Applied Quantitative Finance in Python: Selected theories and examples

Book · November 2024

DOI: 10.31490/9788024847481

1 author:

Aleš Kresta
VSB - Technical University of Ostrava
36 PUBLICATIONS **527** CITATIONS

VSB TECHNICAL UNIVERSITY OF OSTRAVA | FACULTY OF ECONOMICS

Aleš Kresta

APPLIED QUANTITATIVE FINANCE IN PYTHON: SELECTED THEORIES AND EXAMPLES

vol. 38 | 2024

ISBN 978-80-248-4748-1 (on-line)
DOI 10.31490/9788024847481

Series of Economics Textbooks
Faculty of Economics, VSB – Technical University of Ostrava

Aleš Kresta

APPLIED QUANTITATIVE FINANCE IN PYTHON: SELECTED THEORIES AND EXAMPLES

Ostrava, 2024

Aleš Kresta
Department of Finance
Faculty of Economics
VSB – Technical University of Ostrava
17. listopadu 2172/15
708 00 Ostrava, CZ
ales.kresta@vsb.cz

Reviews
Vittorio Moriggia, University of Bergamo, Italy
Jan Górecki, Silesian University in Opava

Writing of the textbook was supported by the Student Grant Competition of the Faculty of Economics, VSB – Technical University of Ostrava, project number SP2024/047, and the European Union under the REFRESH Research Excellence For Region Sustainability and High Tech Industries project number CZ.10.03.01/00/22_003/0000048 via the Operational Programme Just Transition. All support is greatly appreciated.

During the preparation of this textbook, the author used ChatGPT and Writefull to check and improve the grammar, spelling, and style of the text. After using these tools, the author reviewed and edited the content as needed, taking full responsibility for the final publication.

The text should be cited as follows: Kresta, A. (2024). *Applied Quantitative Finance in Python: Selected Theories and Examples*, SOET, vol. 38. Ostrava: VSB – Technical University of Ostrava.

The text has not been proofread.

Printed at VSB – Technical University of Ostrava
Cover design by Editorial Centre, VSB – Technical University of Ostrava

ISBN 978-80-248-4747-4 (print)
ISBN 978-80-248-4748-1 (on-line)

Preface

Welcome to *Applied Quantitative Finance in Python*, a comprehensive textbook designed to explore the intricate intersection of finance, statistics, and computer science. This field plays a crucial role in understanding and navigating through the complexities of financial markets, where mathematical models and computational tools are indispensable.

The objective of the textbook is to introduce and explain the domain of quantitative finance while keeping a balance between theory and practical application. We have tried to explain complicated models in a simplified manner while providing real-world, applicable code snippets. Python (Python Software Foundation, 2024), a modern and rapidly evolving programming language, is our tool of choice for that. Its popularity and extensive community support offer numerous packages that allow the implementation of complex models by writing just a few lines of code. While Python's versatility is a strength, it occasionally poses challenges related to the dependencies. However, distributions such as *Anaconda* (Anaconda Inc., 2024) can address these issues by seamlessly managing dependencies.

This textbook is not designed as an introductory guide to Python programming. It should be noted that, while it covers advanced topics and in-depth concepts within the realm of Python, it assumes a basic understanding of the language. If you are new to Python and looking for a beginner-friendly introduction, we can recommend three well-written books on this topic (Hilpisch, 2018; Intrigue, 2024; Shaw, 2024).

This textbook does not dive into mathematical derivations or proofs but focuses on demonstrating practical applications, listing assumptions, highlighting potential drawbacks, and pointing out common pitfalls. To enhance applicability, we have included Python code snippets, which are also available from the author's webpage. Recognizing the potential difficulties for beginners, we provide detailed comments and explanations for the code. However, as the complexity increases later in the book, we limit our explanations only to key points.

Our objective is not to provide exhaustive answers to every question; some questions, pointing out advanced ideas, are even posed within the textbook. Instead, we aim to guide the reader, explaining essential principles, and showcasing practical applications. While the code examples in the textbook serve illustrative purposes, the real-world scenarios undoubtedly entail greater complexity. The code snippets in this textbook were tested and executed using Anaconda in version 2022.10 with Python version 3.9.13.

The textbook comprises five chapters, each focusing on a specific aspect of quantitative finance. The first chapter introduces the basics of time series, covering data download, return calculation, data visualization, random variables, and stochastic

processes. Key points include understanding the calculation of returns, generating random numbers, and understanding stochastic processes.

The second chapter delves into risk measurement and management, defining risk measures, exploring calculation methods, and introducing backtesting techniques for the verification of methods.

Chapters 3 to 5 are likely to captivate the reader the most. The third chapter presents portfolio optimization and performance measurement. Starting with the wealth calculation, we elucidate the complexities involved. Subsequently, we explore portfolio performance measurement and dive into the intriguing world of portfolio optimization. We explain the CAPM model and the Fama-French models, providing practical insights into parameter estimation. The chapter concludes with an illustration of how portfolio performance can be verified.

The fourth chapter centers on technical analysis, a methodology for predicting market prices based on historical data. We cover basic tools, with a focus on technical indicators and the construction of an automated trading system. Attention is paid to potential pitfalls, including look-ahead bias, survivorship bias, and over-optimization bias. The final subchapter showcases Python packages for technical analysis.

The fifth chapter focuses on options, a well-known financial derivative. After introducing them, we explore three valuation approaches: the Black-Scholes formula, lattice models, and Monte Carlo simulation.

The journey through the chapters is completely up to the reader. However, we particularly recommend that beginners do not skip the first chapter, where fundamental concepts are introduced.

We hope that you will find value in the content, and whether you are a novice or an experienced practitioner, this book aspires to be a companion in your exploration of applied quantitative finance. Your feedback is valuable to us, and we welcome you to interact with the material and share your thoughts on future improvements. Happy reading!

Aleš Kresta, Ostrava, 2024

Content

Detailed Content

Chapter 1
Prices and Returns: Download, Calculation, and Visualization

In this chapter, we work with prices and returns, which are typical examples of financial time series. Time series can be defined following Hansen (2022, p. 444) as "*a process which is sequentially ordered over time.*" In this textbook, we focus on a discrete time series where t is an integer, though there is also considerable literature on continuous-time processes where t is a real number. Time series data come in two types: univariate and multivariate. Univariate time series tracks the values of a single variable over time, while multivariate time series involve multiple variables. The data are recorded at discrete intervals, such as annually, quarterly, monthly, or daily. The frequency represents how many observations occur in a specific period.

In Python, indexing typically starts from 0, and the last value is excluded; however, in many textbooks, including ours, time series samples are denoted as t=1, ..., n, which might be a bit confusing.[1] When dealing with time series in Python, it is common to use data structures like tuples and lists. However, for more complex, multidimensional data, the *Numpy* array and the *Pandas* dataframe provide greater flexibility.

The chapter is structured as follows. Initially, in Subchapter 1.1, we dive into the process of downloading the price data. Following that, in Subchapter 1.2, we demonstrate how to compute returns from these data. We then explore the visualization capabilities offered by Python in Subchapter 1.3. Moving forward, we introduce random variables in Subchapter 1.4 and dive into stochastic processes in Subchapter 1.5.

1.1 Prices

Generally, the price is the amount of money for which something is sold, demanded, or offered. Based on the conditions, we can distinguish three instances:

- ***historical price***, for which something was traded, such as closing price or adjusted closing price (see Section 1.1.1) or last trade price (see option data example in Section 1.1.2),
- ***ask price*** is a quotation of the best, i.e., the lowest, price for which any counterparty is willing to sell,

[1] Indexing of t=1 ,…, n in theory corresponds to sequence indexes in Python [0, n–1] or sequence generated by *range(0,n)*.

- ***bid price*** is a quotation of the best, i.e., the highest, price for which any counterparty is willing to buy.

Question 1-1 Ask and bid prices

Which is higher: ask or bid price? What is the ask-bid spread?

When working with historical data, we typically deal with observed prices, representing the last known prices of trades for an asset. However, it is crucial to recognize that each observation could have contained more information than just the last-trade price. Consider the scenario of extremely illiquid assets that are rarely traded. Although the most recent trade, for example, was at €100, the actual value of the asset could have changed due to new fundamental information. The lowest price someone is willing to sell (*ask price*) could be €110, and the highest price someone is willing to buy (*bid price*) could be €108. However, in the absence of new trades for this illiquid asset, we continue to record the last observed trade price, which is €100. This situation highlights the nuances of illiquid markets, where the recorded price may not fully capture the current supply and demand dynamics.

In addition, we have the option to analyze historical price series using a dual-quotation approach, where both the *ask* and *bid* prices are recorded together. Then generally the price is calculated as the arithmetic average of the ask and bid prices,

$$midprice = \frac{ask+bid}{2}. \tag{1.1}$$

Question 1-2 Order book

Find the information and explain the term *order book*.

1.1.1 Adjusted closing prices

In stock trading, the closing price signifies the final trading price during the regular trading period, usually a day. Investors commonly use it to assess a stock's performance. However, the closing price may not capture all relevant stock information, especially when corporate actions such as dividends, stock splits, or new offerings occur. In such cases, the adjusted closing price is preferred, particularly for historical return analysis.

The key distinction between the historical price and adjusted price lies in the latter's consideration of factors affecting the stock after the market closes (and before it opens again). Adjustments to the historical prices are based on corporate actions, such as dividends, stock splits, and new offerings. Dividends, whether in cash or additional shares, reduce a stock's value. Adjusted price helps to convey this information. For example, if a stock has a closing price of $100 and a $2 dividend per share is paid, the adjusted price should become $102, accounting for the decreased value due to dividends. In fact, the most recent price is kept and the historical prices are adjusted. Thus, in our example, the adjusted prices after the dividend are equal to the historical prices, and the adjusted prices before the dividend are approximately[2] $2 lower than the historical.

[2] In reality, the adjusted prices are calculated based on dividend multipliers, which capture the calculation correctly. Subtraction of $2 is a simplification for better clarity.

Stock splits involve reducing the market price by increasing the number of shares outstanding.[3] For a 2-for-1 split, each old share is exchanged for two new shares, effectively doubling the shares outstanding and halving the share price. While this nominally halves the share price, it is crucial to understand that this does not reduce the overall value. To provide clarity and avoid any perception of a sudden 50% decrease in the prices, adjustments are made by dividing the historical price by 2 for all days before the split.

In cases where a corporation issues new shares, such as through a rights offering, current shareholders are given the right to purchase new shares at a lower price. This increases the number of outstanding shares, leading to a drop in the historical price. The adjusted prices appropriately reflect these impacts.

Overall, adjusted prices provide investors with more accurate information compared to historical prices. In research, adjusted prices are applied for portfolio optimization, allowing analysis to focus solely on capital returns without the need to handle dividends and splits, and thus simplifying analyses.

Question 1-3 Reverse split

Find the information and explain the term *reverse split*. What and why is it used for?

1.1.2 Yahoo Finance and *Yfinance* package

The package *Yfinance* (Aroussi, 2024) is a useful package that allows downloading and importing financial data from the Yahoo Finance website[4]. It allows us to fetch both the fundamental data and the stock price data as open, high, low, close, adjusted close, and volume in different frequencies. It is also usable to obtain option price data. In the following sections, we present the illustration of scripts downloading stock price data, option chain prices, and fundamental data. For more comprehensive examples, please refer to the documentation[5].

Stock price data

In Code 1-1 we show a simple example of downloading the historical data for four stocks. First, when the package is installed, we can simply import it to the Python main script, see line 2. On line 4, we create a list of tickers for which we want to download the data. To get the data, we can use the *download* function; see lines 6 and 8. The input parameters are the tickers that we want to download, the start date and end date, the frequency of the data, and a few other parameters. Specifically, by *group_by* parameter, we specify how the data should be grouped: whether in ticker-column or column-ticker format (as dataframe multi-index). By setting the *auto_adjust* parameter to True, we obtain the prices automatically adjusted for splits and dividends, see Section 1.1.1. By setting the *auto_adjust* parameter to False, we would obtain the *Close* price and the *Adjusted Close* price separately. The function returns the *Pandas* dataframe with the data. From the dataframe, we can obtain the closing prices, see line 10, and from these prices, we can calculate simple and log returns in lines 13 and 15, further discussed in Subchapter 1.2.

[3] There can be also the reverse split.
[4] htttp://finance.yahoo.com
[5] https://pypi.org/project/yfinance/

Code 1-1 Code downloading the prices of MSFT, AAPL, TSLA, and PG stocks

```
 1. import numpy as np
 2. import yfinance as yf  # we import the package and alias it to yf
 3.
 4. tickers=["MSFT", "AAPL", "TSLA", "PG"]  # we specify the list of tickers we want to get
 5.
 6. data = yf.download(tickers, start="2017-01-01", end="2017-04-30", interval = "1d",
group_by="ticker", auto_adjust = True)
 7.
 8. data = yf.download(tickers, start="2017-01-01", end="2017-04-30", interval = "1d",
group_by="column", auto_adjust = True)
 9.
10. prices=data["Close"]
11.
12. # we can calculate percentage changes (simple/discrete returns)
13. pct_returns = prices.pct_change().dropna()
14. # or we can calculate the log-returns
15. log_returns = np.log(prices/prices.shift(1)).dropna()
16.
17. print(log_returns.describe())
18. log_returns.info()
19. print(log_returns.head())
```

In Code 1-1, we present a straightforward example of downloading historical data for four stocks. However, in real-world applications, the datasets used are typically much more extensive. It involves the inclusion of components of the chosen index, which can be utilized as a benchmark. A significant advantage of Python lies in its simplicity in accomplishing this task. In Code 1-2, we provide an illustrative example of fetching such data from internet websites. We use the *Pandas* function *read_html* to obtain the list of tables on a given URL. From the list, we extract the given table: [0] means the first table on a webpage, [1] means the second table on the webpage.[6] Then, we use the dataframe functions *dropna* to drop empty lines and *tolist* to return the dataframe column as a list.

Code 1-2 Code fetching the symbols in the DJIA and S&P 500 indexes

```
1. import pandas as pd
2.
3. # tickers in DJIA
4. url = 'https://en.wikipedia.org/wiki/Dow_Jones_Industrial_Average'
5. tlDJIA = pd.read_html(url)[1]['Symbol'].dropna().tolist()
6.
7. # tickers in SP500
8. url = 'https://en.wikipedia.org/wiki/List_of_S%26P_500_companies'
9. tlSP500 = pd.read_html(url)[0]['Symbol'].tolist()
```

Option data

It is also possible to obtain the option data (see Chapter 5) via the *Yfinance* package. The example is shown in Code 1-3. First, on line 2 we import the package *Yfinance* and alias it to *yf*; see Appendix A. Then, on line 5, we create the object *ticker*, from which we obtain the list of expiration dates, see line 7. For each expiration date, we obtain the option chain of call options, see lines 17-21.

[6] Remember that indexing in Python starts from 0.

Code 1-3 Code downloading the option chain data for AAPL stock

```
import pandas as pd
import yfinance as yf  # we import the package and alias it to yf
import matplotlib.pyplot as plt

ticker = yf.Ticker("AAPL")

expirations = ticker.options
print(expirations)

c = ticker.option_chain(expirations[5]).calls
print(c.head())

fig = plt.figure(figsize=(10, 6))
plt.plot(c.strike,c.lastPrice, label=expirations[5])

all_calls=pd.DataFrame()
for expiration in expirations[6:-2]:
    c = ticker.option_chain(expiration).calls
    plt.plot(c.strike,c.lastPrice, label=expiration)
    c["expiration"]=expiration
    all_calls = pd.concat([all_calls, c], axis=0)

all_calls['expiration'] = pd.to_datetime(all_calls['expiration'])
all_calls['days_to_expiry'] = (all_calls['expiration'] - pd.Timestamp.now()).dt.days
all_calls['midPrice'] = (all_calls['ask']+all_calls['bid'])/2

# Adding legend, x-axis and y-axis labels, and a figure title
plt.legend(loc='upper right')
plt.xlabel('Strike Price')
plt.ylabel('Last Price')
plt.title('Option Prices for AAPL')
plt.grid(True, which='both', linestyle='--', linewidth=0.5)
plt.tight_layout()
plt.show()
> ('2023-10-27', '2023-11-03', '2023-11-10', '2023-11-17', '2023-11-24', '2023-12-01',
'2023-12-15', '2024-01-19', '2024-02-16', '2024-03-15', '2024-04-19', '2024-06-21', '2024-
09-20', '2024-12-20', '2025-01-17', '2025-06-20', '2025-12-19', '2026-01-16')
        contractSymbol             lastTradeDate  ...  contractSize  currency
0  AAPL231201C00130000 2023-10-16 15:54:47+00:00  ...       REGULAR       USD
1  AAPL231201C00135000 2023-10-23 15:33:27+00:00  ...       REGULAR       USD
2  AAPL231201C00140000 2023-10-23 17:41:58+00:00  ...       REGULAR       USD
3  AAPL231201C00150000 2023-10-23 19:23:14+00:00  ...       REGULAR       USD
4  AAPL231201C00155000 2023-10-20 16:56:20+00:00  ...       REGULAR       USD

[5 rows x 14 columns]
```

In Code 1-3, we demonstrate how to fetch and visualize option chain data for Apple Inc. (ticker AAPL) using the *Yfinance* package. For the explanation of visualization under the *Matplotlib* package, please refer to Subchapter 1.3, especially page 17. The description of the code is as follows:

- **Lines 1-3**: We start by importing the necessary packages. The *Pandas* package is used for data manipulation, *Yfinance* for fetching financial data, and *matplotlib.pyplot* for data visualization.
- **Line 5**: We initialize the *ticker* object for Apple Inc. (AAPL) using the *Yfinance* package.
- **Lines 7-8**: The *options* attribute of the *ticker* object provides us with the expiration dates available for AAPL options. We print these dates in the console.

- **Lines 10-12**: We fetch the call option data for the sixth expiration date in the list using the *option_chain* method. The attribute *calls* of the result gives us the call options data, from which we print the first five rows using the *head* method (line 11).
- **Lines 14-16**: We set up a figure for plotting using *Matplotlib*. The figure size is set to 10x6 inches. We then plot the last traded prices against the strike prices of the options for the sixth expiration date.
- **Lines 18-23**: We loop through the expiration dates from the seventh to the third last, fetching the call options data for each date. For each expiration date, we plot the strike prices against the last traded prices. We also add a new column to the data, indicating the expiration date, and concatenate these data into the *all_calls* dataframe.
- **Lines 25-27**: We convert the expiration column of the *all_calls* dataframe to a *datetime* format. We then calculate the number of days to each option's expiry and store it in a new column called *days_to_expiry*. We also compute the mid-price of each option (average of ask and bid prices) and store it in a new column called *midPrice*.
- **Lines 29-34**: We add finishing touches to our plot. We include a legend to differentiate the data for each expiration date, label the x-axis as “Strike Price” and the y-axis as “Last Price”, and give the plot the title “Option Prices for AAPL”. We also add a grid for better readability and adjust the layout to ensure that all elements fit well. Finally, we display the plot using the *show* method.

The output provides the available expiration dates for AAPL options and a visual representation of how the option prices vary with different strike prices for selected expiration dates, see Figure 1-5 on page 17.

Fundamental data

In the realm of finance, fundamental data refers to essential financial information about a company, such as its earnings, balance sheets, and cash flow statements. These data are crucial for analysts and investors to evaluate a company's financial health and to make informed investment decisions. Moreover, we can fetch information about the holders, news, and others from Yahoo Finance via *Yfinace* package, see Code 1-4.

Code 1-4 Fetching fundamental data for the INTC stock

```
import yfinance as yf

ticker = yf.Ticker('INTC')

info = ticker.info # get all the information
CAPMbeta = info['beta'] # get the beta of CAPM model, see Section 3.4.1

# basic information
holders = ticker.major_holders # info about holders
holder_inst = ticker.institutional_holders # institutional holders and their share
holder_mf =ticker.mutualfund_holders # mutual funds holders and their share

# dividends and splits
actions = ticker.actions # get the dividends and splits
dividennds = ticker.dividends # get only the dividends
splits =  ticker.splits # get only the splits

#financials
stat_inc = ticker.income_stmt # Income statement for last 4 years
```

```
stat_inc_q = ticker.quarterly_income_stmt # Income statement for last 4 quarters
stat_bal = ticker.balance_sheet # Balance sheet for last 4 years
stat_bal_q = ticker.quarterly_balance_sheet # Balance sheet for last 4 quarters
stat_cf = ticker.cashflow # Cash-flow statement for last 4 years
stat_cf_q = ticker.quarterly_cashflow # Cash-flow statement for last 4 quarters


news = ticker.news # get related news
```

1.2 Returns

When discussing an asset's returns, we can categorize them into two types: simple returns (or discrete returns) and log returns (or continuously compounded returns). The distinction between these two types extends beyond their calculation formulas; it also affects how they are aggregated over time and how portfolio returns are computed from them. Although logarithmic returns are easier to aggregate over time, calculating portfolio returns is more straightforward with simple returns.

1.2.1 Simple returns

In the discrete case, the return R_t is computed as a relative change in the asset's price P and also considering the dividend yield DIV_t/P_{t-1},

$$R_t = \frac{P_t - P_{t-1} + DIV_t}{P_{t-1}} = \frac{P_t + DIV_t}{P_{t-1}} - 1, \tag{1.2}$$

we thus assume that the price changes over time in fixed discrete steps Δ. It is sometimes useful to work with gross returns, i.e. we add one to the returns,

$$_gR_t = R_t + 1 = \frac{P_t + DIV_t}{P_{t-1}}. \tag{1.3}$$

If we do not want to work with the dividends or keep in mind splits and other corporate actions, we can utilize the adjusted prices instead of using the historical prices, see Section 1.1.1. In such a case there is no *DIV* in the equations.[7] Also, in the equations above, we assume that there is no split of the stock, or that the stock prices are split-adjusted.

There are a few disadvantages of the simple/discrete returns. First, it is not straightforward to aggregate the returns over time, see Section 1.2.3. Second, when we want to calculate the mean return, the geometric mean should be used instead of the arithmetic; otherwise, one can obtain nonsense information, see the example in Table 1-2 on page 10 with a continuation in Table 1-3.

There is also another drawback in the simple returns related to the calculation of the mean. When our investment or speculation undergoes a loss, we naturally think that the same profit will cover the loss, which is, however, true only for small changes. For instance, let us consider that in one year we experience a loss of 66% of the investment. One would think that the next year's profit of 66% will make the speculation break even. However, that is not true, as can be simply demonstrated when we calculate the return over two years,

[7] In this textbook we further assume that we work with the adjusted prices.

$$R_{2years} = (1 - 0.66) \cdot (1 + 0.66) - 1 = -45.22\%, \quad (1.4)$$

which in this case is a loss of around 45% (over these two years). The profit needed to cover the 66% loss is 194% as can be seen in Table 1-1. For other possible losses see also Table 1-1. If you think that the loss of 66% cannot (or should not) happen, at least not for the blue-chip stocks, the opposite is true. Many examples of bluechip stocks decreasing by more than 50% of their value in one year can be shown. For example, consider the Meta company (ticker META) in the period from November 1, 2021, to November 1, 2022, losing more than 66%, see Figure 1-1.

Table 1-1 Profit required to cover the loss

Loss	Required profit
-10%	11%
-20%	25%
-25%	33%
-33%	49%
-50%	100%
-66%	194%
-80%	400%
-90%	900%

Figure 1-1 The prices of META (from November 1, 2021, to November 1, 2022)

Source: https://finance.yahoo.com/quote/META?p=META

1.2.2 Logarithmic returns

In contrast to discrete returns, with logarithmic returns (log returns or continuously compounded returns), it is assumed that the time over which the price of the asset changes is infinitely small. Then, continuously compounded returns r_t can be calculated as follows,

$$r_t = ln\frac{P_t+DIV}{P_{t-1}} = ln(P_t + DIV) - ln\,P_{t-1}. \tag{1.5}$$

Again, we can utilize adjusted prices, which both take care of splits and cancel out *DIV*. Obviously, continuously compounded returns can be calculated from discrete returns and vice versa as follows,

$$r_t = ln(R_t + 1)\ , \tag{1.6}$$

$$R_t = e^{r_t} - 1. \tag{1.7}$$

1.2.3 Aggregation of the returns in time

To aggregate the returns over time, log returns are more intuitive. If you want to compute the total return over n periods, you simply sum up the log returns for those periods:

$$r_{t+1,t+n} = ln(P_{t+n}) - ln(P_t) = \sum_{i=t+1}^{t+n} r_i. \tag{1.8}$$

For the average return in a given period, we can use the arithmetic mean:

$$\bar{r} = \frac{\sum_{i=t+1}^{t+n} r_i}{n} = \frac{r_{t,t+n}}{n}. \tag{1.9}$$

For example, with monthly returns of 1%, the annual return is as follows,

$$r_{1,12} = \sum_{i=1}^{12} 1\% = 12 \cdot 1\% = 12\%, \tag{1.10}$$

and the average daily return is:

$$\bar{r} = \frac{r_{t+1,t+n}}{n} = \frac{r_{1,21}}{21} = \frac{1\%}{21} = 0{,}0476\%. \tag{1.11}$$

As can be seen, we calculate 12 months in the year and 21 days in a month. We assume 21 days in a month because we work with the business days, which on average there are 21 in a month. Typically, there are 252 business days in a year ($12 \cdot 21$), but 250 is often used as a rounded number. Carver (2015) suggests that assuming 256 days in a year simplifies calculations involving the square root of time, such as adjusting the Sharpe ratio calculated from daily data, see Section 3.2.3.[8]

On the other hand, working with simple returns is not as straightforward as one might assume. To compute the total return over n periods, although the straightforward idea would be to do the sum of them, the product of gross simple returns must be calculated and one must be subtracted,

$$R_{t+1,t+n} = \frac{P_{t+n}}{P_t} - 1 = \prod_{i=t+1}^{t+n}(1 + R_i) - 1. \tag{1.12}$$

Also, the calculation of mean return should not be the arithmetic but geometric mean, when working with the gross returns,

$$\bar{R} = \sqrt[n]{\prod_{i=t+1}^{t+n}(1 + R_i)} - 1 = \sqrt[n]{1 + R_{t+1,t+n}} - 1, \tag{1.13}$$

which is usually referred to as the *compound average growth rate* (CAGR) to distinguish it from the arithmetical average, which is sometimes used as a simplification of the

[8] Then, Sharpe ratio calculated from daily data can be simply multiplied by 16 as beeing a square root of 256.

calculation. For instance, if we observe monthly returns of 1%, the annual return is as follows:

$$R_{1,12} = \prod_{i=1}^{12}(1 + 1\%) - 1 = 12{,}6825\%, \tag{1.14}$$

and the average daily return,

$$\bar{R} = \sqrt[21]{1 + 1\%} - 1 = 0{,}0474\%. \tag{1.15}$$

These equations are of particular significance. In practical scenarios, it is tempting to think that the difference between arithmetic and geometric calculations is minor. One might think that using the arithmetic average would not introduce significant errors. To highlight the nuances between these calculations, let us consider an example. Look at four mutual funds and their respective monthly returns, as detailed in Table 1-2.

Table 1-2 presents three funds, each with an arithmetic average of monthly simple returns of 1%. However, they vary in terms of return volatility. *Fund A* has the lowest volatility, *Fund C* the highest, and *Fund B* falls between. At first glance, it might seem that all three funds are similar since they have the same average return. There is also *Fund D* to consider. Its performance, as gauged by the arithmetic average, exceeds that of the first three funds. This might lead one to conclude that, despite its higher volatility, *Fund D* offers better returns. Another perspective, explored in Chapter 3, suggests that *Fund D*'s superior performance is a result of its increased riskiness.

However, we would be surprised when we add the prices of the funds, see Table 1-3. Let us start with a hypothetical scenario where each fund begins with a price of €100. As we investigate the data, an interesting pattern emerges. We can see that, although outperforming by 1.5 percentage points in the arithmetic average return, *fund D* ends up at the same starting price of €100, as indicated in Table 1-3, and is outperformed by *funds A and B*. Also, *funds A, B, and C* end up with different values. We can also check the geometric averages, which, as we see in Table 1-3, correspond to the ranking of the final values of the funds.

Table 1-2 Wrong logic of comparing the performance based on arithmetic average

Month	Simple returns			
	Fund A	Fund B	Fund C	Fund D
1	3,0%	7,0%	22%	25%
2	-1,0%	-5,0%	-20%	-20%
3	3,0%	7,0%	22%	25%
4	-1,0%	-5,0%	-20%	-20%
5	3,0%	7,0%	22%	25%
6	-1,0%	-5,0%	-20%	-20%
7	3,0%	7,0%	22%	25%
8	-1,0%	-5,0%	-20%	-20%
9	3,0%	7,0%	22%	25%
10	-1,0%	-5,0%	-20%	-20%
11	3,0%	7,0%	22%	25%
12	-1,0%	-5,0%	-20%	-20%
Arithmetic average	1,0%	1,0%	1,0%	2,5%

Table 1-3 Extended analysis highlighting common misconception

Month	Simple returns				Prices			
	Fund A	Fund B	Fund C	Fund D	Fund A	Fund B	Fund C	Fund D
0					100,00	100,00	100,00	100,00
1	3,0%	7,0%	22%	25%	103,00	107,00	122,00	125,00
2	-1,0%	-5,0%	-20%	-20%	101,97	101,65	97,60	100,00
3	3,0%	7,0%	22%	25%	105,03	108,77	119,07	125,00
4	-1,0%	-5,0%	-20%	-20%	103,98	103,33	95,26	100,00
5	3,0%	7,0%	22%	25%	107,10	110,56	116,21	125,00
6	-1,0%	-5,0%	-20%	-20%	106,03	105,03	92,97	100,00
7	3,0%	7,0%	22%	25%	109,21	112,38	113,43	125,00
8	-1,0%	-5,0%	-20%	-20%	108,12	106,77	90,74	100,00
9	3,0%	7,0%	22%	25%	111,36	114,24	110,70	125,00
10	-1,0%	-5,0%	-20%	-20%	110,25	108,53	88,56	100,00
11	3,0%	7,0%	22%	25%	113,55	116,12	108,05	125,00
12	-1,0%	-5,0%	-20%	-20%	**112,42**	**110,32**	**86,44**	**100,00**
Arithmetic average	1,0%	1,0%	1,0%	2,5%				
Geometric average	0,98%	0,82%	-1,21%	0,00%				

Question 1-4 The reason for performance differences

Why do funds A, B, and C have different performances despite the same arithmetic average?
Why is it crucial to understand the method used to calculate the average return?

1.2.4 Aggregation of the returns in a portfolio

Now, we consider the return on the portfolio. The value (the price) of the portfolio $P_{P,t}$ is given as the sum of the products of particular assets' prices ($P_{i,t}$) and the corresponding quantities (v_i) of these assets in the portfolio, see Chapter 3,

$$P_{P,t} = \sum_{i=1}^{N} P_{i,t} \cdot v_i. \tag{1.16}$$

The discrete return of the portfolio can be computed as follows,

$$R_{P,t} = \frac{P_{P,t} - P_{P,t-1}}{P_{P,t-1}} = \frac{\sum_{i=1}^{N} P_{i,t-1}(1+R_{i,t})v_i - \sum_{i=1}^{N} P_{i,t-1}v_i}{\sum_{i=1}^{N} P_{i,t-1}v_i}, \tag{1.17}$$

as well as the continuously compounded return,

$$r_{P,t} = ln\ \ P_{P,t} - ln\ \ P_{P,t-1} = ln\ \ \left(\sum_{i=1}^{N} P_{i,t-1}\, e^{r_{i,t}} v_i\right) - ln\ \ \left(\sum_{i=1}^{N} P_{i,t-1}\, v_i\right). \tag{1.18}$$

To simplify these equations, we introduce the weight w_i of the i-th asset in the portfolio. The weight is equivalent to the proportion of the amount invested in i-th asset to the total amount invested,

$$w_i = \frac{P_{i,t-1}v_i}{\sum_{i=1}^{N} P_{i,t-1}v_i}. \tag{1.19}$$

After modification of the equations, we obtain the following formulas for simple and log returns of portfolio,

$$R_{P,t} = \sum_{i=1}^{N} w_i R_{i,t}, \tag{1.20}$$

$$r_{P,t} = ln \sum_{i=1}^{N} (w_i e^{r_{i,t}}) \ . \tag{1.21}$$

So, as we can see, the portfolio return is more straightforward to calculate with the discrete returns.

Question 1-5 The portfolio

What do we mean by a *portfolio* in finance?

1.2.5 Returns calculation in Python

A simple example of returns calculation is shown in lines 13 and 15 of Code 1-1. To calculate the simple returns, we can use the *Pandas* dataframe method *pct_change()*. For the calculation of log returns, the *Numpy log* function can be applied. It is important to note that in both cases the first return is *Not a Number* (*NaN*) since the calculation of this return requires a day-before price, which we do not have. Thus, we apply the *Pandas* dataframe method *dropna()* to drop all rows with any *NaN* value.

Question 1-6 Quantity of returns

How many returns can we derive from *n* prices?

Later in this textbook, we also work with the cumulative (gross) return, see Code 3-3 in Subchapter 3.1. For that, we use the *Pandas* dataframe method *cumsum()* and *cumprod()*. The calculation follows the above-mentioned equations.

1.3 Data visualization

In Python, there are several packages and tools available for data visualization. Here are some of the most popular ones:

- ***Bokeh*** (Bokeh Team, 2024) is designed to easily create interactive plots. Bokeh outputs plots in HTML format, making them easy to be embed into web applications. It is particularly suited for web-based dashboards and applications.
- ***Matplotlib*** (Hunter & Droettboom, 2023) is one of the most widely used and versatile plotting packages in Python. It provides a wide range of plotting options and is highly customizable. You can create line plots, scatter plots, bar plots, histograms, and more.
- ***Pandas*** (The Pandas Development Team, 2024), primarily used for data manipulation, also has built-in plotting capabilities (based on *Matplotlib*). It is particularly handy for quick and basic plotting directly from dataframes.
- ***Plotly*** (Chris P, 2023) is an interactive graphing package. *Plotly*'s Python graphing package makes interactive, publication-quality graphs online. It supports a variety of charts and plots, including 3D plots and geographic maps.
- ***Seaborn*** (Waskom, 2024) is built on top of *Matplotlib* and provides a higher-level interface for creating visually appealing and informative statistical graphics. It comes with several built-in themes and color palettes to make attractive plots.

In the following, we demonstrate the usage of these packages in the example of three simple graphs: scatter plot, area chart, and line chart, but first, we download and calculate some data. Similarly to Subchapter 3.1, we use the FAANG portfolio as a reference, which

comprises Meta Platforms (formerly Facebook), Apple, Amazon, Netflix, and Google. For an illustrative example, let us consider that we bought 1, 2, 3, 4, and 5 stocks of them, respectively, and we held this portfolio from January 1, 2013, to January 1, 2023, see Code 1-5.

Code 1-5 The initial calculations

```
import yfinance as yf
import pandas as pd

import matplotlib.pyplot as plt
import seaborn as sns
import plotly.express as px

# Download data
tickers = ["META",  # Facebook/Meta Platforms, Inc.
           "AAPL",  # Apple
           "AMZN",  # Amazon
           "NFLX",  # Netflix
           "GOOGL"  # Google (Alphabet Inc.)
          ]

data = yf.download(tickers=tickers, start='2013-01-01', end='2023-01-01', interval='1d', group_by='column', auto_adjust=True)
prices = data["Close"].dropna()
simple_returns = prices.pct_change().dropna()
v = pd.DataFrame(index=prices.columns, columns=['quantity'], data=[1, 2, 3, 4, 5])
P_times_v = prices.multiply(v['quantity'], axis=1)
wealth = P_times_v.sum(1)
```

In Code 1-6, we illustrate the creation of scatter plots using various packages. Specifically, we plot the daily simple returns of Meta Platforms (previously known as Facebook) and Apple. As observed, the figures generated by the *Matplotlib*, *Seaborn*, and *Pandas* plotting functions bear a strong similarity to Figure 1-2, which was produced using the *Matplotlib* package. In contrast, the *Plotly* package produces an interactive figure, offering a dynamic user experience that is especially suited for environments like Jupyter Notebook or Jupyter Lab. To enhance the clarity and presentation of these plots, several helpful functions are employed:

- The *xlabel()* function is used to set the label for the x-axis, providing context about the data being plotted along this axis.
- The *ylabel()* function sets the label for the y-axis, offering insights into the data being plotted vertically.
- The *title()* function assigns a title to the plot, giving viewers an immediate understanding of the plot's purpose or the significance of the data.
- By invoking *grid()*, a grid is overlaid on the plot. This grid can aid in reading and interpreting the plot, especially when trying to gauge the values of specific data points.
- The function *show()*, particularly relevant for *Matplotlib*, ensures that the plot is displayed to the user.

Collectively, these functions enhance the visual appeal, clarity, and comprehensibility of the plots, making them more informative and user-friendly.

Similarly, in Code 1-7 we generate line plots of the portfolio value calculated already in Code 1-5. Again, the plots generated by the packages are very similar to the plot produced by *Matplotlib* shown in Figure 1-3, and the *Plotly* package produces an interactive figure.

Code 1-6 The scatter plots utilizing different packages

```
# Scatter plot using Matplotlib
plt.figure(figsize=(10,6))
plt.scatter(simple_returns['META'], simple_returns['AAPL'])
plt.title('Scatter plot of Returns: Meta vs. Apple (Matplotlib)')
plt.xlabel('Meta Returns')
plt.ylabel('Apple Returns')
plt.grid(True)
plt.show()

# Scatter plot using Seaborn
sns.scatterplot(x=simple_returns['META'], y=simple_returns['AAPL'])
plt.title('Scatter plot of Returns: Meta vs. Apple (Seaborn)')
plt.xlabel('Meta Returns')
plt.ylabel('Apple Returns')
plt.show()

# Scatter plot using Pandas
simple_returns.plot.scatter(x='META', y='AAPL', figsize=(10,6), grid=True)
plt.title('Scatter plot of Returns: Meta vs. Apple (Pandas)')
plt.xlabel('Meta Returns')
plt.ylabel('Apple Returns')
plt.show()

# Scatter plot using Plotly
fig = px.scatter(simple_returns, x='META', y='AAPL', title='Scatter plot of Returns:
Meta vs. Apple (Plotly)')
fig.show()
```

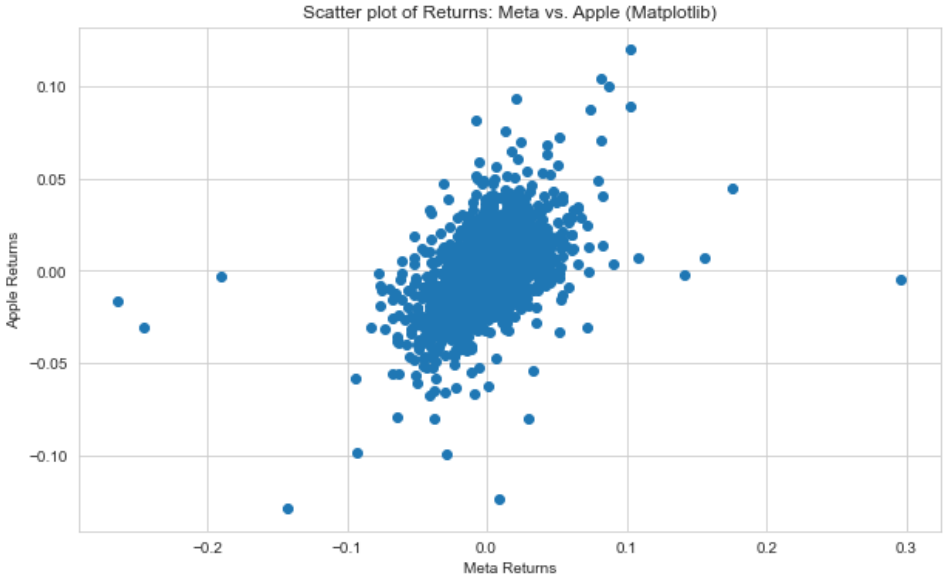

Figure 1-2 The scatter plot produced by *Matplotlib*

Code 1-7 The line chart of portfolio value

```
# Line plot using Matplotlib
plt.figure(figsize=(10,6))
plt.plot(wealth, label='Wealth', color='blue')
plt.title('Wealth Over Time (Matplotlib)')
plt.xlabel('Date')
plt.ylabel('Wealth')
plt.legend()
plt.grid(True)
plt.show()

# Line plot using Seaborn
plt.figure(figsize=(10,6))
sns.lineplot(x=wealth.index, y=wealth.values, label='Wealth')
plt.title('Wealth Over Time (Seaborn)')
plt.xlabel('Date')
plt.ylabel('Wealth')
plt.legend()
plt.grid(True)
plt.show()

# Line plot using Pandas
wealth.plot(figsize=(10,6), label='Wealth', title='Wealth Over Time (Pandas)',
grid=True)
plt.xlabel('Date')
plt.ylabel('Wealth')
plt.legend()
plt.show()

# Line plot using Plotly
fig = px.line(wealth, x=wealth.index, y=wealth.values, title='Wealth Over Time
(Plotly)', labels={'y':'Wealth'})
fig.show()
```

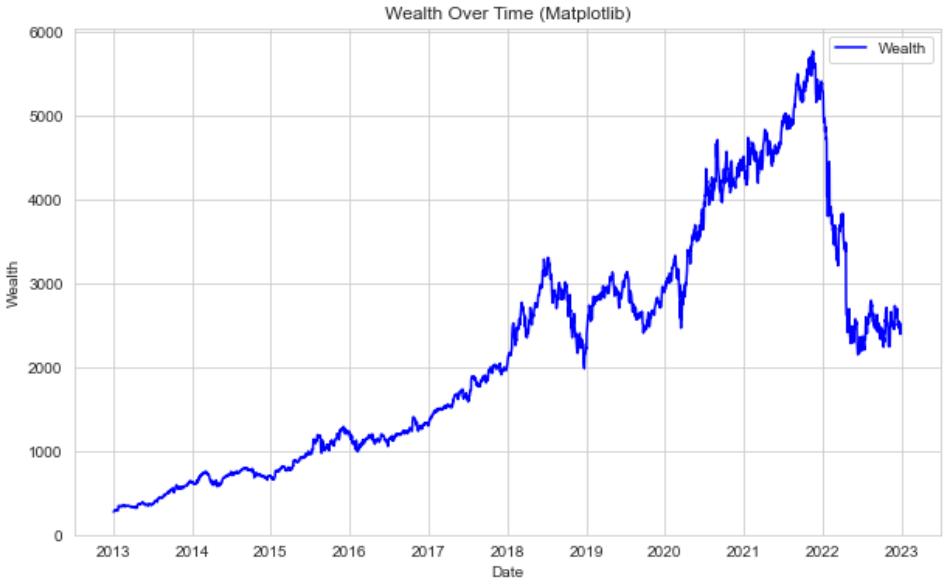

Figure 1-3 The line chart produced by *Matplotlib*

In Code 1-8, we illustrate the progression of portfolio value over time, with distinct colors highlighting individual stocks. Figure 1-4 generated using the *Matplotlib* package, provides a visual representation of this. Unlike Figure 1-3, which only displays the total value of the portfolio, Figure 1-4 offers a detailed breakdown. It showcases the value of holdings in various stocks: 1 unit of Meta Platforms, 2 units of Apple, 3 units of Amazon, 4 units of Netflix, and 5 units of Google. Another application of the stacked area plot can be seen in Subchapter 3.1 in Figure 3-1 and Figure 3-3.

Code 1-8 The stacked area chart of portfolio value

```
# Stacked area plot using Matplotlib
plt.figure(figsize=(10,6))
plt.stackplot(P_times_v.index, [P_times_v[col] for col in P_times_v.columns],
labels=P_times_v.columns)
plt.title('Stock Value Over Time (Matplotlib)')
plt.xlabel('Date')
plt.ylabel('Value')
plt.legend(loc='upper left')
plt.grid(True)
plt.show()

# Stacked area plot using Pandas
P_times_v.plot.area(figsize=(10,6), title='Stock Value Over Time (Pandas)')
plt.xlabel('Date')
plt.ylabel('Value')
plt.legend(loc='upper left')
plt.grid(True)
plt.show()

# Stacked area plot using Plotly Express
import plotly.express as px
fig = px.area(P_times_v, x=P_times_v.index, y=P_times_v.columns,
              title='Stock Value Over Time (Plotly Express)')
fig.show()
```

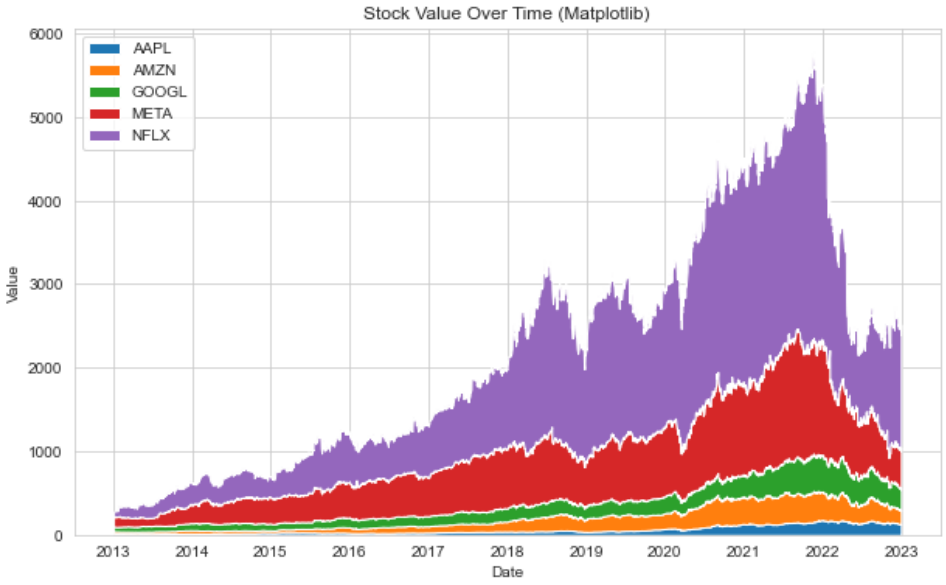

Figure 1-4 The stacked area chart produced by *Matplotlib*

The graphs generated by different packages appear quite similar at first glance. However, it is essential to recognize that our focus was on basic visualizations, which both *Matplotlib* and *Pandas* can handle easily. For more intricate visualizations, *Seaborn* is a recommended choice due to its advanced plotting capabilities. You can refer to its official website[9] for more details. An example of advanced plotting capabilities is the function *regplot* applied in Code 3-18.

1.3.1 plt.show()

When creating a plot using *Matplotlib*, the plot is built step by step in the background. However, it does not automatically display the plot. The *plt.show()* function is used to display the figure to the user. When you call this function, it opens a window that visually presents the plot or figure you have created.

In the context of Jupyter Notebooks or Jupyter Lab, *plt.show()* is not always necessary because plots can be displayed automatically in the output cells. However, in traditional Python scripts, you should use *plt.show()* to explicitly display the plot.

It is also used to draw multiple graphs in one figure as was used in line 19 in Code 1-3. As can be seen, in the cycle on lines 17-21, we always draw a new curve in the same figure. On lines 28-33, we even set some options for the figure. The command *plt.show()* on line 34 finishes the figure and shows it. If we input another *plt.plot()* command afterward, Python will create a new figure. The resulting figure can be seen in Figure 1-5.

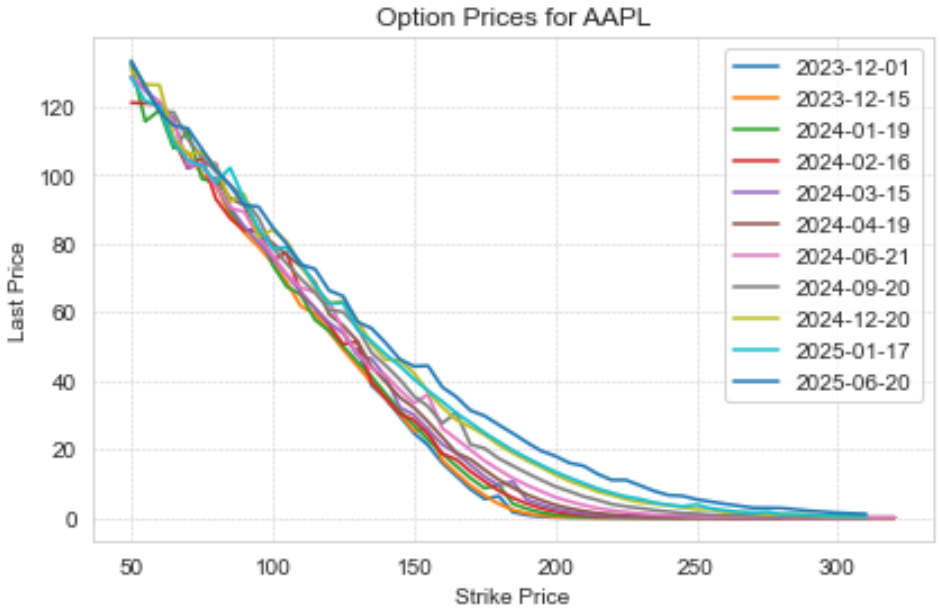

Figure 1-5 Last-trade prices of AAPL options (the result of Code 1-3)

[9] https://seaborn.pydata.org/

Question 1-7 Smooth lines

Why do the curves in Figure 1-5 appear jagged instead of smooth?

1.3.2 Multiple plots and shared axes

In Python it is possible to have multiple graphs in one figure. The useful functions are *plt.subplot()* or *plt.subplots()* and *plt.tight_layout()*.

- The *plt.subplot()* function in *Matplotlib* is used to create a subplot within a larger figure, allowing for the arrangement of multiple plots in a grid format within a single figure. This is particularly useful when we want to display multiple related visualizations side by side for comparison or to provide a comprehensive view of the data.
- The *plt.subplots()* function in *Matplotlib* is a versatile tool that allows to creation of multiple subplots within a single figure, returning both the figure object and an array of axes objects. This function provides a more streamlined way to create and manage subplots compared to *plt.subplot()*. The example of *plt.subplots()* usage is shown in Code 1-9.
- The *plt.tight_layout()* function in *Matplotlib* is used to automatically adjust the size and positions of axes on the figure canvas so that they fit within the figure area without overlapping. This function is particularly useful when we have multiple subplots or when elements of the plot, such as axis labels or titles, are being clipped or overlapping. By calling *plt.tight_layout()*, you ensure that the layout is optimized to make the best use of available space on the figure, leading to a cleaner and more readable presentation.

There may be situations where aligning the x axes of two graphs is beneficial. An example of this, where we plot the prices and returns of META with synchronized x-axes, is presented in Code 1-9. In the code, we intentionally skip the first 1,001 prices, see lines 6 and 26. The resulting figure is shown in Figure 1-6.

Question 1-8 How would the figure look like?

What would happen if in Code 1-9 we omitted the parts "*, sharex=ax1*" and "*,sharex=True*"? Show the figures.

Code 1-9 The example of synchronized x-axes

```
1. # Creating a figure using plt.subplot with the shared x-axis
2. plt.figure(figsize=(10, 6))
3.
4. # Creating the first subplot for prices
5. ax1 = plt.subplot(2, 1, 1)  # 2 rows, 1 column, first plot
6. ax1.plot(prices['META'].iloc[1001:], label='META Prices', color='blue')
7. ax1.set_title('META Stock Prices')
8. ax1.set_ylabel('Price')
9.
10. # Creating the second subplot for simple returns
11. ax2 = plt.subplot(2, 1, 2, sharex=ax1)  # 2 rows, 1 column, second plot
12. ax2.plot(simple_returns['META'], label='META Simple Returns', color='red')
13. ax2.set_title('META Stock Simple Returns')
14. ax2.set_xlabel('Date')
15. ax2.set_ylabel('Simple Return')
16. plt.tight_layout()
17. plt.show()
18.
```

```
19. # Creating a figure with two subplots with aligned x-axes
20. fig, (ax1, ax2) = plt.subplots(2, 1, sharex=True, figsize=(10, 6))
21.
22. # Plotting prices on the first subplot
23. ax1.plot(prices['META'].iloc[1001:], label='META Prices', color='blue')
24. ax1.set_title('META Stock Prices')
25. ax1.set_ylabel('Price')
26.
27. # Plotting simple returns on the second subplot
28. ax2.plot(simple_returns['META'], label='META Simple Returns', color='red')
29. ax2.set_title('META Stock Simple Returns')
30. ax2.set_xlabel('Date')
31. ax2.set_ylabel('Simple Return')
32. plt.tight_layout()
33. plt.show()
```

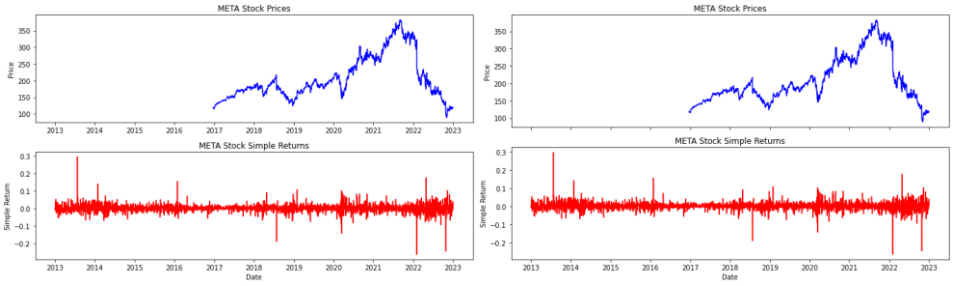

Figure 1-6 The example of synchronized x-axes

Question 1-9 How many prices are we skipping?

