2. Test several targets and check whether the problem is feasible.
3. Repeat the estimation using two different historical windows.
4. Comment on weight stability and the limitations of estimating mean returns.

```

# Installation: pip install yfinance pandas numpy scipy

import yfinance as yf
import pandas as pd
import numpy as np
from scipy.optimize import minimize

# 1. Download prices and calculate daily returns
actifs = ["AAPL", "MSFT", "NVDA", "JNJ", "XOM"]

prix = yf.download(
    actifs,
    start="2018-01-01",
    end="2025-01-01",
    auto_adjust=True,
    progress=False
)["Close"].dropna()

R = prix.pct_change().dropna()

# 2. Define two historical windows to study
# how sensitive weights are to the period used
fenetres = {
    "2018-2021": R.loc["2018":"2021"],
    "2022-2024": R.loc["2022":"2024"]
}

# Target annual returns to test
cibles = [0.05, 0.10, 0.15]
N = len(actifs)

resultats = []

for nom, donnees in fenetres.items():

    # Estimate mean returns and the covariance matrix
    # on an annualized basis from each window
    mu = donnees.mean().values * 252
    cov = donnees.cov().values * 252

    # Objective function: minimize portfolio variance
    def variance(w):
        return w @ cov @ w

    for cible in cibles:

        # Weights must sum to 1 and the estimated return
        # must be at least equal to the target
        contraintes = [
            {"type": "eq", "fun": lambda w: np.sum(w) - 1},
            {"type": "ineq", "fun": lambda w: w @ mu - cible}
        ]

```

```
        # Each asset receives a weight between 0% and 60%
        res = minimize(
            variance,
            np.ones(N) / N,
            method="SLSQP",
            bounds=[(0, 0.60)] * N,
            constraints=contraintes
        )

        # Check that the solution satisfies the constraints
        faisable = (
            res.success
            and abs(res.x.sum() - 1) < 1e-5
            and res.x @ mu >= cible - 1e-5
        )

        if faisable:
            resultats.append({
                "Window": nom,
                "Cible (%)": cible * 100,
                "Estimated return (%)": res.x @ mu * 100,
                "Volatility (%)": np.sqrt(variance(res.x)) * 100,
                **{actifs[i]: res.x[i] * 100 for i in range(N)}
            })
        else:
            resultats.append({
                "Window": nom,
                "Cible (%)": cible * 100,
                "Estimated return (%)": np.nan,
                "Volatility (%)": np.nan,
                **{a: np.nan for a in actifs}
            })

# 3. Compare the results obtained
table = pd.DataFrame(resultats)
print(table.round(2).to_string(index=False))
```

**Conclusion :** **(i)** The optimization seeks the lowest-risk portfolio whose estimated return reaches the target. If the target is too high given the estimated returns and weight constraints, no feasible solution exists.
**(ii)** Historical sensitivity : weights may change between the 2018–2021 and 2022–2024 windows because returns and correlations change.
**(iii)** Main limitation : mean returns are difficult to forecast. A small estimation error can lead to a large change in optimal weights.

**Exercise 4 — Stress Testing**

1. Define at least three market-decline scenarios.
2. Calculate the portfolio return under each scenario.
3. Identify the assets that contribute most to the loss.
4. Discuss the limitations of a stress test based on a single scenario.

```
import yfinance as yf
import pandas as pd
import numpy as np

# 1. Download prices and calculate returns
```

```python
actifs = ["AAPL", "MSFT", "NVDA", "JNJ", "XOM"]

prix = yf.download(
    actifs,
    start="2018-01-01",
    end="2025-01-01",
    auto_adjust=True,
    progress=False
)["Close"].dropna()

R = prix.pct_change().dropna()

# Construct an equally weighted portfolio
poids = np.ones(len(actifs)) / len(actifs)

# 2. Define three hypothetical decline scenarios
scenarios = pd.DataFrame({
    "Moderate scenario": [-0.10, -0.10, -0.15, -0.05, -0.10],
    "Severe scenario": [-0.25, -0.30, -0.40, -0.15, -0.25],
    "Technology crisis": [-0.40, -0.35, -0.50, -0.10, -0.15]
}, index=actifs)

# 3. Calculate the portfolio return under each scenario
rendements = poids @ scenarios.values

resultats = pd.DataFrame({
    "Portfolio return (\%)": rendements * 100
}, index=scenarios.columns)

print("Stress-test results:")
print(resultats.round(2))

# 4. Identify each asset's contribution to the loss
for scenario in scenarios.columns:
    contributions = poids * scenarios[scenario]

    print(f"\n{scenario} : contributions to loss (\%)")
    print((contributions * 100).round(2).sort_values())

    print("Asset contributing most to the loss:",
          contributions.idxmin())

# Plot the estimated losses
resultats.plot(
    kind="bar",
    legend=False,
    title="Portfolio return by scenario"
)
import matplotlib.pyplot as plt
plt.ylabel("Rendement (\%)")
plt.axhline(0, color="black", linewidth=0.8)
plt.tight_layout()
plt.show()
```

**Remark 6.8.3** **(i)** *Scenarios : the declines are hypothetical and represent different possible crises.*
**(ii)** *Contributions : each contribution is the asset's weight multiplied by its return under the scenario. The more negative the contribution, the more the asset adds to the loss.*
**(iii)** *Limitations : one scenario cannot cover every possible crisis. Several shocks should be tested because correlations and asset behavior can change during a crisis.*