In Code 1-9 on lines 6 and 26 we plot the prices from 1,001st. So, it should be said that we skipped 1,000 prices, not 1,001 prices, right? What is correct and why? Discuss it!

1.3.3 Histograms

For the example of histogram plotting, see Code 1-13 and Figure 1-9 in Subchapter 1.4.

1.3.4 3D plots

Matplotlib also offers the possibility to create 3D plots. In Code 1-10 the illustrative example of a 3D scatter plot is presented. In fact, in the code, we plot the *lastPrice* option data obtained in Code 1-3. The resulting figure is shown in Figure 1-7.

Code 1-10 3D scatter plot with *Matplotlib* package (continuation of Code 1-3)

```
 1. import matplotlib.pyplot as plt # 3D plot via matplotlib
 2. fig = plt.figure(figsize=(10, 6))
 3. ax = fig.add_subplot(111, projection='3d')
 4. ax.scatter(all_calls['days_to_expiry'], all_calls['strike'], all_calls['lastPrice'])
 5. ax.set_xlabel('Days to Expiry')
 6. ax.set_ylabel('Strike Price')
 7. ax.set_zlabel('Option Price')
 8. ax.set_title('Matplotlib: Option Prices vs. Maturity and Strike Price')
 9. plt.show()
10.
11. import plotly.express as px # 3D plot via plotly
12. fig = px.scatter_3d(all_calls, x='days_to_expiry', y='strike', z='lastPrice',
13.                     title='Plotly: Option Prices vs. Maturity and Strike Price')
14. fig.show()
```

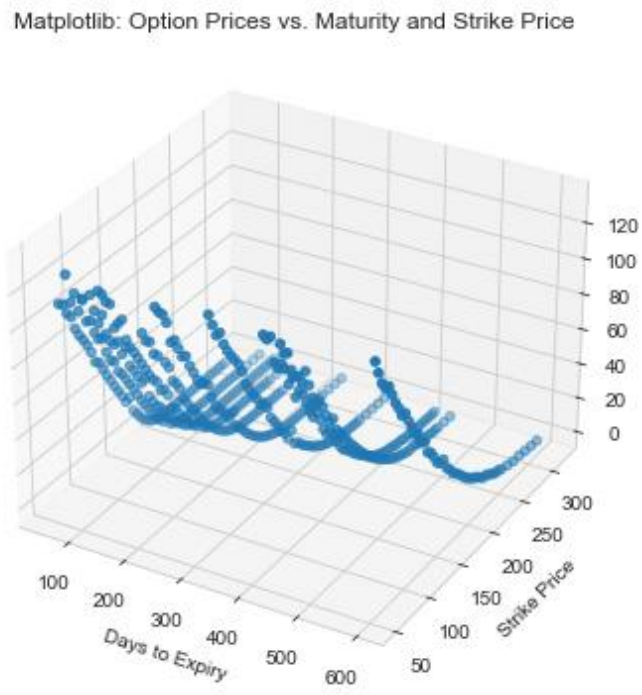

Figure 1-7 Last-trade prices of AAPL options (the result of Code 1-10)

Creating a 3D surface plot is a bit more involving than a 3D scatter plot, especially when using packages like *Matplotlib* and *Plotly*. Typically, in 3D visualizations, a surface plot is used as an equivalent to an area plot in 2D. In Code 1-11 we show how a 3D surface (area) plot can be created for the option prices in dependence on maturity and strike price. The resulting figure is shown in Figure 1-8. Code 1-12 does the same in the *Plotly* package.

Code 1-11 3D surface plot with *Matplotlib* package (continuation of Code 1-3)

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Pivot the data to get a grid suitable for a surface plot
pivot_table=all_calls.pivot(index='days_to_expiry',columns='strike',values='lastPrice')

X = pivot_table.columns
Y = pivot_table.index
X, Y = np.meshgrid(X, Y)
Z = pivot_table.values

fig = plt.figure(figsize=(10, 6))
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(X, Y, Z, cmap='viridis')

ax.set_xlabel('Strike Price')
ax.set_ylabel('Days to Expiry')
ax.set_zlabel('Option Price')
ax.set_title('Matplotlib: Option Prices Surface Plot')
plt.show()
```

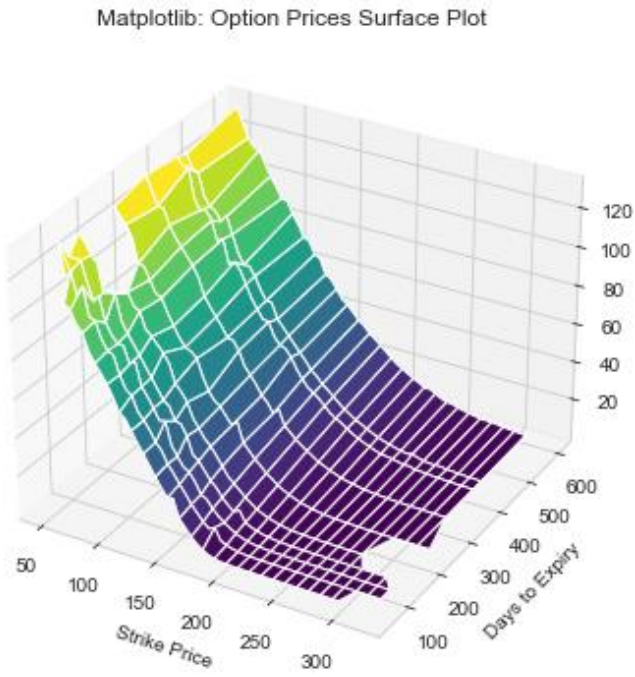

Figure 1-8 Last-trade prices of AAPL options (the result of Code 1-11)

Code 1-12 3D surface plot with *Plotly* package (continuation of Code 1-3)

```
 1. import plotly.graph_objects as go
 2.
 3. # Pivot the data to get a grid suitable for a surface plot
 4. pivot_table = all_calls.pivot(index='days_to_expiry', columns='strike',
values='lastPrice')
 5.
 6. fig = go.Figure(data=[go.Surface(z=pivot_table.values, x=pivot_table.columns,
y=pivot_table.index)])
 7. fig.update_layout(title='Plotly: Option Prices Surface Plot', scene=dict(
 8.                   xaxis_title='Strike Price',
 9.                   yaxis_title='Days to Expiry',
10.                   zaxis_title='Option Price'))
11. fig.show()
```

1.4 Random variables

In the complex world of finance, the concept of randomness is of special significance. Financial markets, which are intrinsically volatile and unpredictable, often behave in ways that seem random. From stock prices to interest rates, the unpredictable nature of financial data has led to the development of numerous models and simulations that rely heavily on random numbers.

Thus, random numbers play a pivotal role in various financial applications. For instance, in the realm of options pricing, the Monte Carlo method uses random number generation to simulate thousands of potential price paths, helping determine the option's

value. Similarly, in risk management, random numbers are used to simulate various economic scenarios to assess potential losses.

It must be noted that the computers work in a deterministic way and there is nothing like randomness or random numbers. What we call random numbers are, in fact, pseudorandom numbers, which are simply sequences of numbers generated by a deterministic process that appear to be random. The advantage of pseudo-random numbers is the repeatability of the generation, which can be guaranteed by setting the so-called seed.

The Python, *Random* module provides basic random number generation, suitable for many financial modeling tasks. However, for more sophisticated financial applications, Python's *Numpy* package provides an enhanced *random* sub-package. An example of the random number generation is shown in Code 1-13. In the code, we also plot the histogram of *rn6* random numbers, see Figure 1-9.

Code 1-13 Random number generation under different probability distributions

```
import numpy.random as rn
import matplotlib.pyplot as plt

rn.seed(100)  # fixes the seed for the reproducibility

rnu = rn.rand(100000,1) # generate the list with 100,000x1 pseudorandom numbers from uniform (0,1) distribution
rnn = rn.randn(100000,1) # generate the list with 100,000x1 pseudorandom numbers from standard normal distribution
rn1 = rn.beta(2, 5, (100000, 1)) # generate 100,000 pseudorandom numbers from beta(2,5) distribution
rn2 = rn.binomial(20, 0.5, (100000, 1)) # generate 100,000 pseudorandom numbers from binomial(20,0.5) distribution
rn3 = rn.chisquare(4, (100000, 1)) # generate 100,000 pseudorandom numbers from chisquare(4) distribution
rn4 = rn.exponential(1, (100000, 1)) # generate 100,000 pseudorandom numbers from exponential(1) distribution
rn5 = rn.lognormal(0,0.5, (100000, 1)) # generate 100,000 pseudorandom numbers from lognormal(0,0.5) distribution
rn6 = rn.normal(0,0.5, (100000, 1)) # generate 100,000 pseudorandom numbers from lognormal(0,0.5) distribution
rn7 = rn.uniform(0,1, (100000, 1)) # generate 100,000 pseudorandom numbers from uniform(0,1) distribution

plt.figure(figsize=(10,6))
plt.hist(x=rn6, bins=100, density=True)
plt.xlabel('Value')
plt.ylabel('Probability')
plt.title('Histogram of pseudorandom numbers')
plt.grid(False)
plt.show()
```

Question 1-10 Generate all the histograms in one figure.

Using the *subplot()* or *subplots()* function generate one figure with the histograms of rn1,…, rn6.

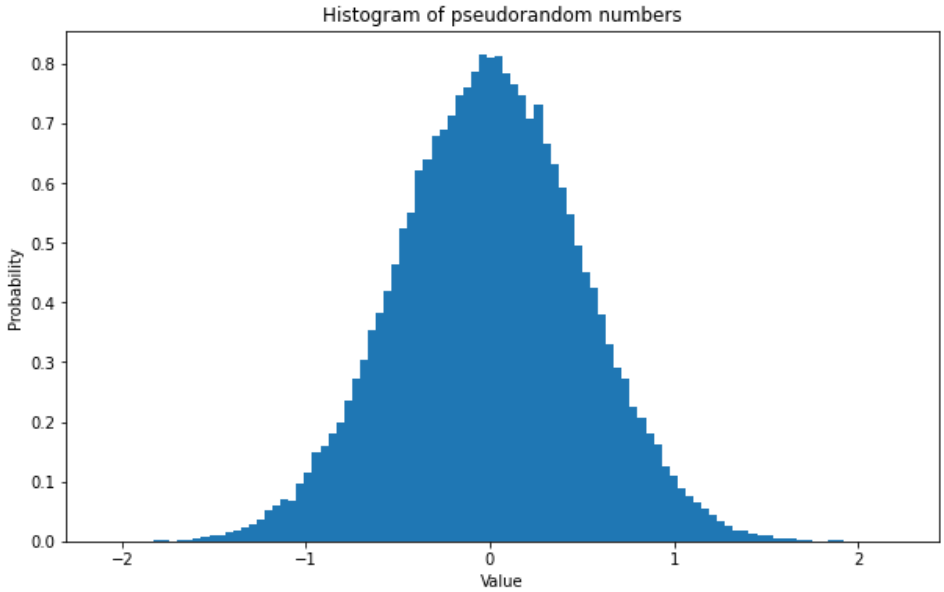

Figure 1-9 Histogram of 100,000 randomly generated N(0,1) numbers

1.4.1 Variance reduction techniques

In this section, we discuss the variance reduction techniques. We focus on simple techniques of *antithetic variates* and *moment matching* discussed by Hilpisch (2018, pp. 372–373).

As we generate random numbers, we expect them to follow the assumed distribution. According to the *Law of Large Numbers*, the more random numbers we generate, the closer the resulting statistics should be to the expected values, see Code 1-14.

Question 1-11 Law of large numbers

Search online and explain the law or large numbers.

Code 1-14 Means and standard deviations of randomly generated numbers

```
 1. import numpy as np
 2. import numpy.random as rn
 3.
 4. rn.seed(100)
 5. for i in range(1,8):
 6.     random = rn.randn(10**i)
 7.     print(f"Random numbers:{10**i:9.0f}  Mean: {np.mean(random):.12f}. Standard
deviation: {np.std(random):.12f}.")
 8.
 9. > Random numbers:       10  Mean: 0.020569627676. Standard deviation: 0.842042313297.
10. Random numbers:      100  Mean: -0.173826616443. Standard deviation: 0.994982483532.
11. Random numbers:     1000  Mean: -0.003117544049. Standard deviation: 1.038720025481.
12. Random numbers:    10000  Mean: 0.000741772285. Standard deviation: 1.002159818386.
13. Random numbers:   100000  Mean: 0.002248399090. Standard deviation: 0.995990272623.
14. Random numbers:  1000000  Mean: -0.000090243735. Standard deviation: 0.998491032509.
15. Random numbers: 10000000  Mean: -0.000278756647. Standard deviation: 0.999713136376.
```

As can be seen, even for 10,000,000 random trials, the mean and standard deviation are not equal to 0 and 1 respectively. We describe two methods to improve the alignment. The first approach involves the use of *antithetic variates*. In this method, only half of the required random samples are drawn, and then an equal set of random numbers with inverted signs is appended, see Code 1-15. As can be seen, the mean is equal to 0 even for a small number of trials, but the standard deviation is not affected.

Code 1-15 Means and standard deviations using antithetic variates

```
rn.seed(100)
for i in range(1,8):
    random = rn.randn(round(10**i/2))
    random=np.concatenate((random, -random))
    print(f"Random numbers:{10**i:9.0f}  Mean: {np.mean(random):.12f}. Standard
deviation: {np.std(random):.12f}.")
> Random numbers:       10  Mean: 0.000000000000. Standard deviation: 1.052170620285.
Random numbers:      100  Mean: -0.000000000000. Standard deviation: 0.918011934621.
Random numbers:     1000  Mean: 0.000000000000. Standard deviation: 1.081068321997.
Random numbers:    10000  Mean: 0.000000000000. Standard deviation: 1.010370099564.
Random numbers:   100000  Mean: 0.000000000000. Standard deviation: 0.995601603746.
Random numbers:  1000000  Mean: -0.000000000000. Standard deviation: 0.997794718807.
Random numbers: 10000000  Mean: -0.000000000000. Standard deviation: 0.999679256402.
```

In the second approach, *moment matching*, we subtract the mean and then divide the random numbers by their standard deviation, see Code 1-16. Under this approach, the first two moments, i.e. the mean and standard deviation, are perfectly aligned with the required values.

Code 1-16 Means and standard deviations using moment matching

```
rn.seed(100)
for i in range(1,8):
    random = rn.randn(10**i)
    random = (random - np.mean(random)) / np.std(random)
    print(f"Random numbers:{10**i:9.0f}  Mean: {np.mean(random):.12f}. Standard deviation:
{np.std(random):.12f}.")
> Random numbers:       10  Mean: -0.000000000000. Standard deviation: 1.000000000000.
Random numbers:      100  Mean: 0.000000000000. Standard deviation: 1.000000000000.
Random numbers:     1000  Mean: -0.000000000000. Standard deviation: 1.000000000000.
Random numbers:    10000  Mean: 0.000000000000. Standard deviation: 1.000000000000.
Random numbers:   100000  Mean: 0.000000000000. Standard deviation: 1.000000000000.
Random numbers:  1000000  Mean: 0.000000000000. Standard deviation: 1.000000000000.
Random numbers: 10000000  Mean: -0.000000000000. Standard deviation: 1.000000000000.
```

1.5 Stochastic processes

Stochastic (or random) processes are sequences of random variables. Most processes applied in finance are of Markov type, i.e. each random value generation depends solely on the preceding one, without being influenced by any previous values. The stochastic processes are usually described by stochastic differential equations.

1.5.1 Simulation of stock prices via GBM

The typical and most utilized stochastic process in finance is Geometric Brownian motion (henceforth GBM). It can be described by the following equation,

$$S_{t+\Delta t} = S_t \cdot e^{\left(r-\frac{1}{2}\sigma^2\right)\cdot\Delta t+\sigma\cdot\sqrt{\Delta t}\cdot z_{t+\Delta t}}. \tag{1.22}$$

In this equation, S_t represents the stock price at time *t*, *r* is the expected annual return, and σ is the standard deviation of the annual return. The variable *z* is a random number drawn from the standard normal distribution. An example of stock price simulation using GBM with a *NumPy* array is provided in Code 1-17. The breakdown with a step-by-step overview of the code is as follows:

- **Lines 1-2**: The necessary packages, *Numpy* for numerical operations and *matplotlib.pyplot* for plotting, are imported.
- **Lines 5-11**: Parameters are defined for the stock price simulation. These include the drift (*mu*), volatility (*sigma*), initial stock price (*S0*), time step Δt (*dt*), total time (T), and the number of simulation trials (*num_trials*).
- **Line 13**: The total number of time steps is calculated on the basis of the total time and the time step.
- **Line 16**: A fixed seed for the random number generator is set to ensure the reproducibility of the simulation results.
- **Lines 19-20**: A matrix is initialized to store the simulated stock prices for each trial. The first column of this matrix is set to the initial stock price.
- **Lines 23-26**: Using a cycle, the stock prices are simulated. For each time step, the loop calculates the drift and diffusion terms and then updates the stock prices based on the geometric Brownian motion formula.
- **Lines 30-38**: For visualization purposes, the simulated stock prices for the first 10 trials are plotted on the left side of the figure.
- **Lines 39-44**: On the right side of the figure, a histogram of the last prices (at time *T*) for all trials is plotted and displayed horizontally.
- **Lines 45-46**: Titles, labels, and grid lines are added to the plots, and the layout is adjusted for better visualization.
- **Line 47**: The entire figure with both subplots is displayed (see Figure 1-10).
- **Lines 50-51**: The mean and standard deviation of the stock prices at the end of the simulation are calculated and printed.

The application usage of Geometric Brownian motion in option pricing can be seen in Code 5-6.

Code 1-17 Simulation of GBM using *Numpy* array

```
 1. import numpy as np
 2. import matplotlib.pyplot as plt
 3.
 4. # Parameters
 5. mu = 0.1  # Drift
 6. sigma = 0.2  # Volatility
 7. S0 = 100  # Initial stock price
 8. dt = 0.01  # Time step
 9. T = 1  # Total time
10. num_trials = 10000  # Number of simulation trials
11.
```

```
# Number of time steps
num_steps = int(T / dt)

# fix the seed
np.random.seed(100)

# Initialize an array to store stock prices for each trial
stock_prices_matrix = np.zeros((num_trials, num_steps + 1))
stock_prices_matrix[:, 0] = S0

# Simulate stock prices for multiple trials using geometric Brownian motion
for i in range(1, num_steps + 1):
    drift = (mu - (sigma**2)/2) * dt
    diffusion = sigma * np.sqrt(dt) * np.random.normal(0, 1, num_trials)
    stock_prices_matrix[:, i] = stock_prices_matrix[:, i - 1] * np.exp(drift +
diffusion)

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

# Plot the simulated stock prices for the first 5 trials
ax1 = plt.subplot(1,2,1)
for i in range(10):  # Only plot the first 5 trials
    plt.plot(np.arange(0, T + dt, dt), stock_prices_matrix[i, :], label=f'Trial {i +
1}')
plt.title('Stock Price Simulation with Numpy (First 5 Trials)')
plt.xlabel('Time')
plt.ylabel('Stock Price')
plt.legend()

# plot the histogram of the last prices (at time T) with horizontal orientation
plt.subplot(1,2,2, sharey=ax1)
plt.hist(stock_prices_matrix[:, -1], bins=100, color='blue', alpha=0.7,
edgecolor='black', orientation='horizontal')
plt.title('Histogram of Last Prices from Stock Price Simulation')
plt.ylabel('Stock Price')
plt.xlabel('Frequency')
plt.grid(True, which='both', linestyle='--', linewidth=0.5)
plt.tight_layout()
plt.show()

# Print mean and standard deviation of the prices at the end
print(f"Mean of the prices at the end: {np.mean(stock_prices_matrix[:, -1])}")
print(f"Standard deviation of the prices at the end: {np.std(stock_prices_matrix[:, -
1])}")
> Mean of the prices at the end: 110.51017671262413
Standard deviation of the prices at the end: 22.038613650062274
```

Question 1-12 The mean and standard deviation

Why are the mean and standard deviation not equal to 10% and 20% respectively? Calculate the mean and standard deviation of logarithmic returns for the whole simulated period.

Question 1-13 The type of distribution of the one-year-ahead prices

What type of probability distribution can be seen in Figure 1-10?

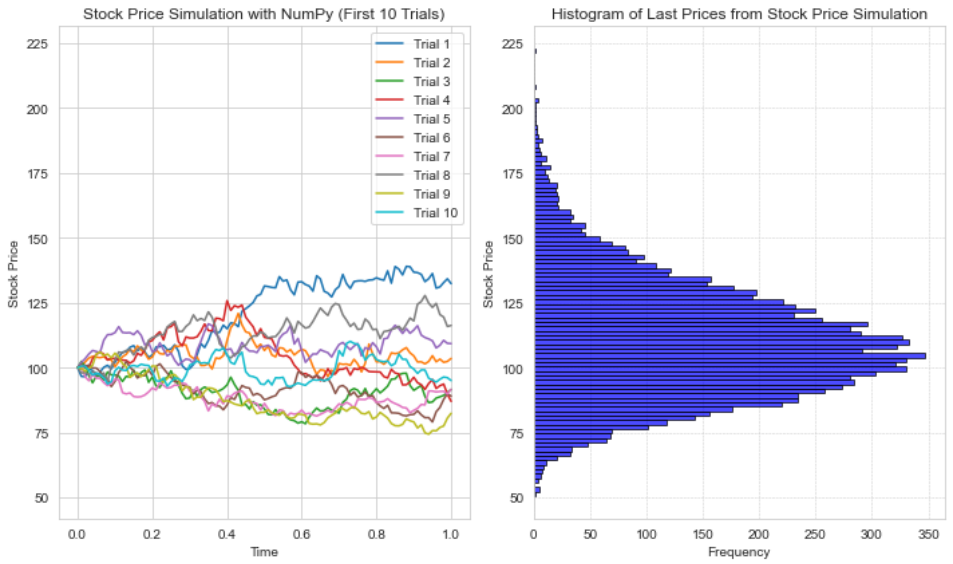

Figure 1-10 Figure generated by Code 1-17

In Code 1-18, an example of stock price simulation via GBM is given using *Pandas* dataframe. In this version, the *stock_prices_matrix* array from Code 1-15 has been replaced with the *stock_prices_df* dataframe. Other lines have been adjusted to work with the dataframe structure. The resulting figure is completely the same as in Figure 1-10.

Question 1-14 The reproducibility

Why do we obtain the same results in Code 1-18 as in Code 1-17?

Code 1-18 Simulation of GBM using *Pandas* dataframe

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

# Parameters
mu = 0.1  # Drift
sigma = 0.2  # Volatility
S0 = 100  # Initial stock price
dt = 0.01  # Time step
T = 1  # Total time
num_trials = 10000  # Number of simulation trials

# Number of time steps
num_steps = int(T / dt)

# fix the seed
np.random.seed(100)

# Initialize a DataFrame to store stock prices for each trial
stock_prices_df = pd.DataFrame(index=range(num_trials), columns=range(num_steps + 1))
stock_prices_df.iloc[:, 0] = S0

# Simulate stock prices for multiple trials using geometric Brownian motion
for i in range(1, num_steps + 1):
    drift = (mu - (sigma**2)/2) * dt
```

```
    diffusion = sigma * np.sqrt(dt) * np.random.normal(0, 1, num_trials)
    stock_prices_df.iloc[:, i] = stock_prices_df.iloc[:, i - 1] * np.exp(drift +
diffusion)

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

# Plot the simulated stock prices for the first 10 trials
ax1 = plt.subplot(1,2,1)
for i in range(10):
    plt.plot(stock_prices_df.columns * dt, stock_prices_df.iloc[i, :], label=f'Trial {i
+ 1}')
plt.title('Stock Price Simulation with Pandas (First 10 Trials)')
plt.xlabel('Time')
plt.ylabel('Stock Price')
plt.legend()

# plot the histogram of the last prices (at time T) with horizontal orientation
plt.subplot(1,2,2, sharey=ax1)
plt.hist(stock_prices_df.iloc[:, -1], bins=100, color='blue', alpha=0.7,
edgecolor='black', orientation='horizontal')
plt.title('Histogram of Last Prices from Stock Price Simulation')
plt.ylabel('Stock Price')
plt.xlabel('Frequency')
plt.grid(True, which='both', linestyle='--', linewidth=0.5)
plt.tight_layout()
plt.show()

# Print mean and standard deviation of the prices at the end
print(f"Mean of the prices at the end: {stock_prices_df.iloc[:, -1].mean()}")
print(f"Standard deviation of the prices at the end: {stock_prices_df.iloc[:, -
1].std()}")
> Mean of the prices at the end: 110.51017671262389
Standard deviation of the prices at the end: 22.039715663396468
```

1.5.2 Simulation of stock returns via ARIMA-GARCH processes

Volatility models have become an important tool in time series analyses, particularly in financial applications. Engle (1982) observed that although the future value of many financial time series is unpredictable, there is a clustering in volatility. He proposed the autoregressive conditional heteroskedasticity (ARCH) process, which was later expanded to the generalized autoregressive conditional heteroskedasticity (GARCH) model by Bollerslev (1986). Later, some more advanced models based on the GARCH model were introduced, such as GJR-GARCH, IGARCH, FIGARCH, GARCH-M, EGARCH, etc. For their description, see Francq and Zakoian (2011).

For time series modeling, the conditional mean can be assumed. The time series of the returns $\{r_{i,t}\}_{t=1}^{T}$ can be modeled using the autoregressive process (AR),

$$r_t = \mu_0 + \sum_{i=1}^{O} \mu_i \cdot r_{t-i} + \sigma_t \cdot \tilde{\varepsilon}_t, \tag{1.23}$$

$$\tilde{\varepsilon}_t \sim N(0,1) \text{ or } \tilde{\varepsilon}_t \sim t_\nu(0,1), \tag{1.24}$$

where μ_0 and μ_i are parameters of the autoregressive model, σ_t is the standard deviation (volatility) modeled by the GARCH model and $\tilde{\varepsilon}_t$ is a random number from the chosen probability distribution, further we assume Gaussian or Student probability distributions. The volatility, σ_t, can then be modeled by the GARCH model which takes the following form,

$$\sigma_t^2 = \omega + \sum_{i=1}^{P} \alpha_i \cdot \sigma_{t-i}^2 + \sum_{j=1}^{Q} \beta_j \cdot \varepsilon_{t-j}^2, \tag{1.25}$$

where ω, α_i and β_j are parameters that need to be estimated. Positive variance is assured if $\omega > 0$, $\alpha_i \geq 0 \ \forall i$ and $\beta_j \geq 0 \ \forall j$. The model is stationary if $\sum_{i=1}^{P} \alpha_i + \sum_{j=1}^{Q} \beta_j < 1$.

However, in order to model the returns soundly, it is important to set the order of AR-GARCH models, i.e. variables *O*, *P*, *Q*. These orders, or lags, are generally set on the basis of chosen criteria. The most straightforward criterion would be the value of likelihood or log-likelihood function. However, it is possible to increase the value of the likelihood function by adding parameters to the model. By doing so, we may end up overfitting the data – we start to model the residuals[10] instead of the underlying relationship. Thus, it is necessary to introduce a penalty term for the number of parameters in the model.

Below we present the Bayesian information criterion (BIC) and Akaike information criterion (AIC), two well-known criteria, that take the number of parameters into account. They can be computed as follows,

$$AIC = 2k - 2\,log(L), \tag{1.26}$$

$$BIC = k\,log(n) - 2\,log(L), \tag{1.27}$$

where *k* is the number of parameters in the model, *n* is the number of observations from which the parameters are estimated, and *L* is the likelihood value of the estimated model.

When deciding which model[11] to apply, the one with the lowest value of AIC (BIC) should be preferred. The difference between the criteria is in the penalty term for the quantity of parameters. The penalty of BIC is a function of the sample size, and so it is typically more severe than that of AIC.

The handy package for ARCH/GARCH modeling is *Arch* (Sheppard, 2024). Below we show only a simple application; for more complex examples, refer to the package webpage[12]. The application of estimating AR(1)-GARCH(1,1) model and simulating the next 1,000 returns is presented in Code 1-19. The code essentially fits an AR(1)-GARCH(1,1) model to the log returns of INTC stock, simulates future returns based on the fitted model, and then plots both the observed and simulated returns and volatilities in a single figure with two subplots. It can be described as follows (different model specifications are depicted in Code 1-20):

- **Lines 1-5**: Necessary packages are imported. This includes *Yfinance* for downloading stock data, *Numpy* and *Pandas* for data manipulation, *Matplotlib* for plotting, and *Arch* for AR-GARCH modeling.
- **Lines 8-11**: The INTC stock data are downloaded from Yahoo Finance for the specified date range. Adjusted closing prices are extracted, and log returns are calculated. The factor of 100 is multiplied to express returns in percentage terms for better *Arch* estimation purposes.
- **Lines 13-16**: An AR(1)-GARCH(1,1) model is specified for logarithmic returns. The model is then fitted to the data and a summary of the model fit is printed.

[10] Note that residuals represent random error, which is assumed to be independent.
[11] The lags in AR-GARCH model.
[12] https://bashtage.github.io/arch/index.html

- **Lines 19-22**: A new AR(1)-GARCH(1,1) model is specified without any data. This model is then used to simulate 1,000 daily returns based on the parameters estimated from the previous model.
- **Lines 24-25**: The index of the simulated data is set to 1,000 business days following the last date of the observed log returns.
- **Line 27**: Two subplots are created: one for returns and the other for volatilities.
- **Lines 28-32**: The observed logarithmic returns are plotted in blue and the simulated returns are plotted in red on the first subplot.
- **Lines 34-38**: The observed volatilities (conditional volatilities from the model) are plotted in blue, and the simulated volatilities are plotted in red on the second subplot.
- **Line 40**: Adjust the layout of the plots to ensure they do not overlap.
- **Line 41**: Displays the plots.

Code 1-19 Estimation and simulation of INTC returns using *Arch* package

```
 1. import yfinance as yf
 2. import numpy as np
 3. import pandas as pd
 4. import matplotlib.pyplot as plt
 5. from arch import arch_model
 6.
 7.
 8. # Download INTC stock data and calculate log returns
 9. data = yf.download('INTC', start='2010-01-01', end='2022-01-01', auto_adjust=True)
10. close_prices = data['Close']
11. log_returns = 100 * np.log(close_prices / close_prices.shift(1)).dropna()
12.
13. # Estimate AR(1)-GARCH(1,1) model
14. model = arch_model(log_returns, mean='AR', lags=1, vol='Garch', p=1, q=1)
15. results = model.fit()
16. print(results.summary()) # Print the summary of the model
17.
18.
19. # Simulate 1000 daily returns according to fitted parameters
20. sim_mod = arch_model(None, mean='AR', lags=1, vol='Garch', p=1, q=1)
21. sim_data = sim_mod.simulate(results.params, 1000)
22. sim_data.head()
23.
24. # Extend the index of datafame by adding another 1000 business days
25. sim_data.index = pd.bdate_range(log_returns.index[-1], periods=1001)[1:]
26.
27. fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(10, 6), sharex=True)
28. # Plot returns
29. ax1.plot(log_returns.index, log_returns, color='blue', label='Observed Returns')
30. ax1.plot(sim_data.index, sim_data['data'], color='red', label='Simulated Returns')
31. ax1.set_title('Returns (Observed and Simulated)')
32. ax1.legend()
33.
34. # Plot volatilities
35. ax2.plot(log_returns.index, results.conditional_volatility, color='blue',
label='Observed Volatility')
36. ax2.plot(sim_data.index, sim_data['volatility'], color='red', label='Simulated
Volatility')
37. ax2.set_title('Volatility (Observed and Simulated)')
38. ax2.legend()
39.
40. plt.tight_layout()
41. plt.show()
42. > [**********100%%**********]  1 of 1 completed
43. Iteration:      1,   Func. Count:      7,   Neg. LLF: 62865122909610.6
44. Iteration:      2,   Func. Count:     17,   Neg. LLF: 29396534822.501152
```

```
Iteration:      3,   Func. Count:     26,   Neg. LLF: 28677965204.258327
Iteration:      4,   Func. Count:     35,   Neg. LLF: 595024064.9076681
Iteration:      5,   Func. Count:     42,   Neg. LLF: 5881.7475074015365
Iteration:      6,   Func. Count:     49,   Neg. LLF: 241488089.770838
Iteration:      7,   Func. Count:     56,   Neg. LLF: 5862.569421587768
Iteration:      8,   Func. Count:     63,   Neg. LLF: 5843.480148391649
Iteration:      9,   Func. Count:     70,   Neg. LLF: 5843.151997779187
Iteration:     10,   Func. Count:     77,   Neg. LLF: 5838.895462096332
Iteration:     11,   Func. Count:     84,   Neg. LLF: 5840.9782672218425
Iteration:     12,   Func. Count:     91,   Neg. LLF: 5841.6369807898445
Iteration:     13,   Func. Count:     98,   Neg. LLF: 5843.692680831037
Iteration:     14,   Func. Count:    105,   Neg. LLF: 5846.038586643464
Iteration:     15,   Func. Count:    112,   Neg. LLF: 5846.398604045644
Iteration:     16,   Func. Count:    119,   Neg. LLF: 6083.48311652338
Iteration:     17,   Func. Count:    126,   Neg. LLF: 6084.0657179763875
Iteration:     18,   Func. Count:    133,   Neg. LLF: 2466345.0526524307
Iteration:     19,   Func. Count:    142,   Neg. LLF: 6182261.40858335
Iteration:     20,   Func. Count:    152,   Neg. LLF: 6107.197989046836
Iteration:     21,   Func. Count:    159,   Neg. LLF: 5869.369971160472
Iteration:     22,   Func. Count:    166,   Neg. LLF: 5860.528120346197
Iteration:     23,   Func. Count:    173,   Neg. LLF: 5850.671859493443
Iteration:     24,   Func. Count:    180,   Neg. LLF: 5852.020227334855
Iteration:     25,   Func. Count:    187,   Neg. LLF: 559578996.2896701
Iteration:     26,   Func. Count:    195,   Neg. LLF: 6455.744175879281
Iteration:     27,   Func. Count:    203,   Neg. LLF: 5831.372434123069
Iteration:     28,   Func. Count:    210,   Neg. LLF: 7767.258857438392
Iteration:     29,   Func. Count:    219,   Neg. LLF: 5812.288041230389
Iteration:     30,   Func. Count:    226,   Neg. LLF: 5811.9938579847985
Iteration:     31,   Func. Count:    232,   Neg. LLF: 5811.993456411456
Iteration:     32,   Func. Count:    238,   Neg. LLF: 5811.993444495178
Iteration:     33,   Func. Count:    243,   Neg. LLF: 5811.99344449389
Optimization terminated successfully    (Exit mode 0)
            Current function value: 5811.993444495178
            Iterations: 34
            Function evaluations: 243
            Gradient evaluations: 33
                          AR - GARCH Model Results
==============================================================================
Dep. Variable:                  Close   R-squared:                       0.011
Mean Model:                        AR   Adj. R-squared:                  0.011
Vol Model:                      GARCH   Log-Likelihood:               -5811.99
Distribution:                  Normal   AIC:                           11634.0
Method:            Maximum Likelihood   BIC:                           11664.1
                                        No. Observations:                 3019
Date:                Tue, Oct 24 2023   Df Residuals:                     3017
Time:                        18:01:03   Df Model:                            2
                                 Mean Model
==============================================================================
                 coef    std err          t      P>|t|        95.0% Conf. Int.
------------------------------------------------------------------------------
Const          0.0425  2.821e-02      1.506      0.132  [-1.282e-02,9.777e-02]
Close[1]      -0.0485  2.307e-02     -2.104  3.536e-02 [-9.377e-02,-3.329e-03]
                            Volatility Model
==============================================================================
                 coef    std err          t      P>|t|       95.0% Conf. Int.
------------------------------------------------------------------------------
omega          0.0298  3.584e-02      0.831      0.406   [-4.046e-02,  0.100]
alpha[1]       0.0257  1.435e-02      1.788  7.379e-02 [-2.469e-03,5.378e-02]
beta[1]        0.9647  2.591e-02     37.230 2.274e-303      [  0.914,  1.016]
==============================================================================

Covariance estimator: robust
```

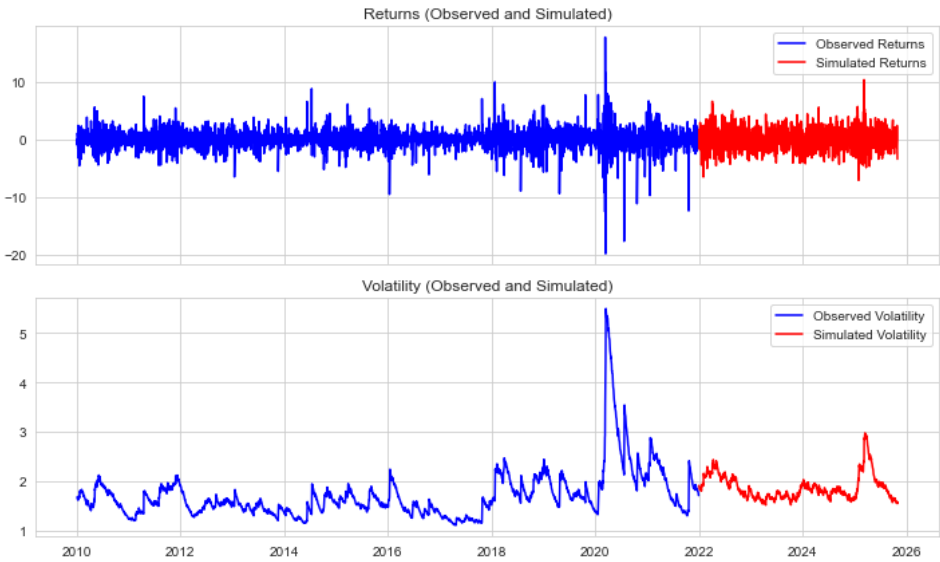

Figure 1-11 Observed and simulated returns and volatilities (result of Code 1-19)

Code 1-20 Different model specifications under *Arch* package

```
# Different model specifications with N(0,1) distribution
model = arch_model(log_returns) # GARCH(1,1) without AR() process
model = arch_model(log_returns, p=1, o=1, q=1) #GJR-GARCH model without AR() process
model = arch_model(log_returns, p=1, o=1, q=1, power=1.0) # TARCH(1,1) (also known as ZARCH) model without AR() process

# Different model specifications with the Student t distribution
model = arch_model(log_returns, dist="StudentsT") # GARCH(1,1) without AR() process
model = arch_model(log_returns, p=1, o=1, q=1, dist="StudentsT") #GJR-GARCH model without AR() process
model = arch_model(log_returns, p=1, o=1, q=1, power=1.0, dist="StudentsT") # TARCH(1,1) (also known as ZARCH) model without AR() process
```

Question 1-15 Statistical signifance of the parameters in the model

Are the parameters of the estimated model presented in Code 1-19 statistically significant?

Question 1-16 Find the best model

Estimate all model specifications in Code 1-20 (you can specify even higher lags) and find the best model. Which criterion did you apply?

Chapter 2
Risk Measurement and Management

By reviewing the widely used measures of investment risk, Sortino and Satchell (2001) pointed out that there are three common properties of different risk measures:

- asymmetry,
- relativity to benchmark,
- multidimensionality.

In the context of investment risk, ***asymmetry*** refers to the idea that investors may not treat positive and negative outcomes equally. For instance, losses (negative returns) might have a greater impact on an investor's perception of risk than gains (positive returns) of the same magnitude. Traditional measures such as standard deviation consider both upward and downward volatility equally. However, measures that account for asymmetry, like semideviance or all safety-first risk measures (Value at Risk, Conditional Value at Risk, etc.), focus only on the downside volatility or the left tail of the probability distribution of returns.

Many risk measures are ***relative*** to a specific benchmark. A benchmark is a point of reference against which things can be compared or assessed. In finance, common benchmarks include market indices such as the S&P 500. In terms of risk management for financial institutions, such as banks and insurance companies, the benchmark can be a risk-free investment or a zero return.

The risk in investments is not one-dimensional. There are ***multiple*** factors that contribute to the overall risk of an investment. For instance, an investment might have market risk (i.e. how investment value changes related to the overall market changes), credit risk (risk of default), liquidity risk (risk associated with the inability to buy/sell the investment quickly without causing a significant change in its price), and many others. A comprehensive assessment of investment risk requires considering all these dimensions.

Based on the properties of risk measures, Rachev et al. (2005) put the risk measures from the literature on portfolio selection problems into two broad categories: dispersion risk measures and safety-first risk measures.

Dispersion risk measures provide insights into the variability or spread of a set of data points, typically returns on an investment. The most common measure of dispersion is the standard deviation, or its square, the variance, which quantifies the average deviation of returns from their mean. Another example is mean absolute deviation, which measures the absolute deviation from the mean of portfolio return. Compared with the standard deviation, the mean absolute deviation is more robust to the outliers. Dispersion risk

measures are fundamental in portfolio management, as they help to construct portfolios that aim to optimize returns for a given level of risk. These measures are discussed in more detail in Section 3.2.2.

Safety-first risk measures are rooted in the principle of prioritizing capital preservation over the pursuit of returns. Unlike dispersion risk measures, which focus on the variability of returns, safety-first measures emphasize avoiding returns that fall below a certain threshold or target. Safety-first measures are particularly relevant for conservative investors, institutions with specific liabilities (like pension funds), or any investment scenario where the primary concern is to avoid significant losses.

The most classic example of a safety-first measure is the Roy safety-first criterion, which evaluates the probability that portfolio returns fall below a threshold level. The lower this probability, the better the portfolio is considered from a safety-first perspective. Another common measure is the Value at Risk (VaR), which estimates the potential loss an investment portfolio could face over a specified period at a given confidence level.

An alternative perspective considers whether a risk measure aligns with logical and rational expectations. Artzner et al. (1999) outlined four axioms that these risk measures should satisfy. When a measure adheres to all four of these criteria, it is termed a ***coherent risk measure***.

The chapter is structured as follows. In Subchapter 2.1, we begin by delving into the axioms of coherent risk measures as defined by Artzner et al. (1999). Subsequently, in Subchapter 2.2, we explore two safety-first risk measures: the widely-used Value at Risk, which is not coherent, and the increasingly popular Conditional Value at Risk, which is coherent. Our discussion also covers the three primary methods of estimating risk measures as introduced in Subchapter 2.3, namely the analytical method, historical simulation, and Monte Carlo simulation. We provide specific examples of the calculation for both the Value at Risk and the Conditional Value at Risk. Concluding the chapter, in Subchapter 2.4, we present and elucidate the process of validating risk measure estimations, commonly referred to as backtesting.

2.1 Axioms of coherent risk measures

According to Artzner et al. (1999), "*A risk measure satisfying the four axioms of translation invariance, subadditivity, positive homogeneity, and monotonicity is called coherent.*" To illustrate the four axioms, we first introduce the relevant symbols. Considering Ω as the set of finite outcomes of an experiment and G as the set of all real-valued functions on Ω, and $\rho(X)$ is defined as the risk measure ρ to X, where $X \in G$.

Axiom I. Monotonicity.

If for all $X1$ and $X2 \in G$, $X1 \leq X2$, then $\rho(X1) \geq \rho(X2)$. This inequality is used to describe the monotonicity of risk measures, which means that the risk of good assets should be lower than the risk of inferior assets.

Axiom II. Subadditivity.

The subadditivity of the risk measures can be expressed as $\rho(X1 + X2) \leq \rho(X1) + \rho(X2)$, for all $X1$ and $X2 \in G$. This means that the overall risk of a portfolio investment does not exceed the sum of all individual risks.

Axiom III. Positive homogeneity.

For any $\alpha \geq 0$, there is $\rho(\alpha \cdot X) = \alpha \cdot \rho(X)$. This equality is used to describe the positive homogeneity of the risk measures, which means that the risk measure is not affected by its unit. In addition, this property can also be regarded as a special case of subadditivity, which reflects the case of no-diversification of risk.

Axiom IV. Translation invariance.

For any real number α, we have $\rho(X + \alpha) = \rho(X) - \alpha$. That means that adding (subtracting) the amount of cash α to the initial position will simply decrease (increase) the risk measurement by α.

2.2 Selected risk measures

In this subchapter, we focus specifically on two safety-first risk measures: Value at Risk and Conditional Value at Risk. Dispersion risk measures, which are more related to portfolio optimization theory, are discussed in Section 3.2.2.

2.2.1 Value at Risk

Value at Risk (VaR) is a risk measure that is mainly used by financial institutions. Jorion (2006, p. 106) defines Value at Risk as "*the worst loss over a target horizon such that there is a low, prespecified probability that the actual loss will be larger. This definition involves two quantitative factors, the horizon and the confidence level*." Simply speaking, VaR essentially expresses the maximum possible loss at a certain significance level α (one minus the confidence level) over a selected time period Δt. Formally, it can be defined as follows:

$$VaR_{X,\alpha} = -min\{x|Pr(X \leq x) \leq \alpha\}, \tag{2.1}$$

where *X* is a random variable profit on the target horizon Δt and α is the significance level. Simply speaking, for parametric VaR under some continuous probability distribution, VaR is, in fact, the negative value of the percentile of the probability distribution of the random variable *X* (profit). By profit in this case we mean the return in monetary terms. It should be noted that VaR can also be defined using the confidence level, usually represented by beta, $\beta = 1 - \alpha$.

Using VaR as a measure of risk, various risks can be quantified in monetary terms. However, in many applications, it is sufficient to work only with returns and define VaR as the negative value of the relevant percentile of the probability distribution of returns instead of profits in monetary units.

It is clear from the definition of VaR that the probability that a continuous random variable *X* is less than the negative VaR is α,

$$Pr\left(X \leq -VaR_{X,\alpha}\right) = \alpha, \tag{2.2}$$

which illustrates Figure 2-1.

The most commonly applied significance levels are 15%, 1%, and 0.5%. These levels are also embedded in the methodologies set out in the legislation. The time horizon over which VaR is calculated can also vary from institution to institution: 10 days for banks

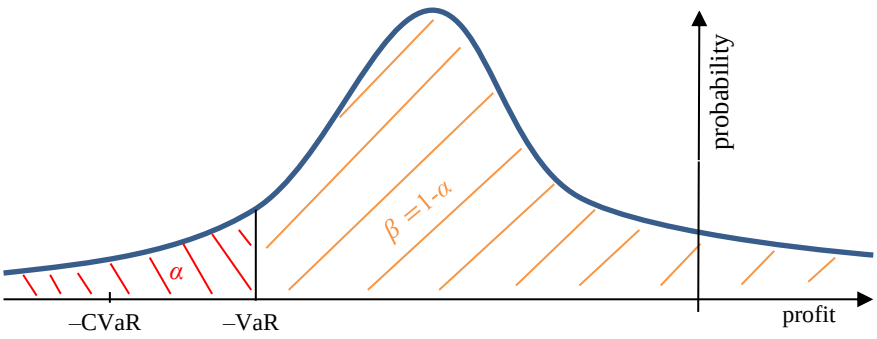

Figure 2-1 VaR and CVaR under continuous probability distribution of profit

Source: Kresta (2016, p. 76)

under Basel II methodology and one year for insurance companies under Solvency II methodology. In general, however, VaR is calculated for shorter time intervals, most often one day, and then recalculated for longer time intervals. Based on the daily VaR, the accuracy of the model is also verified via the so-called back-testing, see Subchapter 2.4.

Many authors criticize the Value at Risk methodology. The main reason for the criticism is that VaR says nothing about the distribution of extreme losses that are greater than this value. The VaR also fails to satisfy the properties of coherent measures of risk, specifically the subadditivity condition, under some special conditions (for some probability distributions). For these reasons, it is preferable to use the Conditional Value at Risk methodology. However, despite these criticisms, VaR is still often used as a measure of risk in the regulation of financial institutions, probably due to its simple and clear interpretation.

Question 2-1 Finish the sentences with VaR

Assume that we are holding a portfolio of stocks with a value of $1,000,000. We calculated that the market risk measured by Value at Risk with a time horizon of 1 year and a confidence level of 95% is 200,000$. Finish the following sentences:
With 95% probability (i.e. in 95 cases out of 100) the loss over a year will not be …
You are 95% sure that the loss over a year will not be …
In only 5% of cases, the loss over a year will be …
With 95% probability (i.e. in 95 cases out of 100) the profit over a year will not be …
You are 95% sure that the profit over a year will not be …
In only 5% of cases, the profit over a year will be …
To finish the sentences, use both monetary units and simple returns.

2.2.2 Conditional Value at Risk

The Conditional Value at Risk (CVaR), also referred to as Expected Shortfall (ES), Average Value at Risk (AVaR), or Tail Value at Risk, can be defined as the mean loss exceeding the VaR at a given confidence level,

$$CVaR_{X,\alpha} = -E\big(X|-X > VaR_{X,\alpha}\big). \tag{2.3}$$

The calculation represents the average loss (negative profit) if the loss is greater than VaR, see

Figure 2-1. CVaR is the answer to the question of what happens if something goes wrong and the loss is higher than VaR. The disadvantage of CVaR as a risk measure is the difficulty of verifying the accuracy of the model.

Unlike VaR, CVaR satisfies all the axioms of coherent risk measures. One of the key advantages of CVaR over VaR is its applicability in portfolio optimization problems, where it allows for easier optimization because the problem can be formulated as a linear optimization problem. However, a notable drawback of CVaR is the difficulty of directly backtesting its estimates. Typically, the robustness of the CVaR estimation process is indirectly assessed by backtesting VaR estimates at various significance levels, see Subchapter 2.4.

Question 2-2 Finish the sentences with CVaR

Assume that we are holding a portfolio of stocks with a value of $1,000,000. We calculated that the market risk, measured by Conditional Value at Risk with a time horizon of 1 year and confidence level of 95% is 400,000$. Finish the following sentences:
In 5% worst cases/scenarios the loss over a year will be …
In 5% worst cases/scenarios the profit over a year will be …
To finish the sentences, use both monetary units and simple returns.

2.3 VaR and CVaR estimation methods

In line with Resti and Sironi (2007), the methods for calculating VaR (and hence CVaR) can be divided into the following three groups:

- analytical methods,
- historical simulations,
- Monte Carlo simulations.

This division is not limited to VaR and CVaR, but also applies to other risk measures.

The ***analytical method*** is based on the assumption that the returns of a portfolio of assets can be modeled using a suitable joint probability distribution that can be described parametrically by a formula (for example, a joint normal probability distribution). Assuming a suitably chosen joint probability distribution, the VaR can be determined analytically. In the case of a joint normal distribution, only the covariance matrix and the vector of expected returns are needed.

The ***historical simulation*** method does not work with a parametric distribution of asset returns but uses historical empirical distributions or scenarios. The VaR is thus determined on the basis of past observations as a negative value of appropriate quantile and CVaR as a negative value of the mean from a given part of the left tail of the distribution.

The most general and also the most flexible is the ***Monte Carlo simulation*** method. This method consists of generating random numbers and then estimating the VaR as an appropriate quantile of the obtained probability distribution.

Table 2-1 Advantages and disadvantages of particular methods

Method	***Advantages***	***Disadvantages***
analytical method	a fast and simple calculation no need for extensive historical data	less accurate results inability to apply to non-linear portfolios strong assumptions (normal distribution of returns)
historical simulation	suitable for all instruments there is no need to make assumptions about the probability distribution or dependence faster than the Monte Carlo simulation	demands for a long series of historical data problem of structural change in the market (need for long time series - history may no longer correspond to the future) risk of inaccurate results for small significance levels
Monte Carlo simulation	suitable for all instruments various assumptions about the probability distribution can be used (modeling heavy tails) no need for extensive historical data	high time and computational requirements quantifies risk only if scenarios are generated from a suitable probability distribution

An overview of the advantages and disadvantages of using each method is summarized in Table 2-1, and each estimation method is discussed in more detail in the following sections.

In the following sections of this textbook, we embark on an in-depth exploration of market risk. To illustrate our examples, consider a portfolio with a value denoted by W. This portfolio is comprised of an assortment of stocks. The quantity of each stock within the portfolio is symbolized by v_i, while the proportion of each stock relative to the entire portfolio is signified by w_i. To determine the portfolio's value and return, we can refer to formulas in Subchapter 3.1.

In Subchapter 2.2, we introduced the concepts of Value at Risk (VaR) and Conditional Value at Risk (CVaR) in terms of monetary units, which we refer to as the absolute VaR and CVaR. However, it is important to recognize that these risk measures can also be articulated in relative terms, that is, as a percentage indicating the potential percentage loss in value. To elucidate:

$$VaR_{\alpha,\Delta t,\text{absolute}} = W \cdot VaR_{\alpha,\Delta t,\text{relative}}, \tag{2.4}$$

$$CVaR_{\alpha,\Delta t,\text{absolute}} = W \cdot CVaR_{\alpha,\Delta t,\text{relative}}. \tag{2.5}$$

Here, ***relative*** implies that VaR and CVaR are expressed as a percentage, representing the loss per one dollar of portfolio value (or any other currency unit). On the contrary, ***absolute*** denotes that these measures reflect the loss in actual currency units for the entire portfolio value W. In subsequent code examples, we focus on calculating the relative values. However, the computation of absolute values can be effortlessly achieved in one of two ways:

- by multiplying the relative values by the portfolio value W, which relates to the third axiom of coherency (positive homogeneity, see Subchapter 2.1),
- by substituting the relative weights w with the absolute weights $u = W \cdot w$ as the function inputs.

Furthermore, we endeavor to compute both VaR and CVaR based on individual risk factors, specifically stock returns, whenever possible. In addition to this, we present a simplified methodology in which a single aggregated risk factor, namely the portfolio return, is presumed. This approach offers a streamlined alternative to the more granular analysis based on individual stock returns.

2.3.1 The analytical method

The analytical approach to estimating the Value at Risk (VaR) and the Conditional Value at Risk (CVaR) represents the most straightforward methodology to calculate these risk metrics. This technique involves assuming a parametric probability distribution, estimating its parameters, and then utilizing a closed-form formula to compute VaR and CVaR. The accuracy of these calculations is dependent on the precision of the estimated parameters.

A myriad of probability distributions can be postulated, with the Gaussian (or normal) distribution being the most elementary. When the stock returns adhere to a joint Gaussian distribution, the portfolio return also conforms to a normal distribution. The parameters of this distribution, specifically the mean and standard deviation of the portfolio return, can be calculated according to the formulas in Section 3.3.1

In the following sections, we concentrate on the Gaussian distribution. First, we demonstrate the computation of VaR and CVaR for a single risk factor, namely portfolio return. Subsequently, we expand the scope of our analysis to encompass individual risk factors (the returns of portfolio components) by simply plugging in the calculations for the portfolio's mean and standard deviation. This step-by-step approach allows us to build from a foundational understanding of the VaR and CVaR calculation for a single risk factor to a more complex analysis involving multiple risk factors.

Single risk factor

If we assume a normal distribution of returns, then equation (2.1) can be rewritten using the inverse of the normal distribution function F_N^{-1} as follows,

$$VaR_{\alpha,\Delta t} = -F_N^{-1}(\alpha; \mu_P, \sigma_P) = \Phi^{-1}(1-\alpha) \cdot \sigma_P - \mu_P, \tag{2.6}$$

i.e. VaR is the negative value of the corresponding quantile of the portfolio profit distribution, respectively return distribution in the case of relative VaR. The equation can be used to calculate both:

- the VaR in monetary units, then μ_P, σ_P are the mean and standard deviation of the portfolio profit (in currency units),
- and the relative VaR in percentage, then μ_P, σ_P are the mean and standard deviation of the portfolio return (in percentage).

The CVaR calculation assuming a normal distribution for a single risk factor (portfolio profit of return) is as follows,

$$CVaR_\alpha = \alpha^{-1} \cdot \varphi\left(\Phi^{-1}(\alpha)\right) \cdot \sigma_P - \mu_P, \tag{2.7}$$

where φ denotes the probability density function of the standard normal distribution, and Φ^{-1} is the inverse of the distribution function of the standard normal distribution. Again,

the same formula can be used to calculate the absolute CVaR in currency units when μ_P, σ_P are the mean and standard deviation of the portfolio profit (in currency units) and relative CVaR when μ_P, σ_P are the mean and standard deviation of the portfolio return (in percentage). The snippet code is shown in Code 2-1.

Question 2-3 Standard normal distribution

What is the standard normal distribution?

Code 2-1 Single factor analytical VaR and CVaR assuming normal distribution

```
def calcAnalyticalSingle(ret, alpha): # single risk factors
    VaR = norm.ppf(1-alpha) * ret.std() - ret.mean() # we calculate VaR
    CVaR = norm.pdf(norm.ppf(alpha))/(alpha) * ret.std() - ret.mean() # we calculate CVaR
    return VaR.iloc[0], CVaR.iloc[0]
```

Multiple risk factors

A similar approach can be taken for a portfolio of stocks. If we assume that the discrete returns of individual assets have a joint normal probability distribution, then the portfolio return also has a normal probability distribution. Using formulas from Section 3.3.1, the mean and standard deviation of the portfolio return, and hence the VaR, can be determined as follows,

$$VaR_{\alpha,\Delta t} = -F_N^{-1}\left(\alpha; \mu^T \cdot w, \sqrt{w^T \cdot \boldsymbol{Q} \cdot w}\right), \tag{2.8}$$

$$VaR_{\alpha,\Delta t} = \Phi^{-1}(1-\alpha) \cdot \sqrt{w^T \cdot \boldsymbol{Q} \cdot w} - \mu^T \cdot w, \tag{2.9}$$

where **Q** is the covariance matrix of the returns and μ is the vector of expected returns. In this way, we obtain the ***relative VaR*** as a percentage.

The ***absolute VaR*** can be computed if instead of the weight in percentage w, we use the weight in currency units u, that is, how much money we invest in the given asset, $u = W \cdot w$. Then, VaR can be calculated as follows,

$$VaR_{\alpha,\Delta t} = -F_N^{-1}\left(\alpha; \mu^T \cdot u, \sqrt{u^T \cdot \boldsymbol{Q} \cdot u}\right), \tag{2.10}$$

$$VaR_{\alpha,\Delta t} = \Phi^{-1}(1-\alpha) \cdot \sqrt{u^T \cdot \boldsymbol{Q} \cdot u} - \mu^T \cdot u. \tag{2.11}$$

For the portfolio, CVaR can be calculated as follows, either from relative weights w,

$$CVaR_{\alpha,\Delta t} = \alpha^{-1} \cdot \varphi\left(\Phi^{-1}(\alpha)\right) \cdot \sqrt{w^T \cdot \boldsymbol{Q} \cdot w} - \mu^T \cdot w, \tag{2.12}$$

or absolute weights u,

$$CVaR_{\alpha,\Delta t} = \alpha^{-1} \cdot \varphi\left(\Phi^{-1}(\alpha)\right) \cdot \sqrt{u^T \cdot \boldsymbol{Q} \cdot u} - \mu^T \cdot u. \tag{2.13}$$

The snippet code is shown in Code 2-2.

Code 2-2 Multiple factor analytical VaR and CVaR assuming normal distribution

```
def calcAnalyticalMultiple(ret, w, alpha): # multiple risk factors
    mu = ret.mean() # we calculate expected returns
    Q = ret.cov()     # we calculate covariance matrix
    stdp = np.sqrt(w.transpose().dot(Q.dot(w))) # we calculate portfolio standard deviation
    mup = mu.transpose().dot(w) # we calculate portfolio return expected value
    VaR = norm.ppf(1-alpha) * stdp - mup # we calculate VaR
```

```
    CVaR = norm.pdf(norm.ppf(alpha))/(alpha) * stdp - mup # we calculate CVaR
    return VaR.iloc[0,0], CVaR.iloc[0,0]
```

2.3.2 The historical simulation method

The historical simulation method for estimating Value at Risk and Conditional Value at Risk is established on the assumption that historical patterns in portfolio returns (or the returns of individual assets) will persist into the future. This method avoids theoretical probability distributions in favor of empirical evidence, positing that the future distribution of returns will closely resemble the distribution observed in historical data.

At the core of this approach is the substitution of the expected distribution of future returns with the actual, observed distribution of past returns. By employing the empirical distribution, this method leverages real-world data to predict potential risk. This technique is particularly appealing for its simplicity and its reliance on actual historical return data, which can provide a more intuitive and tangible basis for risk estimation compared to parametric methods.

To determine the VaR of a linear portfolio of assets using historical simulation, we follow these steps:

1. choose the duration of the historical return time series for each asset in the portfolio (or risk factors),
2. re-evaluating the current value of individual assets, and thus the overall portfolio, based on historically observed returns, thereby obtaining a probability distribution of the portfolio value,
3. compute the VaR as the difference between the current value and the relevant percentile of the portfolio value probability distribution.

The alternative approach is simply to calculate the quantile of the probability distribution of the past portfolio returns. This quantile is multiplied by the portfolio value *W*. See Code 2-3.

For calculating the CvaR, a similar procedure is applied; however, instead of the percentile, we use the following formula:

$$CVaR_{X,\alpha} = W - \frac{1}{\alpha} \cdot \left(\frac{1}{n} \sum_{a=1}^{[\alpha \cdot n]-1} x_a + \left(\alpha - \frac{[\alpha \cdot n]-1}{n} \right) x_{[\alpha n]} \right), \tag{2.14}$$

where *x* is the time series of re-evaluated portfolio values, $[\alpha \cdot n]$ symbolizes the lowest integer greater than $\alpha \cdot n$ and *n* is the number of elements in vector *x*. If *n* is large enough, the part $\left(\alpha - \frac{[\alpha \cdot n]-1}{n} \right) x_{[\alpha n]}$ would be small enough to be omitted, and we can calculate CVaR simply as,

$$CVaR_{X,\alpha} = W - \frac{1}{[\alpha \cdot n]-1} \cdot \sum_{a=1}^{[\alpha \cdot n]-1} x_a. \tag{2.15}$$

Again, alternatively, we can simply calculate CVaR as the mean of the returns below relative VaR and multiply it with the portfolio value *W*, see Code 2-3.

The above-mentioned formulas can be used both in historical simulation and in Monte Carlo simulation.

Code 2-3 VaR and CVaR calculation via historical simulation method

```
def calcHistoricalSingle(ret, alpha): # single risk factors
    VaR = -ret.quantile(q=alpha, interpolation='linear')
    CVaR = -ret[ret<-VaR].mean()
    return VaR.iloc[0], CVaR.iloc[0]


def calcHistoricalMultiple(ret, w, alpha): # multiple risk factors
    # calculates VaR by means of historical simulation
    returns_historical = ret.dot(w) # we calculate new/future portfolio returns if the
historical returns repeat
    VaR = -returns_historical.quantile(q=alpha, interpolation='linear')
    CVaR = -returns_historical[returns_historical<-VaR].mean()
    return VaR.iloc[0], CVaR.iloc[0]
```

In line with Resti and Sironi (2007), the following advantages of the historical simulation method can be highlighted. The concept of VaR estimation is easy to understand: the resulting VaR estimate can be interpreted as the loss at a given significance level that would be realized if historical conditions were repeated. No assumptions are made about the parametric form of the probability distribution of returns on individual assets. In fact, the probabilistic behavior of the returns is assumed to be stable over time. There is no need to estimate a covariance matrix. Again, this method assumes that the dependence structure of the returns of the individual assets in the portfolio is invariant. The VaR estimate using the historical simulation method is stable over time.

However, the disadvantages of this method should also be highlighted. The probability distribution of the returns is assumed to be invariant over time. This disadvantage is overcome by the filtered historical simulation, see below. The choice of the length of the historical time series is problematic, as the choice of a short time series leads to poor modeling of the ends of the probability distribution and hence inaccurate risk estimation; conversely, the choice of too long a time series increases the probability that the stability of probability distribution is invalid.

Filtered historical simulation

Filtered historical simulation is a semiparametric method that was designed and further elaborated in publications by Duffie and Pan (1997), Hull and White (1998), and Barone-Adesi et al. (1998, 1999; 2002). Unlike the historical simulation, constant volatility is not assumed; instead, it is assumed that volatility evolves stochastically. Assuming a very simple stochastic process of the portfolio return,

$$R_{P,t} = \sigma_t \cdot \varepsilon_t, \tag{2.16}$$

where volatility σ_t is modeled by some GARCH-type model, for example by GARCH(1,1) as discussed in Section 1.5.2.

In the filtered historical simulation, the procedure is such that, after estimating the coefficients of the GARCH process, the historical returns are adjusted (filtered) using the estimated historical volatility:

$$e_t = \frac{R_{P,t}}{\sigma_t}. \tag{2.17}$$

This gives us the standardized residuals e, which are independent and from the same probability distribution. Future returns are then modeled using historical simulation of

these residuals adjusted for the estimate of future volatility $\hat{\sigma}_{t+1}$ forecasted by the GARCH model,

$$R_{P,t+1} \approx e \cdot \hat{\sigma}_{t+1}. \tag{2.18}$$

Therefore, the relative value of the risk (in percentage) can be estimated as the α quantile of the standardized residuals multiplied by the estimated value of the future standard deviation. The absolute Value at Risk (in currency units) is simply the relative VaR multiplied by *W*.

2.3.3 The Monte Carlo simulation method

The Monte Carlo simulation is based on the law of large numbers, which expresses the idea that with a large number of independent trials, one can almost certainly expect the relative frequency to be close to the theoretical probability value. This can also be expressed in such a way that the frequencies and their characteristics will converge to the theoretical distribution and to the theoretical characteristics as the number of random trials increases. This convergence is understood as probabilistic convergence, which is different from mathematical convergence.

A Monte Carlo simulation is essentially a generalization of a historical simulation. While the historical simulation considers an empirical probability distribution for modeling returns, the Monte Carlo simulation considers some appropriate parametric probability distribution. If the VaR is dependent on multiple random variables, e.g. if we are estimating the VaR of a portfolio of assets, we need to take into account the dependence of these random variables. In the simplified case and for elliptical multivariate probability distributions, only correlation can be considered, but more accurate results are provided by the use of copula function modeling.

The procedure for estimating the VaR of a portfolio of assets using a Monte Carlo simulation is described below.

1. First, appropriate marginal probability distributions and copula functions are chosen for each asset, including the estimation of their parameters.
2. Then, based on the estimated parameters of the chosen copula function, random number vectors in the (0,1) interval are generated, which have an appropriate dependence structure.
3. The generated random numbers are then transformed into random returns of the individual assets using the appropriate inverse distribution functions of the chosen marginal distributions (the inverse transformation method is used here).
4. For each vector (scenario) of random asset returns, the portfolio return followed by the portfolio value is calculated.
5. The simulated portfolio values are ranked in ascending order from the lowest to the highest, and the VaR is determined as the difference between the appropriate percentile (specifically, the $n\alpha$-th value, where *n* is the number of simulated scenarios) and the actual portfolio value. The CVaR is determined in a similar way to the historical simulation method according to the equation in Section 2.3.2.

The fact that the Monte Carlo simulation method is the most flexible is demonstrated by the fact that by using appropriate marginal distributions and copula functions, a similar model can be obtained to the analytical and historical simulation methods. Monte Carlo simulation using a Gaussian copula function and marginal probability distributions in the

form of a normal distribution can be viewed as a numerical approximation of the analytical method. Furthermore, the historical simulation can be approximated using the empirical copula function and empirical marginal distributions.

We can streamline the process by considering the portfolio return as a single aggregated risk factor, rather than treating individual stock returns as separate risk factors. This approach eliminates the need to address dependencies and copula functions. The steps for this simplified method are:

1. Identify the suitable stochastic process for portfolio return and determine its probability distribution, including its parameters.
2. Using the estimated parameters, generate random scenarios of portfolio return. These scenarios can span the entire time horizon in one step or be broken down into smaller intervals, mirroring the method outlined in Subchapter 1.5.
3. For each simulated scenario of portfolio returns, compute the cumulative portfolio return over the entire period and then determine the portfolio's value.
4. Rank the simulated portfolio values in ascending order. VaR is then calculated as the difference between the specific percentile (specifically, the $n \cdot \alpha$-th value, where n is the number of simulated scenarios) and the current portfolio value. CVaR is computed similarly to the method used in historical simulations according to the equation in Section 2.3.2.

As can be seen, this procedure is similar to the historical simulation method. In Code 2-4 we show only a very simple application of Monte Carlo simulation assuming single risk factor with Gaussian distribution and multiple risk factors with joint Student distribution. Note that a more advanced approach based on stochastic processes can be applied; see, for example, codes Code 1-17, Code 1-18, Code 1-19, and Code 1-20.

Code 2-4 VaR and CVaR calculation via Monte Carlo simulation method

```
from scipy.stats import multivariate_normal, multivariate_t
# calculates VaR and CVaR by means of Monte Carlo simulation method assuming a normal
distribution
def calcMCSMultiple_n(ret, w, alpha):
    # calculates VaR by means of Monte Carlo simulation assuming joint normal dist.
    mu = ret.mean() # we calculate mean/expected returns
    Q = ret.cov() # we calculate covariance matrix
    returns_simulated = multivariate_normal.rvs(mean=mu, cov=Q, size=1000000,
random_state=100)
    returns_simulated = pd.DataFrame(returns_simulated, columns=ret.columns).dot(w)
    VaR = -returns_simulated.quantile(q=alpha, interpolation='linear')
    CVaR = -returns_simulated[returns_simulated<-VaR].mean()
    return VaR.iloc[0], CVaR.iloc[0]

# calculates VaR and CVaR by means of Monte Carlo simulation method assuming student t
distribution
def calcMCSMultiple_t(ret, w, alpha, df):
    # calculates VaR by means of Monte Carlo simulation assuming joint Student distribution
with df degrees of freedom
    mu = ret.mean() # we calculate mean/expected returns
    Q = ret.cov() # we calculate covariance matrix
    returns_simulated = multivariate_t.rvs(loc=mu, shape=Q, df=df, size=100000,
random_state=100)
    returns_simulated = pd.DataFrame(returns_simulated, columns=ret.columns).dot(w)
    VaR = -returns_simulated.quantile(q=alpha, interpolation='linear')
    CVaR = -returns_simulated[returns_simulated<-VaR].mean()
    return VaR.iloc[0], CVaR.iloc[0]
```

2.3.4 Example of risk quantification for FAANG stock portfolio

To better understand these methods, consider a straightforward example of a 5-stock portfolio. We use the FAANG portfolio as a reference, which comprises Meta Platforms (formerly Facebook), Apple, Amazon, Netflix, and Google. For this example, our portfolio consists of one stock from each of these companies. In Code 2-5 we download the data and calculate the basic portfolio parameters such as *v*, *u*, *w*, *W*, see above or in Subchapter 3.1.

Code 2-5 Data download and basic calculations

```
tickers = ["META",  # Facebook/Meta Platforms, Inc.
           "AAPL",  # Apple
           "AMZN",  # Amazon
           "NFLX",  # Netflix
           "GOOGL"  # Google (Alphabet Inc.)
           ]
alpha=0.05

# we download the data
prices = yf.download(tickers, start="2018-01-01", end="2022-12-31", interval = "1d",
group_by="column", auto_adjust = True,)["Close"]
returns = prices.pct_change().dropna() # we calculate percentage changes (simple/discrete
returns)

v = pd.DataFrame(data=[1, 1, 1, 1, 1], index=tickers, columns=['weight']) # quantitites held
u = v * prices.iloc[-1:].transpose().values  # weights in absolute value (in $)
W = u.sum()  # portfolio value
w = u /W # relative weights (in %)

print(f'Portfolio value(W): {W[0]:.2f}')
print(f'Portfolio absolute weights(u): {u}')
print(f'Portfolio relative weights(u): {w}')
> Portfolio value(W): 716.83
Portfolio absolute weights(u):            weight
META   129.378006
AAPL    84.000000
AMZN    88.230003
NFLX   120.339996
GOOGL  294.880005
Portfolio relative weights(u):          weight
META   0.180487
AAPL   0.117183
AMZN   0.123084
NFLX   0.167878
GOOGL  0.411368
```

In Code 2-6, we calculate the one-day Value at Risk and the Conditional Value at Risk at 5% significance levels for the FAANG portfolio. First, we calculate the relative VaR and CVaR on line 2 (univariate risk factor) and line 4 (multivariate risk factors). As can be seen, both calculations lead to the same estimates. Then, we calculate the absolute VaR and CVaR using various approaches:

- we multiply the relative VaR and CVaR by the portfolio value, see lines 8-10 and 14-16,
- we use the same function with absolute weights instead of relative weights, see lines 12 and 18.

As can be seen, the resulting estimates are always the same no matter which approach we choose.

Question 2-4 Interpretation of results in Code 2-5 and Code 2-6

Interpret the results printed by Code 2-5 and Code 2-6.

Code 2-6 VaR and CvaR calculations via analytical method

```
# Calculate and print the relative VaR and CVaR
relative_VaR, relative_CVaR = calcAnalyticalSingle(returns.dot(w), alpha)
print(f"Relative VaR: {relative_VaR:.6f}, relative CVaR: {relative_CVaR:.6f}")
relative_VaR, relative_CVaR = calcAnalyticalMultiple(returns, w, alpha)
print(f"Relative VaR (Multiple): {relative_VaR:.6f}, relative CVaR (Multiple):
{relative_CVaR:.6f}")

# Calculate and print the absolute VaR and CVaR
relative_VaR, relative_CVaR = calcAnalyticalSingle(returns.dot(w), alpha)
absolute_VaR = W.values[0] * relative_VaR
absolute_CVaR = W.values[0] * relative_CVaR
print(f"Absolute VaR: {absolute_VaR:.6f}, absolute CVaR: {absolute_CVaR:.6f}")
absolute_VaR, absolute_CVaR = calcAnalyticalSingle(returns.dot(u), alpha)
print(f"Absolute VaR: {absolute_VaR:.6f}, absolute CVaR: {absolute_CVaR:.6f}")
relative_VaR, relative_CVaR = calcAnalyticalMultiple(returns, w, alpha)
absolute_VaR = W.values[0] * relative_VaR
absolute_CVaR = W.values[0] * relative_CVaR
print(f"Absolute VaR: {absolute_VaR:.6f}, absolute CVaR: {absolute_CVaR:.6f}")
absolute_VaR, absolute_CVaR = calcAnalyticalMultiple(returns, u, alpha)
print(f"Absolute VaR: {absolute_VaR:.6f}, absolute CVaR: {absolute_CVaR:.6f}")
> Relative VaR: 0.031500, relative CVaR: 0.039650
Relative VaR (Multiple): 0.031500, relative CVaR (Multiple): 0.039650
Absolute VaR: 22.580193, absolute CVaR: 28.422431
Absolute VaR: 22.580193, absolute CVaR: 28.422431
Absolute VaR: 22.580193, absolute CVaR: 28.422431
Absolute VaR: 22.580193, absolute CVaR: 28.422431
```

Question 2-5 Assumption of joint normal distribution

Consider the assumption of a joint normal distribution for asset returns. Is this assumption always appropriate in the context of financial markets? Reflect on the characteristics of empirical returns, such as skewness and kurtosis. How might these characteristics deviate from a normal distribution? Can we empirically test the returns for normality? If so, which statistical tests would be suitable for this purpose? Can we test the returns obtained in Code 2-5 for normality?

In Code 2-7, we calculate the one-day Value at Risk and the Conditional Value at Risk at 5% significance levels for the FAANG portfolio by means of a historical simulation method. First, we calculate the relative VaR and CVaR on line 2 (univariate risk factor) and line 4 (multivariate risk factors). As can be seen, both calculations lead to the same estimates. Then, we calculate the absolute VaR and CVaR using various approaches:

- we multiply the relative VaR and CVaR by the portfolio value, see lines 8-10 and 14-16,
- we use the same function with absolute weights instead of relative weights, see lines 12 and 18.

As can be seen, the resulting estimates are always the same no matter which approach we choose.

Code 2-7 VaR and CvaR calculations via historical simulation

```
# Calculate and print the relative VaR and CVaR
relative_VaR, relative_CVaR = calcHistoricalSingle(returns.dot(w), alpha)
print(f"Relative VaR: {relative_VaR:.6f}, relative CVaR: {relative_CVaR:.6f}")
relative_VaR, relative_CVaR = calcHistoricalMultiple(returns, w, alpha)
print(f"Relative VaR (Multiple): {relative_VaR:.6f}, relative CVaR (Multiple):
{relative_CVaR:.6f}")

# Calculate and print the absolute VaR and CVaR
relative_VaR, relative_CVaR = calcHistoricalSingle(returns.dot(w), alpha)
absolute_VaR = W.values[0] * relative_VaR
absolute_CVaR = W.values[0] * relative_CVaR
print(f"Absolute VaR: {absolute_VaR:.6f}, absolute CVaR: {absolute_CVaR:.6f}")
absolute_VaR, absolute_CVaR = calcHistoricalSingle(returns.dot(u), alpha)
print(f"Absolute VaR: {absolute_VaR:.6f}, absolute CVaR: {absolute_CVaR:.6f}")
relative_VaR, relative_CVaR = calcHistoricalMultiple(returns, w, alpha)
absolute_VaR = W.values[0] * relative_VaR
absolute_CVaR = W.values[0] * relative_CVaR
print(f"Absolute VaR: {absolute_VaR:.6f}, absolute CVaR: {absolute_CVaR:.6f}")
absolute_VaR, absolute_CVaR = calcHistoricalMultiple(returns, u, alpha)
print(f"Absolute VaR: {absolute_VaR:.6f}, absolute CVaR: {absolute_CVaR:.6f}")
> Relative VaR: 0.032629, relative CVaR: 0.047030
Relative VaR (Multiple): 0.032629, relative CVaR (Multiple): 0.047030
Absolute VaR: 23.389373, absolute CVaR: 33.712144
Absolute VaR: 23.389373, absolute CVaR: 33.712144
Absolute VaR: 23.389373, absolute CVaR: 33.712144
Absolute VaR: 23.389373, absolute CVaR: 33.712144
```

In Code 2-8, we calculate the one-day Value at Risk and the Conditional Value at Risk at 5% significance levels for the FAANG portfolio by means of a Monte Carlo simulation assuming (joint) normal distribution. First, we calculate the relative VaR and CVaR on line 2 (multivariate risk factors). Then, we calculate the absolute VaR and CVaR by various approaches:

- we multiply the relative VaR and CVaR by the portfolio value, see lines 6-8,
- we use the same function with absolute weights instead of relative weights, see line 10.

As can be seen, the resulting estimates are always the same no matter which approach we choose.

Code 2-8 VaR and CvaR calculations via MC simulation (joint normal distribution)

```
# Calculate and print the relative VaR and CVaR
relative_VaR, relative_CVaR = calcMCSMultiple_n(returns, w, alpha)
print(f"Relative VaR: {relative_VaR:.6f}, relative CVaR: {relative_CVaR:.6f}")

# Calculate and print the absolute VaR and CVaR
relative_VaR, relative_CVaR = calcMCSMultiple_n(returns, w, alpha)
absolute_VaR = W.values[0] * relative_VaR
absolute_CVaR = W.values[0] * relative_CVaR
print(f"Absolute VaR: {absolute_VaR:.6f}, absolute CVaR: {absolute_CVaR:.6f}")
absolute_VaR, absolute_CVaR = calcMCSMultiple_n(returns, u, alpha)
print(f"Absolute VaR: {absolute_VaR:.6f}, absolute CVaR: {absolute_CVaR:.6f}")
> Relative VaR: 0.031464, relative CVaR: 0.039594
Absolute VaR: 22.554041, absolute CVaR: 28.381794
Absolute VaR: 22.554041, absolute CVaR: 28.381794
```

Question 2-6 Different values

In our examples, both the analytical method and Monte Carlo simulation methods assume the same probability distribution for returns modeling (joint normal distribution). Why do the estimated VaR and CVaR differ? What should we do for the Monte Carlo simulation to converge to the analytical method?

In Code 2-9, we calculate the one-day Value at Risk and the Conditional Value at Risk at 5% significance levels for the FAANG portfolio by means of a Monte Carlo simulation assuming (joint) Student distribution. In this code, we limit the example to relative VaR and CVaR only, although the extension to absolute VaR and CVaR would be as simple as in the previous examples. On the contrary, we focus on the influence of parameter *df* on the resulting estimates. We calculate the relative VaR and CVaR for degrees of freedom in the joint Student distribution equal to 4, 40, and 400. By this parameter, we model the tails of the probability distribution. The lower the value of *df*, the fatter (or heavier) the tails. The fatter the tails of the joint probability distribution, the higher the probability of extreme losses (and gains), resulting in higher estimates of VaR and CVaR.

Question 2-7 Student t distribution and Gaussian distribution

Under what conditions does the Student t distribution converge to the Gaussian distribution?

Code 2-9 VaR and CvaR calculations via MC simulation (joint Student distribution)

```
# Calculate and print the relative VaR and CVaR
relative_VaR, relative_CVaR = calcMCSMultiple_t(returns, w, alpha, df=4)
print(f"Degrees of freedom 4. Relative VaR: {relative_VaR:.6f}, relative CVaR:
{relative_CVaR:.6f}")
relative_VaR, relative_CVaR = calcMCSMultiple_t(returns, w, alpha, df=40)
print(f"Degrees of freedom 40. Relative VaR: {relative_VaR:.6f}, relative CVaR:
{relative_CVaR:.6f}")
relative_VaR, relative_CVaR = calcMCSMultiple_t(returns, w, alpha, df=400)
print(f"Degrees of freedom 400. Relative VaR: {relative_VaR:.6f}, relative CVaR:
{relative_CVaR:.6f}")
> Degrees of freedom 4. Relative VaR: 0.040752, relative CVaR: 0.061721
Degrees of freedom 40. Relative VaR: 0.032096, relative CVaR: 0.041009
Degrees of freedom 400. Relative VaR: 0.031530, relative CVaR: 0.039719
```

In the preceding examples (Code 2-6, Code 2-7, Code 2-8, Code 2-9), we observed that each method of estimating Value at Risk (VaR) and Conditional Value at Risk (CVaR) yields slightly different results. This naturally leads to the question of which method provides the most accurate estimate. The answer to this question can be found through the backtesting process, which we explore in detail in the next section. Backtesting allows us to evaluate the performance of each estimation procedure by comparing predicted risks with actual outcomes.

2.4 VaR estimation backtest

The importance of backtesting is stressed by Hull (2021, p. 534): "*Whatever the method used for calculating VaR, an important reality check is back testing. It involves testing how well the VaR estimates would have performed in the past.*"

Within the backtesting procedure, a model's ability to predict future losses is tested. It involves estimating the risk (focusing on Value at Risk in this subchapter) at the time t for time $t + \Delta t$, $VaR_{X,\alpha}(t; t + \Delta t)$, and comparing this with the actual loss observed between these times, $L_X(t; t + \Delta t)$. Typically, the period for which VaR is calculated is one business day, according to Basel II banking supervision standards. In this comparison, two outcomes are possible: the actual loss is either higher than or lower than the estimated loss. A higher actual loss is marked as an ***exception*** (***VaR violation***), represented by "1", while a lower loss is marked as "0",

$$I_t = \begin{cases} 1 & if\ L_X(t; t + \Delta t) > VaR_{X,\alpha}(t; t + \Delta t) \\ 0 & if\ L_X(t; t + \Delta t) \leq VaR_{X,\alpha}(t; t + \Delta t) \end{cases}. \tag{2.19}$$

The backtesting process is applied in rolling windows across the entire data set. This results in a sequence of values indicating whether an exception occurred. From this sequence, we can determine whether the number of exceptions matches our initial assumptions. For a detailed explanation, refer to Resti and Sironi (2007) or Hull (2021).

It is important to note that the total number of days used for backtesting is not equal to the dataset's size. Some data are reserved for initial risk estimation and are not used for backtesting. For clarity, refer to Figure 2-2, which illustrates the rolling windows for the VaR backtest. In this figure, the length of the dataset is 9 days. In order to estimate the risk for the sixth day, values from days 1-5 are utilized, i.e. we assume a historical window of 5 days. Then, in another iteration based on the values from days 2–6, the risk for the seventh day is calculated. Then, the same applies to the eighth day (based on data from the third to seventh day) and the ninth day (based on data from the fourth to eighth day). It is obvious that while the entire dataset contains data from 9 days (n + m), the first 5 days are used for the VaR estimation and thus cannot be used for backtesting. In this case, as the VaR is estimated from the preceding five days (historical window of length m), only four days are used for backtesting purposes (n).

The backtest aims to verify that the observed probability of exceptions' occurring π_{obs} is equal to the expected probability of exceptions' occurring π_{ex}. This equality can be tested statistically. Consider the following null hypothesis:

$$H_0: \pi_{obs} = \pi_{ex}. \tag{2.20}$$

The definition of VaR implies that the expected probability of the exception being observed is equal to α. Then, since the distribution of exceptions over time should be identically and independently (i.i.d.) distributed, the probability of occurrence of n_1 exceptions in n observations is given by the binomial probability distribution,

$$Pr(n_1|\pi_{ex}, n) = \frac{n!}{(n-n_1)!\cdot n_1!} \cdot \pi_{ex}^{n_1}(1 - \pi_{ex})^{n-n_1}. \tag{2.21}$$

The probability that the observed quantity of exceptions is equal to or lower than the selected threshold is given by the cumulative distribution function of the binomial distribution,

$$F_{Bi}(x; \pi_{ex}, n) = Pr(n_1 \leq x|\pi_{ex}, n) = \sum_{i=1}^{x} \frac{n!}{(n-i)!i!} \pi_{ex}^{i}(1 - \pi_{ex})^{n-i}. \tag{2.22}$$

The probability of observing more than x exceptions can be computed as $1 - F_{Bi}(x; \pi_{ex}, n)$.

Backtesting iteration	*Days*								
1	1	2	3	4	5	6	7	8	9
2	1	2	3	4	5	6	7	8	9
3	1	2	3	4	5	6	7	8	9
4	1	2	3	4	5	6	7	8	9

Figure 2-2 Rolling windows for VaR backtesting

Source: (Kresta, 2015, p. 91)

Making any statistical inference about the null hypothesis is connected to the type I and type II errors, see Table 2-2. Type I error is related to the rejection of a valid null hypothesis (the null hypothesis is that the model is accurate). Type II error is associated with the acceptance of an invalid null hypothesis (that is, an imprecise model is accepted as accurate). There is an inverse relationship between type I and type II errors. Thus, by minimizing the probability of type I error, the probability of type II error increases, and vice versa. Since accepting an inaccurate model (type II error) is more costly than rejecting an accurate model (type I error), it is better to be exposed to a higher type I error. This somehow contradicts the standard logic of statistical testing, in which the lower the p-value, the better. The reason is that, as opposed to standard statistical tests, in this case, we want to prove the null hypothesis and reject the alternative hypothesis.

There are also other statistical tests that can be applied for statistical inference about the results of the backtest. Generally, two assumptions are to be tested:

- whether the number of exceptions can be accepted,
- whether the exceptions occur during time randomly and are not dependent on each other (i.e., they are independent and identically distributed).

Further explained statistical tests are the unconditional coverage test proposed by Kupiec (1995) and the conditional coverage test proposed by Christoffersen (1998). However, there are also other statistical tests proposed in the literature, see, e.g., the dynamic quantile test (Engle & Manganelli, 2004), duration-based approach (P. Christoffersen & Pelletier, 2004), and improved duration-based test (Haas, 2005). For an overview, see also Berkowitz et al. (2011).

Table 2-2 Type I and II errors

	The null hypothesis is valid	The null hypothesis is invalid
Reject the null hypothesis	Type I error	Correct statistical inference
Fail to reject the null hypothesis	Correct statistical inference	Type II error

2.4.1 Kupiec's unconditional coverage test

Kupiec's test is derived from a relative amount of exceptions. The test's null hypothesis is that the observed probability of an exception occurring is equal to the expected. A given likelihood ratio test based on χ^2 probability distribution with one degree of freedom is formulated as follows:

$$LR_{Kupiec} = -2\,ln\left[\frac{\pi_{ex}^{n_1}(1-\pi_{ex})^{n_0}}{\pi_{obs}^{n_1}(1-\pi_{obs})^{n_0}}\right], \tag{2.23}$$

where π_{ex} is the expected probability of exception occurring, π_{obs} is the observed probability of exception occurring, n_0 is the number of zeros, and n_1 is the number of ones (exceptions). We know that $\pi_{ex} = \alpha$, $\pi_{obs} = \frac{n_1}{n_0+n_1}$ and $n = n_0 + n_1$. Then, the likelihood ratio can be rewritten as follows,

$$LR_{Kupiec} = 2n_0\,ln\left(\frac{n_0}{n}\right) + 2n_1\,ln\left(\frac{n_1}{n}\right) - 2n_0\,ln(1-\alpha) - 2n_1\,ln(\alpha), \tag{2.24}$$

and the test statistics have asymptotically χ^2 distribution with one degree of freedom.

When making conclusions about the accuracy of the model based on the p-value of the Kupiec test, the following rule of thumb can be applied: the higher the p-value, the more accurate the model. This may seem contradictory to the principles of statistics[13] and thus we explain it further.

In general statistical tests, we formulate the null hypothesis so that we want to disprove it (and accept the alternative hypothesis). In such a framework, we are concerned about the type I error (see Table 2-2), which we want to be as small as possible (we reject the null hypothesis). However, if the framework is twisted and we want to prove the null hypothesis (and reject the alternative), as is the case of Kupiec's test, we are not worried about the type I error but about the type II error.

Roughly speaking, the p-value can be seen as the probability of committing the type I error when rejecting the null hypothesis. As we already discussed, there is an inverse relationship between type I and type II errors. Thus, there is an inverse relationship between the p-value and type II error, which means that the higher the p-value, the lower the probability of committing type II error by accepting the null hypothesis.[14]

The drawbacks of the Kupiec test are twofold:

- it takes into account only the quantity of exceptions,
- the test has a poor power to distinguish accurate and inaccurate models.

The function taking the series of VaR exceptions and the expected probability of exceptions occurring (significance level) as input and returning the p-value of the Kupiec test is shown in Code 2-10. The implementation is straightforward, following equation (2.24). However, the cases in which n_0 or n_1 are equal to 0 (i.e. all or no exceptions respectively) must be taken care of as it is not possible to calculate the log(0).

[13] Typically we compare p-values to the levels like 5%, 1% or 0.1% and the lower the p-value the more significant the statistical inference is.
[14] Thus, in the case of Kupiec's test we assume the significance levels of 5%, 10% and 15%.

Code 2-10 Function for Kupiec's Unconditional Coverage Test

```
def kupiecTest(I, p):
    n = len(I)  # the length of backtested series
    n1 = sum(I) # no. violations
    n0 = n-n1   # no. nonviolations
    kupiec = -2 * n0 * np.log(1-p) - 2 * n1 * np.log(p)  # the first part of the formula
    if n0 > 0: # add the second part of the formula only if n0>0, otherwise log(0) would
cause an error
        kupiec = kupiec + 2 * n0 * np.log(n0/n)
    if n1 > 0: # add the third part of the formula only if n1>0, otherwise log(0) would
cause an error
        kupiec = kupiec + 2* n1 * np.log(n1/n)
    pvalue = 1-chi2.cdf(kupiec, 1) # calculate the p-value, note that the null hypothesis is
that the observed and expected probability of violation occurring, i.e. we want the p-value
to be as high as possible
    return pvalue
```

2.4.2 Christoffersen's conditional coverage test

By contrast, to assess whether the exceptions are distributed equally in time, i.e. without any dependence (autocorrelation), we should first define the dependence of exceptions in time. In Christoffersen (1998), it is defined as the stage when an exception occurring on one day can significantly help to identify whether another exception will follow on a subsequent day. Therefore, we replace the original sequence with a new one, where sequences of 01, 00, 11, or 10 are recorded. The null hypothesis is that the probability of an exception occurring is independent of the information on whether the exception occurred also the previous day,

$$H_0: \pi_{01} = \pi_{11}, \tag{2.25}$$

where π_{01} is the probability of an exception occurring if an exception did not occur previously and π_{11} is the probability of an exception occurring if an exception did occur previously. Then, we can formulate the following likelihood ratio:

$$LR_{Christoffersen1} = -2\,ln\left[\frac{\pi_{obs}^{n_1}(1-\pi_{obs})^{n_0}}{\pi_{01}^{n_{01}}(1-\pi_{01})^{n_{00}}\pi_{11}^{n_{11}}(1-\pi_{11})^{n_{10}}}\right], \tag{2.26}$$

where $\pi_{ij} = Pr(I_t = j|I_{t-1} = i)$ and $\pi_{obs} = \frac{n_{01}+n_{11}}{n_{00}+n_{01}+n_{10}+n_{11}}$. This test statistic has an asymptotically χ^2 probability distribution with one degree of freedom.

Obviously, we can evaluate these two tests together by constructing the following likelihood ratio as proposed by Christoffersen (1998),

$$LR_{Christoffersen2} = LR_{Kupiec} + LR_{Christoffersen1}, \tag{2.27}$$

$$LR_{Christoffersen2} = -2\,ln\left[\frac{\pi_{ex}^{n_1}(1-\pi_{ex})^{n_0}}{\pi_{01}^{n_{01}}(1-\pi_{01})^{n_{00}}\pi_{11}^{n_{11}}(1-\pi_{11})^{n_{10}}}\right], \tag{2.28}$$

which has an asymptotically χ^2 probability distribution with two degrees of freedom.

2.4.3 Example of risk estimation backtesting for FAANG stock portfolio

Continuing the example in Code 2-5, assuming the FAANG portfolio, we backtest the above-mentioned risk estimation methods, see Code 2-11.

- The code initiates a backtesting framework for evaluating various Value at Risk (VaR) estimation methods (lines 1-4). It starts by defining the significance levels (*alphas*) and the lengths of the in-sample and out-of-sample periods (*m* and *n*).
- Two dataframes, *exceptions* and *pValueKupiec*, are prepared (lines 7-8) to store the number of VaR violations and the p-values from the Kupiec test, respectively.
- A grid of subplots (lines 11-12) is set up to visualize the true losses and VaR estimates across different significance levels.
- The core of the code is a cycle (lines 14-44) that iterates over each significance level. Within this cycle:
  - Relative VaR estimates are calculated using analytical, historical, and Monte Carlo simulation methods.
  - True losses for the out-of-sample period are computed as the negative of the portfolio return.
  - The number of exceptions (times the observed loss exceeded VaR) and the Kupiec test p-values are calculated for each method.
- Each subplot visualizes the true losses and VaR estimates (lines 33-44), with the x-axis displaying only the years and the y-axis starting from 0.
- The layout of the subplots is adjusted, and the figure with six subplots is displayed (lines 47-48), showcasing the performance of each VaR estimation method at different significance levels.
- The final output (lines 49-50) includes tables displaying the number of VaR violations and the Kupiec test p-values, providing a statistical basis for evaluating the reliability of each VaR estimation method.

The resulting figure is shown in Figure 2-3.

Code 2-11 Backetesting of VaR with Kupiec test (continuation of Code 2-5)

```
alphas = [0.001, 0.005, 0.01, 0.05, 0.10, 0.15]

m = 250 # the length of in-sample period
n = len(returns) - m # the lengt of backtesting period

# inizicalize the dataframes for storing the results
exceptions = pd.DataFrame(columns = alphas, index=['analytical', 'historical', 'Monte
Carlo', 'expected']) # prepare empty dataframe for results
pValueKupiec = pd.DataFrame(columns = alphas, index=['analytical', 'historical', 'Monte
Carlo']) # prepare empty dataframe for results

fig, axes = plt.subplots(nrows=3, ncols=2, figsize=(12.5, 17))  # Create a grid of 3x2
subplots
axes = axes.flatten()  # Flatten the 2D array of axes to 1D for easy iteration

for i, alpha in enumerate(alphas):
    VaRs = pd.DataFrame(index = returns.index[m:], columns=['analytical', 'historical',
'Monte Carlo', 'True loss']) # create an empty DataFrame
    for t in range(m, m+n):
        inSample = returns[t-m:t-1] # define in-sample period for VaR estimation
        outOfSampleReturns = returns[t:t+1] # define out-of-sample period for VaR estimate
verification
```

```
        VaRs.iloc[t-m,0], _ = calcAnalyticalMultiple(inSample, w, alpha) # calculate VaR
estimate via analytical method
        VaRs.iloc[t-m,1], _ = calcHistoricalMultiple(inSample, w, alpha) # calculate VaR
estimate via historical simulation method
        VaRs.iloc[t-m,2], _ = calcMCSMultiple_t(inSample, w, alpha, 4)  # calculate VaR
estimate via Monte Carlo simulation method assuming student t-distribution with 3 df
        VaRs.iloc[t-m,3] = -pd.DataFrame(data=outOfSampleReturns).dot(w).iloc[0,0] #
calculate true observed loss

        print(f"Cycle {t-m+1} out of {n} for alpha={alpha} done.")

    for column in VaRs.columns[0:-1]:
        I = VaRs['True loss'] > VaRs[column] # get the series of exceptions (VaR violations)
        exceptions[alpha][column] = sum(I) # get the number of exceptions (VaR violations)
        pValueKupiec[alpha][column] = kupiecTest(I, alpha) # calculate the p-value of Kupiec
test
    exceptions[alpha]['expected'] = alpha * n # calculate the expected number of exceptions

    ax = axes[i]
    VaRs['x'] = VaRs.index  # Add new column generated from the index to plot the figures
    VaRs.plot.scatter(x='x', y='True loss', marker='.', ax=ax)  # Plot the true losses on
the ith subplot
    VaRs.plot(y=['analytical', 'historical', 'Monte Carlo'], ax=ax)  # Add the VaR estimates
on the ith subplot
    #ax.set_xticklabels(VaRs['x'], rotation=90)  # Rotate the labels of x-axis
    ax.xaxis.set_major_locator(mdates.YearLocator())  # Set major ticks to be at the start
of each year
    ax.xaxis.set_major_formatter(mdates.DateFormatter('%Y'))  # Format major ticks to show
only the year
    ax.set_ylim(bottom=0)  # Set the lower limit of the y-axis to 0
    ax.set_title(f'VaRs for alpha={alpha}')  # Set the title of the ith subplot
    ax.set_ylabel('VaRs & True Loss')  # Set the label of y-axis
    ax.set_xlabel('Date')  # Set the label of x-axis
    ax.legend(loc="best")


plt.tight_layout()  # Adjust subplots to fit into the figure area
plt.show()  # Display the figure with 6 subplots>              0.001 0.005  0.010 0.050
0.100  0.150
print(exceptions)
print(pValueKupiec)
> analytical      10    20     24    71     98    133
historical       6     9     14    64    113    156
Monte Carlo      0     6      7    33     82    107
expected     1.008  5.04  10.08  50.4  100.8  151.2
                0.001     0.005     0.010     0.050     0.100     0.150
analytical        0.0       0.0  0.000183  0.004921  0.767843  0.102369
historical   0.000716  0.111531  0.241206  0.058753  0.208016  0.673327
Monte Carlo  0.155545  0.677319  0.302193  0.007437  0.042025  0.000045
```

The results indicate that there is not a universally superior estimation method that is statistically significant across all alpha levels. It is important to note that our analysis was limited to a few selected methods. For instance, the Monte Carlo simulation was conducted with a Student's t distribution, assuming only 4 degrees of freedom. Other values could have been chosen for this parameter. Additionally, the length of the in-sample period is a variable that can significantly influence the outcomes. Therefore, it is clear that we have backtested only a small subset of the potential models and methods available.

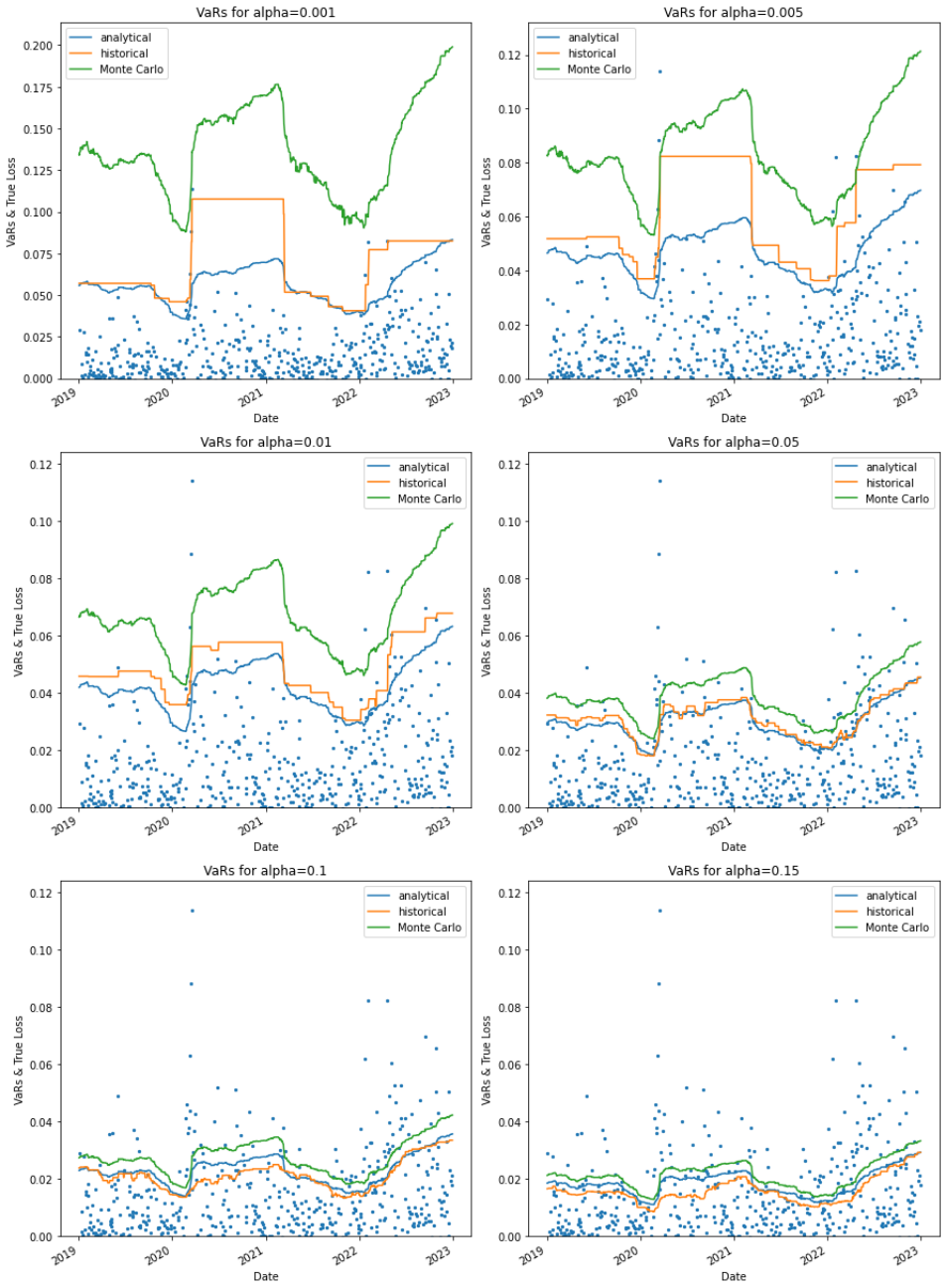

Figure 2-3 Backtested VaRs obtained by running Code 2-11

Question 2-8 VaR exceptions identification

Identify the VaR exceptions in Figure 2-3.
Are the exceptions randomly distributed in the period? If not, identify the periods in which the exceptions are clustered.

Question 2-9 Evolution of VaR estimates in historical simulation method

When we look at Figure 2-3, it is evident that the VaR estimates from the historical simulation method tend to remain constant over time, especially at lower alpha levels. In particular, the lower the alpha, the more prolonged the periods of stability (the "flat regions") in the VaR estimates. This stability is due to the method's reliance on historical data, where lower alphas focus on more extreme historical quantiles, leading to less fluctuation in the VaR estimates. However, another factor that can affect the duration of these flat regions is the look-back period, which is the amount of historical data considered in the simulation. A longer look-back period generally results in more extended flat regions, as the VaR estimates are based on a broader range of historical data, making them less sensitive to recent market movements. On the other hand, a shorter look-back period can make the VaR estimates more reactive to recent changes, potentially shortening the flat regions.
Plot the VaR estimates for different alphas and different look-back periods (m) and verify this information.

Chapter 3
Portfolio: Optimization and Performance Measurement

"*Investors' portfolios are simply their collections of investment assets. Once a portfolio is established, it is updated or 'rebalanced' by selling existing securities and using the proceeds to buy new securities, by investing additional funds to increase the overall size of the portfolio, or by selling securities to decrease the size of the portfolio*." (Bodie et al., 2024, p. 10)

The portfolio is thus a collection of assets. In this textbook, for simplicity, we assume only stocks as the assets. In the third chapter, we explore the intricacies of portfolio management, balancing theoretical foundations with practical applications. The chapter begins with wealth path calculation in Subchapter 3.1, unveiling the complexities of determining wealth trajectory over time.

Subchapter 3.2 delves into portfolio performance measurement, covering final wealth, compound average growth rate, risk measures, and performance ratios. Practical implementation of performance measurement in Python using the *Pyfolio* package is introduced in Section 3.2.4, offering hands-on insights.

The core of the chapter lies in Subchapter 3.3, where portfolio optimization takes center stage. Starting from the classical mean-variance optimization in Section 3.3.1, we progress to more complex strategies. Alternative approaches and Python packages for portfolio optimization are discussed in Sections 3.3.4 and 3.3.5.

In Subchapter 3.4, we introduce the Capital Asset Pricing Model (CAPM) and Fama-French factor models, enriching the understanding of the risk-return relationship. The chapter concludes with Subchapter 3.5, focusing on backtesting portfolio optimization models. It covers potential biases, a simple two-period backtest, and the rolling-window approach, offering examples of a robust methodology for evaluating portfolio strategies over time.

3.1 Wealth path calculation

From a quantitative finance perspective, a portfolio's composition is characterized by the weights of the assets it contains. These weights can be expressed in several ways:

- ***Quantities***, *v*, refer to the actual number of units or shares of each asset held in the portfolio.

- ***Absolute weights***, *u*, represent the monetary value invested in each asset. The sum of these absolute weights equals the total value of the portfolio, denoted as *W*.
- ***Relative weights***, *w*, are the proportional values of each asset in the portfolio, expressed as a percentage of the total portfolio value. They indicate the fraction of the total portfolio that each asset represents.

Typically, the portfolio construction process begins with an understanding of either the quantities of assets held, as demonstrated in Code 3-1, or with a predetermined investment amount and desired portfolio composition in terms of relative weights, as shown in Code 3-2. It is important to note that when we calculate the quantities of assets from the total investment and relative weights, the resulting figures may not always be whole numbers. This implies that such a portfolio might not be feasible to construct in the real world due to the indivisibility of certain assets, especially when dealing with stocks or other securities that cannot be purchased in fractional units.

Code 3-1 Portfolio definition by quantities

```
import yfinance as yf  # we import the package and alias it to yf
import pandas as pd

tickers = ["META",  # Facebook/Meta Platforms, Inc.
           "AAPL",  # Apple
           "AMZN",  # Amazon
           "NFLX",  # Netflix
           "GOOGL"  # Google (Alphabet Inc.)
          ]
prices = yf.download(tickers, start="2013-01-02", end="2013-01-03", interval = "1d",
group_by="column", auto_adjust = True,)["Close"]


# defining portfolio by v
v = pd.DataFrame(data=[1, 1, 1, 1, 1], index=tickers, columns=['weight']) # quantitites held
u = v * prices.iloc[-1:].transpose().values  # weights in absolute value (in $)
W = u.sum()  # portfolio value
w = u /W # relative weights (in %)

print(f'Portfolio value(W): {W[0]:.2f}')
print(f'Portfolio quantities(v): {v}')
print(f'Portfolio absolute weights(u): {u}')
print(f'Portfolio relative weights(w): {w}')
> Portfolio value(W): 88.90
Portfolio quantities(v):        weight
META        1
AAPL        1
AMZN        1
NFLX        1
GOOGL       1
Portfolio absolute weights(u):           weight
META   16.791189
AAPL   12.865500
AMZN   18.099348
NFLX   28.000000
GOOGL  13.144286
Portfolio relative weights(w):          weight
META   0.188877
AAPL   0.144718
AMZN   0.203591
NFLX   0.314959
GOOGL  0.147854
```

Code 3-2 Portfolio definition by its value and relative weights

```
# defining portfolio by W and w
W = 1000
w = pd.DataFrame(data=1/5, index=tickers, columns=['weight']) # relative weights (in %)
u = w.multiply(W)  # weights in absolute value (in $)
v = u.div(prices.iloc[-1:].values.transpose())

print(f'Portfolio value(W): {W}')
print(f'Portfolio quantities(v): {v}')
print(f'Portfolio absolute weights(u): {u}')
print(f'Portfolio relative weights(w): {w}')
> Portfolio value(W): 1000
Portfolio quantities(v):           weight
META   11.911009
AAPL   15.545450
AMZN   11.050122
NFLX    7.142857
GOOGL  15.215737
Portfolio absolute weights(u):        weight
META    200.0
AAPL    200.0
AMZN    200.0
NFLX    200.0
GOOGL   200.0
Portfolio relative weights(w):        weight
META      0.2
AAPL      0.2
AMZN      0.2
NFLX      0.2
GOOGL     0.2
```

Once a portfolio is established, a key area of interest is tracking its performance over time. With access to historical price or return data, it is possible to compute the portfolio's historical trajectory. This involves calculating the ***wealth path***, which charts the portfolio's value as it evolves through time. Such an analysis provides valuable insights.

However, it is important to stress that, although the number of stocks held v_i stays the same, the relative weight w_i and absolute weight u_i change period to period (day by day) due to the returns. To deal with the situation, we have four options for what and how we want to model:

- In a basic analysis, we might assume a fixed relative weight, w_i. This implies that during each period, for instance, every day, we would rebalance the portfolio to ensure that the relative weights remain unchanged. However, it is crucial to recognize that this method would be expensive in reality. Rebalancing daily would result in numerous trades, incurring significant costs. Practical applications should account for these potential costs.
- We do not assume rebalancing, i.e. we fix the portfolio on the initial day. To address this challenge, we have three potential strategies:
  - First, work directly with v instead of relative weights w, as shown in Code 3-5.
  - Second, utilize the benefits of log returns, which can be aggregated over time. Alternatively, we can correctly aggregate simple returns. This approach is detailed in Section 1.2.3. Under this framework, we must work with the time series of cumulated returns from the initial period, see Code 3-6.
  - Third, for each period, recalculate the weight vector w based on the returns. This process is described in Code 3-7.

It is vital to note that all three methods should yield the same path for portfolio wealth.

The wealth path of a portfolio can be depicted in two distinct manners. The first approach involves plotting the ***absolute value*** of the portfolio in monetary units over a period of time. Alternatively, the wealth path can be represented relatively, as a function of each dollar of initial investment. This ***relative wealth path*** is calculated by dividing the absolute wealth path by the initial portfolio value. Essentially, this method tracks the evolution of one unit of currency invested in the portfolio, effectively representing the cumulative simple gross return.

To better understand these strategies, let us consider a straightforward example of a 5-stock portfolio. We use the FAANG portfolio as a reference, which comprises Meta Platforms (formerly Facebook), Apple, Amazon, Netflix, and Google. For this example, our portfolio consists of one stock from each of these companies.

In Code 3-3 we do some initial calculations. The code begins by importing essential packages for financial data analysis and visualization, including *Yfinance* for fetching stock data, *Pandas* for data manipulation, *Numpy* for numerical operations, and *Matplotlib* for plotting. It then specifies a list of ticker symbols representing five major tech companies. Using the *Yfinance* package, daily stock data for these companies is downloaded for the period from January 1, 2013, to January 1, 2023. From this dataset, only the adjusted closing prices are extracted and stored.

To establish an investment strategy, we compose a portfolio of one unit of each stock. We calculate the initial portfolio value and the corresponding initial weights based on the stock prices on the first day, the approach demonstrated in Code 3-3. The code then calculates simple daily returns, which represent the percentage change in stock prices day by day. Additionally, it computes logarithmic returns. Cumulative gross simple returns are also determined, both through the accumulation of log returns, and as the cumulative product of gross simple returns.[15] The results, including the initial weights and the initial portfolio value, are then printed for review. As can be seen, the initial portfolio value would be $88.9 and the portfolio composition would be as follows: 18.89% in META, 14.47% in AMZN, 20.36% in GOOGL, 31.50% in META and 14.79% in NFLX.

Code 3-3 The initial calculations

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

# Download data
tickers = ["META",  # Facebook/Meta Platforms, Inc.
           "AAPL",  # Apple
           "AMZN",  # Amazon
           "NFLX",  # Netflix
           "GOOGL"  # Google (Alphabet Inc.)
          ]

data = yf.download(tickers=tickers, start='2013-01-01', end='2022-12-31', interval='1d',
group_by='column', auto_adjust=True)
prices = data["Close"].dropna()
```

[15] Note that the second approach overwrites the values calculated by the first approach, however, the values are the same. The code is intended only to demonstrate the possible calculations.

```
# Initial weights and portfolio value
v = pd.DataFrame(index=prices.columns, columns=['weight'], data=1)
initial_portfolio_value = prices.iloc[0].dot(v).iloc[0]
initial_w = prices.iloc[0].multiply(v['weight']) / initial_portfolio_value
# Calcualation of returns time series
simple_returns = prices.pct_change().fillna(0)
log_returns = np.log(prices / prices.shift(1)).fillna(0)

cumulative_log_returns = log_returns.cumsum()
cumulative_gross_simple_returns = np.exp(cumulative_log_returns)
# alternatively we calculate from simple returns
cumulative_gross_simple_returns = (simple_returns + 1).cumprod()

print(initial_w)
print(initial_portfolio_value)
> [**********100%%**********]  5 of 5 completed
AAPL     0.188877
AMZN     0.144718
GOOGL    0.203591
META     0.314960
NFLX     0.147854
dtype: float64
88.90031623840332
```

In Code 3-4, we demonstrate the computation of both relative and absolute wealth paths by applying daily rebalancing. In Code 3-5, the absolute wealth path without rebalancing is calculated as the product of prices and quantities (the first approach). Furthermore, Code 3-6 illustrates how wealth paths can be calculated using cumulative gross returns (the second approach).

Code 3-4 Simple approach of day-to-day rebalancing (i.e. fixed relative weights)

```
# Wealth path via simple approach (rebalancing)
reb_weights = pd.DataFrame(index=simple_returns.index, columns=initial_w.index)
reb_weights[:] = initial_w.values
# vector multiplication returns matrix multiplied by vector of weights
reb_portfolio_return = simple_returns.dot(initial_w)
# alternatively element by element multiplication of returns matrix by weights matrix
reb_portfolio_return2 = simple_returns.multiply(reb_weights).sum(axis=1)
reb_wealth_path_relative = (1+reb_portfolio_return).cumprod()
reb_wealth_path_absolute = reb_wealth_path_relative * initial_portfolio_value
```

Code 3-5 Simple approach of using quantities instead of weights

```
# 1. Wealth path evolution
fir_wealth_path_absolute = prices.dot(v)
fir_wealth_path_relative = prices.dot(v) / initial_portfolio_value
```

Code 3-6 Utilizing the cumulative gross simple returns

```
# 2. Cumulative returns
sec_wealth_path_relative  = cumulative_gross_simple_returns.dot(initial_w)
sec_wealth_path_absolute  = cumulative_gross_simple_returns.dot(initial_w) *
initial_portfolio_value.sum()
```

Code 3-7 showcases the method for calculating the evolution of relative weights over time in a portfolio that is not subject to rebalancing. These dynamically changing relative weights can then be used directly to compute the relative wealth path using equation (1.20).

Code 3-7 Calculating the day-to-day weights

```
# 3. New weights matrix
# Correct weights calculation
weights = pd.DataFrame(index=simple_returns.index, columns=initial_w.index)
weights.iloc[0] = initial_w # Set the initial weights for the first date
# Iterate over each date and recalculate the weights
for i in range(1, len(simple_returns)):
    weights.iloc[i] = weights.iloc[i-1] * (1 + simple_returns.iloc[i-1]) # Calculate the new
weights based on returns
    weights.iloc[i] /= weights.iloc[i].sum() # Normalize the weights so they sum up to 1
thi_portfolio_return = simple_returns.multiply(weights).sum(axis=1)
thi_wealth_path_relative = (1+thi_portfolio_return).cumprod()
thi_wealth_path_absolute = thi_wealth_path_relative * initial_portfolio_value
```

Code 3-8 is dedicated to visualizing the results. It specifically plots the progression of both the weights and the wealth paths over time, comparing scenarios with and without portfolio rebalancing. The graphical representations of these comparisons are presented in Figure 3-1 and Figure 3-2, respectively, providing a clear visual distinction between the two strategies.

Code 3-8 Plotting the results

```
# Plot the resuls

# Plotting reb_weights and weights
fig, axes = plt.subplots(2, 1, figsize=(12, 10), sharex=True)
reb_weights.plot.area(ax=axes[0], stacked=True, title="Rebalanced Weights Over Time")
axes[0].set_ylabel("Weights")
axes[0].legend(loc="upper right")
weights.plot.area(ax=axes[1], stacked=True, title="Weights Over Time (Without Rebalancing)")
axes[1].set_ylabel("Weights")
axes[1].legend(loc="upper right")
plt.tight_layout()
plt.show()


# Plotting relative and absolute wealth paths
fig, axes = plt.subplots(2, 1, figsize=(12, 10), sharex=True)
# Plot for relative wealth paths
axes[0].plot(reb_wealth_path_relative, label="Rebalanced")
axes[0].plot(fir_wealth_path_relative, label="Quantities")
axes[0].plot(sec_wealth_path_relative, label="Cumulative Returns")
axes[0].plot(thi_wealth_path_relative, label="Dynamic Weights")
axes[0].set_title("Relative Wealth Paths Over Time")
axes[0].set_ylabel("Relative Wealth")
axes[0].legend(loc="upper left")

# Plot for absolute wealth paths
axes[1].plot(reb_wealth_path_absolute, label="Rebalanced")
axes[1].plot(fir_wealth_path_absolute, label="Quantities")
axes[1].plot(sec_wealth_path_absolute, label="Cumulative Returns")
axes[1].plot(thi_wealth_path_absolute, label="Dynamic Weights")
axes[1].set_title("Absolute Wealth Paths Over Time")
axes[1].set_ylabel("Absolute Wealth (in USD)")
axes[1].legend(loc="upper left")
plt.tight_layout()
plt.show()
```

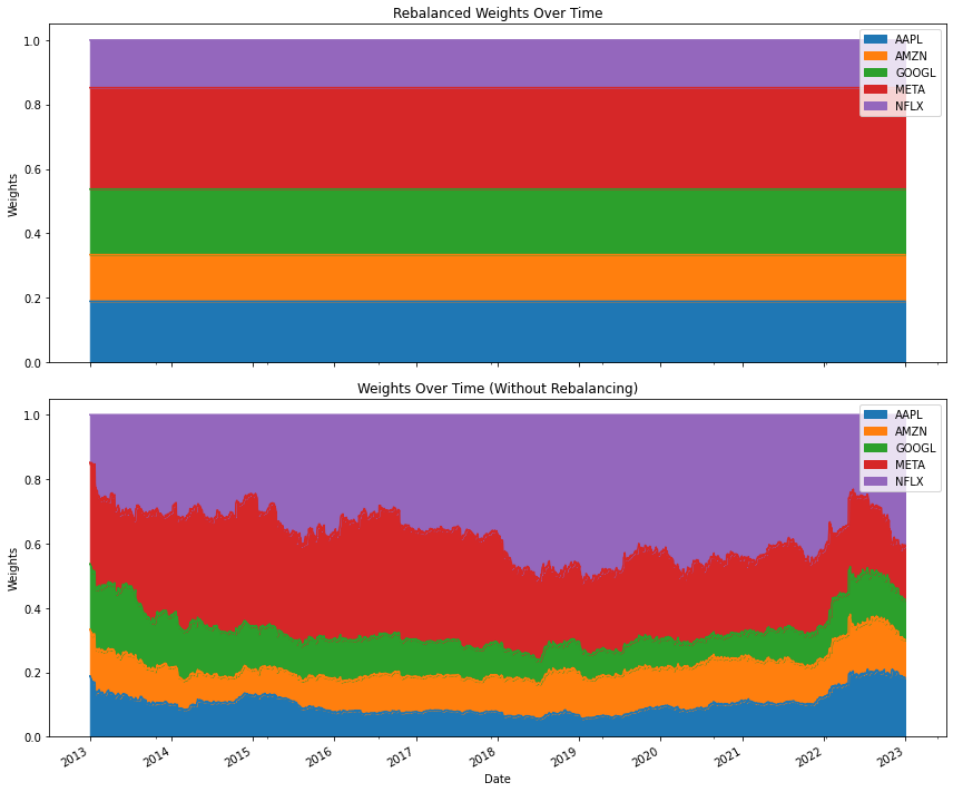

Figure 3-1 FAANG weights evolution in time

Note: With rebalancing (top) vs. without rebalancing (bottom).

Figure 3-1 clearly illustrates the impact of rebalancing on portfolio weights. In the rebalancing scenario, the weights remain constant throughout the entire period, reflecting the periodic adjustment to maintain the original asset allocation. Conversely, in the absence of rebalancing, the relative and absolute weights change in response to changes in asset prices. In particular, Netflix exhibits the most significant increase in relative weights. The reason is that its prices (and thus also absolute weights) increased over time more than the other four stocks, resulting in increased relative weights.

Furthermore, Figure 3-2 reveals that the overall trajectories of wealth paths are quite similar between the rebalanced and non-balanced portfolios. Furthermore, the overlapping lines in the figure indicate that the wealth paths calculated using different methods for the non-rebalanced portfolio result in the same values.

Figure 3-1 and Figure 3-2 might give the impression that the differences between rebalancing and not rebalancing a portfolio are minimal. However, this perception changes when considering assets with divergent price trajectories. This is exemplified in the portfolio comprising Tesla (TSLA) and Intel (INTC), as detailed in Code 3-9. The corresponding outcomes are depicted in Figure 3-3 and Figure 3-4.

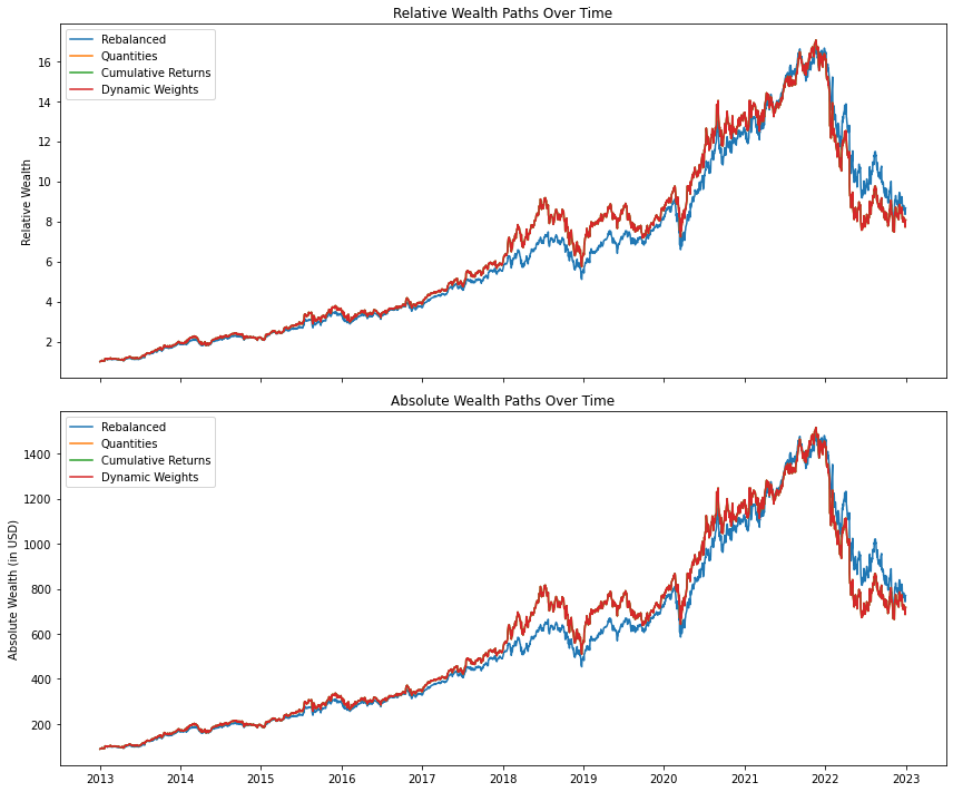

Figure 3-2 FAANG wealth evolutions in time: relative (top) vs. absolute (bottom)

During the period under review, Tesla's stock price experienced a significant rise, while Intel's declined. This divergence led to a substantial shift in the relative weights of the assets over time, as shown in Figure 3-3. Consequently, the wealth paths also differed markedly, as illustrated in Figure 3-4. Specifically, the rebalancing strategy effectively involved selling shares of the outperforming asset (Tesla) and purchasing more of the underperforming asset (Intel) to maintain constant relative weights. In contrast, the non-rebalancing approach involved holding the initially purchased quantities throughout the period without any further intervention.

Code 3-9 The changes in initial calculations

```
tickers = ["TSLA", # Tesla
           "INTC"  # Intel
          ]
data = yf.download(tickers=tickers, start='2013-01-01', end='2022-01-01', interval='1d',
group_by='column', auto_adjust=True)
prices = data["Close"].dropna()

# Initial weights and portfolio value
v = pd.DataFrame(index=prices.columns, columns=['weight'], data=[1,5])
> [*********************100%%**********************]  2 of 2 completed
INTC    0.568754
TSLA    0.431246
dtype: float64
27.33166766166687
```

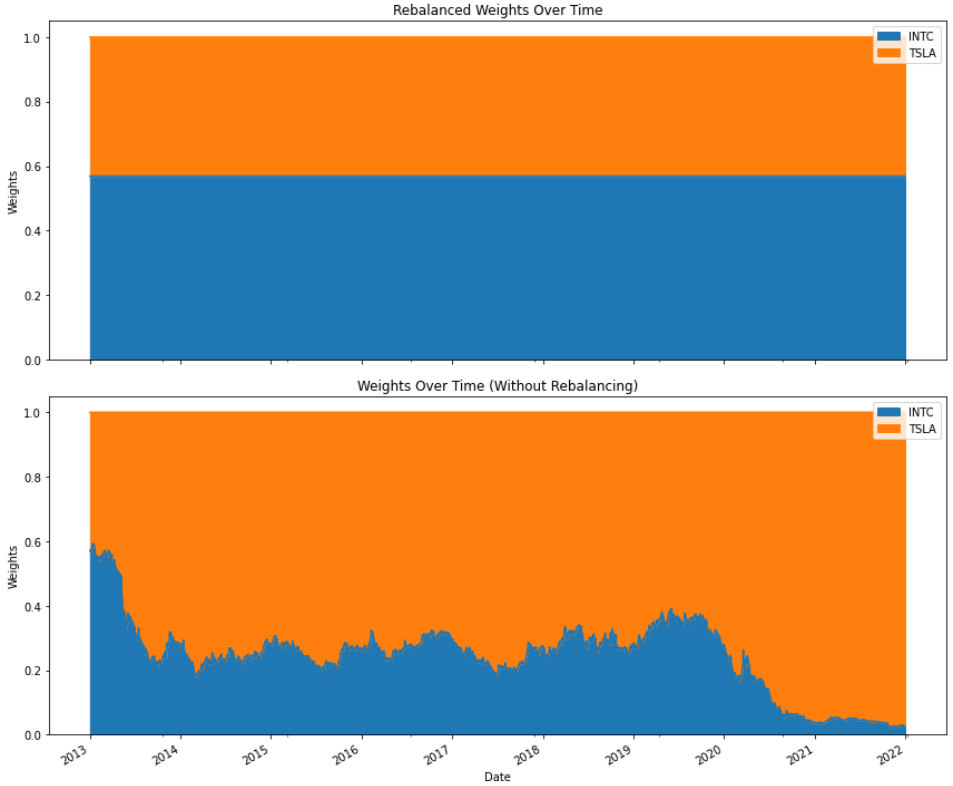

Figure 3-3 TSLA-INTC weight evolution in time

Note: With rebalancing (top) vs. without rebalancing (bottom).

Question 3-1 Is rebalancing possible?

When holding one stock of INTC and five stocks of TSLA is it possible to rebalance the portfolio to keep the weights fixed (56.9% and 43.1% respectively)? Would the possibility increase if the portfolio would be 1,000 INTC stocks and 5,000 TSLA stocks?

3.2 Portfolio performance measurements

Rachev et al. (2005, p. 318) stressed that "m*easuring a strategies performance is an ex-post analysis. The performance measure is calculated using the realized portfolio returns during a specified period back in time (i.e., the past one year). Alternatively, performance measures can be used in an ex-ante analysis, in which certain assumptions for the future behavior of the assets are introduced. In this case, the general goal is to find a portfolio with the best characteristics as calculated by the performance measure.*" As can be seen, the performance measures can be applied both to measure the historical performance of the strategies and as the optimization criterion in portfolio optimization, see Section 3.3.3.

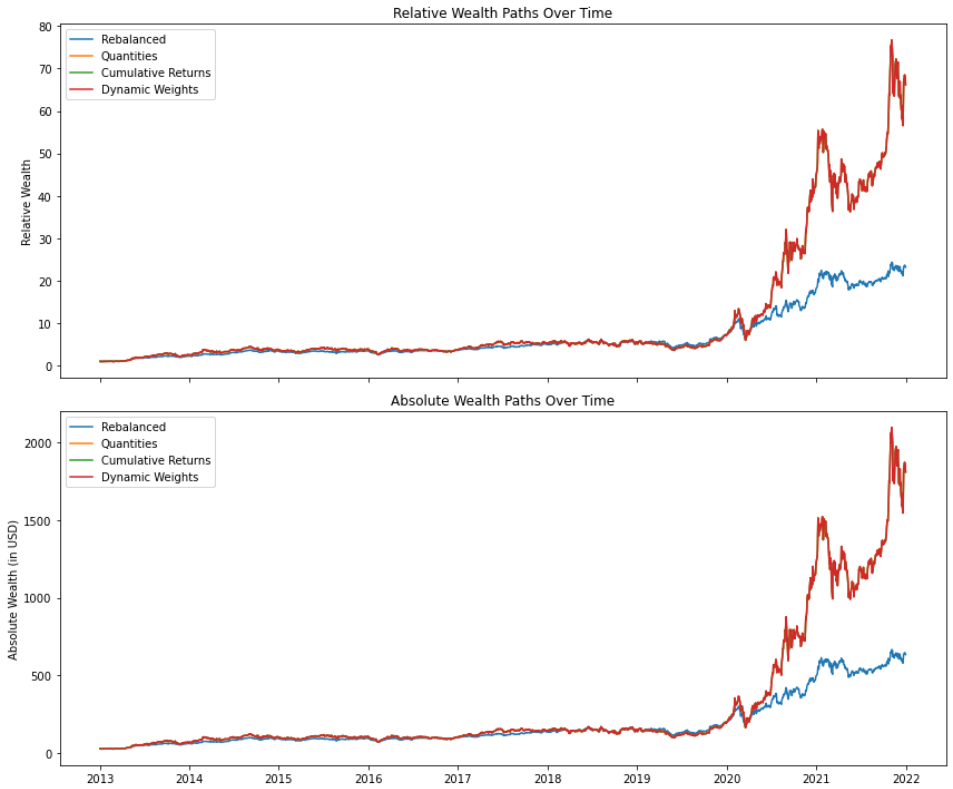

Figure 3-4 TSLA-INTC wealth evolution in time

Note: Relative (top) vs. absolute (bottom).

While we can consider the mean annual return as the performance measure of the portfolio investment or trading strategies, the proper way in the performance measurement is to consider not only the gains, but also the riskiness. Most of the performance measures are the ratios of average (or expected) return divided by the value of the chosen risk measure.

3.2.1 Final wealth and CAGR

The most straightforward and also commonly applied way to compare investment strategies is to compare the final wealth. The idea is very simple: we assume that we start with a wealth of 1 currency unit, and then, based on the portfolio returns, we can calculate the evolution of the wealth over time. The last value, that is, the wealth at the end of the analysis period, is the final wealth.

Another performance measure, which is, however, equivalent to the final wealth, is the compound average growth rate (CAGR), which is simply calculated as geometrical mean return, see Section 1.2.3. The rate should be stated on a per annum basis, preferably annual, to allow for a simple comparison with values reported in other studies.

The drawback of these measures is that they do not consider the risk at all. It is well known and discussed in Subchapter 3.3 that riskier portfolios deliver higher rates of return.

3.2.2 Risk measures

In Chapter 2, we explored two primary categories of risk measures: dispersion risk measures and safety-first risk measures. ***Dispersion risk measures*** evaluate the variability or spread of the portfolio return distribution, while ***safety-first risk measures*** are concerned solely with downside risk. In the following, we dive into some well-known dispersion risk measures, beginning with the standard deviation of portfolio return.

Standard deviation is a widely used measure that quantifies the extent to which the returns deviate from the mean (or expected return). It is mathematically represented as:

$$STD(R_p) = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(R_{p,i} - \overline{R_p}\right)^2} \tag{3.1}$$

$$STD(R_p) = \sqrt{E\left(R_p - E(R_p)\right)^2}, \tag{3.2}$$

where R_p is the portfolio return, $E(R_p)$ is the expected portfolio return, $\overline{R_p}$ is the average portfolio return, and *n* is the number of observations. This measure is a cornerstone of the Markowitz mean-variance framework (see Subchapter 3.3). However, a limitation of standard deviation is that it treats positive and negative deviations from the mean equally, although only negative returns are typically perceived as risk.

To address this, a downside deviation can be introduced. It is calculated similarly to standard deviation, but only considers values below a target return (often the mean return or zero) known as the Minimum Acceptable Return (MAR),

$$DD(R_p) = \sqrt{\frac{1}{n}\sum_{R_p<MAR}\left(R_{p,i} - MAR\right)^2}. \tag{3.3}$$

where *n* is the number of portfolio returns that fall below the MAR.

Another measure of dispersion is the Mean Absolute Deviation (MAD), which is defined as follows,

$$MAD(R_p) = \frac{1}{n}\sum_{i=1}^{n}\left(R_{p,i} - \overline{R_p}\right), \tag{3.4}$$

$$MAD(R_p) = E\left(\left|R_p - E(R_p)\right|\right). \tag{3.5}$$

MAD measures the average absolute deviation from the mean portfolio return. Compared to standard deviation, it is more robust to outliers in the return distribution.

Another example of a dispersion measure is beta in the CAPM model, which measures systematic risk. This measure is introduced in Section 3.4.1.

The second category of risk measures focuses on safety first principles. Among the most prominent measures in this category are Value at Risk (VaR) and Conditional Value at Risk (CVaR), which were introduced in Subchapter 2.2. Another key measure within this category is the Maximum Drawdown.

Maximum Drawdown (MDD) represents the maximum percentage decrease in portfolio value over time. It can be determined from the evolution of the portfolio value W as follows:

$$MDD = \max_{t\in(0,T)} \left(1 - \frac{W_t}{\max_{\tau\in(0,t)} W_\tau}\right). \tag{3.6}$$

This measure is particularly favored among practicing traders as it quantifies the maximum percentage amount the portfolio has lost from its peak value, providing a clear picture of potential losses during investment periods. For the application, see Code 3-10.

Code 3-10 Function calculating MDD

```
def calc_MDD(prices):
    # the function calculates and returns the MDD
    # the function expect that there is only one column in the DataFrame prices or prices
are pandas Series
    df = pd.DataFrame(prices.copy()) # make a copy of input data
    if (df.shape[1]!=1): # check the input data - whether there is only one column
        raise TypeError("Wrong format of the input data") # if there is not only one column,
raise an error and stop the program
    df.rename(columns={df.columns[0]: "prices" }, inplace = True) # rename the first to
column to "price" no matter what the name of the column is
    mdd = (-(df['prices']/df['prices'].cummax() - 1)).max() # calculate MDD in one line
    return mdd
```

Although the absolute value of maximum drawdown can also be calculated,

$$MDD_{abs} = \max_{t\in(0,T)} \left(\max_{\tau\in(0,t)} W_\tau - W_t\right), \tag{3.7}$$

it does not facilitate an easy comparison between different portfolio strategies. Furthermore, its significance can vary depending on whether the drawdown occurred early on, when the portfolio value is lower, or later after the portfolio value has increased.

Bacon (2012, p. 97) argued that "*perhaps the simplest measure of risk in a return series from an absolute return investor's perspective, wishing to avoid losses, is any continuous losing return period, known as a drawdown.*" Drawdowns are depicted in Figure 3-5, where their absolute values are presented. In the figure, D_1, D_2, and D_3 represent examples of individual drawdowns, i.e., an uninterrupted series of losses in a return series. It's crucial to differentiate between the maximum drawdown and the largest individual drawdown. The maximum drawdown signifies the greatest loss from the peak to the subsequent lowest value within the observed period, and its calculation is detailed in the equations above. Alongside the drawdowns, the figure also clearly delineates other important metrics, such as recovery time. Recovery time refers to the duration required to return from a drawdown to its original peak level. This metric can be measured from either the peak or the trough, see Figure 3-5.

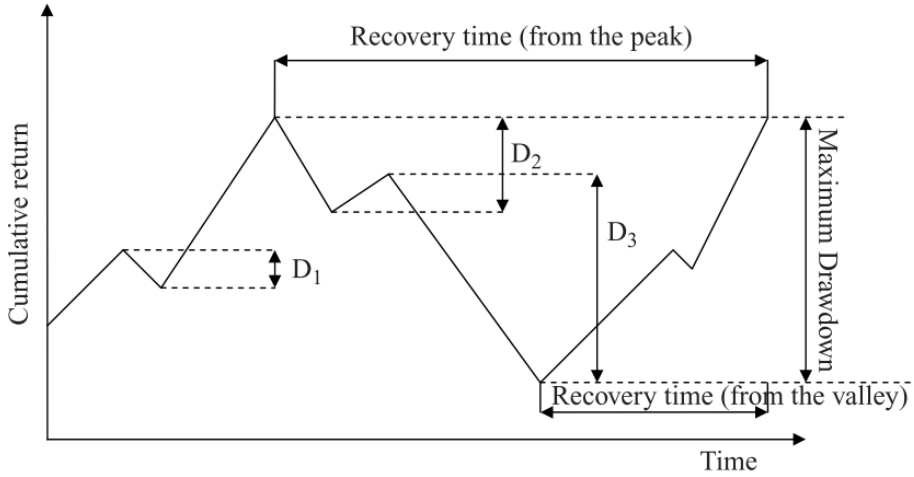

Figure 3-5 Different individual drawdowns in the relative wealth path
Source: Bacon (2012, p. 99)

3.2.3 Performance ratios

As will be discussed in Subchapter 3.3, riskier portfolios tend to deliver higher rates of return. Therefore, when evaluating portfolio performance, it is essential to consider not only the rate of return but also the level of risk involved. In other words, a proper comparison of portfolio performance should involve the evaluation of the risk-adjusted return. This is often measured using portfolio performance ratios, which typically represent the ratio of the portfolio's benefit, such as the excess return over the risk-free rate, to a chosen risk measure. These ratios may vary according to the specific risk measure applied. In the following, we review some well-known performance ratios.

Sharpe ratio

The most well-known performance measure is the Sharpe ratio, closely linked to the mean-variance concept as it considers only the mean and standard deviation of the return distribution,

$$SR(R_p) = \frac{E(R_p - R_f)}{\sigma(R_p)}. \tag{3.8}$$

Originally proposed by Sharpe (1966) as the reward-to-variability ratio to gauge mutual fund managers' performance, it underwent a revision (Sharpe, 1994) in which the risk-free return R_f was replaced with a benchmark return, $R_{benchmark}$,

$$SR(R_p) = \frac{E(R_p - R_{benchmark})}{\sigma(R_p)}. \tag{3.9}$$

The benchmark could be, for instance, the return of a buy-and-hold strategy. In portfolio theory, a market index often serves as the benchmark that portfolio managers are seeking to replicate or even beat. Special cases of the benchmark include the risk-free return ($R_{benchmark} = R_f$) or a zero return ($R_{benchmark} = 0$). The Sharpe ratio defines risk through the standard deviation, favoring strategies with the highest expected return per

unit of this deviation. When comparing two strategies, the one with the higher Sharpe ratio is better, indicating a higher expected return for the same return volatility or lower volatility for the same expected return.

As Carver (2015) emphasized, it is crucial to annualize returns and standard deviations for comparability reasons. For example, when calculating the Sharpe ratio using daily returns, it must be multiplied by the square root of 252 or by 16 as a rule of thumb (Carver, 2015). Depending on a particular area of application, the Sharpe ratio of around 0.5-1 is a good one. The example of Sharpe ratio calculation is shown in Code 3-11.

Code 3-11 Function calculating Sharpe ratio

```
def calc_Sharpe(prices, rfr=0):
    # the function calculates and returns the Sharpe ratio for the price evolution of the
stock/portfolio given as the argument
    # the function expects that there is only one column in the variable prices
    # rfr is the value of the risk-free rate to consider (p.a.)
    df = pd.DataFrame(prices.copy()) # make a copy of input data
    if (df.shape[1]!=1): # check the input data - whether there is only one column
        raise TypeError("Wrong format of the input data") # if there is not only one column,
raise an error and stop the program
    df.rename(columns={df.columns[0]: "prices" }, inplace = True) # rename the first column
to "price" no matter what the name of the column is
    df['returns'] = df["prices"].pct_change().dropna() # calculate simple returns
    mr = df['returns'].mean() * 252 # very simple annualization, please refer to CAGR
calculation
    std = df['returns'].std() * np.sqrt(252) # very simple annualization,
    sharpe = (mr - rfr) / std # calculate Sharpe ratio
    return sharpe
```

Question 3-2 Sharpe ratio annualization

Explain why the Sharpe ratio, calculated from the daily mean return and daily standard deviation, should be multiplied by a factor of 16.

MAD ratio

The Mean Absolute Deviation ratio, *MADR*, proposed by Konno and Yamazaki (1991), considers the mean absolute deviation from the expected return as a measure of risk. The formula for the MAD ratio is as follows:

$$MADR(R_p) = \frac{E(R_p - R_{benchmark})}{MAD(R_p)}, \tag{3.10}$$

where $MAD(R_p)$ is the mean absolute deviation as defined in Section 3.2.2.

Sortino and Sortino-Satchell ratios

The Sharpe ratio, while useful, has a limitation: It penalizes both the upside and downside variability of portfolio returns. However, research by Kahneman and Tversky (1979) suggests that investors react differently to gains and losses. To address this, the ***Sortino Ratio*** (Sortino & Price, 1994), *SOR*, was developed, which considers only downside deviation as a measure of variability. The formula for the Sortino ratio is as follows:

$$SOR(R_p) = \frac{E(R_p - MAR)}{DD(R_p)}, \tag{3.11}$$

where $DD(R_p)$ is the downside deviation of the portfolio return, see Section 3.2.2. Unlike the Sharpe Ratio, the Sortino Ratio focuses solely on returns that fall below the minimum acceptable return and ignores any upside deviation. A higher Sortino ratio indicates better portfolio performance when adjusted for downside risk.

The ***Sortino-Satchell*** ratio, *SSR*, is an extension of the Sortino ratio that allows for a more flexible assessment of downside risk. It is defined as follows:

$$SSR(R_p, u) = \frac{E(R_p - MAR)}{\left\{E\left[(MAR - R_p)_+^u\right]\right\}^{1/u}}, \tag{3.12}$$

where *u* represents the order of the lower partial moment, which determines the degree to which the calculation focuses on the downside risk. When *u*=2, the formula simplifies to the standard Sortino ratio, focusing on the downside deviations. By adjusting the value of *u*, investors can tailor the sensitivity of the SSR ratio to different levels of downside risk, making it a versatile tool for risk-adjusted performance evaluation.

Jensen alpha

Jensen's alpha was first used as a measure in the evaluation of mutual fund management by Michael Jensen in 1968. It measures the excess return of a portfolio over the theoretical expected return which would be predicted by CAPM, i.e., it measures the overperformance related to the optimal portfolio with the same systematic risk measured by the portfolio beta, see Subchapter 3.4. The calculation is as follows,

$$\alpha_{JA} = R_p - \left[R_f + \beta_p \cdot \left(R_m - R_f\right)\right], \tag{3.13}$$

where portfolio beta, β_p, can be calculated as the weighted average of assets' betas, or estimated via OLS regression, see Section 3.4.1. If the value is positive, the portfolio historically delivered a better return than would correspond to its systematic risk. On the other hand, if the value is negative, it means that the portfolio underperformed. This measure can be applied for historical ex-post evaluation but is not applicable as the optimization criterion in ex-ante analysis.

Treynor ratio

The Treynor ratio, *TR*, developed by Jack Treynor, is a measure of risk-adjusted performance that relates the excess return of a portfolio to its systematic risk, represented by the portfolio's beta β_p. It is defined as follows:

$$TR(R_p) = \frac{E(R_p - R_f)}{\beta_p}. \tag{3.14}$$

In contrast to the Sharpe ratio, which considers total risk, the Treynor ratio focuses solely on systematic risk, making it particularly useful for portfolios that are expected to be well-diversified. Similarly to the Sharpe ratio, a higher value of the *TR* means that the given portfolio has better risk-adjusted performance.

A potential issue arises when a portfolio's beta is negative, as the Treynor ratio is based on the assumption of positive beta values. When β_p is negative, the interpretation of the Treynor ratio becomes problematic. A negative β_p would invert the ratio, potentially making a well-performing portfolio appear to have a negative value of Treynor ratio. This is clearly counterintuitive and suggests that the Treynor ratio is not suitable for portfolios

with negative values of beta. In such cases, it might indicate that the chosen benchmark market return is inappropriate for the portfolio in question or that the portfolio's investment strategy is fundamentally different from the market's behavior.

Farinelli-Tibiletti ratio

The Farinelli-Tibiletti (FT) ratio, proposed by Farinelli and Tibiletti (2003), is a performance metric that uses partial moments of different orders to assess a portfolio's risk-adjusted return. Unlike traditional measures that use the expected value of portfolio returns, the FT ratio evaluates the portfolio's reward using an upper partial moment.[16] The formula is as follows:

$$FT(R_p, \gamma, \delta) = \frac{\left\{E\left[(R_p - MAR)_+^{\gamma}\right]\right\}^{1/\gamma}}{\left\{E\left[(MAR - R_p)_+^{\delta}\right]\right\}^{1/\delta}}, \tag{3.15}$$

were $\gamma \geq 1$ and $\delta \geq 1$ represent the orders of the partial moments, reflecting an investor's attitude towards gains (outperformance) and losses (underperformance).

STARR and Rachev ratios

The STARR ratio is a performance measure that can be considered a specific case of the more general Rachev ratio. It resembles the Sharpe ratio but replaces the standard deviation with the Conditional Value at Risk (CVaR) as the risk measure:

$$STARR(R_{p,}\alpha) = \frac{E(R_p - MAR)}{CVaR_{\alpha}(R_p - MAR)}, \tag{3.16}$$

where *MAR* could represent the risk-free rate, and $CVaR_{\alpha}$ is a measure of downside risk.

The Rachev ratio (RR), proposed by Rachev et al. (2005), is a performance ratio that utilizes a reward measure instead of the expected value of portfolio returns. It is defined as follows:

$$RR(R_p, \alpha, \beta) = \frac{CVaR_{\beta}(MAR - R_p)}{CVaR_{\alpha}(R_p - MAR)}. \tag{3.17}$$

In this formula, α is the probability level of the lower tail and β is the probability level of the upper tail of the portfolio return distribution. The numerator represents the average of αtail corresponding to outperformance over a benchmark (i.e., MAR), while the denominator represents the average of βtail corresponding to underperformance under the same benchmark. The selection of α and β, often set at 5% in empirical studies, reflects investors' risk preferences.

The Rachev ratio is a reward-to-risk ratio applicable in both ex-ante and ex-post analysis, optimizing for the Rachev ratio can be numerically challenging due to the convex nature of the two CVaR functions in the ratio, which may lead to local extremes in their ratio.

[16] Compare the Farinelli-Tibiletti Ratio to the Sortino-Satchell ratio.

Calmar ratio

The Calmar ratio (CR) is named after the California Managed Annual Reports. Unlike the Sharpe ratio, which uses standard deviation as a measure of risk, the Calmar ratio employs the maximum drawdown. The formula for the Calmar ratio is as follows:

$$CR(R_p, T) = \frac{E(R_p - R_f)}{MDD(T)}, \tag{3.18}$$

where *MDD(T)* denotes the maximum drawdown over the observed period *T*, which is commonly set to three years in empirical studies. The example is shown in Code 3-12.

Code 3-12 Function to calculate the Calmar ratio

```
def calc_Calmar(prices, rfr=0):
    # the function calculates and returns the Calmar ratio for the price evolution of the
stock/portfolio given as the argument
    # the function expects that there is only one column in the variable prices
    # rfr is the value of the risk-free rate to consider (p.a.)
    df = pd.DataFrame(prices.copy()) # make a copy of input data
    if (df.shape[1]!=1): # check the input data - whether there is only one column
        raise TypeError("Wrong format of the input data") # if there is not only one column,
raise an error and stop the program
    df.rename(columns={df.columns[0]: "prices" }, inplace = True) # rename the first to
column to "price" no matter what the name of the column is
    df['returns'] = df["prices"].pct_change().dropna() # calculate simple returns
    df['returns cumulative gross'] = (1 + df['returns']).cumprod() # calculate the
cummulative returns
    mdd = (-(df['returns cumulative gross']/df['returns cumulative gross'].cummax() -
1)).max() # calculate MDD in one line
    cagr = df['returns cumulative gross'].iloc[-1]**(252/df.shape[0])-1 # calculate
Cumulative Annual Growth Rate (CAGR) - the (geometrical) mean annual return
    calmar = (cagr - rfr) / mdd # calculate calmar ratio
    return calmar
```

3.2.4 Performance measurement in Python: *Pyfolio*

Pyfolio (Quantopian Inc., 2019) is a powerful and versatile Python package designed for the performance and risk analysis of financial portfolios. Developed by Quantopian, it facilitates the evaluation of investment strategies and portfolio management by offering comprehensive tools for analyzing performance metrics, risk factors, and even transactional details. *Pyfolio* is particularly valuable for assessing the robustness of trading strategies and gaining insight into their behavior under various market conditions.

The drawback of the original package is that it is no longer updated and the last version is from April, 2019. In this textbook, we use the fork[17] *Pyportfolio-reloaded* maintained by Jansen (2024).

In Code 3-13, we show the simple snippet code analyzing the returns of INTC over the period from 2012 to 2022, however, any other data, especially those calculated in Subchpater 3.1, can be inputted. The graphical output is shown in Figure 3-6. As can be seen, the *Pyfolio* package provides a detailed graphical analysis of the performance, which is best viewed in Jupyter-Notebook or Jupyter-Lab.

[17] Fork refers to a copy of a repository – a central place where the package is stored.

Figure 3-6 The output of the Code 3-13 (rearranged to fit the page)

Code 3-13 Example of *Pyfolio-reloaded* usage

```
import pyfolio as pf
import yfinance as yf

prices = yf.download(tickers = 'INTC', start='2012-01-01', end='2023-01-01', interval =
'1d', group_by = 'column', auto_adjust = True)
returns = prices['Close'].pct_change().dropna() #calculate returns

# calculate the statistics of the wealth path
print(pf.timeseries.perf_stats(returns))

pf.create_returns_tear_sheet(returns)
#pf.create_simple_tear_sheet(returns)
#pf.create_full_tear_sheet(returns)

> Annual return           0.037789
Cumulative returns      0.502722
Annual volatility       0.300952
Sharpe ratio            0.274484
Calmar ratio            0.061319
Stability               0.745077
```

```
Max drawdown           -0.616262
Omega ratio             1.053516
Sortino ratio           0.385895
Skew                   -0.216072
Kurtosis               13.563240
Tail ratio              0.989255
Daily value at risk    -0.037589
dtype: float64
```

3.3 Portfolio optimization

The portfolio optimization problem addresses the fundamental question of how to construct a portfolio, i.e., determine which assets to include and what weights to assign them. In portfolio optimization, two conflicting goals arise: maximizing the benefit, typically represented by the expected return of the portfolio, while simultaneously minimizing the associated weakness, usually defined as risk.

The pioneering work in portfolio theory is the mean-variance framework, discussed in Section 3.3.1, which can be readily extended to a more general mean-risk framework, explored in Section 3.3.2. Investors may also choose to directly maximize performance ratios, as detailed in Section 3.3.3, or employ alternative approaches outlined in Section 3.3.4.

3.3.1 Mean-variance optimization framework

Markowitz (1952) laid the foundations of modern portfolio theory, which was later recognized with a Nobel Prize in Economics in 1990. Markowitz's model, known as the mean-variance model, is based on two parameters of the portfolio return distribution: the expected (mean) return and its variance.

Like all models, the mean-variance model is based on several assumptions that simplify the reality. The main characteristics of the model include:

- It is the single-period model, i.e. the model is static, considering an investment horizon where the portfolio's structure remains unchanged.
- The risk of the portfolio is represented by the variance (or standard deviation) of the return probability distribution.
- The investor is rational and risk-averse, seeking the highest expected return for the lowest possible risk (variance of returns).
- Markets are efficient, all investors have costless access to market information, and prices respond instantly to new public information.
- There are no frictions, i.e. there are no transaction costs, taxes, and restrictions on short selling.
- There is infinite divisibility of assets, meaning that the funds available to an investor can be allocated among assets in any proportion.
- The distribution of the returns is multivariate normal or the investor’s utility function is quadratic.

These assumptions are foundational to the mean-variance framework, which seeks to optimize portfolio selection based on the trade-off between risk and return.

For the portfolio composed of *n* assets, the portfolio's expected return $E(R_p)$ and the variance of the portfolio return σ_p^2 can be determined as follows:

$$E(R_p) = E(R)^T \cdot x = \sum_{i=1}^{n} E(R)_i \cdot x_i, \tag{3.19}$$

$$\sigma_p^2 = x^T \cdot \boldsymbol{Q} \cdot x = \sum_{i=1}^{n} \sum_{j=1}^{n} x_i \cdot \sigma_{i,j} \cdot x_j, \tag{3.20}$$

where $E(R)$ is the vector of assets' expected returns, $\boldsymbol{Q}$ is the covariance matrix and *x* is the vector of relative weights.

Let us first think about the set of all portfolios that an investor can construct, known as the ***feasible set***. An example of this set is depicted in Figure 3-7. The light gray area in the graph represents all feasible portfolios, all combinations of expected return and standard deviation (square root of variance) that an investor can achieve by composing a portfolio. However, not all portfolios are efficient. Consider *Portfolio A*. As seen in Figure 3-7, this portfolio is not efficient for the investor because,

- for the same expected return, a lower standard deviation of the return can be achieved (*Portfolio B*),
- for the same standard deviation, a higher expected return can be achieved (*Portfolio C*),
- in fact, it is even possible to achieve both a lower standard deviation and a higher expected return (*Portfolio D*).

Since the investor is rational, maximizing expected return and minimizing variance, *Portfolio A* would not be efficient for him.

On the other hand, for *Portfolios B, C,* and *D*, one parameter cannot be improved without worsening the other. For example, *Portfolio C* has a higher expected return than *Portfolios D* and *B,* but also a higher standard deviation. Such portfolios are called efficient. The set of all efficient portfolios, the ***efficient frontier***, is shown in Figure 3-7 as a dark gray line. Essentially, these are Pareto-efficient portfolios, where one parameter cannot be improved without worsening the other.

Remember that in the mean-variance model, the investor has two objectives: maximizing expected return and minimizing variance, while these goals are usually in contradiction. Thus, the following optimization problem can be defined,

$$w = \begin{cases} \underset{x}{argmin}\ k \cdot x^T \cdot \mathbf{Q} \cdot x - \ (1-k) \cdot E(R)^T \cdot x \\ \sum_{i=1}^{n} x_i = 1 \\ x_i \geq 0, \quad i = 1, \ldots, n \end{cases}, \tag{3.21}$$

where the objective function is the weighted sum of the portfolio variance and the negative value of expected return with the weight *k* being a measure of the investor's risk tolerance. If *k*=0, it concerns a risk-neutral investor and his objective function is the maximization of expected return. If *k*>0, it concerns a risk-averse investor. In the case k=1, it concerns an absolutely risk-averse investor, who does not consider the expected return at all, but only minimizes the variance. The optimization task is an example of a quadratic programming problem. The solution to this problem is the vector of relative weights representing the ***optimal portfolio***, i.e., the portfolio with the maximum value of utility function for the investor.

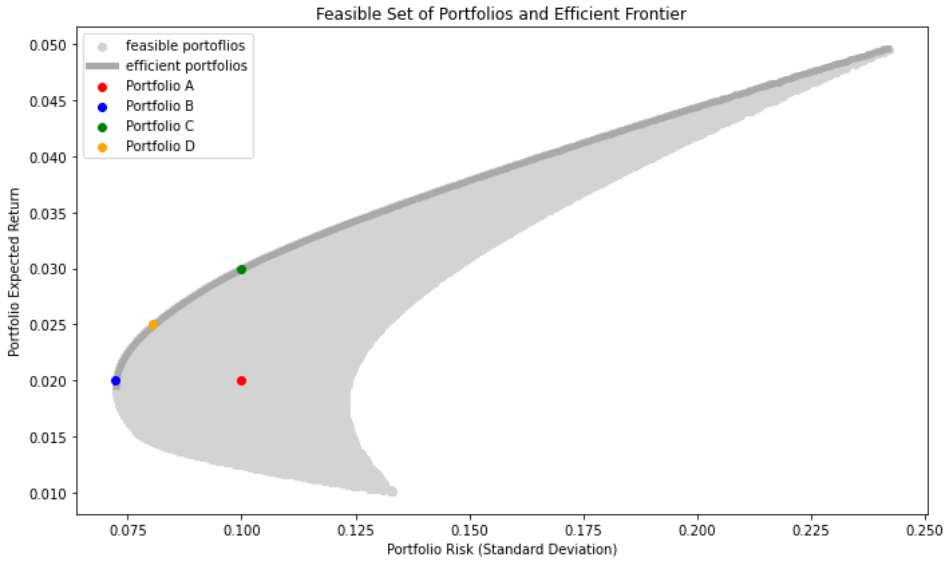

Figure 3-7 Feasible and efficient sets of portfolios for three assets

Note: Parameters as defined in Zmeškal et al. (2013, p. 112).

You should notice that both feasible and efficient set of portfolios consists of many portfolios and are common to all investors. The optimal portfolio is the only one, it is a part of the efficient set, and it is investor-specific (depends on *k*).

When we talk about the efficient set in quantitative finance, we are referring to a visual representation of all portfolios that maximize expected returns for every level of risk. Constructing this set is a key objective and involves a multistep optimization process. Initially, the focus is on identifying the portfolio with the absolute minimum risk. This is achieved by solving a quadratic optimization problem, which can be mathematically represented as:

$$w_{min} = \begin{cases} \underset{x}{argmin}\ (x^T \cdot \mathbf{Q} \cdot x) \\ \sum_{i=1}^{n} x_i = 1 \\ x_i \geq 0, \quad i = 1, \ldots, n \end{cases} . \tag{3.22}$$

The next phase involves finding the portfolio that promises the maximum expected return. Theoretically, this is a simpler task, as it is often the portfolio that is wholly invested in the asset with the highest expected return. This optimization can be expressed as:

$$w_{max} = \begin{cases} \underset{x}{argmax}\ (E(R)^T \cdot x) \\ \sum_{i=1}^{n} x_i = 1 \\ x_i \geq 0, \quad i = 1, \ldots, n \end{cases} . \tag{3.23}$$

However, if multiple assets share the highest expected return, the tiebreaker is to choose the asset with the lowest associated risk.

Next, we must find the inner points of the efficient frontier. For equidistant expected returns,

$$R_{gen} \in \left\{R_{min} + \frac{1}{n}(R_{max} - R_{min}), R_{min} + \frac{2}{n}(R_{max} - R_{min}), \dots\right\}. \tag{3.24}$$

where R_{min} is expected return for minimum risk portfolio, R_{max} is the expected return for the maximum expected return portfolio, we find the portfolios with minimum variance,

$$w = \begin{cases} \underset{x}{argmin}\ (x^T \cdot \mathbf{Q} \cdot x) \\ E(R)^T \cdot x = R_{gen} \\ \sum_{i=1}^{n} x_i = 1 \\ x_i \geq 0, \quad i = 1, \dots, n \end{cases}. \tag{3.25}$$

Alternatively, we can solve the optimization task (3.21) for the equidistant values of k from 0 to 1.

3.3.2 Generalized mean-risk optimization framework

The mean-variance approach can be simply extended to the general mean-risk framework, in which different risk measures (see Section 3.2.2) can replace the variance. The standard deviation can be replaced, e.g., by mean absolute deviation (Konno & Yamazaki, 1991), the downside deviation (Estrada, 2008), or Conditional Value at Risk (Rockafellar & Uryasev, 2000).

To obtain the efficient frontier, considering the chosen risk measure ρ, in line with Section 3.3.1, we solve the following optimization problems,

$$w_{min} = \begin{cases} \underset{x}{argmin}\ \rho(x) \\ \sum_{i=1}^{n} x_i = 1 \\ x_i \geq 0, \quad i = 1, \dots, n \end{cases} \tag{3.26}$$

for minimum-risk portfolio,

$$w_{max} = \begin{cases} \underset{x}{argmax}\ (E(R)^T \cdot x) \\ \sum_{i=1}^{n} x_i = 1 \\ x_i \geq 0, \quad i = 1, \dots, n \end{cases} \tag{3.27}$$

for the maximum-return portfolio,

$$w = \begin{cases} \underset{x}{argmin}\ \rho(x) \\ E(R)^T \cdot x = R_{gen} \\ \sum_{i=1}^{n} x_i = 1 \\ x_i \geq 0, \quad i = 1, \dots, n \end{cases} \tag{3.28}$$

for a given expected return R_{gen}.

3.3.3 Performance ratio maximization

In portfolio optimization, another prevailing strategy is to focus on maximizing a specific performance measure. This method is often rooted in historical analysis, where historical returns serve as the basis for determining the optimal portfolio weights. By analyzing past

performance, investors aim to construct a portfolio that is expected to offer the best possible results according to the chosen performance measure.

The selection of a performance measure is pivotal and varies depending on the investor's goals, see Section 3.2.3. To maximize the historical value of the selected performance measure, the optimization problem can be expressed mathematically as follows:

$$w = \begin{cases} \underset{x}{argmax}\ (Performance\ measure(R_{hist}, x)) \\ \sum_{i=1}^{n} x_i = 1 \\ x_i \geq 0, \quad i = 1, \ldots, n \end{cases} . \tag{3.29}$$

Here, R_{hist} represents the matrix of historical returns, and x is the vector of portfolio weights to be optimized. The constraints ensure that the total portfolio weight sums up to 100% and that no individual weight is negative.

In practice, this optimization process involves backtesting various weight combinations to determine which portfolio composition would have yielded the best historical performance according to the selected metric. Python's powerful data science packages, such as *Pandas* for data handling and *Pyportfolioopt* or *Riskfolio-Lib* for portfolio optimization, provide the necessary tools to run these historical simulations and identify the weight allocation that maximizes the desired performance measure. See Section 3.3.5.

However, it is important to note that past performance is not indicative of future results. Thus, while historical optimization is a common approach, it must be applied with an understanding of its limitations.

3.3.4 Other portfolio optimization approaches

There are also other portfolio optimization strategies. In this section, we list some examples.

Equally weighted portfolio

In an equally weighted portfolio, sometimes also named a naive portfolio, the components of the portfolio are invested at an equal weight,

$$w_{naive} = 1/n, \tag{3.30}$$

where n is the number of components in the portfolio. The naive portfolio is usually applied as a benchmark to compare the performance of the other optimization models, just as Demiguel et al. (2009, pp. 1916–1917) stated: "*There are two reasons for using the naive rule as a benchmark. First, it is easy to implement because it does not rely either on the estimation of the moments of asset returns or on optimization. Second, despite the sophisticated theoretical models developed in the last 50 years and the advances in methods for estimating the parameters of these models, investors continue to use such simple allocation rules for allocating their wealth across assets.*"

Code 3-14 Function returning 1/n portfolio

```
def port_naive(prices):
    n = prices.shape[1] # obtain the number of the columns (assets in prices DataFrame)
    weights = [1/n for _ in range(n)]
    return weights
```

Equally weighted risk contribution portfolio

According to Martínez-Nieto et al. (2021), the naive portfolio exhibits a limited diversification effect when the individual risks vary significantly. In response to this challenge, one can construct an equally weighted risk contribution portfolio, often referred to as the risk parity portfolio. In the equally weighted risk contribution portfolio, each stock contributes an equal amount of risk to the total portfolio risk, as detailed by Maillard et al. (2010).

Any risk measure can be used, see the application in the *Riskfolio-Lib* package, for simplicity reasons, we illustrate this approach on the variance. Following this strategy, allocation is achieved by solving the following optimization problem:

$$w = \begin{cases} \underset{x}{argmin} \ \sum_{i=1}^{n}\sum_{j=1}^{n}\left(x_i \cdot (\mathbf{Q}\cdot x)_i - x_j \cdot (\mathbf{Q}\cdot x)_j\right)^2 \\ \sum_{i=1}^{n} x_i = 1 \\ x_i \geq 0, \quad i = 1,\ldots,n \end{cases} . \tag{3.31}$$

Here, w represents the weights of the portfolio, $\mathbf{Q}$ is the covariance matrix of stock returns, and n is the number of stocks in the portfolio. The objective is to minimize the sum of squared differences in variance contributions across all pairs of stocks, adhering to the conditions that the weights sum to 100% and individual weights are non-negative.

A practical application of the risk parity portfolios applying the *Riskfolio-Lib* package is presented in the code snippet in Code 3-16. The power of the *Riskfolio-Lib* package lies in the freedom to choose any risk measure according to which the risk parity portfolio should be constructed.

Question 3-3 Risk parity portfolios

Applying the *Riskfolio-Lib* package, compute the risk parity portfolios for different risk measures, and visualize the portfolio compositions. Compare and comment on these portfolio compositions.

Other approaches

Carver (2015) presents a powerful, yet simple algorithm for portfolio creation, based on equal risk contribution. His approach is practical and does not involve any optimization.

3.3.5 Portfolio optimization packages in Python

Like for any other financial domain, there are numerous packages available for portfolio optimization. In this section, we provide examples of how to utilize two of them: *PyPortfolioOp*, which is relatively simple yet useful, and *Riskfolio-Lib*, which is easy to use but highly powerful. Particularly, the latter comes highly recommended.

Pyportfolioopt

Pyportfolioopt, short for Python Portfolio Optimization, is a powerful open-source package designed to streamline the process of portfolio optimization in Python. It provides a user-friendly interface for constructing, analyzing, and optimizing investment portfolios. This section illustrates the application of the package on the construction of mean-CVaR efficient frontier; for theory, see Sections 3.3.1 and 3.3.2.

The example is illustrated in Code 3-15. After importing the necessary packages, we retrieve the list of DJIA index components, as shown in Code 1-2, and proceed to download the corresponding data, as shown in Code 1-1. Subsequently, utilizing *PyPortfolioOp* functions, we estimate the expected returns and the covariance matrix.

Following this, we identify portfolios of the following three types: the portfolio with the minimum Conditional Value at Risk (CVaR), the portfolio with the maximum expected return (while technically omitting the constraint on CVaR by allowing it to be –100%), and 48 interior points of the efficient frontier. This is achieved through a cycle where we specify the target return within the range of returns from the minimum-CVaR portfolio to the maximum-expected-return portfolio.

Portfolio weights are consistently determined by creating an instance of the *pypfopt.efficient_frontier.EfficientCVaR* class, followed by optimization using *ef_cvar.min_cvar()*. The expected return and CVaRs are stored in the lists *returns* and *cvars*, respectively. The weights are stored in the dataframe *weights*. Finally, we visualize the efficient frontier and plot the weights of efficient portfolios, see Figure 3-8.

Code 3-15 Calculation of mean-CVaR efficient frontier (*Pyportfolioop*)

```
# Import necessary packages
import yfinance as yf
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import pypfopt

# Make sure to install the required PyPortfolioOpt package if not already installed:
# !pip install PyPortfolioOpt

# Define the in-sample period for historical data analysis.
start_date = '2012-01-01'
end_date = '2017-01-01'

# Download the list of DJIA tickers
url = 'https://en.wikipedia.org/wiki/Dow_Jones_Industrial_Average'
ticker_list = pd.read_html(url)[1]['Symbol'].dropna().tolist()

# Download historical stock data using the yfinance package.
# We will only keep the closing prices for our analysis.
prices = yf.download(tickers=ticker_list, start=start_date, end=end_date, interval='1d',
group_by='column', auto_adjust=True)["Close"]
# Remove any columns with missing values to maintain data consistency
prices.dropna(axis=1, inplace=True)

# Calculate expected returns and historical returns using PyPortfolioOpt functions
mu = pypfopt.expected_returns.mean_historical_return(prices)
historical_returns = pypfopt.expected_returns.returns_from_prices(prices)

# Initialize DataFrame to store the weights
weights = pd.DataFrame(index=prices.columns)
```

```
# Portfolio Optimization: Minimum CVaR (Conditional Value at Risk)
beta = 0.95  # Confidence level used for CVaR calculation
# Initialize the Efficient Frontier object for CVaR
ef_cvar = pypfopt.efficient_frontier.EfficientCVaR(expected_returns=mu,
returns=historical_returns, weight_bounds=(0, 1))
# Optimize for minimum CVaR
weights_min_cvar = ef_cvar.min_cvar()
# Retrieve and clean the weights for the minimum CVaR portfolio
weights['min CVaR'] = pd.Series(ef_cvar.clean_weights())
# Calculate and store the performance of the minimum CVaR portfolio
ret_min_cvar, cvar_min = ef_cvar.portfolio_performance()

# Portfolio Optimization: Maximum Expected Return
# Reinitialize the Efficient Frontier object
ef_max_ret = pypfopt.efficient_frontier.EfficientCVaR(expected_returns=mu,
returns=historical_returns, weight_bounds=(0, 1))
# Optimize for the portfolio that maximizes expected return for the given level of CVaR
ef_max_ret.efficient_risk(target_cvar=1)
# Retrieve and clean the weights for the maximum return portfolio
weights['max ret'] = pd.Series(ef_max_ret.clean_weights())
# Calculate and store the performance of the maximum return portfolio
ret_max, cvar_max = ef_max_ret.portfolio_performance()

# Generate and collect performance data for a range of portfolios
returns = []
cvars = []
n_portfolios = 50  # Number of portfolios to simulate

# Loop through a range of target returns to generate different portfolios
for target_ret in np.linspace(ret_min_cvar, ret_max, n_portfolios):
    # Reinitialize the Efficient Frontier object
    ef = pypfopt.efficient_frontier.EfficientCVaR(expected_returns=mu,
returns=historical_returns, weight_bounds=(0, 1))
    # Optimize for the efficient return for each target return
    ef.efficient_return(target_return=target_ret)
    # Retrieve and clean the weights for the current portfolio
    weights['Target return of {:.2f}%'.format(target_ret * 100)] =
pd.Series(ef.clean_weights())
    # Calculate and store the performance of the current portfolio
    ret, cvar = ef.portfolio_performance()
    returns.append(ret)
    cvars.append(cvar)

# Plot the efficient frontier
plt.figure(figsize=(10, 6))
plt.plot(cvars, returns, label='Efficient Frontier')
plt.title('Efficient Frontier Curve')
plt.xlabel('Conditional Value at Risk (CVaR)')
plt.ylabel('Expected Return')
plt.legend()
plt.tight_layout()
plt.show()

# Plot the weights as a stacked bar chart
weights_transposed = weights.drop(columns=['min CVaR', 'max ret']).T
weights_transposed.plot(kind='bar', stacked=True, figsize=(10, 6), legend=False)
plt.title('Portfolio Weights for Different Optimization Strategies')
plt.xlabel('Assets')
plt.ylabel('Weights')
#plt.legend(title='Optimization Strategies', loc='best')
plt.tight_layout()
plt.show()
```

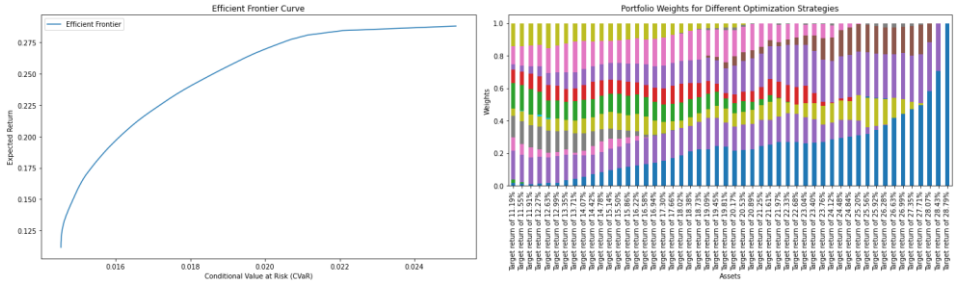

Figure 3-8 Efficient mean-CVaR frontier and portfolio weights (*PyPortfolioOp*)

Question 3-4 Variables in the graph

Describe the variables in Figure 3-8. Is the return and CVaR annualized? Compare the efficient frontier to the graph in Figure 3-10.

Riskfolio-Lib

Riskfolio-Lib (Cajas, 2024) is a powerful package with a simple interface, which students, academics, and practitioners can use to build investment portfolios based on mathematically complex models with low effort.

Its strength lies in the ability to optimize a variety of even highly complex models with just a few lines of code, as demonstrated in Code 3-16. The package can optimize mean-risk portfolios and risk parity portfolios, factor models, and hierarchical risk parity portfolios, offering a wide range of options for risk measures. In addition, it supports various constraints in portfolio optimization, particularly those related to the number of assets (cardinality constraint).

In the code snippet, we download the list of tickers of the components in DJIA index, see Code 1-2, drop the symbols with *NaN* values, and calculate the simple returns, see Code 1-5. Then, we create the portfolio object and set the method for the calculation of expected returns and the covariance matrix. We also set the alpha for CVaR calculation (see Section 2.2.2) and optionally we can set the cardinality constraints (see the commented line in the code). Finally, by calling the functions *optimization* and *rp_optimizaion* of the portfolio object, we can obtain the weights of corresponding portfolios. Using function *plot_pie*, we can visualize the portfolio composition in a graph, see the output in Figure 3-9.

Code 3-16 Optimal portfolio calculations

```
import yfinance as yf
import pandas as pd
import riskfolio as rp

start_date = '2012-01-01'
end_date = '2017-01-01'

# Download the list of DJIA tickers
url = 'https://en.wikipedia.org/wiki/Dow_Jones_Industrial_Average'
ticker_list = pd.read_html(url)[1]['Symbol'].dropna().tolist()

# Download historical stock data using the yfinance package.
# We will only keep the closing prices for our analysis.
```

```
prices = yf.download(tickers=ticker_list, start=start_date, end=end_date, interval='1d',
group_by='column', auto_adjust=True)["Close"]
prices.dropna(axis=1, inplace=True) # Remove any columns with missing values to maintain
data consistency
returns=prices.pct_change().dropna() # Calculate returns

port = rp.Portfolio(returns=returns) # Building the portfolio object
port.assets_stats(method_mu='hist', method_cov='hist') # set the method fro mu and
covariance calculation
port.alpha = 0.05 # set CVaR alpha to 5%

# the cardinality constraint (maximum number of the assets)
#port.card = 5

w = pd.DataFrame()
w['minV'] = port.optimization(model='Classic', rm='MV', obj='MinRisk') # minimum-varinace
portfolio
w['maxR'] = port.optimization(model='Classic', rm='MV', obj='MaxRet') # minimum-varinace
portfolio
w['minMDD'] = port.optimization(model='Classic', rm='MDD', obj='MinRisk') # minimum-MDD
portfolio
w['tanV'] = port.optimization(model='Classic', rm='CVaR', obj='Sharpe', rf=0, hist=True) #
Tangential portfolio of mean-variance efficient frontier
w['tanCVaR'] = port.optimization(model='Classic', rm='CVaR', obj='Sharpe', rf=0, hist=True)
# Tangential portfolio of mean-CVaR efficient frontier

w['maxU'] = port.optimization(model='Classic', rm='CVaR', obj='Utility', l=2) # maximum-
utility portfolio

w['parV'] = port.rp_optimization(model='Classic', rm='MV') # risk parity for variance
w['parCVaR'] = port.rp_optimization(model='Classic', rm='CVaR') # risk parity for CVaR

ax = rp.plot_pie(w=w['minV'].to_frame(), title='Min-CVaR', others=0.05, nrow=25, cmap =
"tab20", height=6, width=10, ax=None)

ax = rp.plot_pie(w=w['tanCVaR'].to_frame(), title='Tangential mean-CVaR', others=0.05,
nrow=25, cmap = "tab20", height=6, width=10, ax=None)
```

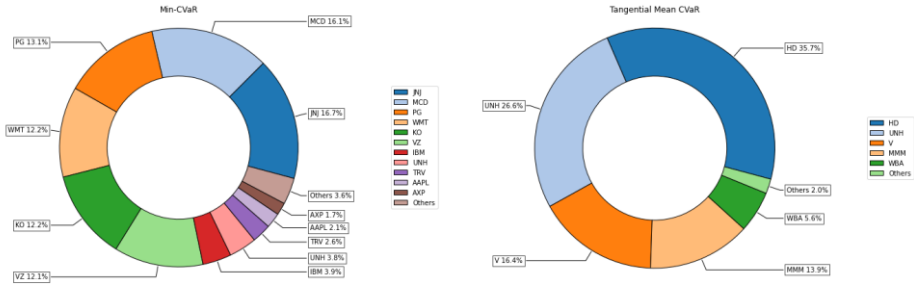

Figure 3-9 The output of Code 3-16

Question 3-5 Portfolio diversification

Compare and discuss the number of assets in the portfolios depicted in Figure 3-9. From how many assets is the maximum-return portfolio composed?

Using *Riskfolio-Lib* we can also simply calculate the whole efficient frontier just by calling the function *efficient_frontier* and plot it by calling the function *plot_frontier*, see Code 3-17. The weights can be plotted by calling the function *plot_frontier_area*. The results are shown in Figure 3-10.

Code 3-17 Calculation of mean-CVaR efficient frontier (*Riskfolio-Lib*)

```
port = rp.Portfolio(returns=returns) # Building the portfolio object
port.assets_stats(method_mu='hist', method_cov='hist')
port.alpha = 0.05 # set CVaR alpha to 5%

n_portfolios = 50

weights = port.efficient_frontier(model='Classic', rm='CVaR', points=n_portfolios, rf=0,
hist=True)

# Plot efficient frontier
ax = rp.plot_frontier(w_frontier=weights, mu=port.mu, cov=port.cov, returns=port.returns,
rm='CVaR', rf=0, alpha=0.05, cmap='viridis', w=w['tanCVaR'], label='Max Risk-Adjusted Return
Portfolio', marker='*', s=16, c='r', height=6, width=10, ax=None)

# Plot efficient frontier portfolio weights
ax = rp.plot_frontier_area(w_frontier=weights, cmap="tab20", height=6, width=10, ax=None)
```

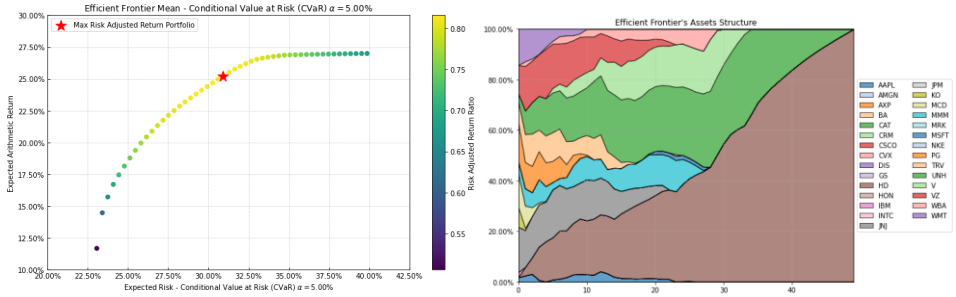

Figure 3-10 Efficient mean-CVaR frontier and portfolio weights (*Riskfolio-Lib*)

Question 3-6 Variables in the graph

Describe the variables in Figure 3-10. Is the CVaR and return annualized? Compare the efficient frontier to the graph in Figure 3-8.

3.4 CAPM model and Fama-French models

In the previous subchapter, especially in Section 3.3.1, we assumed possible investments only in the risky assets (stocks). Adding the risk-free asset among possible investment options would be straightforward, however, the efficient frontier would change dramatically. In this section, we discuss the CAPM model and Fama-French factor models. The implication of the CAPM model is that the total risk (standard deviation of the returns) should not be priced, but only a systematic part of it.

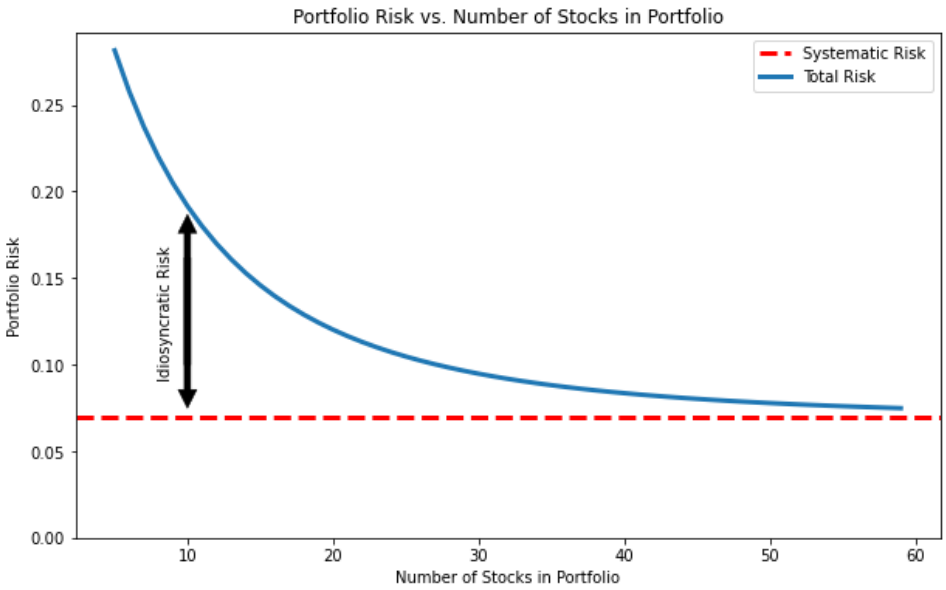

Figure 3-11 Portfolio risk in dependence on the number of stocks

3.4.1 CAPM model

The Capital Asset Pricing Model (CAPM) was introduced independently in the early 1960s by Sharpe (1964), Treynor (1962), Lintner (1965), and Mossin (1966). It builds on the portfolio choice model developed by Markowitz (1952) already discussed in Section 3.3.1. In Markowitz's framework, we measure the risk of the portfolio by the standard deviation. On the contrary, the CAPM model introduces and uses a systematic risk, *beta*. ***Systematic risk*** can be understood as a measure of the risk the asset adds to a well-diversified market portfolio.

Thus, we can distinguish between idiosyncratic risk and systematic risk; see Figure 3-11. ***Idiosyncratic risk***, also known as *unsystematic risk* or *specific risk*, refers to the risk that is specific only to an individual asset. For multiple assets, this risk is not correlated, and therefore the total idiosyncratic risk of the portfolio decreases as the number of assets increases. This is called the *diversification* effect. For well-diversified portfolios, the idiosyncratic risk becomes very small once about 20-30 components are in the portfolio. On the contrary, as can be seen in Figure 3-11, the total risk cannot be decreased below the given threshold, no matter how many assets we include in the portfolio. This threshold represents the systematic risk, which cannot be diversified out.

Question 3-7 Hedging out the systematic risk

Can you think of any way to hedge against systematic risk?

The CAPM model, like any other (financial) model, is based on assumptions, which provide a simplified representation of the (financial) world. These assumptions are as follows:

- Markets are efficient, all investors have costless access to market information, and prices instantly respond to new public information.
- There are no frictions, i.e., there are no transaction costs, taxes, and restrictions on short selling.
- There is infinite divisibility of assets, meaning the funds available to an investor can be allocated among assets in any proportion.
- For any given asset, expected returns are normally distributed.
- Investors are rational and risk-averse, seeking the highest expected return for the lowest possible risk.
- The investors have homogeneous expectations, that is, all investors, given the same information, have the same expectations for future security prices, expected returns, and risks.
- It is the single-period model, i.e. the model is static, considering an investment horizon where the portfolio's structure remains unchanged.

Although these assumptions are not always realistic, they simplify reality and make the theoretical environment possible.

Capital Market Line as an efficient frontier

When we expand Markowitz's mean-variance framework to include a risk-free asset that can be both borrowed and lent, we derive the Capital Market Line (CML) of the CAPM model. The CML illustrates the relationship between the expected return of the portfolio and the standard deviation of the portfolio for efficient portfolios, essentially representing the efficient frontier.

The CAPM-CML is illustrated in Figure 3-12. In the figure, the blue curve shows efficient portfolios composed of risky assets only (see Section 3.3.1). The CML line, represented by the green-red line, is defined by the formula:

$$E(R_P) = R_{RF} + \frac{R_M - R_{RF}}{\sigma_M} \cdot \sigma_P. \tag{3.32}$$

A special portfolio known as the ***market*** or ***tangential portfolio*** resides at the intersection of these two curves. The ***tangential portfolio*** is a portfolio with a maximum return adjusted for the risk, thus it is also called a maximum risk-adjusted portfolio. In the case of standard deviation as a risk measure, we get the maximum Sharpe ratio portfolio, see Section 3.3.3. The ***market portfolio*** comprises all assets in the market, each weighted by its market capitalization. It is assumed to be identical to the tangential portfolio, representing the portfolio with the highest Sharpe ratio.

The point of the tangential portfolio divides the CML line into two segments: lending (green) and borrowing (red) portfolios. The distinction lies in whether we borrow or lend money at a risk-free rate. For lending portfolios, we allocate funds into both risky and risk-free assets, while for borrowing portfolios, we invest in risky assets more than we possess, effectively borrowing money. In practice, it is called a leveraged position.

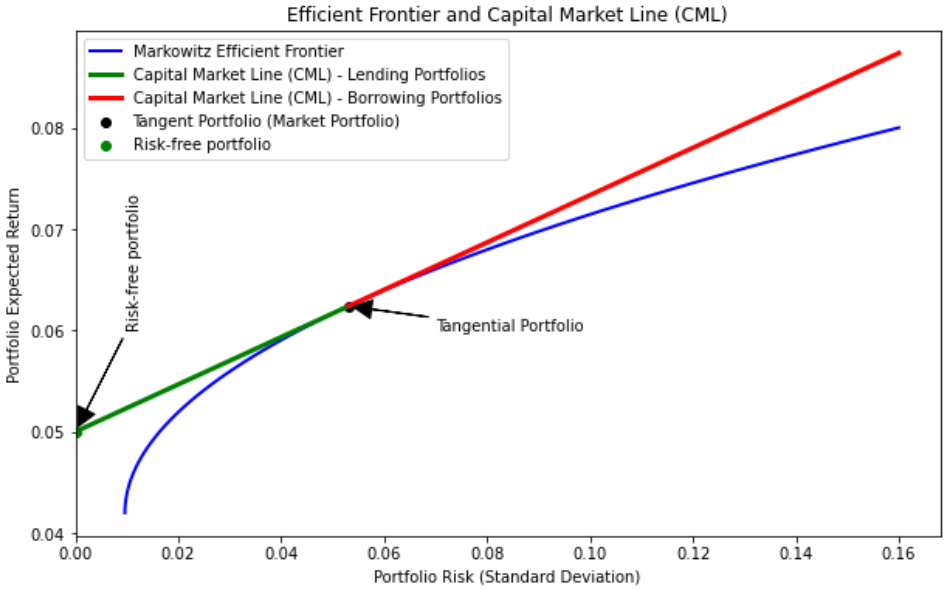

Figure 3-12 An example of Capital Market Line

Security Market Line

For simplicity, it can be said that CML represents the relationship between total risk and the expected return of efficient portfolios. On the contrary, the Security Market Line (SML) represents the relationship between systematic risk and expected return. Note that while standard deviation is a measure of total portfolio risk, beta measures systematic risk. Under CAPM assumptions, the beta can be calculated as follows:

$$\beta = \frac{cov(R_i, R_M)}{\sigma_M}, \tag{3.33}$$

leading to the SML line,

$$E(R_i) = R_{RF} + \beta \cdot (R_M - R_{RF}). \tag{3.34}$$

In these equations, the index i usually represents the individual assets, but can also represent the portfolios. The SML line is shown in Figure 3-13.

As can be seen, the beta parameter is crucial as it represents the riskiness of the asset (stock). On the basis of its value, we can conclude:

- A stock with a ***beta of 1*** tends to move in line with the market. If the market goes up by 1%, the stock, on average, is expected to go up by 1%, and vice versa.
- A stock with a ***beta greater than 1*** is considered more volatile than the market. On average, if the market goes up, this stock tends to rise by a higher percentage, and if the market falls, the stock tends to experience a greater decline.
- A stock with a ***positive beta of less than 1*** is considered less volatile than the market. It tends to move less than the market in percentage terms. On average, if the market goes up, the stock rises but not as much, and if the market falls, the stock declines but not as sharply.

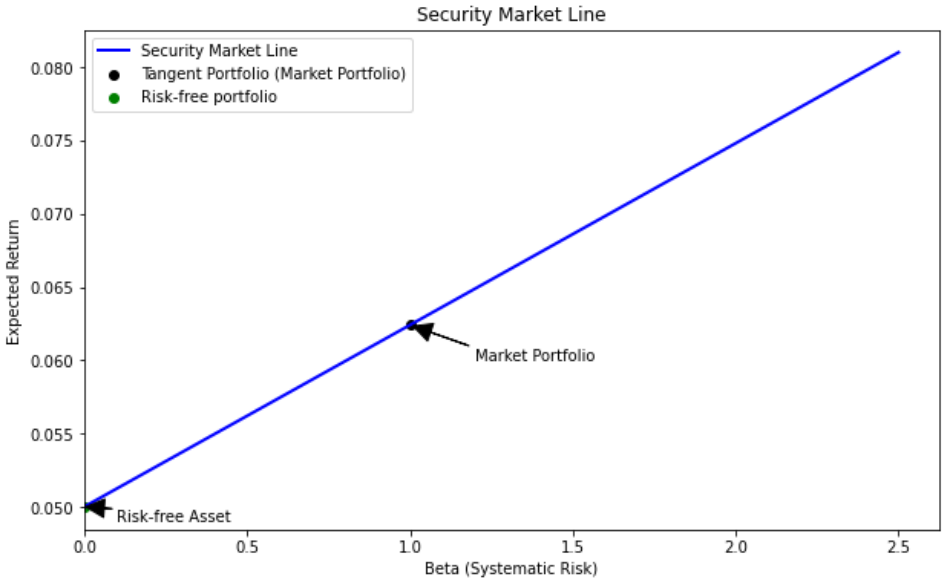

Figure 3-13 An example of Security Market Line

- A ***beta of 0*** indicates that the movement of the stock price is not correlated with the market. Changes in the market do not affect the stock (on average).
- A ***negative beta*** implies an inverse relationship to the market. If the market rises, a stock with negative beta decreases on average and vice versa. This suggests that the stock moves in the opposite direction related to the market.

Parameters estimation

The SML equation can be rewritten into the following form:

$$R_{i,t} = \alpha + (1 - \beta) \cdot R_{RF} + \beta \cdot R_{M,t} + \epsilon_{i,t}, \tag{3.35}$$

where $R_{i,t}$ is the (observed) return of the ith stock, R_{RF} is the risk-free rate, $R_{M,t}$ is the return of the market, ϵ_t represent the random term (a white noise), β is the measure of systematic risk and α represents the stock‘s overperformance (positive value) or underperformance (negative value), see Jensen alpha in Section 3.2.3, leading to the linear regression,

$$R_{i,t} = a + \beta \cdot R_{M,t} + \epsilon_{i,t}. \tag{3.36}$$

The market portfolio used in the CAPM model is purely theoretical and cannot be purchased or invested directly as an alternative to individual stocks. Typically, an index, such as the Standard & Poor's 500, is employed as a practical substitute for the market portfolio. The risk-free rate represents the expected return of an investment with zero risk over a specified period of time. In practice, it is commonly represented by the yield on government bonds or Treasury bills. It is important to note that we make the assumption that the risk-free rate remains constant.

In Code 3-18, which results in Figure 3-14, we present a scatterplot illustrating the relationship of returns between the Intel (ticker INTC) and the S&P 500 index (ticker ^GSPC). As depicted in the figure, a positive relationship is observed, suggesting a positive beta value.

Code 3-18 Download and plotting of daily returns of INTC

```
# download the data for Intel and ^GSPC to mimic S&P 500 index and calculate simple returns
data = yf.download(tickers = "INTC ^GSPC", start="2000-11-01", end="2023-10-01", interval =
"1d", group_by = 'column', auto_adjust = True)
ret=data["Close"].pct_change().dropna() # calculate simple returns

# plot the returns in scatter plot
plt.figure(figsize=(10, 6))
plt.scatter(x=ret['^GSPC'], y=ret['INTC'], s=2)
plt.xlabel('S&P 500 return')
plt.ylabel('INTC return')
plt.title('Dependence of returns')
plt.grid(True)
plt.show()

# plot using seaborn package
sns.jointplot(data=ret, x="^GSPC", y="INTC") # plot the scatter plot with marginal
distributions
# sns.jointplot(data=ret, x="^GSPC", y="INTC", kind="kde") # the same but contour plot
sns.regplot(x=ret['^GSPC'], y=ret['INTC'], ci=None) # plot the scatter plot with the
regression
```

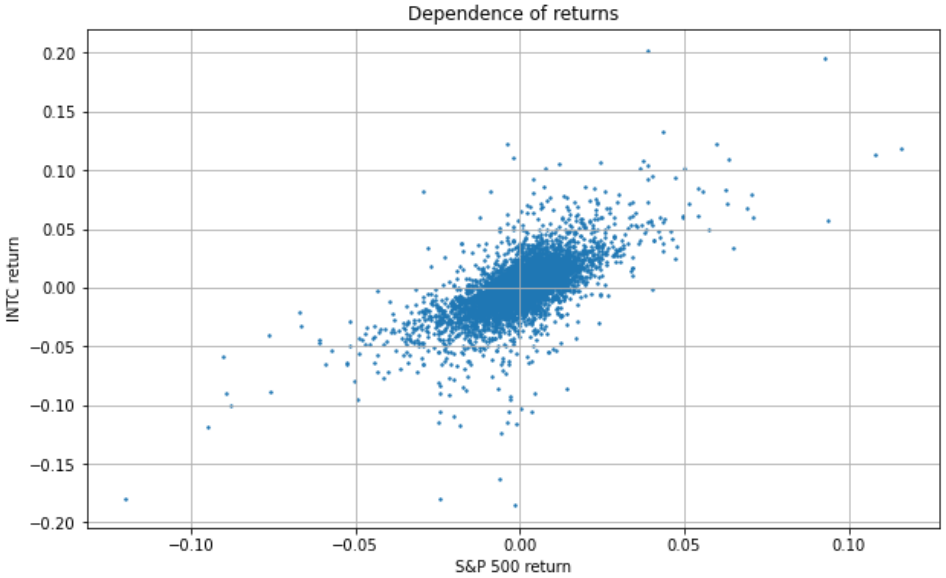

Figure 3-14 Scatterplot of the returns (output of Code 3-18)

To obtain estimates for a and β in equation (3.36), we can utilize the *numpy.polyfit* or *sklearn.linear_model* function, as demonstrated in Code 3-19. In this example, the estimated value of β is 1.247, and the estimated value of the intercept (a) is -0.00193%, approximately -0.486% per annum. This implies that Intel is more volatile than the S&P 500, given that the beta is greater than 1. Also, note that we can get a beta estimate from Yahoo Finance website using the *Yfinance* package, see Code 1-4 in Section 1.1.2.

To draw relevant conclusions, we perform rigorous linear regression using the *statmodel.OLS* in Code 3-20 and Code 3-21. For a detailed overview of linear regression models in Python, see Appendix B.

Question 3-8 Overperformance or underperformance of INTC

Seeing the results in Code 3-19, and assuming the risk-free rate to be 2% p.a., what is the annual underperformance of INTC relative to the correctly priced assets?

Code 3-19 Estimation of regression coefficients via *Numpy* and *Scikit-learn* packages

```
# calculate the linear regression using numpy
slope, intercept = np.polyfit(ret['^GSPC'], ret['INTC'], 1)
print(f"intercept: {intercept}; slope: {slope}")

# calculate linear regression with scikit-learn
from sklearn.linear_model import LinearRegression
x = ret['^GSPC'].to_numpy().reshape(-1, 1)
y = ret['INTC'].to_numpy()
model = LinearRegression().fit(x, y)
print(f"intercept: {model.intercept_}; slope: {model.coef_}")
> intercept: -1.927890223672462e-05; slope: 1.246569358459721
intercept: -1.9278902236727455e-05; slope: [1.24656936]
```

Code 3-20 Estimation of the regression coefficients via *Statsmodel* package

```
# calculate linear regression with statsmodels
import statsmodels.api as sm

x = ret['^GSPC'].to_numpy()#.reshape(-1, 1)
y = ret['INTC'].to_numpy()
x = sm.add_constant(x) # we must add constant in order to have intercept
model = sm.OLS(y, x) # create the model by means of OLS (be carefull about the order of
parameters - different to scikit-learn)
results = model.fit() # fit the model
print(f"intercept: {results.params[0]}; slope: {results.params[1]}")
print(results.summary()) # print results of linear regression
>                             OLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:                       0.456
Model:                            OLS   Adj. R-squared:                  0.456
Method:                 Least Squares   F-statistic:                     4837.
Date:                Tue, 06 Feb 2024   Prob (F-statistic):               0.00
Time:                        14:53:02   Log-Likelihood:                 15374.
No. Observations:                5762   AIC:                         -3.074e+04
Df Residuals:                    5760   BIC:                         -3.073e+04
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const      -1.928e-05      0.000     -0.087      0.931      -0.000       0.000
x1             1.2466      0.018     69.546      0.000       1.211       1.282
==============================================================================
Omnibus:                     1295.441   Durbin-Watson:                   2.007
```

```
Prob(Omnibus):                  0.000   Jarque-Bera (JB):            36814.235
Skew:                          -0.427   Prob(JB):                         0.00
Kurtosis:                      15.354   Cond. No.                         81.0
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
```

Question 3-9 Coefficients interpretation

Given that the risk-free rate is assumed to be zero, how would you interpret the regression coefficients? Are these coefficients statistically significant?
If we do not assume a zero risk-free rate, can the intercept help us determine its value? If so, what would the risk-free rate be?
Is there a need to re-estimate our model under these assumptions? Why or why not?

Code 3-21 Re-estimation of the regression coefficients via *Statsmodel* package

```
# re-estimate the model without the intercept
x = ret['^GSPC'].to_numpy()
y = ret['INTC'].to_numpy()
model = sm.OLS(y, x)  # create the model by means of OLS
results = model.fit()  # fit the model
print(results.summary()) # print results of linear regression
>                                 OLS Regression Results
==============================================================================================
Dep. Variable:                        y   R-squared (uncentered):                   0.457
Model:                              OLS   Adj. R-squared (uncentered):              0.456
Method:                   Least Squares   F-statistic:                              4839.
Date:                  Tue, 06 Feb 2024   Prob (F-statistic):                        0.00
Time:                          14:53:02   Log-Likelihood:                          15374.
No. Observations:                  5762   AIC:                                 -3.075e+04
Df Residuals:                      5761   BIC:                                 -3.074e+04
Df Model:                             1
Covariance Type:              nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
x1             1.2465      0.018     69.566      0.000       1.211       1.282
==============================================================================
Omnibus:                     1295.445   Durbin-Watson:                   2.007
Prob(Omnibus):                  0.000   Jarque-Bera (JB):            36814.807
Skew:                          -0.427   Prob(JB):                         0.00
Kurtosis:                      15.354   Cond. No.                         1.00
==============================================================================

Notes:
[1] R² is computed without centering (uncentered) since the model does not contain a
constant.
[2] Standard Errors assume that the covariance matrix of the errors is correctly specified.
```

Question 3-10 Which model is better?

Which model (with or without intercept) is better? Discuss why.

3.4.2 Fama-French factor models

There may also be other factors that explain the stock returns. A typical example is the 3-factor model introduced by Fama and French (1993), later extended to a four-factor model by adding a momentum factor by Carhart (1997) and applied by Fama and French (2010, 2012). However, Fama and French (2015) have also considered a five-factor model,

which is discussed in the textbook, and extended it even to a six-factor model (Fama & French, 2018).

The Fama-French 5-factor model (Fama & French, 2015) can be described as follows,

$$R_{i,t} - R_{RF,t} = \alpha_i + \beta_{1,i} \cdot MKT_t + \beta_{2,i} \cdot SMB_t + \beta_{3,i} \cdot HML_t + \\ + \beta_{4,i} \cdot RMW_t + \beta_{5,i} \cdot CMA_t + \epsilon_{i,t}, \quad (3.37)$$

where $R_{i,t}$ represents the stock return for asset *i* at time *t*, $R_{RF,t}$ is the market risk-free rate at time *t*, $\beta_{1,i}$ measures the volatility of the stock compared to the market, MKT_t is the market risk premium at time *t*, $\beta_{2,i}$ is the coefficient for the *SMB* factor (size premium), SMB_t, $\beta_{3,i}$ is the coefficient for the *HML* factor (value premium), HML_t, $\beta_{4,i}$ is the coefficient for the RMW factor, RMW_t, $\beta_{5,i}$ is the coefficient for the CMA factor, CMA_t. The factors' values are published by French on his website (French, 2024) and are easily downloadable.

Question 3-11 Find the explanation of the factors

Search online for an explanation of the factors and interpret the results in Code 3-22.

The code snippet that performs the estimation of the Fama-French 5-factor model for Intel is shown in Code 3-22:

- Lines 3-8: The code retrieves financial factor data from a specified URL, downloads them as a zip file, and extracts the content. The financial factor data are stored in a file named *F-F_Research_Data_5_Factors_2x3_daily.csv*.
- Lines 10-14: The *Pandas* package is used to read the financial factor data from the CSV file, skipping the first 3 rows, and formatting the index as datetime.
- Lines 15-16: The data are scaled by dividing each value by 100 and converting percentages into decimals.
- Lines 18-23: Intel stock data (ticker: INTC) is downloaded from Yahoo Finance using the *Yfinance* package. Simple daily returns are calculated based on the closing prices. The time zone information is removed from the index of the return data.
- Lines 25-27: The stock returns data and the financial factors data are merged based on the common date index.
- Lines 29-32: The *Close* column in the merged data is renamed to *R* (representing returns). Excess returns (column *R-RF*) are calculated as the difference between stock returns (column *R*) and risk-free returns (column *RF*).
- Lines 35-40: A linear regression model is created using the OLS method from the *Statsmodels* package. The dependent variable (*y*) is in column *R-RF* and the independent variables (*x*) are the factors in columns *Mkt-RF*, *SMB*, *HML*, *RMW* and *CMA*. The model is fitted to the data, and the summary statistics are printed.

Code 3-22 Estimation of Fama-French 5-factor model

```
import pandas as pd
import statsmodels.api as sm
import urllib.request
import zipfile
import yfinance as yf
```

```
# first we get the data
FF5F_address = 'https://mba.tuck.dartmouth.edu/pages/faculty/ken.french/ftp/F-
F_Research_Data_5_Factors_2x3_daily_CSV.zip'

urllib.request.urlretrieve(FF5F_address,'FF5F-data.zip') # download the file and save it as
'FF5F-data.zip'
zip_file = zipfile.ZipFile('FF5F-data.zip', 'r') # open the zip file, we have downloaded
zip_file.extractall()  # extract the data (the extracted file has the name 'F-
F_Research_Data_5_Factors_2x3_daily.csv')
zip_file.close() # we close zip file

ff_factors = pd.read_csv('F-F_Research_Data_5_Factors_2x3_daily.csv', skiprows = 3,
index_col = 0)
ff_factors.index = pd.to_datetime(ff_factors.index, format= '%Y%m%d') # format the index
ff_factors = ff_factors.apply(lambda x: x/ 100) # convert the percentage to decimals

# download the stock data and calculate the returns
data = yf.download(tickers = "INTC", start="2000-11-01", end="2022-11-01", interval = "1d",
group_by = 'column', auto_adjust = True)
ret=data["Close"].pct_change().dropna() # calculate simple returns
ret.index = ret.index.tz_localize(None) # remove tz information
# Merging the data
all_data = pd.merge(pd.DataFrame(ret),ff_factors, how = 'inner', left_index= True,
right_index= True) # merge data together

all_data.rename(columns={"Close":"R"}, inplace=True) # Rename the column with returns from
Close to R
all_data['R-RF'] = all_data['R'] - all_data['RF'] # Calculate the excess returns

# do the regression with statsmodels
y = all_data['R-RF']
x = all_data[['Mkt-RF', 'SMB', 'HML', 'RMW', 'CMA']]
x = sm.add_constant(x)
model = sm.OLS(y, x)
results = model.fit()
print(results.summary())

>                            OLS Regression Results
==============================================================================
Dep. Variable:                   R-RF   R-squared:                       0.482
Model:                            OLS   Adj. R-squared:                  0.481
Method:                 Least Squares   F-statistic:                     1028.
Date:                Tue, 06 Feb 2024   Prob (F-statistic):               0.00
Time:                        16:46:57   Log-Likelihood:                 14911.
No. Observations:                5533   AIC:                        -2.981e+04
Df Residuals:                    5527   BIC:                        -2.977e+04
Df Model:                           5
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const      -3.857e-05      0.000     -0.175      0.861      -0.000       0.000
Mkt-RF         1.2316      0.020     62.995      0.000       1.193       1.270
SMB           -0.2277      0.038     -6.057      0.000      -0.301      -0.154
HML           -0.3229      0.033     -9.656      0.000      -0.388      -0.257
RMW           -0.2951      0.048     -6.137      0.000      -0.389      -0.201
CMA            0.0926      0.063      1.465      0.143      -0.031       0.217
==============================================================================
Omnibus:                     1330.386   Durbin-Watson:                   2.008
Prob(Omnibus):                  0.000   Jarque-Bera (JB):            39082.221
Skew:                          -0.508   Prob(JB):                         0.00
Kurtosis:                      15.980   Cond. No.                         302.
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
```

Code 3-23 Re-estimation of Fama-French 5-factor model with significant factors only

```
# re-estimate the model with only significant variables
y = all_data['R-RF']
x = all_data[['Mkt-RF', 'SMB', 'HML', 'RMW']]
model = sm.OLS(y, x)
results = model.fit()
print(results.summary())
>                                 OLS Regression Results
==============================================================================================
Dep. Variable:                      R-RF   R-squared (uncentered):                   0.482
Model:                             OLS   Adj. R-squared (uncentered):              0.481
Method:                  Least Squares   F-statistic:                              1285.
Date:                 Tue, 06 Feb 2024   Prob (F-statistic):                        0.00
Time:                         17:02:14   Log-Likelihood:                          14910.
No. Observations:                5533   AIC:                                  -2.981e+04
Df Residuals:                    5529   BIC:                                  -2.978e+04
Df Model:                           4
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Mkt-RF         1.2240      0.019     64.923      0.000       1.187       1.261
SMB           -0.2249      0.038     -5.993      0.000      -0.298      -0.151
HML           -0.2992      0.029    -10.224      0.000      -0.357      -0.242
RMW           -0.2870      0.048     -6.020      0.000      -0.380      -0.194
==============================================================================
Omnibus:                     1335.008   Durbin-Watson:                   2.009
Prob(Omnibus):                  0.000   Jarque-Bera (JB):            38635.312
Skew:                          -0.518   Prob(JB):                         0.00
Kurtosis:                      15.904   Cond. No.                         2.91
==============================================================================

Notes:
[1] R² is computed without centering (uncentered) since the model does not contain a
constant.
[2] Standard Errors assume that the covariance matrix of the errors is correctly specified.
```

3.5 Backtesting of portfolio optimization models

In Subchapter 3.3, we discussed various approaches for finding the optimal portfolio with a specific emphasis on concrete models applicable to portfolio optimization. Two logical questions arise: Which approach or model is the best? And, if applied over a specific period, what would be the performance of the portfolio obtained?

The first question is logically addressed by comparing the performances of different approaches – essentially, answering the second question for all possible methods. ***However, a crucial aspect is not to measure performance on the same dataset used for portfolio optimization.*** The dataset must be strictly split into two parts: one for parameter estimation and weight optimization (commonly known as the ***in-sample period***) and the other for measuring the performance of obtained portfolios (known as the ***out-of-sample period***).

This simple approach is illustrated in Figure 3-15 on the right panel. In this example, the total dataset spans from 1990 to 2000 and is divided into the first four years as an in-sample part, where models from Subchapter 3.3 are applied. The last seven years form the out-of-sample part, where performance is measured by applying measures from Subchapter 3.2. It is crucial to note that the in-sample part must precede the out-of-sample part. Therefore, the K-fold approach, commonly used in the machine learning domain,

cannot be applied in portfolio backtesting but can be used in technical analysis backtesting (see Chapter 4).

In real-world applications, the situation is more complex due to the need to consider portfolio rebalancing. Using our 1990-2000 example, would you hold the same portfolio for seven years, especially when the weights were calculated from data spanning 1990-1993? Or would you change the portfolio, i.e., ***rebalance*** it?

There are also more complex approaches to backtest portfolio optimization. Carver (2015) enumerates the expanding approach and the rolling-window approach, as shown in Figure 3-16. In both approaches, the portfolio is rebalanced at the beginning of each year and held for one year. The difference lies in the length of the in-sample period. In the first approach, we aim to use the time series as long as possible, while in the latter, the length of the in-sample period is fixed, and the periods roll year-by-year.

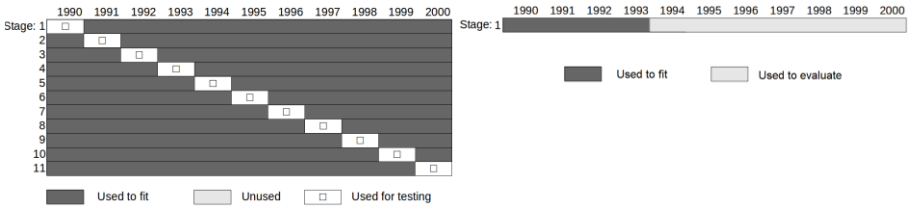

Figure 3-15 The K-folds approach (left) and simple approach (right)

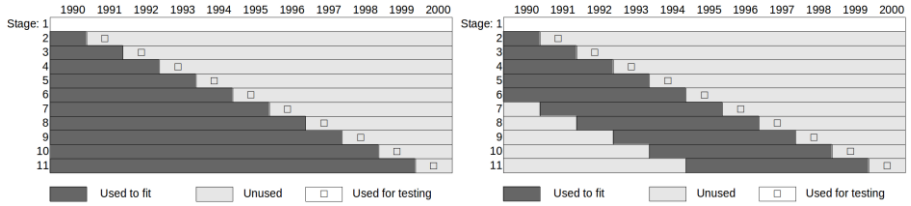

Figure 3-16 The expanding approach (left) and rolling window approach (right)

Source: (Carver, 2015, p. 57)

3.5.1 Possible biases

Except for the need to consider the transaction costs, which can become enormous, especially for strategies with frequent rebalancing, there are other possible pitfalls which are less known.

Look ahead bias

Look-ahead bias occurs when using information or data in a study or simulation that would not have been known or available during the period under analysis. This can happen even unintentionally! For instance, when defining a dataset, students often choose stocks that they "like". However, the problem arises because they "like" stocks that performed exceptionally well in the past, leading to an overestimation of potential outcomes.

A common example is including Tesla in a dataset and analyzing the period from 2019 to 2022. However, at the beginning of 2019, would anyone have considered Tesla as

an option to include in their portfolio? Certainly not, as they would not have even known that it existed.

The suggested approach to avoid look-ahead bias is to precisely define the dataset in a manner consistent with what would have been known at the beginning of the out-of-sample period. The preferred option is to use index components.

Survivorship bias

Even using the index components, our decisions can still be biased if we take the actual composition of the index, which is, however, not known during the analyzed period. This is related to a ***survivorship bias***. The correct approach would be to take the index composition on the last day of the in-sample period, which would make the analysis nonbiased. Survivorship bias is related to the look-ahead bias discussed in the previous section.

Question 3-12 Survivorship bias

Search online for an explanation of the survivorship bias.

In finance, survivorship bias refers to the tendency to exclude failed companies from studies due to their inexistence. This bias frequently skews the results of performance studies toward higher values, since only companies successful enough to survive until the end of the analyzed period are included in the analysis. For instance, the actual composition of an index represents only those stocks that survived, i.e., the companies that have not gone bankrupt or faced other serious issues. A comparable example can be found in the measurement of mutual funds (Elton et al., 1996). This issue was also popularized by Taleb (2007, 2008).

3.5.2 Simple two-period backtest

An example of the simple approach to backtesting is illustrated in Code 3-24 and Code 3-25. The code is intentionally split into two parts corresponding to the in-sample and out-of-sample periods. Note that in the first code, we work with the dataframe *inprices*, while in the latter we work with the dataframe *outprices*.

In the snippet Code 3-24, we perform portfolio optimization using the *Pyportfolioopt* package. The data consist of daily closing prices for stocks from the S&P 500 index downloaded from Yahoo Finance. We divide the data into an in-sample period (from January 1, 2012, to January 1, 2017) and an out-of-sample period (from January 1, 2017, to January 1, 2023).

First, we download the list of ticker symbols for the S&P 500 companies. Next, using the *Yfinance* package, we obtain historical stock prices of companies in the S&P 500 index. We filter out stocks with missing data and split the dataset into in-sample and out-of-sample periods.

The portfolio is constructed using various optimization strategies. The naive strategy assigns equal weights to all assets. We calculate the expected returns and the covariance matrix on the basis of historical data. Three portfolio optimization strategies are implemented: maximizing the Sharpe ratio, minimizing variance, and minimizing Conditional Value at Risk at a confidence level of 95%.

The weights of each strategy are stored in a dataframe *weights*. The resulting weights are then visualized using a stacked bar graph, see Figure 3-17, providing insights into the composition of the optimized portfolios.

Note that the results are survivorship-biased, and it is suggested to perform the analysis using the index composition at the end of the in-sample period. However, to make the code clear, we illustrated the process using this simplification.

Code 3-24 Estimation of portfolio weights

```
import yfinance as yf
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import pypfopt

# in-sample period from 2012-01-01 to 2017-01-01
# out-of-sample period from 2017-01-01 to 2023-01-01

# Just a note: in this example, our analysis suffers from the so-called survivorship bias
because we take the composition of the index (and thus our database) from the out-of-sample
period
# A better approach would be to take the composition of the DJIA index on the last day of
the in-sample period. This would make the analysis survivorship-bias-free.

# First download the list of tickers in DJIA, see Code 2-1
url = 'https://en.wikipedia.org/wiki/List_of_S%26P_500_companies'
tlSP500 = pd.read_html(url)[0]['Symbol'].tolist()

## alternatively download the csv file from https://stockmarketmba.com/stocksinthedjia.php
and save it as 'DJIAlist.csv'
# tclist = pd.read_csv("DJIAlist.csv")['Symbol'].dropna().tolist() # load the csv into the
Pandas Dataframe

# Now we download the data using yfinance package and keep only the Close prices (already
adjusted)
data = yf.download(tickers = tlSP500, start='2012-01-01', end='2023-01-01', interval = '1d',
group_by = 'column', auto_adjust = True)
data = data["Close"].dropna(axis=1, inplace=False) # we drop the symbols with missing data
# we split the data into in-sample and out-of-sample parts, see above
inprices = data[:'2017-01-01'] # the in-sample prices, on these we calculate the weights
outprices =  data['2017-01-01':] # the  out-of-sample prices, on these we check the
performance

# define a function to return a vector of 1/n weights, see Code 4-10
def port_naive(prices):
    n = prices.shape[1] # obtain the number of the columns (assets in prices DataFrame)
    weights = [1/n for _ in range(n)]
    return weights

# create weights dataframe to stroe the weights of strategies, assign the weights of naive
strategy
weights = pd.DataFrame(index=inprices.columns, columns=['naive'], data=port_naive(inprices))

# calculate the expected returns and covariance matrix,
# see also other possibilities on how to improve the estimates
mu = pypfopt.expected_returns.mean_historical_return(inprices) # to improve the forecast see
also:
#mu = pypfopt.expected_returns.ema_historical_return()
#mu = pypfopt.expected_returns.capm_return()

Q = pypfopt.risk_models.risk_matrix(inprices) # to improve the forecast see the 'method'
parameter or pypfopt.risk_models-CovarianceShrinkage(data)
Q = pypfopt.risk_models.fix_nonpositive_semidefinite(Q) # Check the covariance matrix and if
not positive semidefinite, fix it.
```

```
# calculate the weights of the maximum Sharpe ratio portfolio
ef = pypfopt.efficient_frontier.EfficientFrontier(mu, Q) # create the Efficient Frontier
object
weights['max Sharpe'] = pd.Series(ef.max_sharpe()) # add the vector of weights to DataFrame

# calculate the weights of the minimum variance portfolio
ef = pypfopt.efficient_frontier.EfficientFrontier(mu, Q) # create new Efficient Frontier
object
weights['min variance'] = pd.Series(ef.min_volatility())  # add the vector of weights of the
minimum variance portfolio to DataFrame

# calculate the minimum-CVaR portfolio (for a confidence level of 95%)
ef = pypfopt.efficient_frontier.EfficientCVaR(mu, returns=inprices.pct_change().dropna(),
beta=0.95) # create the Efficient Frontier object
weights['min CVaR'] = pd.Series(ef.min_cvar())  # # add the vector of weights of minimum-
CVaR portfolio

# the function ef.clean_weights() can be used to get the weights „rounded“ (to getr id of
the small weights
#weights['min CVaR'] = pd.Series(ef.clean_weights())

# plot the figure with the weights distribution
weights.transpose().plot.bar(stacked=True, legend=False)
plt.title('Weights') # add the title to the figure
plt.ylabel('Weight') # set the label of y axis
#plt.xlabel('Portfolio') # set the label of y axis
# the legend is missing as 29 stocks make the figure very unclear
fig = plt.gcf()
fig.set_size_inches(10, 3)  # Adjust the size of the current figure
plt.show()
```

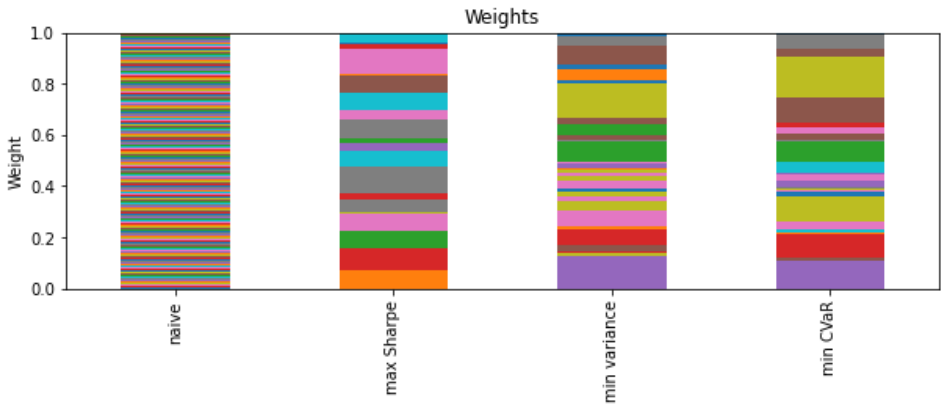

Figure 3-17 Obtained portfolio weights

Question 3-13 Rewrite Code 3-24

In line with Section 3.3.5, rewrite Code 3-24 to utilize *Riskfolio-Lib* instead of *PyPortfolioOp*.

Snippet Code 3-25 is the continuation of the approach. In the code, the wealth paths are computed using gross cumulative returns as shown in Code 3-6. These wealth paths are then plotted, see the resulting Figure 3-18. The code which defines three functions, each serving to compute different financial performance metrics, as discussed in Subchapter 3.2, was omitted, but can be found in Code 3-10, Code 3-11, and Code 3-12.

All functions expect a dataframe *prices* with a single column representing historical daily prices and the annual risk-free rate. Concretely, we calculate the maximum drawdown, the Calmar ratio, and the Sharpe ratio.

The functions are then applied to evaluate the performance of different investment strategies. The code calculates and prints the final wealth, maximum drawdown, Sharpe ratio, and Calmar ratio for each strategy (naive, maximum Sharpe ratio, minimum variance, and minimum CVaR). The results are displayed, providing a comprehensive analysis of the risk-adjusted returns and drawdowns for each investment strategy.

Code 3-25 Measurement of optimized portfolio performances

```
# calculate the wealth path evolutions utilizing gross cumulative returns shown in Code 4 6
wealth = pd.DataFrame(columns=['naive', 'max Sharpe', 'min variance', 'min CVaR'])
wealth['naive'] = ((outprices.pct_change().fillna(0) + 1).cumprod()).dot(weights['naive'])
wealth['max Sharpe'] = ((outprices.pct_change().fillna(0) + 1).cumprod()).dot(weights['max
Sharpe'])
wealth['min variance'] = ((outprices.pct_change().fillna(0) + 1).cumprod()).dot(weights['min
variance'])
wealth['min CVaR'] = ((outprices.pct_change().fillna(0) + 1).cumprod()).dot(weights['min
CVaR'])

# plot the wealth path evolutions
plt.figure(figsize=(10, 6))
plt.plot(wealth)
plt.title('Wealth paths') # add the title to the figure
plt.ylabel('Portfolio Value') # set the label of y-axis
plt.xlabel('Time') # set the label of x-axis
plt.legend(wealth.columns)
plt.show()

#def calc_MDD(prices):
#def calc_Calmar(prices, rfr=0):
#def calc_Sharpe(prices, rfr=0):
#see codes: Code 3-10, Code 3-11, Code 3-11

for col in wealth.columns:
    print(f"Final wealth {col}: ", wealth[col].iloc[-1])

for col in wealth.columns:
    print(f"Maximum drawdown of {col}: ",  calc_MDD(wealth[col]))

for col in wealth.columns:
    print(f"Sharpe ratio of {col}: ",  calc_Sharpe(wealth[col]))

for col in wealth.columns:
    print(f"Calmar ratio of {col}: ",  calc_Calmar(wealth[col]))

> Final wealth naive:  2.1491815450949723
Final wealth max Sharpe:  2.167415904647871
Final wealth min variance:  2.0744746912195673
Final wealth min CVaR:  2.184985948280671
Maximum drawdown of naive:  0.36507489043751085
Maximum drawdown of max Sharpe:  0.3661742330394172
Maximum drawdown of min variance:  0.3049886122784261
Maximum drawdown of min CVaR:  0.3109395253532937
Sharpe ratio of naive:  0.7256918280523825
Sharpe ratio of max Sharpe:  0.725691296986577
Sharpe ratio of min variance:  0.8282125513914923
Sharpe ratio of min CVaR:  0.8756402539954004
Calmar ratio of naive:  0.37305561308719865
Calmar ratio of max Sharpe:  0.37631361090376075
Calmar ratio of min variance:  0.42462090502141925
Calmar ratio of min CVaR:  0.4480950413993361
```

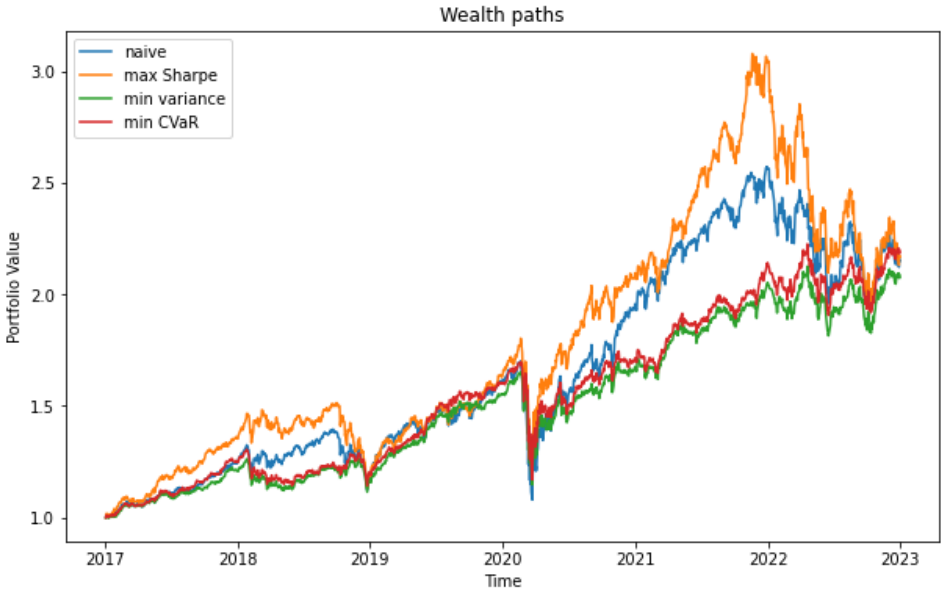

Figure 3-18 Portfolio wealth paths

3.5.3 Rolling-window approach

An example of the rolling-window approach in portfolio optimization is demonstrated in the provided code snippet Code 3-26. This code begins by retrieving the symbols of S&P 500 companies from Wikipedia, as illustrated in Code 1-2. Subsequently, it proceeds to download historical stock prices from Yahoo Finance for the specified period (2012-2023), as illustrated in Code 1-1.

The important part of the code involves iterating over the years 2017-2022 (please note that the cycle excludes the last year, that is why we write 2022+1). Within each iteration of the cycle, the in-sample data are assigned, encompassing the previous five years before the year specified in the variable *year*. These historical data are used for parameter estimation.

Simultaneously, the out-of-sample period is defined by the year specified in the variable *year* and serves as the timeframe to which the weights are assigned. Portfolio optimization is carried out using the *EfficientCVaR* function from the *Pyportfolioopt* package. It is worth noting that any other model can be employed, as demonstrated in Code 3-24. When the cycle is completed, the code generates a dataframe named *weights*, where the calculated weights are stored.

Portfolio performance evaluation is carried out across the entire out-of-sample data range (2017-2022). This evaluation employs a similar approach to the one discussed in the code snippet Code 3-7. Please note that while the adjustment of weights resulting from changes in prices occurs (see Code 3-7) for each year within the main cycle, the cumulative product of the portfolio return, calculated as the product of stock returns and stock weights,

is performed after the completion of the cycle. The resulting wealth path is depicted in Figure 3-19.

Question 3-14 Transaction costs

For simplicity reasons, we do not assume transaction costs in Code 3-26. Can you adjust the code so that it assumes transaction costs?

Code 3-26 Example of rolling window approach in portfolio optimization

```
url = 'https://en.wikipedia.org/wiki/List_of_S%26P_500_companies'
tlSP500 = pd.read_html(url)[0]['Symbol'].tolist()

data = yf.download(tickers = tlSP500, start='2012-01-01', end='2023-01-01', interval = '1d',
group_by = 'column', auto_adjust = True)
data = data["Close"].dropna(axis=1, inplace=False) # we drop the symbols with missing data
weights = pd.DataFrame(columns = data.columns)

for year in range(2017, 2022 + 1):
    inprices = data[f'{year-5}-01-01':f'{year}-01-01']
    outprices = data[f'{year}-01-01':f'{year}-12-31']
    outreturns = outprices.pct_change().fillna(0)
    # calculate expected returns and covariance matrix
    mu = pypfopt.expected_returns.mean_historical_return(inprices)
    Q = pypfopt.risk_models.risk_matrix(inprices)
    Q = pypfopt.risk_models.fix_nonpositive_semidefinite(Q)

    # create Efficient Frontier object
    #ef = pypfopt.efficient_frontier.EfficientFrontier(mu, Q)
    ef = pypfopt.efficient_frontier.EfficientCVaR(mu, returns=inprices.pct_change().dropna(),
beta=0.95)

    # calculate weights and store them as w for a given year
    w = pd.DataFrame(index=outprices.index, columns=outprices.columns)
    w.iloc[0] = ef.min_cvar() # Set the initial weights for the first date
    # Iterate over each date and recalculate the weights
    for i in range(1, len(outreturns)):
        w.iloc[i] = w.iloc[i-1] * (1 + outreturns.iloc[i-1]) # Calculate the new weights based
on returns
        w.iloc[i] /= w.iloc[i].sum() # Normalize the weights so they sum up to 1
    weights = pd.concat([weights, w], axis=0)

weights = weights.astype(float)
plt.figure(figsize=(10, 6))
plt.stackplot(weights.index, weights.T.values)
plt.ylabel("Weights")
plt.show()

outprices = data['2017-01-01':'2023-01-01']
outreturns = outprices.pct_change().fillna(0)
portfolio_return = outreturns.multiply(weights).sum(axis=1)
wealth = (1+portfolio_return).cumprod()

# plot the wealth path evolutions
plt.figure(figsize=(10, 6))
plt.plot(wealth)
plt.title('Wealth paths') # add the title to the figure
plt.ylabel('Portfolio Value') # set the label of y-axis
plt.xlabel('Time') # set the label of x-axis
plt.grid(True)
plt.show()

print(f"Final wealth: ",  wealth.iloc[-1])
print(f"Maximum drawdow: ",  calc_MDD(wealth))
print(f"Sharpe ratio: ",  calc_Sharpe(wealth))
```

```
print(f"Calmar ratio: ",  calc_Calmar(wealth))
> Final wealth:  1.7532483721703136
Maximum drawdow:  0.3349701226044598
Sharpe ratio:  0.6447873413918007
Calmar ratio:  0.29325833571211973
```

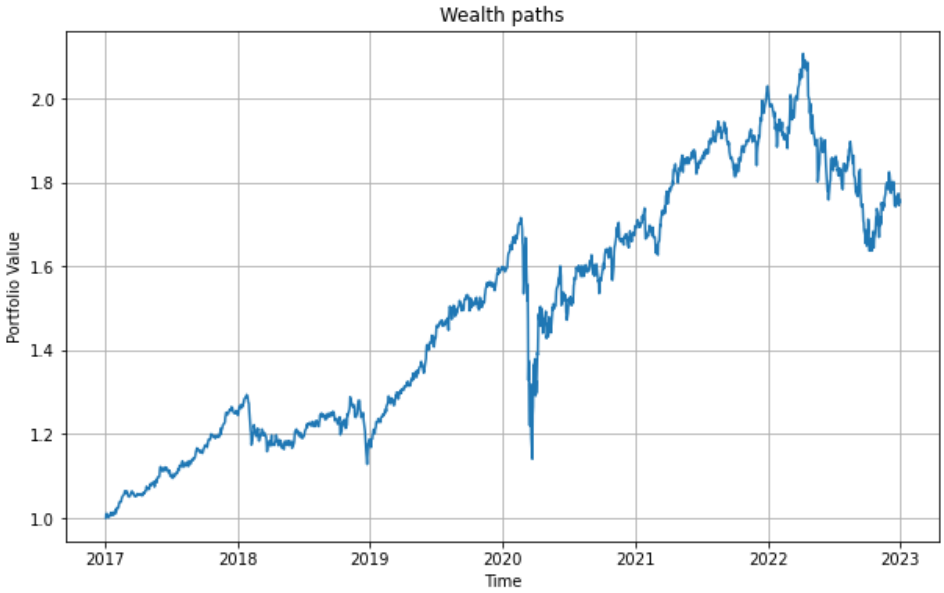

Figure 3-19 Portfolio wealth path

Chapter 4
Technical Analysis and Algorithmic Trading

Technical analysis is a powerful tool that traders and investors use to predict future price movements based on historical patterns and market trends. It is a discipline that applies statistical and mathematical methods to the study of market data, primarily price and volume.

Technical analysis can be used on any security with historical trading data. This includes stocks, futures, commodities, fixed-income securities, currencies, and others. In fact, technical analysis is much more prevalent in commodity and forex markets, where traders focus on short-term price movements. Unlike fundamental analysis, which attempts to evaluate a security's intrinsic value, technical analysis focuses solely on the study of market behavior. Graphical chart patterns, indicators, and statistical analysis are key methods used in technical analysis.

In this chapter, we explore the fundamental concepts of technical analysis and how they can be applied in Python for quantitative finance. In Subchapter 4.1, we look at various technical indicators, chart patterns, and trading strategies that have been used by traders around the world. In Subchapter 4.2 we introduce automated trading systems based on the technical analysis tools with practical applications in Python and we briefly discuss the backtesting procedure and parameter optimizations. The chapter concludes in Subchapter 4.3 with an introduction to Python packages which focuses on technical analysis.

4.1 Basic tools in technical analysis

Technical analysis is a vast field with numerous tools and techniques at its disposal. In this subchapter, we cover some of the fundamental tools used by technical analysts. Technical analysis is grounded in three core assumptions:

1. The first assumption, "***the market discounts everything***", posits that the current price of an asset fully reflects all available and relevant information, thus eliminating the need to consider factors outside of the price movement itself.
2. The second assumption, "***price moves in trends***", is based on the belief that prices follow trends over time, which can be upward (bullish), downward (bearish), or sideways (neutral). Identifying these trends early is a key focus of analysts in making informed trading decisions.

3. The third assumption, "***history tends to repeat itself***" is based on the notion that due to market psychology and collective behavior, there exist patterns in price movements, which tend to repeat over time.

These assumptions collectively form the foundation of technical analysis and explain why technicians believe that technical analysis works.

4.1.1 Charts

Charts are the backbone of technical analysis. They provide a graphical representation of market activity over a specific period. The most common types of charts used in technical analysis are line charts, bar charts, and candlestick charts. Each type of chart can provide unique insight into market trends and patterns.

In Figure 4-1, we show an example of the chart types generated by Code 4-1. In the figure, there are four charts: a line chart (top left), a bar chart (top right), a candlestick chart (bottom left), and a point-and-figure chart (bottom right). All represent the daily price movements of SPY, a well-known ETF tracking the S&P 500 index, in a period of two months.

The ***line chart*** is the simplest type of chart, which plots a series of data points connected by straight line segments. Each point typically represents the closing price of a security in each period. Although the chart is easy to construct and interpret, it is not commonly applied, as it shows only a minimum of information, and some information is even lost, e.g. the lowest price, which can significantly differ from the closing prices, e.g. due to flash crashes[18].

Code 4-1 Generation of SPY price charts

```
import yfinance as yf
import mplfinance as mpf
import matplotlib.pyplot as plt

# Download historical data for desired ticker symbol
ticker = "SPY"
data = yf.download(ticker, start='2022-10-01', end='2022-11-30', interval = '1d', group_by =
'column', auto_adjust = True)

# Create subplots
fig, axs = plt.subplots(2,2, figsize=(19, 10))
# Plot line chart
mpf.plot(data, type='line', ax=axs[0,0], volume=False, show_nontrading=False)
# Plot bar chart
mpf.plot(data, type='ohlc', ax=axs[0,1], volume=False, show_nontrading=False)
# Plot candlestick chart
mpf.plot(data, type='candle', ax=axs[1,0], volume=False, show_nontrading=False)
# Plot PnF chart
mpf.plot(data, type='pnf', ax=axs[1,1], volume=False, show_nontrading=False,
         pnf_params=dict(box_size=2, reversal=3))

# Display the plot
plt.tight_layout()
plt.show()
```

[18] https://en.wikipedia.org/wiki/2010_flash_crash

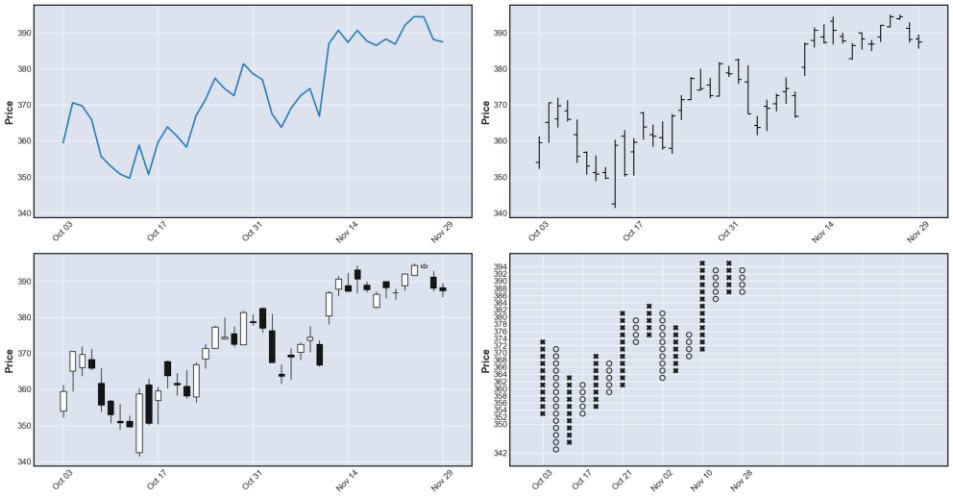

Figure 4-1 An example of the chart types

In the ***bar chart***, each bar represents a period (such as day, week, or hour). The top of the vertical line indicates the highest price for that period, while the bottom of the vertical line indicates the lowest price. A small horizontal line to the left denotes the opening price, and a small horizontal line to the right denotes the closing price. The advantage is that more in-depth information is presented simply and clearly.

The ***candlestick chart*** is similar to a bar chart. Each candlestick represents the open, high, low, and close prices for a given period as specified by the timeframe. If the candlestick is full (usually black or red), it means that the closing price is lower than the opening price, indicating a bearish signal. If the candlestick is hollow (usually white or green), it means that the closing price is higher than the opening price, indicating a bullish signal. The wick above and below the candlestick represents the highest and lowest prices during that timeframe. Compared to the bar chart, the candlestick chart is even more clear and intuitive.

The ***point and figure chart*** is a special, not commonly used, chart type, which focuses on significant price movements and ignores the time factor. Columns alternate between Xs and Os, representing rising and falling prices, respectively. Xs are used when prices are rising, while Os are used when prices are falling. In contrast to the previous types of chart, which are time-sampled, the point and figure chart is price-sampled, i.e. the new price is recorded in dependence on two parameters: the box size and reversal amount. The size of the box is the minimum price change in the direction of the current trend that must take place in order to draw a new box (X or O). The reversal amount is the number of boxes required to cause a reversal in a trend. A reversal would be represented by a movement to the next column and a change in direction. If the amount of reversal increases, the columns corresponding to the less significant trends are removed and it is easier to detect long-term trends. In the chart in Figure 4-1, the box size is 1$ and the reversal amount is 3 boxes.

These charts are tools used in technical analysis to help traders and investors understand price patterns and make informed decisions. Each chart has its strengths and is used in different scenarios, depending on the trader's strategy and the market conditions.

4.1.2 Trends

A trend is the general direction in which a security or market is moving. Identifying trends is one of the key aspects of technical analysis. Trends can be upward (bullish), downward (bearish), or sideways. An *uptrend* is classified as a series of higher highs and higher lows, while a *downtrend* is one of lower lows and lower highs.

When deciding about the trend, one must also keep in mind what period we consider, i.e. how much we zoom in/out the chart. Generally, we can distinguish short-, medium-, and long-term trends, and the trend does not have to be the same in each of them. ***Short-term trends***, ranging from a few days to a few weeks, reveal the immediate fluctuations influenced by daily news and intraday trading activities. ***Medium-term trends***, which span several weeks to months, offer a broader perspective influenced by fundamental factors and evolving market conditions. Medium-term trends are crucial for swing traders and position traders seeking to capture trends that persist beyond short-term noise. ***Long-term trends*** provide a strategic perspective that spans months, years, or even decades. Influenced by fundamental factors such as economic growth and geopolitical events, long-term trends guide buy-and-hold investors and institutional investors in making informed decisions.

To visualize a trend, chartists usually draw a trendline into a graph. The ***trendline*** is a straight line that visually represents the general direction or trend of a series of data points on a chart. In the context of financial markets and technical analysis, trendlines are commonly used to illustrate the overall trajectory of a security's price movement over a specific period. These lines are drawn by connecting consecutive highs or lows in a price chart, creating a visual guide that helps identify the prevailing trend.

In an uptrend, a trendline is drawn by connecting a series of higher lows, highlighting the upward movement in prices. In contrast, in a downtrend, the trendline connects a sequence of lower highs, emphasizing the downward direction of prices.

Question 4-1 Draw trendlines

In Figure 4-1, decide the general trend of SPY and draw an appropriate trendline in each of the graphs.

4.1.3 Chart patterns

Chart patterns play a crucial role in technical analysis. They are patterns that are formed within a chart when prices are graphed. Some of the most common patterns include head and shoulders, channels, double tops and double bottoms, wedges, and triangles. The patterns can be either continuation or reversal patterns, which captures the essence of their predictive nature. ***Continuation patterns*** suggest that the prevailing trend is likely to persist, providing traders with an indication to expect a continuation of the current price movement. On the other hand, ***reversal patterns*** indicate a potential change in the direction of the trend, signaling an opportunity for traders to anticipate a change in market sentiment.

Question 4-2 Chart patterns

Search online for the chart patterns listed above and familiarize yourself with their types (continuation versus reversal).

4.1.4 Support and resistance

Support and resistance levels are fundamental concepts in technical analysis and provide valuable insights into potential price movements. They are key reference points on a price chart that help traders identify areas where buying or selling interest is likely to be concentrated.

Support represents a price level where a downtrend is expected to encounter buying interest, leading to a potential pause or reversal in downward movement. Traders often look for support levels to identify opportunities to enter a long position. ***Resistance*** is a price level where an uptrend is expected to encounter selling interest, leading to a potential pause or reversal in the upward movement. Traders often look for resistance levels to identify opportunities to enter short positions or to close long positions and take profit.

There are various methods on how to identify the resistance and support, some of which include a clear rationale behind them:

- ***Previous lows*** (highs): Examining historical price data to identify levels where the price has historically reversed or bounced.
- ***Psychological levels***: Round numbers or psychologically significant levels often act as support or resistance. For example, for prices ending in 50 or 100, there may be a lot of limit orders waiting to be filled.
- ***Moving averages***: Traders often use moving averages, such as the simple moving average (SMA) or exponential moving average (EMA), to identify dynamic support or resistance levels.
- ***Trendlines***: Drawing trendlines along the lows (highs) of an uptrend (downtrend) can help identify potential support (resistance) levels. When the price approaches this trendline, it may find support (resistance).
- ***Fibonacci retracements***: Traders use Fibonacci retracement levels to identify potential resistance (support) zones based on the ratio of a security's prior move.
- ***Previous support or resistance***: When a support level is broken, it often transforms into a resistance level, and conversely, when a resistance level is breached, it may turn into a support level.

Question 4-3 Support and resistance levels

On the historical chart of the S&P 500 index, try to identify the important support and resistance levels.

4.1.5 Indicators

Technical indicators are mathematical calculations based on the price, volume, or open interest in a security or contract. By analyzing historical data, technical indicators can help predict future price movements or identify oversold and overbought areas. Some of the most widely used technical indicators include moving averages, relative strength index (RSI), and moving average convergence divergence (MACD).

There are also two types of indicator constructions: those that fall in a bounded range and those that do not.

- The ones that are bound within a range are called ***oscillators*** - these are the most common types of indicators. Oscillators have a range, for example, between zero and 100, and they signal periods where the security is overbought (usually near 100) or oversold (usually near zero).
- Nonbounded indicators still form buy and sell signals along with displaying strength or weakness, but they vary in the way they do this.

This corresponds to the two main ways that indicators are used to form buy and sell signals in technical analysis, such as crossovers and divergences.

- ***Crossovers*** are the most popular and are reflected when either the price or another variable (such as, e.g., MACD or RSI) moves through the threshold, its moving average, or when two different moving averages cross over each other.
- The second way indicators are used is through ***divergences***, which occur when the direction of the price trend and the direction of the indicator trend are moving in the opposite trend. This signals that the direction of the price trend is weakening and may reverse.

Moving averages

Moving average (MA) is a technical analysis indicator that is well-known and often used by practitioners. Furthermore, academics use this indicator as a popular example. We can mention, for example, the studies of Anghel (2013) and Stanković et al. (2015). In general, three types of moving averages can be distinguished:

- simple moving average (SMA),
- weighted moving average (WMA),
- exponential moving average (EMA).

For all types of moving averages, it is first necessary to choose the period q from which the moving average is calculated and the time series of prices p_t with which to work. This is essentially the number of past prices that we take into account in the calculation. In the case of a simple moving average, the calculation is the simple arithmetic average of the last q prices,

$$SMA(q)_t = \frac{1}{q} \cdot \sum_{i=t-q+1}^{t} p_i \ . \tag{4.1}$$

In contrast, the weighted moving average is the weighted average of the last q prices,

$$WMA(q)_t = \frac{\sum_{i=t-q+1}^{t} v_{t-i+1} \cdot p_i}{\sum_{i=1}^{q} v_i}, \tag{4.2}$$

where v_i are the weights assigned to each price. Typically, a higher weight is assigned to the most recent prices and a lower weight is assigned to historically distant prices. The last type, the exponential moving average, is calculated using a recurrent calculation,

Table 4-1 Example of MA calculations

Day	Price	SMA(3)	WMA(3;3,2,1)	EMA(3)
1	10	–	–	10,00
2	11	–	–	10,50
3	12	11,00	11,33	11,25
4	15	12,67	13,33	13,13
5	18	15,00	16,00	15,56
6	16	16,33	16,50	15,78
7	14	16,00	15,33	14,89

$$EMA(q)_t = EMA(q)_{t-1} + \frac{2}{q+1} \cdot \left(p_t - EMA(q)_{t-1} \right), \tag{4.3}$$

where the first value is equal to the first price, $EMA(q)_1 = p_1$.

An example of the calculation of these three moving averages is shown in Table 4-1. In this illustrative example, the moving averages are calculated with the period $q = 3$. For the weighted moving average, the weights are 1, 2, and 3.

When working with moving averages, it is important to note the possible error in the analysis, especially if closing prices are used. ***The value of the moving average at time t is known only when the closing price at time t is already known***. If, for example, a moving average is calculated from minute-by-minute data, then the value of the moving average at a particular minute is only known at the end of that minute. Given this, ***the analysis cannot consider buying the asset at this price*** but has to allow for a certain delay. In our case of minute data, for example, the closing price of the following minute should be taken as the reference for the buying/selling price.

The generally recommended rule of thumb is to buy the asset when the price crosses the moving average from below and sell when the price crosses the moving average from above. The most commonly used periods in such a setup are the fifty-day and two-hundred-day moving averages.

However, this trading system based on a single moving average results in a large number of trades, especially at low periods, which, due to the transaction costs, reduces the profitability of the trading system. To filter out false signals and reduce the number of trades, this system can be generalized to use two moving averages: a fast one with a short period and a slow one with a long period. If we denote the period of the fast-moving average *f* and the period of the short-moving average *s*, the trading system can be defined based on the rules for opening and closing a position (signal) as follows:

$$signal_t = \begin{cases} 1 \; if \; MA(f)_t \geq MA(s)_t \text{ and } MA(f)_{t-1} < MA(s)_{t-1} \\ -1 \; if \; MA(f)_t \leq MA(s)_t \text{ and } MA(f)_{t-1} > MA(s)_{t-1} \\ 0 \; otherwise \end{cases} \tag{4.4}$$

The rules for entry and exit are as follows. If the fast-moving average crosses the slow-moving average from below, we enter the long position. If it crosses from above, we

close the long position and, if allowed, take a short position. Thus, we have two possible positions: short (-1, we borrow the asset we sell) and long (1, we buy the asset). We can also consider a neutral position (0), which we take at the beginning of the analysis period before the first crossover occurs. An illustrative example of the system is shown in Figure 4-3. Although this system may seem simple, it is a tested trading system that is used in many studies.

A second possible definition of the system (if we do not consider the initial period before the crossover occurs) is a function defining the current position to be held,

$$position = \begin{cases} 1 \; if \; MA(f)_t > MA(s)_t \\ -1 \; if \; MA(f)_t < MA(s)_t \\ 0 \; if \; MA(f)_t = MA(s)_t \end{cases} \tag{4.5}$$

This represents the automatic trading system (ATS), a well-defined procedure to determine whether to be in a long position (*positon* = 1), short position (*positon* = –1), or neutral position (*positon* = 0*)*. The application of such an ATS can be found in Code 4-2.

Question 4-4 Moving averages

Look at Subchapter 4.2 and try to backtest the system on some real-world data, such as SPY, see Section 4.2.1. Also, discuss the optimization of parameters (the period of fast and slow MA) and potential overfitting.

Moving average converge divergence (MACD)

The Moving Average Convergence Divergence (MACD) indicator was introduced in the 1970s by Gerard Appel. Since then, it has become one of the most widely used indicators in technical analysis. The indicator value is calculated as the difference from two exponential moving averages of closing prices,

$$MACD(f,s)_t = EMA(f)_t - EMA(s)_t, \tag{4.6}$$

where *f* and *s* are the periods of the fast and slow exponential moving averages. The most commonly used periods are 12 and 26, but other periods can be used depending on the particular market.

A positive indicator value detects an uptrend and a negative value detects a downtrend. In this case, we would obtain a trading system similar to the system defined in the previous subsection. However, the more applied system is the one with a signal line. The signal line is determined as the nine-day exponential moving average of the MACD indicator,

$$position_t = \begin{cases} 1 \; if \; MACD(f,s)_t > EMA_{MACD}(9)_t \\ -1 \; if \; MACD(f,s)_t < EMA_{MACD}(9)_t. \\ 0 \; if \; MACD(f,s)_t = EMA_{MACD}(9)_t \end{cases} \tag{4.7}$$

A buy signal is generated if the MACD indicator crosses the signal line from below, and a sell signal is generated if the MACD indicator crosses the signal line from above. Calculating the signal line as an exponential moving average with a period of 9 is

recommended, but again different values can be chosen depending on the specifics of the market.

Bollinger bands

Bollinger bands (BB) were developed in the early 1980s by John Bollinger. In the case of this indicator, it is a plot of three curves on a chart that describes the price evolution. The middle curve represents the medium-term trend, usually expressed as a simple moving average of 20 days. The other two curves represent the mean curve shifted up and down by two standard deviations of the price. These curves measure price volatility.

The calculation of these three curves (upper, middle, and lower) is as follows,

$$BB_u(q,k)_t = SMA(q)_t + k \cdot \sqrt{\sum_{i=t-q+1}^{t} \frac{(p_i - SMA(q)_t)^2}{q}}, \tag{4.8}$$

$$BB_m(q,k)_t = SMA(q)_t, \tag{4.9}$$

$$BB_l(q,k)_t = SMA(q)_t - k \cdot \sqrt{\sum_{i=t-q+1}^{t} \frac{(p_i - SMA(q)_t)^2}{q}}, \tag{4.10}$$

where q denotes the period of the moving average used and k indicates how many standard deviations the upper and lower curves are shifted compared to the mean curve. The most common combination is 20 days (q = 20) and two standard deviations (k = 2). If the return on an asset were a random variable with a normal probability distribution, then the price should move between the lower and upper curves 95% of the time. However, because of the heavy tails of the probability distribution, there is a smaller percentage of cases. Grimes (2012) reported that for most markets, the value is around 88%.

Bollinger bands are a useful tool for displaying market volatility, as shown here by the width of the band formed by the upper and lower curves. When volatility increases, the bandwidth increases, and vice versa. This band also indicates overbought/oversoldness in the market.

Some traders buy when the price touches the lower curve and close the trade when the price touches the moving average, i.e., the middle curve. Other traders buy when the price crosses the lower curve and sell when the price crosses the upper curve. However, it is generally not recommended to use this indicator alone. Even Bollinger himself recommended its use in combination with other indicators.

Relative strength index

The relative strength index (RSI) measures the overboughtness, or oversoldness, of the market. It is credited to trader J. Welles Wilder, who created the indicator to measure the state of the market. The calculation of the RSI is as follows,

$$RSI(q)_t = 100 - \frac{100}{1 + RS(q)_t}, \tag{4.11}$$

$$RS(q)_t = -\frac{\sum_{i=t-q+1}^{t} max(0, p_i - p_{i-1})}{\sum_{i=t-q+1}^{t} min(0, p_i - p_{i-1})}, \tag{4.12}$$

Table 4-2 Example of RSI calculations

Day	*Price*	*RS(4)*	*RSI(4)*
1	12	–	–
2	11	–	–
3	12	–	–
4	15	–	–
5	18	$-\frac{0+1+3+3}{-1+0+0+0}=\frac{7}{1}$	$100-\frac{100}{1+\frac{7}{1}}=87.5$
6	16	$-\frac{1+3+3+0}{0+0+0-2}=\frac{7}{2}$	$100-\frac{100}{1+\frac{7}{2}}=77.78$

where p_i is the closing price and q is the length of the selected period. It is recommended to use a 14-day period, but nine-day and 26-day periods are also popular. The shorter the period, the more the indicator fluctuates and gives more signals. The example of the RS and RSI calculation for q=4 is shown in Table 4-2.

The values of this indicator oscillate between zero and one hundred and if its value is greater than 70, the market is overbought, while if its value is lower than 30, it is oversold. The general recommendation is to sell if the market is overbought (RSI>70) and buy if the market is oversold (RSI<30). It is often recommended to wait until the confirmation - sell if the RSI crosses 70 from above and buy if the RSI crosses 30 from below. However, the rules for closing the trade vary from author to author.

Stochastic Oscillator

The stochastic oscillator is one of the most recognized momentum indicators used in technical analysis. The idea behind this indicator is that, in an upward trend, the price should be closing near the highs of the trading range, signaling upward momentum in the security. In downtrends, the price should close near the lows of the trading range, signaling downward momentum. The stochastic oscillator is plotted within a range of zero and 100 and signals overbought conditions above 80 and oversold conditions below 20.

The stochastic oscillator contains two lines. The first line is the %K, which is essentially the raw measure used to formulate the idea of the momentum behind the oscillator. The second line is the %D, which is simply a moving average of the %K. The %D line is considered to be the more important of the two lines as it is seen to produce better signals. The stochastic oscillator generally uses the past 14 trading periods in its calculation but can be adjusted to meet the needs of the user.

On-Balance Volume

Technical indicators are not limited to price movements and can also be derived from traded volume data. The On-Balance Volume (OBV) indicator stands out as a well-known and straightforward technical tool that reflects changes in traded volume. It is also one of the simplest volume indicators to compute and understand.

Calculating the OBV involves a relatively simple process. For each trading period, the total volume is assigned a positive or negative value based on whether the price moved up or down during that period. If the price increases, the volume is given a positive value;

conversely, a negative value is assigned when the price decreases. These positive or negative volume values are then added to a running total that accumulates from the beginning of the analyzed period.

It is important to focus on the trend in the OBV, which is even more important than its actual value. This measure expands on the basic volume measure by combining volume and price movement.

4.2 Automated trading systems (ATS)

Although the realm of technical analysis is inherently subjective, with each chartist often defining rules in a flexible manner, especially to retroactively fit historical data and yield profitable outcomes, our textbook strives for objectivity. We emphasize concrete, objectively stated rules that can be rigorously tested through backtesting. This approach aligns with the principles elucidated by Aronson (2007), who thoroughly explored the challenges and importance of developing robust technical analysis systems.

In the subsequent section, we present an illustrative example of applying the moving average crossover rule, as discussed in Subchapter 4.2. Our focus extends to addressing the issue of asynchronous trading, elucidating potential pitfalls in the analysis.

Even if an objectively defined system is employed and withstands correct backtesting, challenges such as overfitting, often termed data snooping bias, can emerge. Consider this scenario: presenting a seemingly flawless MA crossover system with stellar results in the trading of the S&P 500. The apparent profitability may be a result of cherry-picking the optimal combination of fast and slow periods from numerous possibilities, see Section 4.2.2. This specific combination could have been determined only after the analyzed period and could not have been applied in the period under consideration, making it impractical for real-time application. Thus, it is advisable not to trust anyone showing you the historical performance as an argument about the technical analysis possibilities.

An additional concern is survivorship bias, as outlined by Taleb (2007). To illustrate, suppose that I send you weekly emails forecasting market movements, accurately predicting up or down movement. Imagine that for the next 8 weeks, I will send you an e-mail in which I will perfectly forecast the next week's movement as up or down. That would sound great making me a forecaster guru, right? However, the truth can be very different.

What if I did it as follows? Initially, I send forecasts to 1024 recipients, half for up and half for down. Subsequently, I continue to work with only those who received the correct forecast the previous week (512), again splitting them for the next forecast, that is, to 256 I send up and 256 down. This process repeats, whittling down the pool until, after eight weeks, only one individual (you) has received eight consecutive correct forecasts. Would it still make me a forecaster guru?

Another Taleb illustration is the idea that, given enough monkeys randomly pressing keyboards, one may eventually write the Odyssey. This is related to Malkiel (1973, p. 24) stating the opinion that "*A blindfolded monkey throwing darts at a newspaper's financial pages could select a portfolio that would do just as well as one carefully selected by experts.*"

4.2.1 Backtesting of automated trading system

In this section, we provide an example of automated trading system backtesting. The system is based on the crossover of two moving averages, see Section 4.1.5. The assumed values of fast- and slow-moving averages are three and twenty-one days, respectively. When backtesting the trading systems, one must be careful about a few important issues.

1) Transaction costs must be taken into account. In the real world, each buying and selling is connected with the transaction costs (such as broker fees, ask-bid spread, slippage, etc.)
2) It must be correctly stated for which price we can buy/sell. The easiest way to make a mistake is when your code assumes that you can buy or sell for the price from which you calculate the trading decision (signal). That is why we shift the position in the Code 4-2.

Code 4-2 ATS of moving averages crossover

```
import yfinance as yf
import numpy as np
import matplotlib.pyplot as plt

# first, we set the parameters of our strategy
FAST = 3 # period of fast MA, in case FAST=1 we have the closing price
SLOW = 21 # period of slow MA
DOLLARTC = 0.05 # transaction costs per one buy/sell transaction
INITIALAMOUNT = 500 # the initial amount of money we start trading with

# first we download the data
data = yf.download(tickers = 'SPY', start='2000-01-01', end='2022-11-01', interval = '1d',
group_by = 'column', auto_adjust = True)

data['Fast'] = data['Close'].rolling(FAST).mean() # we calculate the fast mooving average
data['Slow'] = data['Close'].rolling(SLOW).mean() # we calculate the slow mooving average

conditions = [data['Fast']>data['Slow'], data['Fast']<data['Slow']] # conditions to check
choices = [1, -1] # the position to substitute
# choices = [1, 0] # alternatively we can consider long-only positions
defaultchoice = 0 # position if none of the conditions is evaluated as True, i.e. in case
that fastMA=slowMA or in case of NaN

data['Position'] = np.select(condlist=conditions, choicelist=choices, default=defaultchoice)
# calculate the postion based on conditions

# now, the question is for what price we buy and for what price we sell, see that the
position is calculated based on the closing price, so we probably cannot buy for that price
# let's suppose we cannot buy/sell for the closing price we use to calculate MA
# if we suppose that we CAN buy/sell for the closing price we use to compute MA, comment on
the following line
data['Position'] = data['Position'].shift(1) # we shift the position by one day

data.loc[data.index[-1],'Position'] = 0 # at the end we close all positions
data['Trade'] = data['Position'].diff() # we calculate what should we do, i.e. should we
sell the long position and open short position, or cover short position and buy a long
position
data['CF'] = -data['Close'] * data['Trade'] - DOLLARTC * np.abs(data['Trade']) # we
calculate profit in the currency for trading one share
data['Profit'] = data['CF'].cumsum() + data['Position'] * data['Close']

# calculate profit of B&H strategy
data['Profit B&H'] = data['Close'] - data.loc[data.index[0],'Close'] - DOLLARTC
data.loc[data.index[-1], 'Profit B&H'] = data.loc[data.index[-1], 'Profit B&H'] - DOLLARTC
```

```
# visualize - only last 125 days
# create the figure with 3 plots/graphs
fig, (ax1, ax2, ax3) = plt.subplots(3, sharex=True, figsize=(19.20,10.80))
fig.suptitle('MA trading system') # add the title to the figure
# in the first graph, plot the stock price (close), fast and slow MA
ax1.plot(data[['Close', 'Fast', 'Slow']])
# in the second graph, plot the position and quantity traded (buy/sell)
ax2.plot(data[['Position', 'Trade']])
# in the third graph, plot the cumulative profit
ax3.plot(data[['Profit', 'Profit B&H']])
ax1.set_ylabel('Price & MAs') # set the label of y axis
ax2.set_ylabel('Position & trade signal') # set the label of y axis
ax3.set_ylabel('Total profit') # set the label of y axis
ax1.legend(['Close', 'Fast MA', 'Slow MA']) # add legend
ax2.legend(['Position', 'Buy/Sell']) # add legend
ax3.legend(['Strategy','B & H']) # add legend
plt.tight_layout()
```

In Code 4-2, we present the implementation of a basic moving average crossover trading strategy using Python, incorporating the *Yfinance* package to fetch historical stock data and *Numpy* and *Matplotlib* to analyze and visualize. For further insights into the visualization aspect, please refer to Subchapter 1.3. The breakdown of the code is as follows:

- Lines 1-4: We import the necessary packages, including *Yfinance* for financial data, *Numpy* for numerical operations, and *Matplotlib* for plotting.
- Line 6: We set parameters for the MA strategy, defining the periods for fast- and slow-moving averages, transaction costs, and initial trading capital.
- Lines 8-14: Using *yfinance.download*, we fetch historical data for the S&P 500 ETF (SPY) from January 1, 2000, to November 1, 2022, with a daily interval.
- Lines 16-17: We calculate the fast- and slow-moving averages based on the closing prices of the stock, utilizing the *rolling.mean* function from *Numpy*.
- Lines 19-23: Trading positions (long or short) are determined based on the crossover of fast- and slow-moving averages using *np.select* from *Numpy*. ***Positions are shifted by one day.***
- Lines 25-29: Trade signals, cumulative profits, and a Buy & Hold (B&H) strategy comparison are calculated, considering transaction costs.
- Lines 31-38: Visualization is carried out using *Matplotlib*, with three graphs representing stock prices, MA crossovers, and cumulative profits.
- Lines 40-45: Labels, legends, and other finishing touches are added to improve the readability of the plot. The final plot is displayed using *plt.show()*.

This code serves as a practical illustration of a straightforward MA trading strategy, providing both a numerical and a visual representation of its performance. The comparison with a Buy & Hold strategy offers additional insight into the strategy's effectiveness. For the output and visual representation, see Figure 4-2. As is obvious, the trading system does not outperform a buy-and-hold strategy in terms of final wealth.

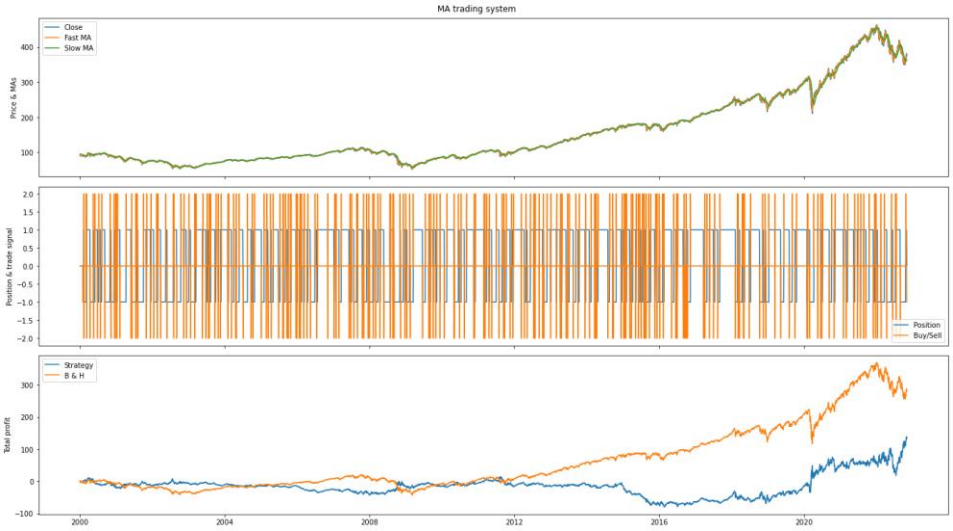

Figure 4-2 The figure generated by Code 4-2

Question 4-5 Other metrics to judge the performance of the trading system

What other metrics can be used to evaluate and compare the performance of the trading system? Are any ratios from Sections 3.2.2 and 3.2.3 applicable?
Search online and explain the measures: risk reward ratio and percentage of profitable trades. In what relation must the values of these two be?

To better illustrate the relations between price, moving averages, position, and trading signal, in Code 4-3 we plot only the last 125 days of our analysis. The resulting figure is depicted in Figure 4-3.

Code 4-3 Visualization of only last 125 days

```
# visualize - only last 125 days
# create the figure with 3 plots/graphs
fig, (ax1, ax2, ax3) = plt.subplots(3, sharex=True, figsize=(19.20,10.80))
fig.suptitle('MA trading system') # add the title to the figure
# in the first graph, plot the stock price (close), fast and slow MA
ax1.plot(data[['Close', 'Fast', 'Slow']][-125:])
# in the second graph, plot the position and quantity traded (buy/sell)
ax2.plot(data[['Position', 'Trade']][-125:])
# in the third graph, plot the cumulative profit
ax3.plot(data[['Profit', 'Profit B&H']][-125:])
ax1.set_ylabel('Price & MAs') # set the label of y axis
ax2.set_ylabel('Position & trade signal') # set the label of y axis
ax3.set_ylabel('Total profit') # set the label of y axis
ax1.legend(['Close', 'Fast MA', 'Slow MA']) # add legend
ax2.legend(['Position', 'Buy/Sell']) # add legend
ax3.legend(['Strategy','B & H']) # add legend
plt.tight_layout()
```

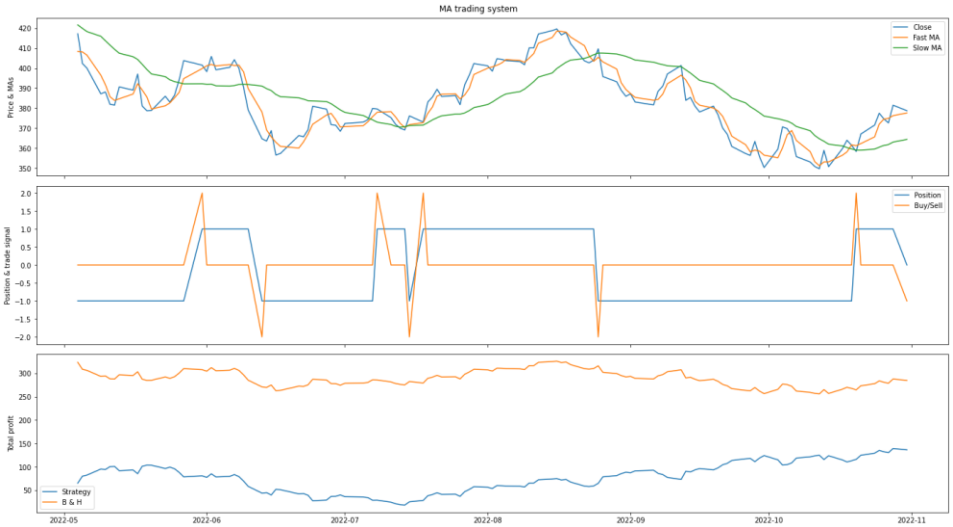

Figure 4-3 The figure generated by Code 4-3

4.2.2 Parameters optimization

As we have seen in the previous section, the moving average crossover ATS with given parameter values was not profitable. Logically, one would be interested in answering the question of what combination of fast and slow periods would yield a profitable trading system. A straightforward example of such an analysis in Python is given in Code 4-4. The generated figure is depicted in Figure 4-4.

As is obvious from Code 4-4, we wrapped Code 4-2 into a function *mastrategy*, which takes two inputs: the periods of fast and slow MA and returns the calculated profit of such strategy. The main script then represents just two nested cycles, which evaluate all possible combinations of values of fast- and slow-moving averages.

Code 4-4 Parameters optimization of the MA crossover ATS

```
import yfinance as yf
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl

DOLLARTC = 0.05

data = yf.download(tickers = 'INTC', start='2000-01-01', end='2022-11-01', interval = '1d',
group_by = 'column', auto_adjust = True)

def mastrategy(data,fast,slow):
    data['Fast'] = data['Close'].rolling(fast).mean()
    data['Slow'] = data['Close'].rolling(slow).mean()

    conditions = [data['Fast']>data['Slow'], data['Fast']<data['Slow']] # conditions to
check
    choices = [1, -1] # the position to substitute
    # choices = [1, 0] # alternatively we can consider long-noly postions
    defaultchoice = 0

    data['Position'] = np.select(condlist=conditions, choicelist=choices,
default=defaultchoice)
```

```
    data['Position'] = data['Position'].shift(1) # we shift the position by one day
    data.loc[data.index[-1],'Position'] = 0 # at the end we close all positions
    data['Trade'] = data['Position'].diff() # we calculate what should we do, i.e. should we
sell the long position and open a short position, or cover the short position and buy a long
position
    data['CF'] = -data['Close'] * data['Trade'] - DOLLARTC * np.abs(data['Trade'])
    data['Profit'] = data['CF'].cumsum() + data['Position'] * data['Close']

    return data.loc[data.index[-1], 'Profit']

profits = np.zeros(shape=(50,200))
for f in range(1,50):  # we will try all values in interval <1,50)
    for s in range(1,200): # we will try all values in interval <1,200)
        profits[f,s] = mastrategy(data=data,fast=f,slow=s)
        print(f"Strategy: fast={f}, slow={s} calculated.")

fig, ax = plt.subplots(figsize=(19.20,10.80))
im=ax.imshow(profits[1:,1:], cmap=mpl.colormaps['RdYlGn'])
ax.set_xlabel('period of slow MA') # set the label of x axis
ax.set_ylabel('period of fast MA') # set the label of y axis
fig.colorbar(im, location='bottom')
plt.tight_layout()
plt.savefig("chapter 4 - MAcrossover - combinations.pdf", dpi=500)
plt.savefig("chapter 4 - MAcrossover - combinations.png", dpi=500)
plt.show()
```

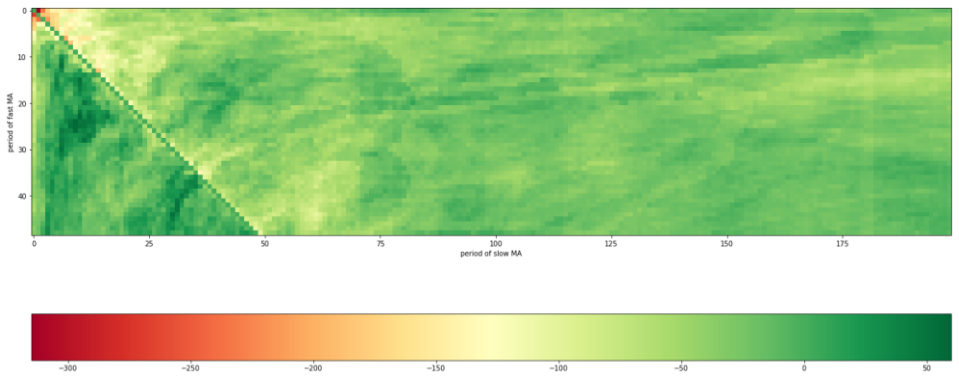

Figure 4-4 The figure generated by Code 4-4

Examining the results, it is evident that the optimal combination consists of a fast-moving average with a 24-day period and a slow-moving average with an 11-day period, yielding a profit of $59.9. In particular, this maximum profit still falls short of the buy-and-hold strategy, as illustrated in Figure 4-2. Additionally, there are also other noteworthy results on which we can illustrate the correct approach to verification and interpretation of the results.

Firstly, it is crucial to acknowledge that the fast-moving period exceeds that of the slow-moving average, which contradicts our initial trading logic outlined in Section 4.1.5. Although these results can be rationalized by considering the INTC stock's tendency towards mean-reverting behavior rather than trend-following, they still deviate from our original rationale. Therefore, caution is advised in drawing a proper conclusion.

Moreover, it is important to recognize that these results cannot be extrapolated into the future. The challenge lies in the inability to use the same dataset for both optimizing parameters and predicting the performance of such an optimized system. For a comprehensive evaluation, adopting one of the backtesting approaches discussed in Carver (2015), as depicted in Figure 3-16, or alternatively, the K-folds approach commonly applied in machine learning problems, shown in Figure 3-15, is recommended. Alternatively, a very simple and straightforward method involves splitting the dataset only into two periods: an in-sample period for parameter optimization, and an out-of-sample period for evaluating the performance of the optimized automated trading system. These concepts are also applicable to backtesting portfolio optimization, as discussed in Subchapter 3.5.

4.3 Technical analysis packages in Python

As is evident, basic technical analysis techniques can be executed using the *Pandas* and *Numpy* packages, complemented by the *Mplfinance* package for more sophisticated financial data visualization. However, there are additional specialized packages that can be beneficial. Examples include the *Ta-Lib*, *Ta*, and *Backtrader* packages, which are introduced in greater detail in the following sections. Other useful packages are *Zipline* and *Pyfolio*.

4.3.1 *Ta-Lib* package

Ta-Lib is a comprehensive package that offers a wide range of functions for technical analysis. It covers an extensive set of indicators, oscillators, and pattern recognition tools, allowing users to perform in-depth analysis and make informed trading decisions.

One potential drawback of *Ta-Lib* is that it needs to be installed separately, which could be a minor inconvenience for users who are new to Python packages and wish for a more streamlined setup process.

An example of an application is depicted in Code 4-5, which utilizes the *Ta-Lib* package to analyze historical stock data for Intel Corporation (INTC) obtained through Yahoo Finance. It calculates the 50-day and 200-day moving averages, as well as the relative strength index with a period of 14 days. The code then identifies the overbought (RSI > 70) and oversold (RSI < 30) regions and visualizes the results in two subplots. The top graph displays the stock's closing price along with the 50-day and 200-day MAs, providing insight into trend directions. The bottom graph focuses on the RSI, highlighting overbought and oversold regions.

Code 4-5 Calculation of MAs and RSI using *Ta-Lib*

```
import talib
import yfinance as yf
import matplotlib.pyplot as plt

# Download historical stock data
data = yf.download(tickers = 'INTC', start='2022-01-01', end='2022-11-01', interval = '1d',
group_by = 'column', auto_adjust = True)

# Calculate 50-day and 200-day moving averages
data['MA_50'] = talib.SMA(data['Close'], timeperiod=50)
data['MA_200'] = talib.SMA(data['Close'], timeperiod=200)
```

```
# Calculate RSI
data['RSI'] = talib.RSI(data['Close'], timeperiod=14)

# Identify overbought and oversold conditions
data['Overbought'] = (data['RSI'] > 70).astype(int)
data['Oversold'] = (data['RSI'] < 30).astype(int)

# Plotting
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(12, 8), sharex=True,
gridspec_kw={'height_ratios': [2.5, 1]})

# Top graph: Stock price and moving averages
ax1.plot(data['Close'], label='AAPL Close Price')
ax1.plot(data['MA_50'], label='50-day MA')
ax1.plot(data['MA_200'], label='200-day MA')
ax1.set_title('AAPL Price and Moving Averages')
ax1.legend()

# Bottom graph: RSI
ax2.plot(data['RSI'], color='orange', label='RSI')
ax2.fill_between(data.index, y1=70, color='red', alpha=0.2, label='Overbought')
ax2.fill_between(data.index, y1=30, color='green', alpha=0.2, label='Oversold')
ax2.set_title('Relative Strength Index (RSI)')
ax2.legend()

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

4.3.2 *Ta* package

The *Ta* package is a Python package designed to simplify the process of performing technical analysis on financial time series data. This package provides a collection of functions to calculate various technical indicators commonly used in trading strategies.

When comparing the *Ta* package with *Ta-Lib* package, the *Ta* package is notable for its user-friendly interface and straightforward installation using pip. Conversely, *Ta-Lib*, renowned for its advanced analysis tools, requires a more involved installation process with additional dependencies. The choice depends on user preferences, with the *Ta* package offering ease of use and quick integration, while *Ta-Lib* provides a comprehensive suite of tools at the cost of a potentially more complex setup.

The example of using the *Ta* package to calculate OBV and RSI is shown in Code 4-6. In this Python code, historical stock data for Apple Inc. (AAPL) is downloaded using the *Yfinance* package, spanning January 1, 2023, to January 1, 2024. Then, the *Ta* package is used to calculate the on-balance volume and the relative strength index. The results are visualized in two subplots. The upper subplot illustrates the closing price of the AAPL along with the OBV, providing information on price trends and volume movements. The bottom subplot focuses on the RSI, indicating potential overbought or oversold conditions.

Code 4-6 The example of calculation of on-balance volume and RSI with *Ta* package

```
import yfinance as yf
import ta
import matplotlib.pyplot as plt

# Download historical stock data
data = yf.download('AAPL', start='2023-01-01', end='2024-01-01')

# Calculate On-Balance Volume (OBV) and Relative Strength Index (RSI)
data['OBV'] = ta.volume.on_balance_volume(data['Close'], data['Volume'])
```

```
data['RSI'] = ta.momentum.RSIIndicator(data['Close']).rsi()

# Create subplots
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(10, 8), sharex=True,
gridspec_kw={'height_ratios': [2.5, 1]})

# Plot closing prices and OBV on the top subplot
ax1.plot(data['Close'], label='AAPL Close Price', color='blue')
ax1.set_ylabel('Close Price', color='blue')
ax1.legend(loc='upper left')

ax1_2 = ax1.twinx()
ax1_2.plot(data['OBV'], label='On-Balance Volume (OBV)', color='orange')
ax1_2.set_ylabel('On-Balance Volume (OBV)', color='orange')
ax1_2.legend(loc='upper right')

# Plot RSI on the bottom subplot
ax2.plot(data['RSI'], label='Relative Strength Index (RSI)', color='green',
linestyle='dashed')
ax2.set_xlabel('Date')
ax2.set_ylabel('RSI', color='green')
ax2.legend(loc='upper right')

# Show the plot
plt.title('AAPL Stock Price with On-Balance Volume (OBV) and RSI')
plt.show()
```

4.3.3 *Backtrader* package

Backtrader (Rodriguez, 2023) is a versatile and user-friendly Python package designed for developing and testing algorithmic trading strategies. With its extensive features, *Backtrader* empowers traders and developers to simulate and analyze trading strategies using historical data.

In Code 4-7, a simple moving average (SMA) crossover strategy is implemented using the *Backtrader* package for backtesting financial strategies. The strategy involves two SMAs with periods of 10 and 30 days, determining buy and sell signals based on their crossover. Historical stock data for Apple Inc. (AAPL) is obtained from Yahoo Finance using the *Yfinance* package. The backtest is conducted on these data, and the strategy performance is visualized with a plot, showcasing the starting and ending portfolio values. The code demonstrates the application of *Backtrader* for the development and evaluation of strategies in a financial context.

Code 4-7 Backtesting example with *Backtrader*

```
import backtrader as bt
import yfinance as yf

class SmaCross(bt.Strategy):
    pfast=10  # Period for the fast moving average
    pslow=30   # Period for the slow moving average

    def __init__(self):
        sma1 = bt.ind.SMA(period=self.pfast)
        sma2 = bt.ind.SMA(period=self.pslow)
        self.crossover = bt.ind.CrossOver(sma1, sma2)

    def next(self):
        if not self.position:
            if self.crossover > 0:
                self.buy()
        elif self.crossover < 0:
```

```
            self.close()

cerebro = bt.Cerebro() # Create a Cerebro engine
dataframeAAPL = yf.download('AAPL', start='2000-10-01', end='2024-11-30', interval = '1d',
group_by = 'column', auto_adjust = True)
data = bt.feeds.PandasData(dataname=dataframeAAPL)
cerebro.adddata(data) # Add the data feed to Cerebro
cerebro.addstrategy(SmaCross) # Add the strategy to Cerebro
cerebro.broker.set_cash(100000) # Set the initial cash amount for the backtest
print('Starting Portfolio Value:', cerebro.broker.getvalue()) # Print the starting cash
amount
cerebro.run() # Run the backtest
print('Ending Portfolio Value:', cerebro.broker.getvalue()) # Print the final cash amount
after the backtest
cerebro.plot() # Plot the results with a single command
```

Chapter 5
Options

In Chapter 5, we immerse ourselves in the intricate world of financial options, exploring the various models and methodologies used for their valuation. Options, as financial derivatives, provide investors with unique opportunities to manage risk or speculate on market movements. This chapter serves as an introductory guide to understanding and valuing options, with a focus on both theoretical foundations and practical implementation.

Subchapter 5.1 begins our exploration with an introduction to options, explaining the basics. Then, in Subchapter 5.2, we delve into the Black-Scholes valuation formula for the valuation of European options. We then introduce two numerical approaches, specifically applicable to the valuation of American options. In Subchapter 5.3, we introduce lattice models, specifically the binomial option pricing model. In Subchapter 5.4, we take a look at the Monte Carlo simulation and its application in the valuation of European options and the least squares Monte Carlo method for the valuation of American options.

By combining theoretical insights with practical examples, this chapter equips the reader with the knowledge and skills necessary to navigate the world of option trading and valuation.

5.1 Option introduction

Options are essentially a contract whose terms bind one side (the seller) to a tight position and the other (the buyer) to a loose position. By issuing an option, the seller promises to sell (call option) or buy (put option) the ***underlying asset*** on which the option is issued from the option buyer at a predetermined price (***strike price*** or ***exercise price***) and date if the buyer requests so. By purchasing an option, the buyer acquires the option to buy (call option) or sell (put option) the underlying asset at a predetermined price in the future.

Depending on when the buyer can make his request to buy/sell the asset, we can distinguish:

- ***American-style options*** can be exercised by the buyer at any time from purchase until maturity.
- ***European-style options*** can only be exercised at maturity.
- ***Bermudan-style options*** can be exercised in a finite number of predetermined time periods during its lifetime.

The most common and most traded options are American-style options.

Depending on the complexity of the ***payoff function***, we distinguish between simple options (referred to as plain vanilla) and exotic options. The payoff function of ***plain vanilla options*** is one of the simplest:

$$\psi = max(0; S_T - K) \tag{5.1}$$

for a plain vanilla call, and

$$\psi = max(0; K - S_T) \tag{5.2}$$

for a plain vanilla put, where K is the strike price, i.e. the price for which the underlying asset can be bought or sold, S_T is the price of the underlying asset at maturity and ψ is the payoff at maturity.

On the other hand, there are many types of ***exotic options***, each of which has an individual complex payoff function. Examples of exotic options include barrier options, Asian options, and digital options, among others. Barrier options have a distinctive feature, where the payoff depends on whether the underlying asset's price crosses a predefined barrier during the option's life. Asian options derive their payoff from the average price of the underlying asset over a specified period. Digital options, on the other hand, have a binary payoff based on whether the asset's price at expiration is above or below a predetermined level. However, what is common for all options is that the buyer has a right, not a duty, to exercise the option.

For simplicity, we further limit ourselves to plain vanilla options only. It can be seen that the payoff of both the call and the put options is always nonnegative. This means that the buyer of an option does not incur any risk of financial expenditure by purchasing the option other than the payment of the ***option premium*** (the option price). Profit can be computed simply as payoff minus the option premium.

The dependence of payoff and profit on the price of the underlying asset at maturity is shown in Figure 5-1. In the figure, we assume that the strike price is K = \$100, the option premium is 5\$, and we plot the payoffs and profits for the price of the underlying asset S_T in the range of \$80 to \$120. As is obvious from the payoff functions, for the buyer (long position), there is a positive payoff for underlying asset prices below (put option) or above (call option) strike price. The profit is shifted down by the option premium; thus, we can observe the break-even point for the buyer (long position) at 95\$ (put option) and 105\$ (call option). The profit of the put option is limited to 95\$ as the strike price minus the option premium. The profit of the call option is theoretically unlimited. The above description is from the point of view of the buyer of the option. For the seller (short position), or option underwriter, the payoff and profit are the same with the opposite sign. **Note that the potential loss of the underwritten call option is unlimited**.

The value of an option consists of an intrinsic value and a time value. The *intrinsic value* is defined as the benefit from the immediate exercise of the option and therefore corresponds to the payoff function. The intrinsic value also allows for the division of options into in-the-money (ITM), at-the-money (ATM), and out-of-the-money (OTM) options. *In-the-money options* have a positive intrinsic value, i.e. the underlying current price is above the strike price for call options and below the strike price for put options. *At-the-money options* have zero (or around zero) intrinsic value, i.e. the underlying current price is equal (or around) the strike price for both types of option. ***Out-of-the-***

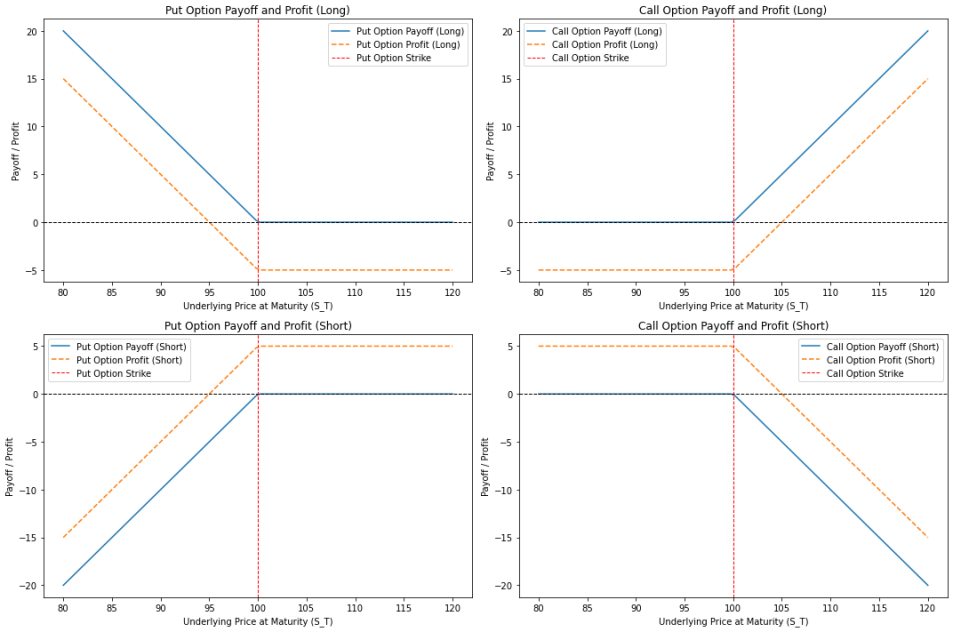

Figure 5-1 Payoff and profit diagrams

Note: Diagrams for put option (left) and call option (right) in long (top) and short position (bottom).

money options have a zero intrinsic value, i.e. the underlying current price is below the strike price for call options and above the strike price for put options.

The ***time value*** of an option can be defined as the difference between the value of the option, the option premium, and the intrinsic value of the option, thus indicating how the value of the option differs from its intrinsic value. It follows that if the time value is negative, it would be reasonable to exercise the option because the payoff would be higher than the option premium. This can happen in put options, see Figure 5-2.

In Figure 5-2 we compare the prices of the call and put options for different times to maturity and different prices of the underlying asset, S_0. From the graph, it is clear that for the call option, the time value is always positive and increases with time to maturity. In the case of a put option, the situation is more complex and depends on the specific price of the underlying asset. In fact, there is a level for which the time value is zero. This level also represents the threshold at which the sign of the time value of the option changes. At a lower price of the underlying asset, the time value is negative and decreases with increasing time to maturity; at a higher price of the underlying asset, it is positive and increases with increasing time to maturity.

5.1.1 Put-Call Parity

Put-call parity is a fundamental concept in option pricing that establishes a relationship between the prices of *European call and put options* of the same parameters (underlying asset, strike price, time to maturity, etc.) differing only in the option type. By combining

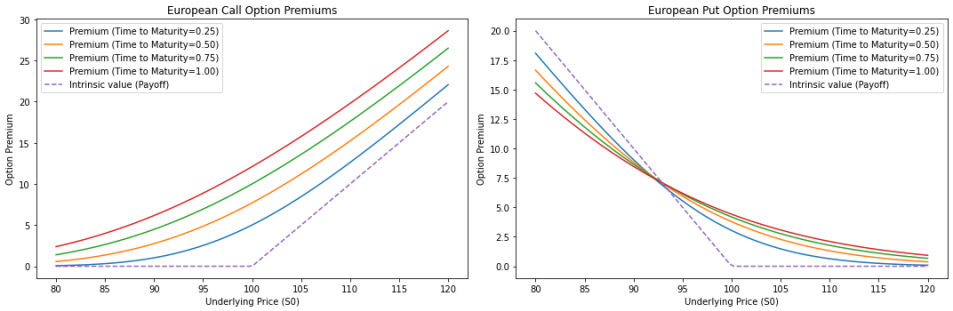

Figure 5-2 Option premiums for different times to maturity and underlying prices

Note: European call and put options with K=100, σ=20%, and r=8%.

the long position in a call option and the short position in a put option, we obtain the same profit function as having a long position in the underlying asset which is sold at maturity for the strike price and adjusted for interest payments. To be more precise, we can write put-call parity as the following equation:

$$c - p = S_0 - \frac{K}{(1+r)^T}, \tag{5.3}$$

where c is call option premium, p is put option premium, K is a strike price, S_0 is the actual underlying price, r is the risk-free rate and T is the time to maturity in years.

5.1.2 Risk neutral valuation principle

In option valuations, there is one assumption, or, say, principle, which is common to all valuation methods described further, and it is called ***risk-neutral valuation***. As explained by Hull (2021, p. 292): "*This states that, when valuing a derivative, we can make the assumption that investors are risk neutral. This assumption means investors do not increase the expected return they require from an investment to compensate for increased risk.*" Of course, the world we live in is not risk-neutral. However, it turns out that the risk-neutral valuation gives us the right option price for the world we live in. The consequence of this assumption is that the expected return on a stock is the same as the discount rate, and both are equal to the risk-free rate.

5.2 Black-Scholes valuation formula

The Black-Scholes model (Black & Scholes, 1973) can be used to value ***European-type options only***. It is based on assumptions that must be met if the valuation of a given option is to be correctly determined:

- we assume that there are no transaction costs for buying and selling as well as no spread between ask and bid prices,
- a constant value of volatility and a risk-free rate,
- we assume an underlying asset whose holding does not generate any income (e.g. dividends) or expenses (e.g. storage expenses),
- trading takes place in continuous time,
- prices are independent of expected returns,
- identical risk-free borrowing and lending rates,

- infinite divisibility of the underlying assets,
- no short selling restrictions, full use of the cash possible (no margin required),
- does not take into account taxation,
- perfectly competitive market - no market participant is able to influence the asset price or the risk-free rate,
- the assumption of the law of one price and the impossibility of arbitrage,
- the evolution of the underlying asset price can be described by a geometric Brownian motion with logarithmic prices, see Section 1.5.1.

The price of a call option can be calculated as follows,

$$c_{BS}(S_t, K, r, \sigma, T-t) = S_t \cdot \Phi(d_1) - e^{-r\cdot(T-t)} \cdot K \cdot \Phi(d_2), \tag{5.4}$$

where:

$$d_1 = \frac{ln\left(\frac{S_t}{K}\right) + \left(r + \frac{\sigma^2}{2}\right)\cdot(T-t)}{\sigma\cdot\sqrt{T-t}}, \tag{5.5}$$

$$d_2 = d_1 - \sigma \cdot \sqrt{T-t}. \tag{5.6}$$

Similarly, the price of a put option is calculated as follows,

$$p_{BS}(S_t, K, r, \sigma, T-t) = e^{-r\cdot(T-t)} \cdot K \cdot \Phi(-d_2) - S_t \cdot \Phi(-d_1). \tag{5.7}$$

In these formulas, S_t represents the current price of the underlying asset, σ is volatility measured as the standard deviation of log returns, r is the risk-free rate, K is the strike price, T is the maturity date and $T-t$ represents the time to maturity in years.

Code 5-1 Option valuation according to BS model

```
import numpy as np
from scipy.stats import norm

def BS_option(S, K, T, r, sigma, option_type):
    d1 = (np.log(S/K) + (r + (sigma**2)/2)*T) / (sigma*np.sqrt(T))
    d2 = d1 - sigma * np.sqrt(T)

    if option_type == "call":
        return S*norm.cdf(d1) - K*np.exp(-r*T)*norm.cdf(d2)
    elif option_type == "put":
        return K*np.exp(-r*T)*norm.cdf(-d2) - S*norm.cdf(-d1)
    else:
        raise ValueError("Invalid option type. Use 'call' or 'put'.")

print("Call Option Price:", BS_option(S=100, K=100, T=1, r=0.05, sigma=0.2,
option_type="call"))
print("Put Option Price:", BS_option(S=100, K=100, T=1, r=0.05, sigma=0.2,
option_type="put"))
> Call Option Price: 10.450583572185565
Put Option Price: 5.573526022256971
```

The basic example of option valuation is given in Code 5-1. As can be seen, according to the BS model, the option premium for call and put options with a strike price of $100, current underlying price of $100, time to maturity of one year, risk-free rate of 5%, and annual volatility of returns 20% is 10.45$ and 5.57$, respectively.

Question 5-1 Calculate the option premium and compare it to the market value

Choose the option on some stock, determine the input values, and compute the option premium according to Black-Scholes model utilizing Code 5-1. Compare your results with the market price, which you find at *http://finance.yahoo.com*.
What can cause the difference you observe? Can dividends play a role? How do you estimate the volatility of the returns?

5.2.1 Option greeks

Sometimes, for example, in option hedging or option trading, it is important to calculate the sensitivity of the option price to the change of input parameters. These sensitivities are called ***option greeks,*** and they are mathematically calculated as the partial derivatives of the option price with respect to the parameter values.

Delta is the first derivative of the option price with respect to the change in the underlying price. It roughly approximates the change in the option price with respect to a change of $1 in the underlying asset price. However, it is not precise as there is no strictly linear relationship. ***Vega*** is the first derivative of the option price with respect to the change in volatility. ***Theta*** is the first derivative of the option price with respect to the change in time to maturity. It roughly approximates the change in the option price with respect to a 1-year change in the time to maturity. ***Rho*** is the first derivative of the option price with respect to the change in the risk-free interest rate. ***Gamma*** is the first-order derivative of the delta with respect to the change in the underlying price or the second-order derivative of the option price with respect to the change in the underlying price.

Assuming the Black-Scholes option pricing formula, the option greeks can be computed analytically, see Table 5-1. There are also other option greeks, which are not discussed as often, such as lambda, epsilon, vomma, vera, speed, zomma, color, ultima. These Greeks are second- or third-order derivatives or derivatives of option greeks with respect to some other factor. The example of option greeks calculation under the BS model is given in Code 5-2.

Table 5-1 Analytical formulas for option greeks in the BS

	call options	***put options***
delta	$\Phi(d_1)$	$-\Phi(-d_1) = \Phi(d_1) - 1$
gamma	$\frac{\phi(d_1)}{S_t \cdot \sigma \cdot \sqrt{T-t}}$	
vega	$\phi(d_1) \cdot S_t \cdot \sqrt{T-t}$	
theta	$-\frac{\phi(d_1) \cdot S_t \cdot \sigma}{2 \cdot \sqrt{T-t}} - \frac{r \cdot K \cdot \Phi(d_2)}{e^{r \cdot (T-t)}}$	$\frac{r \cdot K \cdot \Phi(-d_2)}{e^{r \cdot (T-t)}} - \frac{\phi(d_1) \cdot S_t \cdot \sigma}{2 \cdot \sqrt{T-t}}$
rho	$\frac{K \cdot (T-t) \cdot \Phi(d_2)}{e^{r \cdot (T-t)}}$	$-\frac{K \cdot (T-t) \cdot \Phi(-d_2)}{e^{r \cdot (T-t)}}$

Code 5-2 Greeks calculation according to BS model

```
import numpy as np
from scipy.stats import norm

def BS_greeks(S, K, T, r, sigma, option_type):
    d1 = (np.log(S / K) + (r + 0.5 * sigma**2) * T) / (sigma * np.sqrt(T))
    d2 = d1 - sigma * np.sqrt(T)

    if option_type == "call":
        delta = norm.cdf(d1)  # Delta
        gamma = norm.pdf(d1) / (S * sigma * np.sqrt(T))  # Gamma
        vega = S * np.sqrt(T) * norm.pdf(d1)  # Vega
        theta = -S * sigma * norm.pdf(d1) / (2 * np.sqrt(T)) - r * K * np.exp(-r * T) *
norm.cdf(d2)  # Theta
        rho = K * T * np.exp(-r * T) * norm.cdf(d2)  # Rho
    elif option_type == "put":
        delta = norm.cdf(d1) - 1  # Delta for a put option
        gamma = norm.pdf(d1) / (S * sigma * np.sqrt(T))  # Gamma for a put option
        vega = S * np.sqrt(T) * norm.pdf(d1)  # Vega for a put option
        theta = -S * sigma * norm.pdf(d1) / (2 * np.sqrt(T)) + r * K * np.exp(-r * T) *
norm.cdf(-d2)  # Theta for a put option
        rho = -K * T * np.exp(-r * T) * norm.cdf(-d2)  # Rho for a put option
    else:
        raise ValueError("Invalid option type. Use 'call' or 'put'.")

    return delta, gamma, vega, theta, rho

delta_call, gamma_call, vega_call, theta_call, rho_call = BS_greeks(S=100, K=100, T=1,
r=0.05, sigma=0.2, option_type="call")
delta_put, gamma_put, vega_put, theta_put, rho_put = BS_greeks(S=100, K=100, T=1, r=0.05,
sigma=0.2, option_type="put")

print("Call Option Greeks:")
print("Delta:", delta_call)
print("Gamma:", gamma_call)
print("Vega:", vega_call)
print("Theta:", theta_call)
print("Rho:", rho_call)

print("\nPut Option Greeks:")
print("Delta:", delta_put)
print("Gamma:", gamma_put)
print("Vega:", vega_put)
print("Theta:", theta_put)
print("Rho:", rho_put)
> Call Option Greeks:
Delta: 0.6368306511756191
Gamma: 0.018762017345846895
Vega: 37.52403469169379
Theta: -6.414027546438196
Rho: 53.232481545376345

Put Option Greeks:
Delta: -0.3631693488243809
Gamma: 0.018762017345846895
Vega: 37.52403469169379
Theta: -1.6578804239346256
Rho: -41.89046090469506
```

Question 5-2 Plot the dependence of option greeks values

Using Code 5-2 plot the values of the option greeks for different prices of the underlying asset. Assume the interval [$80, $100].

Question 5-3 Calculate and interpret the option greeks for real-world option

Choose an option (for which the data is available), calculate the greeks, and interpret delta and theta.

5.2.2 Implied volatility smile and implied volatility surface

In the Black-Scholes pricing formula, all parameters in the model other than the volatility are directly observable, i.e. we can find or state their values. However, the volatility σ in the model represents the standard deviation of the future log returns and as such cannot be observed directly in the market. As a proxy, we can use the historical value of the standard deviation of the log returns, however, as a well-known theorem says "*the past performance is not indicative of future results*", also the past volatility says a little about the future volatility.

In practice, we usually work with implied volatility. Knowing the market option premium, we can find the volatility for which the Black-Scholes formula prices the option at a given market premium. Such volatility is called implied volatility. We do the following optimization,

$$iv = \underset{\sigma}{\operatorname{argmin}} |p_{BS}(\sigma) - p_{market}|, \tag{5.8}$$

i.e., we are looking for the implied volatility value, *iv*, for which the absolute value of the difference between the Black-Scholes price and market price is minimized. Note that we can also minimize the square of the difference, or the square of the ratio minus one, see Code 5-3; however, the absolute value of the difference is more illustrative.

Whereas historical volatilities are backward looking, i.e. they measure the historical volatility, the implied volatilities are forward-looking, i.e. they measure what the market thinks of the volatility to be.

It would be also reasonable to assume that if we calculate implied volatility for different strikes (and times to maturity) having all other parameters the same, we would get one value of the implied volatility, however, that is not what we observe in the market. The reason for this is that the assumptions of the Black-Scholes model, especially the normal distribution of the returns with constant volatility, are too simple and thus the model is not accurate.

We can also plot the implied volatility as a function of strike price (volatility smile) or as a function of strike price and time to maturity (volatility surface). A typical shape of implied volatility curves for stocks' underlying assets is a skewed curve with implied volatility higher for higher and lower strike values. The implied volatility is substantially higher for low strikes and slightly lower for high strikes, so the curve seems like a smile, and thus it is called the volatility smile curve.

Question 5-4 The different fun functions in Code 5-3

Why in Code 5-3 do we change the *fun* function, which is minimized by *scipy.optimize.minimize*? Why are we not using the commented one (the absolute difference between the price and BS value)?

Code 5-3 Calculation of implied volatility (reverse from Code 5-1)

```
def BS_iv(price, S, K, T, r, option_type='call'):
    def fun(sigma):
        # return abs(BS_option(S, K, T, r, sigma, option_type) - price)
        return (BS_option(S, K, T, r, sigma, option_type) / price - 1) ** 2

    res = scipy.optimize.minimize_scalar(fun, bounds=(0.001, 6), method='bounded')
    return res.x

iv_call = BS_iv(price=10.45058357, S=100, K=100, T=1, r=0.05, option_type="call")
iv_put = BS_iv(price=5.573526022, S=100, K=100, T=1, r=0.05, option_type="put")
print("Implied volatility of call option: ", iv_call)
print("Implied volatility of put option: ", iv_put)
> Implied volatility of call option:  0.1999990177142531
Implied volatility of put option:  0.19999901776564558
```

Code 5-4 Calculation of volatility smile for META stock

```
import pandas as pd
import numpy as np
import scipy
import datetime as dt
import yfinance as yf
import matplotlib.pyplot as plt
# https://finance.yahoo.com/news/read-options-table-080007410.html

r = 0.05
stock = yf.Ticker("META") # define the ticker for which we want to download the data; we
chose META as it does not pay dividends
expirations = stock.options # get the list of possible expiration dates data
print(expirations[-1])

calls = stock.option_chain(expirations[-1]).calls # get all the call options for the
expiration data third from the last one
calls.insert(loc=6, column='price', value=(calls['ask']+calls['bid'])/2) # if traded and the
ask,bid is known
#calls.insert(loc=6, column='price', value=(calls['lastPrice'])) # if ask bid prices are not
available
S = stock.history(period='1d', interval='1m')['Close'].iloc[-1] # we get the last price of
the underlying asset(stock)
ttm = (pd.to_datetime(expirations[-1]) - dt.datetime.now()).total_seconds() /60/60/24/252
calls.insert(8, 'our IV', np.nan)

for index, row in calls.iterrows():
    calls['our IV'].values[index]=BS_iv(row['price'], S, row['strike'], ttm, r, 'call')

plt.scatter(x=calls['strike'], y=calls['our IV'])
plt.xlabel('Strike Price')
plt.ylabel('Our IV')
plt.title('Our IV vs. Strike Price for Call Options')
plt.show()
```

5.3 Lattice models

Lattice models are a class of numerical methods used in option pricing, offering a flexible framework to value financial derivatives. These models are particularly well suited for handling complex option structures and incorporating various market dynamics. One of the most popular lattice models is the binomial option pricing model.

5.3.1 Binomial option pricing model

The binomial option pricing model (BOPM), unlike the Black-Scholes pricing formula, can be used for the valuation of American-style options. The model is based on the absence of arbitrage opportunities and the discrete evolution of the underlying asset over time. It assumes that there are only two possible states following the initial state: increase and decrease; hence, the name of the model. To understand it, a single-period binomial model can be constructed, but multi-period models are typically used (a part of such a model can be seen in Figure 5-3).

The assumptions on which the model is built are as follows:

- trading occurs at discrete time moments,
- prices are independent of expected returns,
- equal risk-free borrowing and lending rates,
- infinite divisibility of underlying assets,
- assuming a perfectly competitive market,
- not considering the taxation of returns,
- assuming no transaction costs for buying and selling, as well as no spread between ask and bid prices,
- assumption of the validity of the law of one price and the absence of arbitrage opportunities.

The binomial process can be understood as an approximation of a continuous stochastic process in discrete time intervals. Assuming that the price movement follows a

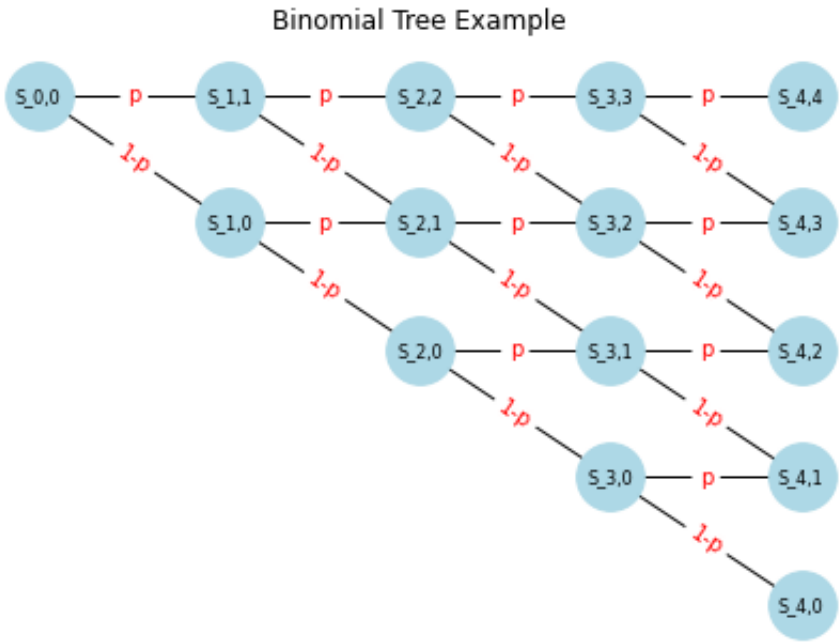

Figure 5-3 Example of binomial tree with four steps

geometric Brownian motion, see Section 1.5.1, we can relate the parameters of the binomial model (p, u, and d) to the parameters of GBM ($\mu = r_{RF}$ and σ). Assuming equality of the expected values and variances of both processes, we obtain the following two equations:

$$E(S_{t+\Delta t}) = S_t \cdot e^{r_{RF} \cdot \Delta t} = p \cdot u \cdot S_t + (1-p) \cdot d \cdot S_t \,, \tag{5.9}$$

$$var\left[\frac{dS(\Delta t)}{S_t}\right] = \sigma^2 \Delta t = p \cdot u^2 + (1-p) \cdot d^2 - [p \cdot u + (1-p) \cdot d]^2. \tag{5.10}$$

As these two equations contain three unknown parameters, p, u, and d, we have the flexibility to choose one of them. A common choice is to set the probability of a decrease and increase to be equal:

$$p = (1-p) = \tfrac{1}{2}, \tag{5.11}$$

or alternatively, set the product of the growth and decrease indices equal to one:

$$u = \tfrac{1}{d}. \tag{5.12}$$

This gives us three equations with three unknowns. We follow the parameter determination overview in (Clewlow & Strickland, 1998, p. 18). The models published as CRR (Cox et al., 1979) and JR (Jarrow & Rudd, 1983) are derived from equations (5.11) and (5.12) respectively. Table 5-2 summarizes the parameters of these binomial models.

Table 5-2 Binomial tree parameters

CRR	***JR***
$u = e^{\left(r_{RF} - \frac{1}{2}\sigma^2\right) \cdot \Delta t + \sigma \cdot \sqrt{\Delta t}}$ $d = e^{\left(r_{RF} - \frac{1}{2}\sigma^2\right) \cdot \Delta t - \sigma \cdot \sqrt{\Delta t}}$ $p_u = p_d = \frac{1}{2}$	$u = e^{\sigma \cdot \sqrt{\Delta t}}$ $d = e^{-\sigma \cdot \sqrt{\Delta t}}$ $p_u = \frac{1}{2} + \frac{r_{RF} - \frac{1}{2}\sigma^2}{2\sigma} \cdot \sqrt{\Delta t}$ $p_d = \frac{1}{2} - \frac{r_{RF} - \frac{1}{2}\sigma^2}{2\sigma} \cdot \sqrt{\Delta t}$

When valuing an option using the binomial model, we proceed in the following consecutive steps:

- First, we calculate all the parameters of the binomial model:
  - step length Δt as the ratio of time to maturity and the desired number of steps,
  - growth index u and decrease index d according to the formulas in Table 5-2,
  - probability of growth p_u and probability of decrease p_d according to the formulas in Table 5-2.
- We create a binomial tree representing the development of the underlying asset's price over time from $t = 0$ to $t = T$ using the formulas:

$$S_{t+\Delta t}^{d} = d \cdot S_t, \tag{5.13}$$

$$S_{t+\Delta t}^{u} = u \cdot S_t. \tag{5.14}$$

- We evaluate the option for individual nodes of the binomial tree at maturity time T by assigning its value based on the payoff function.
- We value the option backwards at each node from $t = T - \Delta t$ until $t = 0$ depending on whether it is a European or American option:
  - for a European option, we discount the average option price in nodes $t + \Delta t$:

$$f_t = (1 + r_{RF})^{-\Delta t} \cdot (f_{t+\Delta t}^{u} \cdot p_u + f_{t+\Delta t}^{d} \cdot p_d) \tag{5.15}$$

  - for an American option, we consider the possibility of exercising the option at any time until maturity. Therefore, we determine the option price as the maximum of the intrinsic value *IV* and the discounted average option price in the following nodes:

$$f_t = max\{IV; (1 + r_{RF})^{-\Delta t} \cdot (f_{t+\Delta t}^{u} \cdot p_u + f_{t+\Delta t}^{d} \cdot p_d)\} \tag{5.16}$$

- By iteratively repeating the last step, we obtain the option price for $t = 0$, representing the valuation moment.

The example of the implementation is shown in Code 5-5 in the function *BOPM_option*. In the code, we price European and American options with the specifications as in Code 5-1. As can be seen, the resulting option prices (option premiums) for European options are similar to the valuation provided by the Black-Scholes model. The difference is only in the valuation of the American put option.

Code 5-5 Option valuation according to BOPM (the same option as in Code 5-1)

```
import numpy as np

def BOPM_option(S, K, T, r, sigma, option_type, num_steps=100):
    # First, we calculate all the parameters of the binomial model:
    delta_t = T / num_steps
    u = np.exp(sigma * np.sqrt(delta_t))
    d = 1 / u
    p = (np.exp(r * delta_t) - d) / (u - d)

    # Generate a binomial tree for the underlying asset price
    underlying_tree = np.zeros((num_steps + 1, num_steps + 1))
    for i in range(num_steps + 1):
        for j in range(i + 1):
            underlying_tree[j, i] = S * u**(i-j) * d**j

    # We evaluate the option for individual nodes of the binomial tree at maturity time
    option_values = np.zeros((num_steps + 1, num_steps + 1))
    if "call" in option_type:
        option_values[:, -1] = np.maximum(0, underlying_tree[:, -1] - K)
    elif "put" in option_type:
        option_values[:, -1] = np.maximum(0, K - underlying_tree[:, -1])
    else:
        raise ValueError("Invalid option type. Use 'European call', 'European put',
'American call', or 'American put'.")

    # Option valuation using binomial tree (backward induction)
    for i in range(num_steps - 1, -1, -1):
        for j in range(i+1):
            option_values[j, i] = np.exp(-r * delta_t) * (p * option_values[j, i + 1] + (1 -
p) * option_values[j+1, i + 1])
            if option_type.startswith("American"):
```

```
                # Check for early exercise
                early_exercise_value = np.maximum(0, K - underlying_tree[j, i]) if "put" in
option_type else np.maximum(0, underlying_tree[j, i] - K)
                option_values[num_steps - j, i] = max(option_values[num_steps - j, i],
early_exercise_value)
    return option_values[0, 0]

print("European Call Option Price:", BOPM_option(S=100, K=100, T=1, r=0.05, sigma=0.2,
option_type="European call"))
print("European Put Option Price:", BOPM_option(S=100, K=100, T=1, r=0.05, sigma=0.2,
option_type="European put"))
print("American Call Option Price:", BOPM_option(S=100, K=100, T=1, r=0.05, sigma=0.2,
option_type="American call"))
print("American Put Option Price:", BOPM_option(S=100, K=100, T=1, r=0.05, sigma=0.2,
option_type="American put"))
> European Call Option Price: 10.430611662249113
European Put Option Price: 5.553554112321353
American Call Option Price: 10.430611662249113
American Put Option Price: 6.082354409142444
```

Question 5-5 The difference in option price

Why do the prices of the option obtained in Code 5-1 and Code 5-5 differ? What must be done to decrease the difference?

5.4 Monte Carlo and least squares Monte Carlo

Monte Carlo simulation is a powerful numerical method widely employed in finance for valuing financial instruments, with a particular focus on options. It relies on random sampling to obtain numerical results for complex problems that may not have a closed-form solution. In the context of option pricing, Monte Carlo simulations provide a flexible approach to option valuing.

In financial applications, Monte Carlo simulation is commonly applied to simulate the possible future paths of the underlying asset's price, as discussed in Subchapter 1.5. In this subchapter, we explicitly assume a geometric Brownian motion, but other stochastic processes can be considered. This choice aligns with the original Black-Scholes model (Black & Scholes, 1973) applied in Subchapter 5.2, ensuring comparability of results.

5.4.1 Valuation of European options via Monte Carlo simulation

The valuation of European options through Monte Carlo simulation is straightforward and involves the following key steps:

1. ***Selecting a Mathematical Model:*** Choose a mathematical model to depict the stochastic process governing the underlying asset's price, as detailed in Subchapter 1.5. In our case, we employ geometric Brownian motion, see Section 1.5.1. Note that under the risk-neutral valuation, see Section 5.1.2, the expected return is assumed to be a risk-free rate.
2. ***Generating Random Paths:*** Simulate multiple paths of the underlying asset's price by sampling from the chosen model.
3. ***Calculating Option Payoff:*** Since European options can be exercised only at maturity, determine the payoff for each simulated path at the maturity date. The payoff depends on the type of option (call or put) and its parameters, such as the strike price (K).

4. ***Discounting Future Payoffs:*** Given that the payoff occurs at maturity, i.e., in the future, we discount these future payoffs to the present value using the risk-free rate.
5. ***Averaging Discounted Payoffs:*** Average the discounted payoffs across all simulated paths. This average serves as an estimate for the option's expected present value and, consequently, its price (option premium).

The entire process is illustrated in Code 1-17. While this approach is essentially a numerical approximation of the continuous Black-Scholes model, we should arrive at similar results.

Code 5-6 Example of option valuation utilizing MC simulation

```
import numpy as np

def MC_option(S, K, T, r, sigma, option_type, num_trials=10000, num_steps=100):
    np.random.seed(100) # fix the seed

    mu = r # under risk-neutral probability mu=r
    dt = T / num_steps

    # Simulate stock price paths using geometric Brownian motion

    # Initialize an array to store stock prices for each trial
    stock_prices = np.zeros((num_trials, num_steps + 1))
    stock_prices[:, 0] = S

    # Simulate stock prices for multiple trials using geometric Brownian motion
    for i in range(1, num_steps + 1):
        drift = (mu - (sigma**2)/2) * dt
        diffusion = sigma * np.sqrt(dt) * np.random.normal(0, 1, num_trials)
        stock_prices[:, i] = stock_prices[:, i - 1] * np.exp(drift + diffusion)

    # Calculate option payoffs at maturity
    if option_type == 'call':
        option_payoffs = np.maximum(stock_prices[:, -1] - K, 0)
    elif option_type == 'put':
        option_payoffs = np.maximum(K - stock_prices[:, -1], 0)
    else:
        raise ValueError("Invalid option type. Use 'call' or 'put'.")

    # Discount the expected payoffs to present value
    option_price = np.mean(option_payoffs * np.exp(-r * T))

    return option_price

print("European Call Option Price:", MC_option(S=100, K=100, T=1, r=0.05, sigma=0.2,
option_type="call"))
print("European Put Option Price:", MC_option(S=100, K=100, T=1, r=0.05, sigma=0.2,
option_type="put"))
> European Call Option Price: 10.347557847965888
European Put Option Price: 5.476757334688844
```

Question 5-6 The difference in option price

Why do the prices of the option obtained in Code 5-1 and Code 5-6 differ? What must be done to decrease the difference?
Can the variance reduction techniques introduced in Section 1.4.1 improve the accuracy? Implement them into the code and compare the results.

5.4.2 Valuation of American options via least squares Monte Carlo

Valuation of European options under Monte Carlo simulation is straightforward due to their exercisability only at maturity. On the contrary, American options grant the right to exercise at any time before maturity. The optimal strategy for an American option holder involves comparing the immediate exercise payoff with the expected payoff from continuation, choosing to exercise when the immediate payoff is higher. This requires a backward valuation approach, similar to the one used in the binomial option pricing model (refer to Section 5.3.1).

The main challenge in valuing American options lies in determining the payoff from continuation. Longstaff and Schwartz (2001) addressed this challenge through a simple but effective method employing ordinary least-square regression (see Appendix B). The process of valuing American put/call options shares similarities with that of European options, especially in terms of path simulations. However, the approach diverges in the following manner.

"*At the final exercise date, the optimal exercise strategy for an American option is to exercise the option if it is in the money. Prior to the final date, however; the optimal strategy is to compare the immediate exercise value with the expected cash flows from continuing, and then exercise if immediate exercise is more valuable. Thus, the key to optimally exercising an American option is identifying the conditional expected value of continuation. In this approach, we use the cross-sectional information in the simulated paths to identify the conditional expectation function. This is done by regressing the subsequent realized cash flows from continuation on a set of basis functions of the values of the relevant state variables. The fitted value of this regression is an efficient unbiased estimate of the conditional expectation function and allows us to accurately estimate the optimal stopping rule for the option.*" (Longstaff & Schwartz, 2001, pp. 3–4)

To estimate the expected cash flow from continuing the option's life conditional on the stock price, we regress the future option value on a constant, stock price, and square of stock price. This specification is one of the simplest, and more general specifications can be applied.

The entire process is demonstrated in Code 5-7. In the code, we also allowed for European options valuation; however, obviously, we arrived at the same results as in Section 5.4.1.

Code 5-7 Example of option valuation utilizing LSMC

```
import numpy as np

def LSMC_option(S, K, T, r, sigma, option_type, num_trials=10000, num_steps=100):
    # Function to calculate the payoff of the option
    def payoff(S, K, option_type):
        if "call" in option_type:
            return np.maximum(0, S - K)
        elif "put" in option_type:
            return np.maximum(0, K - S)
        else:
            raise ValueError("Invalid option type. Use 'European call', 'European put',
'American call', or 'American put'.")

    np.random.seed(100) # Fix the seed for reproducibility
```

```
    mu = r  # under risk-neutral probability mu=r
    dt = T / num_steps

    # Simulate stock price paths using geometric Brownian motion
    stock_prices = np.zeros((num_trials, num_steps + 1))
    stock_prices[:, 0] = S

    for t in range(1, num_steps + 1):
        drift = (mu - 0.5 * sigma**2) * dt
        diffusion = sigma * np.sqrt(dt) * np.random.normal(0, 1, num_trials)
        stock_prices[:, t] = stock_prices[:, t - 1] * np.exp(drift + diffusion)

    # Matrix to store whether the option is exercised at each time step
    excercise = np.zeros((num_trials, num_steps + 1))

    # Matrix to store the cash flows of the option at each time step
    CFs = np.zeros((num_trials, num_steps + 1))
    CFs[:, -1] = payoff(stock_prices[:, -1], K, option_type)

    if option_type.startswith("European"):
        # European Option Pricing
        return np.mean(CFs[:, -1] * np.exp(-r * T))

    elif option_type.startswith("American"):
        # American Option Pricing using Least Squares Monte Carlo
        for t in range(num_steps - 1, -1, -1):
            # Calculate payoffs
            payoffs = payoff(stock_prices[:, t], K, option_type)

            # Find paths where the option is in-the-money
            hold = np.where(payoffs > 0)[0]

            if len(hold) > 3:
                # Apply least square method of future CF on current stock price
                regression = np.polyfit(stock_prices[hold, t].flatten(), CFs[hold,
t+1].flatten()*np.exp(-r * dt), 3)
                # calculate continuation values for ITM options
                CV = np.polyval(regression, stock_prices[hold, t].flatten())

                # Check whether to exercise the option now or continue holding it
                excercise[hold, t] = np.where(payoffs[hold] >= CV, 1, 0)
                # Update cash flows based on exercise decision
                CFs[:, t] = np.where(excercise[:, t].flatten()==1, payoffs, CFs[:,
t+1].flatten()*np.exp(-r * dt))
            else:
                # If there are very few valid paths with ITM options, do not perform
regression
                CFs[:, t] = CFs[:, t+1].flatten()*np.exp(-r * dt)

        return np.mean(CFs[:, 0])

    else:
        raise ValueError("Invalid option type. Use 'European call', 'European put',
'American call', or 'American put'.")


# Example usage
print("European Call Option Price:", LSMC_option(S=100, K=100, T=1, r=0.05, sigma=0.2,
option_type="European call"))
print("European Put Option Price:", LSMC_option(S=100, K=100, T=1, r=0.05, sigma=0.2,
option_type="European put"))
print("American Call Option Price:", LSMC_option(S=100, K=100, T=1, r=0.05, sigma=0.2,
option_type="American call"))
print("American Put Option Price:", LSMC_option(S=100, K=100, T=1, r=0.05, sigma=0.2,
option_type="American put"))
> European Call Option Price: 10.347557847965888
European Put Option Price: 5.476757334688844
```

```
American Call Option Price: 10.375712050593638
American Put Option Price: 5.991380001549636
```

Appendices

Appendix A

List of Helpful Packages

In Python, we usually work with *modules*, *packages*, and *libraries*. The ***module*** is simply a *.py* file that contains Python code, usually in the form of functions. Such a module or function can be imported into the main script. For example, consider the module *portfolio.py* shown in Code A–1.

Code A-1 Module *portfolio.py*

```
1. def log2disc(returns):
2.     return exp(returns)-1
3.
4. def disc2log(returns):
5.     return log(1+returns)
```

As can be seen from the code, the module contains two functions: the first calculates the log returns from the discrete returns, and the second calculates the discrete returns from the log returns. In order to be able to call these functions, the module must be imported first, see Code A-2. Either we can import only the particular function from the module, see line 1, or the whole module (line 2). When we import the whole module we can also alias it, see line 5. In these cases, if we want to call the function we must call it with the name of the module as the prefix, see lines 3 and 6. However, we can also import the function from the module using the construct from, see line 8, in this case, we call the function with its name only, i.e. we do not add the prefix of the name of the module, see line 9.

Code A-2 Importing module and calling its functions

```
1. import portfolio.log2disc  # we import only the particular function
2. import portfolio  # or we import the whole module
3. portfolio.log2disc(0.1)  # then we call the function with module. Prefix
4.
5. import portfolio as pf  # we can change name with which we call the module
6. pf.log2disc(0.1)  # we call the function with the changed name of module
7.
8. from portfolio import log2disc  # or we can import just the function
9. log2disc(0.1)  # then we call the function without the module prefix
```

The ***package*** is simply a directory with one or more modules. For the directory, in order to be considered a package, it must contain a *file __init__.py*, which usually initializes the code for the corresponding package, but can be empty as well. The packages allow hierarchical structure, i.e. we can organize the modules in sub-folders to a different

level (as long as each directory contains *__init__.py* file). Let us consider the following package structure shown in Figure A-1. If we have this package structure, the different ways to import the functions and call them are shown in Code A-3.

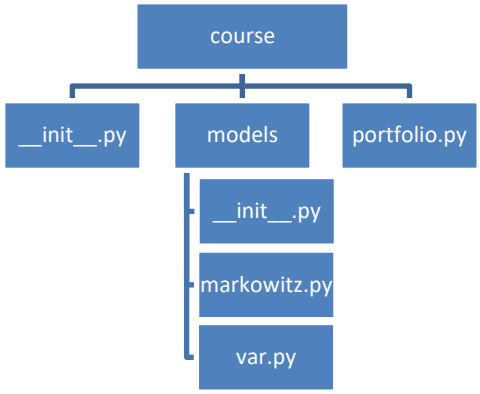

Figure A-1 The structure of the course package

Code A-3 Import the function from the module *portfolio* from the package *course*

```
1. import course  # import the whole package
2. course.portfolio.log2disc(0.1) # call the function by specifying the package name and
  module name
3. from course import portfolio  # or we import only the module portfolio
4. portfolio.log2disc(0.1)  # and call the function with module name
5. from course.portfolio import.log2disc  # alternatively we can import only
   the function
6. log2disc(0.1)  # and call the function
```

While the package is the collection of modules, the ***library*** is the reference for the collection of packages. However, the package and library are often interchangeable as the package can contain sub-packages. There are many packages available[19], and below we list some of the common packages utilized in the textbook.

Arch (Sheppard, 2024) is a Python package that provides tools for volatility modeling utilizing GARCH-type models.

Backtrader (Rodriguez, 2023) is a Python package designed for vectorized backtesting of trading strategies. It leverages *Numpy* and *Pandas* for efficient computation and analysis of financial data, providing a framework for testing and optimizing trading algorithms.

Empyrial (Passoubady, 2024) is a package for generating portfolio reports. It simplifies the process of creating comprehensive and visually appealing reports for investment portfolios, helping to analyze and communicate portfolio performance.

[19] For the repository of availiable packages see https://pypi.org/.

Matplotlib (Hunter & Droettboom, 2023) is a versatile plotting package for Python. It provides a wide range of high-quality 2D and 3D visualizations, making it a go-to choice for creating static, interactive, and animated plots. *Matplotlib* integrates seamlessly with *Numpy* and *Pandas*, making it an essential tool for data visualization in various domains.

Mplfinance (MPL Developers, 2023) extends *Matplotlib* with utilities for the visualization, and visual analysis, of financial data.

Numpy (Oliphant, 2024) is a fundamental package for scientific computing in Python. It provides support for large multidimensional arrays and matrices, along with mathematical functions to operate on these arrays. *Numpy* is widely used in data science, machine learning, and scientific research due to its efficiency and versatility.

Pandas (The Pandas Development Team, 2024) is a powerful Python Python data manipulation and analysis package. It offers data structures like dataframes for efficient handling and manipulation of structured data. Pandas simplifies tasks such as data cleaning, exploration, and transformation, making it a cornerstone in data science workflows.

Plotly (Chris P, 2023) is an interactive Python plotting package that allows users to create dynamic, interactive, and shareable visualizations. It supports a wide range of chart types and is often used for creating web-based dashboards and applications.

Pyfolio (Quantopian Inc., 2019) is a powerful but outdated package for performance analysis. It allows the user to run a complex performance analysis in a few lines of code.

Pyfolio-reloaded (Jansen, 2024) is an updated version of *Pyfolio*, a package that allows running a complex performance analysis in a few lines of code.

Pyportfolioopt (Martin, 2023) is a Python package for portfolio optimization. It offers tools for constructing optimal portfolios based on various risk and return metrics. *PyPortfolioOpt* is used in quantitative finance to build and analyze diversified investment portfolios.

Riskfolio-Lib (Cajas, 2024) is a package for risk-based portfolio optimization. It allows users to analyze and optimize portfolios based on risk measures and constraints. *Riskfolio-Lib* is particularly useful for investors looking to manage and mitigate risks in their portfolios.

Seaborn (Waskom, 2024) is a statistical data visualization package based on *Matplotlib*. It simplifies the creation of attractive and informative statistical graphics. *Seaborn* comes with built-in themes and color palettes to enhance the aesthetics of plots, making it particularly useful for exploratory data analysis.

Ta (Padial, 2023) is a technical analysis package for financial time series built on the *Pandas* package.

Ta-lib (Benediktsson, 2023) is a technical analysis package widely used in financial markets. It provides tools for the calculation of various technical indicators such as moving averages, relative strength index, and more, making it valuable for quantitative analysis in trading strategies.

Tushare (Liu, 2024) is a Python package for easy data acquisition for the China stock market. It provides tools for accessing a wide range of financial data related to Chinese stocks, making it a valuable resource for analysts and investors focusing on the Chinese market.

Yfinance (Aroussi, 2024) is a Python package that facilitates easy data acquisition from the Yahoo Finance server. It provides a simple interface to access financial data, making it convenient for researchers and analysts interested in historical stock market information.

Zipline (Quantopian Inc., 2020) is a Python package for advanced algorithmic trading backtests. Developed by Quantopian, *Zipline* provides a framework for testing trading algorithms using historical data, allowing users to evaluate the performance of their strategies.

Appendix B

Linear Regression in Python

Linear regression is a statistical technique that is used to establish a linear relationship between a target variable and one or more predictor variables. In this context:

- The target variable, also known as dependent, endogenous, regressand, response, measured, or criterion, is what we aim to predict or explain.
- The predictor variables, often referred to as independent, exogenous, regressors, covariates, or input, are used to make predictions or explain variations in the target variable.

When there are two or more independent variables, the model is called multiple linear regression. The equation is as follows:

$$y = \beta_0 + \beta_1 \cdot x_1 + \cdots + \beta_k \cdot x_k + \epsilon, \tag{5.17}$$

where y is a dependent variable, $\beta_0, \beta_1, \ldots, \beta_k$ are the regression coefficients of the independent variables, $x_1, \ldots, x_k$ are independent variables and ϵ represents the error term (residuals).

Regression coefficients can be estimated via various methods. The simplest one is the ***Ordinary Least Squares*** (OLS) method, which minimizes the sum of the squared differences (residuals) between the observed values and the values predicted by the model. There are several assumptions of the model:

- ***Linearity***: The relationship between the independent and dependent variables is linear.
- ***Independence***: The residuals (errors) are independent of each other.
- ***Homoscedasticity***: The residuals have constant variance at every level of the independent variables.
- ***Normality***: For any fixed value of the independent variable, the dependent variable is normally distributed.

In Python, multiple packages are available to perform linear regression. We will demonstrate some of these methods using a specific example from the stock market. In addition to individual stocks, the stock market also trades exchange traded funds (ETFs). An ETF is similar to a mutual fund, but it is traded like a company's stock. Typically,

ETFs aim to replicate the performance of specific indices, such as the Dow Jones Industrial Index or the Standard & Poor's 500 index. Unlike stocks, ETFs usually have annual fees, though these are often lower than those of mutual funds.

Some ETFs are leveraged, aiming to achieve daily returns that are 2x or 3x the performance of their respective indices in a single day.[20] There are also inverse ETFs that seek to replicate the opposite performance of an index, magnified by 1x, 2x, or 3x. Examples include the following.

- **SPY** tracks the S&P 500 index.
- **UPRO** and **SPXL** aim for 3x the daily return of the S&P 500 index.
- **SPXS** seeks to achieve –3x the daily return of the S&P 500 index.
- **YINN** targets 3x the daily return of the FTSE China 50 index.
- **YANG** aims for –3x the daily return of the FTSE China 50 index.

In our demonstration, we will focus on how accurately the SPXL ETF tracks the performance of the S&P 500 index.

Question B-1 Compare the performance of SPY and the S&P 5000 index

Visually compare the cumulative performance of SPY and S&P 5000 index over a selected period. Discuss why the cumulative performance differs.

Before diving into linear regression in Python, we first retrieve our data as outlined in Section 1.1.2. We'll be sourcing the daily adjusted closing prices for the SPXL exchange-traded fund, as well as the values for the S&P 500 index. The relevant code snippet can be found in Code B-1. Then, different approaches and packages can be utilized: *Numpy*, *Statsmodels*, *Scikit-learn*, *Seaborn*, and *Statsmodels*. Below, we show the application of the regression estimation function of each package.

Code B-1 Downloading data

```
1. import yfinance as yf
2. # Download data for SPXL and S&P 500 index
3. spxl = yf.download('SPXL', start='2020-01-01', end='2023-01-01', auto_adjust = True)
4. sp500 = yf.download('^GSPC', start='2020-01-01', end='2023-01-01', auto_adjust = True)
5. # Calculate returns from adjusted closing prices
6. spxl_ret = spxl['Close'].pct_change().dropna()
7. sp500_ret = sp500['Close'].pct_change().dropna()
```

Numpy

The *polyfit* function (see Code B-2) in the *Numpy* package is the simplest and quickest way to obtain the slope and intercept coefficients. As *Numpy* is a commonly used package, no other special package is required to import. However, the function provides only the values of the coefficients and no detailed statistics about the regression are provided.

Code B-2 Linear regression in *Numpy*

```
1. import numpy as np
2. # Using polyfit for linear regression
3. slope, intercept = np.polyfit(sp500_ret, spxl_ret, 1)
4. predicted_values = np.polyval([slope, intercept], sp500_ret)
```

[20] These funds should not be expected to provide three times the (cummulative) return of the benchmark for periods greater than a day. For explanation refer to the example in Table 1-3.

```
5. print([slope, intercept])
6. > [2.9470224035613706, 2.292954192608204e-05]
```

Scipy

SciPy is a package built on top of *Numpy* and provides a more extensive suite of statistical functions. The function *linregress* from the *scipy* package provides additional statistics such as the correlation coefficient, the p-value, and the standard error; however, it still does not offer as comprehensive statistics as some specialized statistical packages. The code snippet is shown in Code B-3.

Code B-3 Linear regression in *Scipy*

```
1. from scipy.stats import linregress
2. # Using linregress for linear regression
3. slope, intercept, r_value, p_value, std_err = linregress(sp500_ret, spxl_ret)
4. predicted_values = intercept + slope * sp500_ret
5. print([slope, intercept, r_value, p_value, std_err])
6. > [2.947022403561368, 2.2929541926078106e-05, 0.9989999116673105, 0.0,
0.0048035009670335214]
```

Scikit-learn

Scikit-learn is a machine learning package that also includes linear regression models. Again, it does not offer as comprehensive statistics as some specialized statistical packages. The code snippet is shown in Code B-4.

Code B-4 Linear regression in *Scikit-learn*

```
1. from sklearn.linear_model import LinearRegression
2. #Using LinearRegression
3. reg = LinearRegression().fit(sp500_ret.values.reshape(-1, 1), spxl_ret)
4. predicted_values = reg.predict(sp500_ret.values.reshape(-1, 1))
5. print([reg.coef_, reg.intercept_])
6. > [array([2.9470224]), 2.2929541926078323e-05]
```

Seaborn

Seaborn is primarily a visualization package, see Subchapter 1.3. It offers a convenient function to plot data with a linear regression model. The function *regplot* provides a quick visual representation of the data and the regression line, in which it automatically calculates and plots the regression, see Code B-5. However, it does provide neither regression coefficient nor regression statistics and as that is not suitable for actual regression analysis; it's more for visualization.

Code B-5 Visualisation of linear regression in *Seaborn*

```
1. import seaborn as sns
2. import matplotlib.pyplot as plt
3. # Create the regression plot
4. sns.regplot(x=sp500_ret, y=spxl_ret)
5. # Add labels and title
6. plt.xlabel('S&P 500 Returns')
7. plt.ylabel('SPXL ETF Returns')
8. plt.title('Linear Regression of SPXL ETF on S&P 500 Returns')
9. plt.show() # Display the plot
```

Statsmodels

The last package we mention is *Statsmodels*. It is a comprehensive package for statistical modeling and it provides a more detailed summary of the regression results. It also allows for more complex statistical models beyond just linear regression. See the code and results in Code B-6.

Code B-6 Linear regression in *Statsmodels*

```
import statsmodels.api as sm
#Using statsmodel package
X = sm.add_constant(sp500_ret) # First add a constant to regressors
model = sm.OLS(spxl_ret, X).fit() # Fit the model
predicted_values = model.predict(X)
print(model.summary()) # print the detailed summary of the regression results
>                             OLS Regression Results
==============================================================================
Dep. Variable:                  Close   R-squared:                       0.998
Model:                            OLS   Adj. R-squared:                  0.998
Method:                 Least Squares   F-statistic:                 3.764e+05
Date:                Fri, 20 Oct 2023   Prob (F-statistic):               0.00
Time:                        11:21:25   Log-Likelihood:                 3583.9
No. Observations:                 756   AIC:                            -7164.
Df Residuals:                     754   BIC:                            -7154.
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const       2.293e-05    7.7e-05      0.298      0.766      -0.000       0.000
Close          2.9470      0.005    613.516      0.000       2.938       2.956
==============================================================================
Omnibus:                      427.176   Durbin-Watson:                   2.715
Prob(Omnibus):                  0.000   Jarque-Bera (JB):            25379.504
Skew:                          -1.756   Prob(JB):                         0.00
Kurtosis:                      31.167   Cond. No.                         62.4
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
```

Alternatively, we can perform a similar analysis on the YINN (Direxion Daily FTSE China Bull 3x Shares ETF) and ^HSI (Hang Seng Index), see Code B-7.

Code B-7 Linear regression in *Statsmodels* of YINN and ^HSI

```
# YINN and ^HSI
import yfinance as yf
import statsmodels.api as sm
# Download data for YINN and ^HSI at once and drop NaN values
data = yf.download(['YINN', '^HSI'], start='2020-01-01', end='2023-01-01',
auto_adjust=True).dropna()
# Calculate returns from adjusted closing prices
yinn_ret = data['Close']['YINN'].pct_change().dropna()
hsi_ret = data['Close']['^HSI'].pct_change().dropna()
# Perform regression using statsmodel package
X = sm.add_constant(hsi_ret)
model = sm.OLS(yinn_ret, X).fit()
print(model.summary())
>                             OLS Regression Results
==============================================================================
Dep. Variable:                   YINN   R-squared:                       0.405
Model:                            OLS   Adj. R-squared:                  0.404
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                    ^HSI   R-squared:                       0.404
Model:                            OLS   Adj. R-squared:                  0.403
Method:                 Least Squares   F-statistic:                     488.7
Date:                Fri, 20 Oct 2023   Prob (F-statistic):           4.97e-83
Time:                        11:56:09   Log-Likelihood:                 1134.2
No. Observations:                 721   AIC:                            -2264.
Df Residuals:                     719   BIC:                            -2255.
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const         -0.0002      0.002     -0.125      0.901      -0.004       0.003
^HSI           2.4844      0.112     22.107      0.000       2.264       2.705
==============================================================================
Omnibus:                      188.078   Durbin-Watson:                   2.835
Prob(Omnibus):                  0.000   Jarque-Bera (JB):             2504.548
Skew:                           0.773   Prob(JB):                         0.00
Kurtosis:                      11.999   Cond. No.                         60.0
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
```

Question 5-7 Interpretation of the results

Interpret the results shown in Code B-6 and Code B-7.
Are the coefficients statistically significant?
In addition to the coefficients, what other statistical measures are presented in the results?
How would you interpret these additional statistics in the context of the regression analysis?
How would you interpret the coefficients in the context of the ETF and index return relationship?
Based on the coefficients, did the ETF achieve tripling the daily returns of the index?
Can you determine the daily costs/fees associated with the exchange-traded fund from the results?
How would you calculate the annual costs based on these daily fees?
Is it possible to annualize these costs? (Refer to Section 1.2.3 for guidance.)

Appendix C

Additional Codes

Code C-1 Automated trading system based on MACD

```
import yfinance as yf
import numpy as np
import matplotlib.pyplot as plt

# first we set the parameters of our strategy
FAST = 12 # period of fast MA, in case FAST=1 we have the closing prices
SLOW = 26 # period of slow MA
SGNLN = 9 # period of signal line
DOLLARTC = 0.05 # transaction costs per one buy/sell transaction, is influenced by the number of shares we trade
INITIALAMOUNT = 500 # the initial amount of money we start trading with


data = yf.download(tickers = 'BTC-USD', start='2000-01-01', end='2022-11-01', interval = '1d', group_by = 'column', auto_adjust = True)
#data = yf.download(tickers = 'SPY', start='2000-01-01', end='2022-11-01', interval = '1d', group_by = 'column', auto_adjust = True)


data['Fast'] = data['Close'].ewm(span=FAST, adjust=False).mean()
data['Slow'] = data['Close'].ewm(span=SLOW, adjust=False).mean()
data['MACD'] = data['Fast'] - data['Slow']
#data['MACD'] = data['MACD'] / data['Close']
data['Signal line'] = data['MACD'].ewm(span=SGNLN, adjust=False).mean()

conditions = [data['MACD']>data['Signal line'], data['MACD']<data['Signal line']] # conditions to check
choices = [1, -1] # the position to substitute
# choices = [1, 0] # alternatively we can consider long-only positions
defaultchoice = 0 # position if none of the conditions is evaluated as True, i.e. in case that fastMA=slowMA or in case of NaN

data['Position'] = np.select(condlist=conditions, choicelist=choices, default=defaultchoice) # calculate the position based on conditions

# now the question is for what price we buy and for what price we sell, see that the position is calculated based on the closing price, so we probably cannot buy for that price
# let's suppose we cannot buy/sell for the closing price we use to calculate MA
# if we suppose that we CAN buy/sell for the closing price we use to compute MA, comment the following line
data['Position'] = data['Position'].shift(1) # we shift the position by one day

data.loc[data.index[-1],'Position'] = 0 # at the end we close all positions
data['Trade'] = data['Position'].diff() # we calculate what should we do, i.e. should we sell the long position and open a short position, or cover the short position and buy a long position
```

```
data['CF'] = -data['Close'] * data['Trade'] - DOLLARTC * np.abs(data['Trade']) # we calculate profit in currency for trading one share
data['Profit'] = data['CF'].cumsum() + data['Position'] * data['Close']

# calculate profit of B&H strategy
data['Profit B&H'] = data['Close'] - data.loc[data.index[0],'Close'] - DOLLARTC
data.loc[data.index[-1], 'Profit B&H'] = data.loc[data.index[-1], 'Profit B&H'] - DOLLARTC

print(f"No. of stcoks traded in total: {data['Trade'].abs().sum()}.")
print(f"Final profit: ${data['Profit'][-1].round(2)}.")
print(f"Profit of B&H strategy: ${data['Profit B&H'][-1].round(2)}.")



# visualize
fig, (ax1, ax2, ax3, ax4) = plt.subplots(4, sharex=True, figsize=(19.20,10.80)) # create the figure with 3 plots/graphs
fig.suptitle('MA trading system') # add the title to the figure
ax1.plot(data[['Close', 'Fast', 'Slow']]) # in the first graph, plot the stock price (close), fast and slow MA
ax2.plot(data[['MACD', 'Signal line']])
ax3.plot(data[['Position', 'Trade']]) # in the second graph, plot suggested position and quantity traded
ax4.plot(data[['Profit', 'Profit B&H']])
ax1.set_ylabel('Price & MAs') # set the label of y-axis
ax2.set_ylabel('MACD & Signal line') # set the label of y-axis
ax3.set_ylabel('Position & trade signal') # set the label of y-axis
ax4.set_ylabel('Total profit') # set the label of y-axis
ax4.legend(['Strategy','B & H'])
del ax1, ax2, ax3, ax4  # delete unnecessary variables
```

Bibliography

ANACONDA Inc. (2024). *Anaconda | Unleash AI innovation and value*. Anaconda. https://www.anaconda.com/

AROUSSI, R. (2024). *yfinance: Download market data from Yahoo! Finance API* (0.2.36) [Python; OS Independent]. https://github.com/ranaroussi/yfinance

ARTZNER, P., DELBAEN, F., EBER, J.-M., HEATH, D. (1999). Coherent Measures of Risk. *Mathematical Finance*, *9*(3), 203–228. https://doi.org/10.1111/1467-9965.00068

BACON, C. R. (2012). *Practical Risk-Adjusted Performance Measurement*. John Wiley & Sons.

BARONE-ADESI, G., BOURGOIN, F., GIANNOPOULOS, K. (1998). Don't look back. *Risk*, *11*(August), 100–103.

BARONE-ADESI, G., GIANNOPOULOS, K., VOSPER, L. (1999). VaR without Correlations for N Linear Portfolios. *Journal of Futures Markets*, *19*(5), 583–602. https://doi.org/10.1002/(SICI)1096-9934(199908)19:5<583::AID-FUT5>3.0.CO;2-S

BARONE-ADESI, G., GIANNOPOULOS, K., VOSPER, L. (2002). Backtesting derivative portfolios with filtered historical simulation (FHS). *European Financial Management*, *8*(1), 31–58. https://doi.org/10.1111/1468-036X.00175

BENEDIKTSSON, J. (2023). *TA-Lib: Python wrapper for TA-Lib* (0.4.28) [Cython, Python; OS Independent]. http://github.com/ta-lib/ta-lib-python

BERKOWITZ, J., CHRISTOFFERSEN, P., PELLETIER, D. (2011). Evaluating value-at-risk models with desk-level data. *Management Science*, *57*(12), 2213–2227. https://doi.org/10.1287/mnsc.1080.0964

BLACK, F., SCHOLES, M. (1973). The Pricing of Options and Corporate Liabilities. *Journal of Political Economy*, *81*(3), 637–654. https://doi.org/10.1086/260062

BODIE, Z., KANE, A., MARCUS, A. J. (2024). *Investments* (13th edition). McGraw-Hill.

BOKEH TEAM. (2024). *bokeh: Interactive plots and applications in the browser from Python* (3.3.4) [JavaScript, Python; OS Independent]. https://pypi.org/project/bokeh/

BOLLERSLEV, T. (1986). Generalized autoregressive conditional heteroskedasticity. *Journal of Econometrics*, *31*(3), 307–327. https://doi.org/10.1016/0304-4076(86)90063-1

CAJAS, D. (2024). *Riskfolio-Lib: Portfolio Optimization and Quantitative Strategic Asset Allocation in Python* (5.0.1) [Python; MacOS, Microsoft, Unix]. https://github.com/dcajasn/Riskfolio-Lib

CARHART, M. M. (1997). On Persistence in Mutual Fund Performance. *The Journal of Finance*, *52*(1), 57–82. https://doi.org/10.1111/j.1540-6261.1997.tb03808.x

CARVER, R. (2015). *Systematic Trading: A unique new method for designing trading and investing systems*. Harriman House Limited.

CHRIS, P. (2023). *plotly: An open-source, interactive data visualization library for Python* (5.18.0) [Python; OS Independent]. https://plotly.com/python/

CHRISTOFFERSEN, P. F. (1998). Evaluating Interval Forecasts. *International Economic Review*, *39*(4), 841–862. https://doi.org/10.2307/2527341

CHRISTOFFERSEN, P., PELLETIER, D. (2004). Backtesting Value-at-Risk: A Duration-Based Approach. *Journal of Financial Econometrics*, *2*(1), 84–108. https://doi.org/10.1093/jjfinec/nbh004

CLEWLOW, L., STRICKLAND, C. (1998). *Implementing Derivative Models*. John Wiley & Sons.

COX, J. C., ROSS, S. A., RUBINSTEIN, M. (1979). Option pricing: A simplified approach. *Journal of Financial Economics*, *7*(3), 229–263. https://doi.org/10.1016/0304-405X(79)90015-1

DeMIGUEL, V., GARLAPPI, L., UPPAL, R. (2009). Optimal Versus Naive Diversification: How Inefficient is the 1/N Portfolio Strategy? *The Review of Financial Studies*, *22*(5), 1915–1953. https://doi.org/10.1093/rfs/hhm075

DUFFIE, D., PAN, J. (1997). An overview of value at risk. *The Journal of Derivatives*, *4*(3), 7–49. https://doi.org/10.3905/jod.1997.407971

ELTON, E. J., GRUBER, M. J., BLAKE, C. R. (1996). Survivor Bias and Mutual Fund Performance. *The Review of Financial Studies*, *9*(4), 1097–1120. https://doi.org/10.1093/rfs/9.4.1097

ENGLE, R. F. (1982). Autoregressive Conditional Heteroscedasticity with Estimates of the Variance of United Kingdom Inflation. *Econometrica*, *50*(4), 987–1007. https://doi.org/10.2307/1912773

ENGLE, R. F., MANGANELLI, S. (2004). CAViaR: Conditional Autoregressive Value at Risk by Regression Quantiles. *Journal of Business & Economic Statistics*, *22*(4), 367–381. https://doi.org/10.1198/073500104000000370

ESTRADA, J. (2008). Mean-semivariance optimization: A heuristic approach. *Journal of Applied Finance*, *18*(1). https://papers.ssrn.com/sol3/papers.cfm?abstract_id=2698700

FAMA, E. F., FRENCH, K. R. (1993). Common risk factors in the returns on stocks and bonds. *Journal of Financial Economics*, *33*(1), 3–56. https://doi.org/10.1016/0304-405X(93)90023-5

FAMA, E. F., FRENCH, K. R. (2010). Luck versus Skill in the Cross-Section of Mutual Fund Returns. *The Journal of Finance*, *65*(5), 1915–1947. https://doi.org/10.1111/j.1540-6261.2010.01598.x

FAMA, E. F., FRENCH, K. R. (2012). Size, value, and momentum in international stock returns. *Journal of Financial Economics*, *105*(3), 457–472. https://doi.org/10.1016/j.jfineco.2012.05.011

FAMA, E. F., FRENCH, K. R. (2015). A five-factor asset pricing model. *Journal of Financial Economics*, *116*(1), 1–22. https://doi.org/10.1016/j.jfineco.2014.10.010

FAMA, E. F., FRENCH, K. R. (2018). Choosing factors. *Journal of Financial Economics*, *128*(2), 234–252. https://doi.org/10.1016/j.jfineco.2018.02.012

FRANCQ, C., ZAKOIAN, J.-M. (2011). *GARCH Models: Structure, Statistical Inference and Financial Applications*. John Wiley & Sons.

FRENCH, K. R. (2024). *Kenneth R. French*. Kenneth R. French – Data Library. https://mba.tuck.dartmouth.edu/pages/faculty/ken.french/data_library.html

GRIMES, A. (2012). *The Art and Science of Technical Analysis: Market Structure, Price Action and Trading Strategies*. John Wiley & Sons.

HAAS, M. (2005). Improved duration-based backtesting of value-at-risk. *Journal of Risk*, *8*(2), 17–38. https://doi.org/10.21314/JOR.2006.126

HANSEN, B. (2022). *Econometrics*. Princeton University Press.

HILPISCH, Y. (2018). *Python for Finance: Mastering Data-Driven Finance*. O'Reilly Media, Inc.

HULL, J. C. (2021). *Options, Futures, and Other Derivatives, Global Edition*. Pearson Higher Ed.

HULL, J., WHITE, A. (1998). Incorporating volatility updating into the historical simulation method for value-at-risk. *Journal of Risk*, *1*(1), 5–19. https://doi.org/10.21314/JOR.1998.001

HUNTER, J. D., DROETTBOOM, M. (2023). *matplotlib: Python plotting package* (3.8.2) [Python; OS Independent]. https://matplotlib.org

INTRIGUE, A. (2024). *Python Programming for Beginners: Zero to Hero: Mastering Python Step-by-Step*. Amazon Digital Services LLC – Kdp.

JANSEN, S. (2024). *pyfolio-reloaded: Performance and risk analysis of financial portfolios with Python* (0.9.5) [Python; OS Independent]. https://pypi.org/project/pyfolio-reloaded/

JARROW, R. A., RUDD, A. (1983). *Option Pricing*. R.D. Irwin.

JORION, P. (2006). *Value at Risk: The New Benchmark for Managing Financial Risk* (3rd edition). McGraw Hill.

KAHNEMAN, D., TVERSKY, A. (1979). Prospect Theory: An Analysis of Decision under Risk. *Econometrica*, *47*(2), 263–291. https://doi.org/10.2307/1914185

KONNO, H., YAMAZAKI, H. (1991). Mean-Absolute Deviation Portfolio Optimization Model and Its Applications to Tokyo Stock Market. *Management Science*, *37*(5), 519–531. https://doi.org/10.1287/mnsc.37.5.519

KRESTA, A. (2015). *Financial Engineering in Matlab: Selected Approaches and Algorithms*. VSB-Technical University of Ostrava.

KRESTA, A. (2016). *Kvantitativní metody investování s aplikacemi v prostředí Matlab*. VSB-Technical University of Ostrava.

KUPIEC, P. H. (1995). Techniques for Verifying the Accuracy of Risk Measurement Models. *The Journal of Derivatives*, *3*(2), 73–84. https://doi.org/10.3905/jod.1995.407942

LINTNER, J. (1965). Security Prices, Risk, and Maximal Gains from Diversification. *The Journal of Finance*, *20*(4), 587–615. https://doi.org/10.1111/j.1540-6261.1965.tb02930.x

LIU, J. (2024). *tushare: A utility for crawling historical and Real-time Quotes data of China stocks* (1.3.7) [Python; OS Independent]. https://tushare.pro

LONGSTAFF, F. A., SCHWARTZ, E. S. (2001). Valuing American Options by Simulation: A Simple Least-Squares Approach. *The Review of Financial Studies*, *14*(1), 113–147. https://doi.org/10.1093/rfs/14.1.113

MAILLARD, S., RONCALLI, T., TEÏLETCHE, J. (2010). The Properties of Equally Weighted Risk Contribution Portfolios. *The Journal of Portfolio Management*, *36*(4), 60–70. https://doi.org/10.3905/jpm.2010.36.4.060

MARKOWITZ, H. (1952). Portfolio Selection. *The Journal of Finance*, *7*(1), 77–91. https://doi.org/10.2307/2975974

MARTIN, R. A. (2023). *pyportfolioopt: Financial portfolio optimization in python* (1.5.5) [Python; OS Independent]. https://github.com/robertmartin8/PyPortfolioOpt

MOSSIN, J. (1966). Equilibrium in a Capital Asset Market. *Econometrica*, *34*(4), 768–783. https://doi.org/10.2307/1910098

MPL DEVELOPERS. (2023). *mplfinance: Utilities for the visualization, and visual analysis, of financial data* (0.12.10b0) [Python; OS Independent]. http://github.com/matplotlib/mplfinance

OLIPHANT, T. E. (2024). *numpy: Fundamental package for array computing in Python* (1.26.4) [C, Python; OS Independent]. https://numpy.org

PADIAL, D. L. (2023). *ta: Technical Analysis Library in Python* (0.11.0) [Python; OS Independent]. https://github.com/bukosabino/ta

PASSOUBADY, S. (2024). *empyrial: An Open Source Portfolio Management Framework for Everyone* (2.0.2) [Python; OS Independent]. https://github.com/ssantoshp/Empyrial

PYTHON SOFTWARE FOUNDATION. (2024). *Welcome to Python.org* [Computer software]. https://www.python.org/

QUANTOPIAN Inc. (2019). *pyfolio: Pyfolio is a Python library for performance and risk analysis of financial portfolios* (0.9.2) [Python; OS Independent]. https://github.com/quantopian/pyfolio

QUANTOPIAN Inc. (2020). *zipline: A backtester for financial algorithms.* (1.4.1) [Python; OS Independent]. https://zipline.io

RACHEV, S. T., MENN, C., FABOZZI, F. J. (2005). *Fat-Tailed and Skewed Asset Return Distributions: Implications for Risk Management, Portfolio Selection, and Option Pricing*. Wiley.

ROCKAFELLAR, T. R., URYASEV, S. (2000). Optimization of conditional value-at-risk. *Journal of Risk*, *2*(3), 21–42. https://doi.org/10.21314/JOR.2000.038

RODRIGUEZ, D. (2023). *backtrader: BackTesting Engine* (1.9.78.123) [Python; OS Independent]. https://github.com/mementum/backtrader

SHARPE, W. F. (1964). Capital Asset Prices: A Theory of Market Equilibrium Under Conditions of Risk. *The Journal of Finance*, *19*(3), 425–442. https://doi.org/10.1111/j.1540-6261.1964.tb02865.x

SHARPE, W. F. (1966). Mutual Fund Performance. *The Journal of Business*, *39*(1), 119–138. https://doi.org/10.1086/294846

SHARPE, W. F. (1994). The Sharpe Ratio. *The Journal of Portfolio Management*, *21*(1), 49–58. https://doi.org/10.3905/jpm.1994.409501

SHAW, Z. A. (2024). *Learn Python the Hard Way*. Addison-Wesley Professional.

SHEPPARD, K. (2024). *arch: ARCH for Python* (6.3.0) [Cython, Python; OS Independent]. https://pypi.org/project/arch/

SIRONI, A., RESTI, A. (2007). *Risk Management and Shareholders' Value in Banking: From Risk Measurement Models to Capital Allocation Policies*. John Wiley & Sons.

SORTINO, F. A., PRICE, L. N. (1994). Performance Measurement in a Downside Risk Framework. *The Journal of Investing*, *3*(3), 59–64. https://doi.org/10.3905/joi.3.3.59

SORTINO, F. A., SATCHELL, S. (eds.). (2001). *Managing Downside Risk in Financial Markets*. Butterworth-Heinemann.

TALEB, N. N. (2007). *Fooled by Randomness: The Hidden Role of Chance in Life and in the Markets*. Penguin Books.

TALEB, N. N. (2008). *The Black Swan: The Impact of the Highly Improbable*. Penguin Books.

THE PANDAS DEVELOPMENT TEAM. (2024). *pandas: Powerful data structures for data analysis, time series, and statistics* (2.2.0) [Cython, Python; OS Independent]. https://pandas.pydata.org

TIBILETTI, L., FARINELLI, S. (2003). Upside and downside risk with a benchmark. *Atlantic Economic Journal*, *31*(4), 387–387. https://doi.org/10.1007/BF02298499

TREYNOR, J. L. (1962). *Toward a theory of market value of risky assets*.

WASKOM, M. (2024). *seaborn: Statistical data visualization* (0.13.2) [Python; OS Independent]. https://pypi.org/project/seaborn/

ZMEŠKAL, Z., DLUHOŠOVÁ, D., TICHÝ, T. (2013). *Finanční modely: Koncepty, metody, aplikace* (3rd edition). Ekopress.

List of Figures

List of Tables

List of Codes

List of Questions

Index

APPLIED QUANTITATIVE FINANCE IN PYTHON: SELECTED THEORIES AND EXAMPLES

Aleš Kresta

Summary

The textbook *Applied Quantitative Finance in Python* focuses on a comprehensive exploration of the intricate convergence of finance, statistics, and computer science. Balancing theory and practical application, the textbook simplifies complex models into real-world Python code snippets. The book prioritizes practical applications over mathematical derivations, providing codes and detailed comments. The objective is guidance rather than exhaustive answers, with five chapters covering time series basics, risk measurement, portfolio optimization, technical analysis and automated trading, and options valuation. Whether a novice or an experienced practitioner, the book aims to be a valuable companion in the exploration of applied quantitative finance.

About the author

doc. Ing. Aleš Kresta, Ph.D. (1984)

Aleš Kresta is an Associate Professor in the Department of Finance at the Faculty of Economics, VSB – Technical University of Ostrava, where he earned his Ph.D. in Finance. His publication activity comprises more than twenty articles in international refereed journals and several books. He contributed to several research projects funded by the Czech Science Foundation. His primary research interests encompass risk estimation, backtesting, portfolio optimization, financial time series modeling, and soft computing within the realm of quantitative finance.

EDITORS' OFFICE
VSB – Technical University of Ostrava
Faculty of Economics, 17. listopadu 2172/15
708 00 Ostrava, Czech Republic
Assistant Editor: *Martina HUDÁKOVÁ*

PUBLISHER
VSB – Technical University of Ostrava
Faculty of Economics, 17. listopadu 2172/15
708 00 Ostrava, Czech Republic
IČ 61989100

Series of Economics Textbooks (SOET) is published by Faculty of Economics, VSB – Technical University of Ostrava since 2012. It covers a broad set of topics in business/economics disciplines, but mainly current issues in economics, finance, management, business economy, and informatics. Whereas the series of scientific monographs SAEI welcomes original results of research focusing on any of the topics mentioned above, within the sibling series SOET the publication of texts with more clear linkage to courses thought at the Faculty of Economics is encouraged.

The series addresses students and practitioners interested in up-to-date treatment of all economic disciplines. Manuscripts can be submitted to *soet@vsb.cz*. We kindly ask potential authors to follow the instructions about the structure of the book before they proceed to submission procedure. The text can be written either in *Czech* or *English*. The text's length shouldn't be less 100 pages, when the template is followed. Before publishing, each manuscript must be reviewed at least by two independent reviewers. The reviewing procedure is strictly double-blind. For further information authors may visit www.ekf.vsb.cz/soet.

APPLIED QUANTITATIVE FINANCE IN PYTHON: SELECTED THEORIES AND EXAMPLES

Aleš Kresta

Vydala VŠB – Technická univerzita Ostrava
1. vydání 2024
Tisk Ediční středisko VŠB-TUO
Náklad 40 ks, neprodejné
Počet stran 173

ISBN 978-80-248-4747-4 (print)
ISBN 978-80-248-4748-1 (on-line)

Volume 1
Hakalová, J., Palochová, M., Pšenková, Y., Bartková, H. (2012). *Účetnictví podnikatelských subjektů I*, SOET, vol. 1. Ostrava: VSB-TUO.
Volume 2
Navrátil, B., Kaňa, R., Zlý, B. (2012). *Evropská unie a integrační procesy. Terminologický slovník (aktualizovaný po Lisabonské smlouvě)*. SOET, vol. 2. Ostrava: VSB-TUO.
Volume 3
Velčovská, Š. (2013). *Product Management,* SOET, vol. 3. Ostrava: VSB-TUO.
Volume 4
Tichá, M. (2015). *Česká ekonomika na prahu 21. století v kontextu společenského vývoje, 2. aktualizované vydání,* SOET, vol. 4. Ostrava: VSB-TUO.
Volume 5
Horváthová, P. a kol. (2017). *Základy managementu, 2. vydání,* SOET, vol. 5. Ostrava: VSB-TUO.
Volume 6
Šotkovský, I. (2013). *Demografie. Teorie a praxe v regionálních souvislostech,* SOET, vol. 6. Ostrava: VSB-TUO.
Volume 7
Šebestíková, V. a kol. (2013). *Účetnictví podnikatelských subjektů II,* SOET, vol. 7. Ostrava: VSB-TUO.
Volume 8
Horváthová, P. (2013). *Essential of Management,* SOET, vol. 8. Ostrava: VSB-TUO.
Volume 9
Mruzková, J., Lisztwanová, K. (2013). *Teorie nákladů, kalkulace a ceny,* SOET, vol. 9. Ostrava: VSB-TUO.
Volume 10
Ministr, J., Novák, V., Pochyla, M., Rozehnal, P. (2013). *Osobní informatika,* SOET, vol. 10. Ostrava: VSB-TUO.
Volume 11
Němec, R. (2014). *Principy projektování a implementace systémů Business Intelligence,* SOET, vol. 11. Ostrava: VSB-TUO.
Volume 12
Horváthová, P. a kol. (2014). *Řízení lidských zdrojů pro pokročilé,* SOET, vol. 12. Ostrava: VSB-TUO.
Volume 13
Václavková, R. a kol. (2014). *Efektivní řízení obce − strategie, marketing, projekty, veřejné zakázky*, SOET, vol. 13. Ostrava: VSB-TUO.
Volume 14
Mikušová, M., Papalová, M. (2014). *Krizový management,* SOET, vol. 14. Ostrava: VSB-TUO.
Volume 15
Matusiková L. a kol. (2017). *Strategický management, 2. vydání*, SOET, vol. 15. Ostrava: VSB-TUO.

Volume 16
Macurová, P., Klabusayová, N., Tvrdoň, L. (2014). *Logistika*, SOET, vol. 16. Ostrava: VSB-TUO.
Volume 17
Horváthová, P., Čopíková, A. (2014). *Odměňování zaměstnanců v organizacích*, SOET, vol. 17. Ostrava: VSB-TUO.
Volume 19
Kirovová, I. (2017). *Organizační chování, 1. díl, 2. aktualizované vydání*, SOET, vol. 19. Ostrava: VSB-TUO.
Volume 20
Bláha, J., Černek, M. (2015). *Podnikatelská etika a CSR*, SOET, vol. 20. Ostrava: VSB-TUO.
Volume 21
Čopíková, A., Bláha, J., Horváthová, P. (2015). *Řízení lidských zdrojů*, SOET, vol. 21. Ostrava: VSB-TUO.
Volume 22
Černek, M., Staňková, Š. (2015). *Mezinárodní a interkulturální management*, SOET, vol. 22. Ostrava: VSB-TUO.
Volume 23
Fojtíková, L., Vahalík, B. (2017). *Praktická hospodářská politika ve vybraných zemích světové ekonomiky*, SOET, vol. 23. Ostrava: VSB-TUO.
Volume 24
Vrabková, I. (2016). *Veřejná správa*, SOET, vol. 24. Ostrava: VSB-TUO.
Volume 25
Bartková, H. (2016). *Historie, vývoj a regulace účetnictví v České republice*, SOET, vol. 25. Ostrava: VSB-TUO.
Volume 26
Horváthová, P., Čopíková A. (2017). *Human Resource Management*, SOET, vol. 26. Ostrava: VSB-TUO.
Volume 27
Mikušová, M. (2017). *Crisis management*, SOET, vol. 27. Ostrava: VSB-TUO.
Volume 28
Funioková, T. (2017). *Introduction to Financial Mathematics*, SOET, vol. 28. Ostrava: VSB-TUO.
Volume 29
Friedrich, V., Hradecký, P., Michalcová, Š., Pomp, M. (2018). *Vybrané statistické metody*, SOET, vol. 29. Ostrava: VSB-TUO.
Volume 30
Mikušová, M. (2018). *Family Business*, SOET, vol. 30. Ostrava: VSB-TUO.
Volume 31
Kovářová, E. (2019). *Globalizace a vybrané globální problémy: Vybraná témata a souvislosti*, SOET, vol. 31. Ostrava: VSB-TUO.
Volume 32
Horváthová, P. (2019). *Teams and Teamwork*, SOET, vol. 32. Ostrava: VSB-TUO.

Volume 33
Horváthová, P., Mikušová M. a kol. (2019). *Trendy v managementu*, SOET, vol. 33. Ostrava: VSB-TUO.
Volume 34
Vrabková, I. (2022). *Úvod do ekonomie veřejného sektoru*, SOET, vol. 34. Ostrava: VSB-TUO.
Volume 35
Novák, V. (2023). *Analýza dat v Microsoft Excelu*, SOET, vol. 35. Ostrava: VSB-TUO (on-line).
Volume 36
Halásková, M. (2023). *Veřejná správa v zemích Evropské unie*, SOET, vol. 36. Ostrava: VSB-TUO (on-line).
Volume 37
Tománek, P. (2023). *Veřejné rozpočty*, SOET, vol. 37. Ostrava: VSB-TUO (on-line).

Aleš Kresta

Applied Quantitative Finance in Python: Selected Theories and Examples

vol. 38 | 2024

The book Applied Quantitative Finance in Python provides a comprehensive guide to the intersection of finance, statistics, and computer science. Balancing theory and practical application, it uses Python to simplify complex financial models with real-world code snippets. It covers essential principles and advanced topics, guiding readers through real-world financial problems.

Whether you are a novice or an experienced practitioner, this book aims to be a companion in exploring applied quantitative finance. Starting with the basics of time series, the book progresses to risk measurement, portfolio optimization, technical analysis, and option valuation. Each chapter offers hands-on Python examples, highlighting potential pitfalls and providing practical insights.

The book was written with the support of VSB – Technical University of Ostrava under project number SP2024/047 and the European Union operational program Just Transition under project number CZ.10.03.01/00/22_003/0000048.

Published by

VSB TECHNICAL UNIVERSITY OF OSTRAVA | FACULTY OF ECONOMICS

17. listopadu 2172/15
708 00 Ostrava
Tel.: +420 597 322 421
E-mail: soet@vsb.cz

ISBN 978-80-248-4748-1 (on-line)
DOI 10.31490/9788024847481