Hochschule für Technik Stuttgart

Masterarbeit

vorgelegt von
Axel Roth
Masterstudiengang Mathematik
Fakultät Vermessung, Informatik und Mathematik
Wintersemester 2022/2023

Asset Allocation using Particle Swarm Optimization in R

ErstprüferIn: Prof. Dr. Harald Bauer
ZweitprüferIn: Prof. Dr. Stefan Reitz

Contents

List of Figures

List of Abbreviations

Abbreviation	Definition
xts	R object used to store time series data
IDE	Integrated development environment
MVP	Mean-variance portfolio
ITP	Index tracking problem
TEV	Tracking error variance
MSTE	Mean square tracking error
QP	Quadratic programming
SP500	Standard and Poor's 500 Index
SP500TR	Standard and Poor's 500 Total Return Index

Abstract

Asset allocation aims at constructing portfolios that follow a specific investment strategy, such as index tracking or optimal risk return. These strategies are formulated as optimization problems whose solutions represent an optimal portfolio depending on the strategy. In practice, these problems can be too complicated to be solved directly by deterministic optimization methods, which can lead to simplified models that may contain faulty assumptions or inaccuracies. When this is the case, metaheuristics such as Particle Swarm Optimization (PSO) may be needed. PSO is able to find optimal solutions to complex problems by evaluating random positions in the search space to iteratively approach the optimum. In this thesis, common asset allocation problems are first explained and solved using deterministic optimization methods to generate benchmark problems. After introducing the PSO, it is illustrated with simple examples and the convergence behavior is analyzed before it is subsequently tested on benchmark problems. Then, the PSO and its variants are used to solve optimization problems that cannot be solved by deterministic optimization methods. Finally, the most promising PSO variant is used to solve a discrete index tracking problem considering transaction costs and a rebalancing constraint. Backtests show that this PSO variant can provide time-stable results for such problems.

Chapter 1

Introduction

In the past, active portfolio management was the only approach used, where the portfolio manager actively bought and sold assets to create a portfolio that meets the client's objectives. The goal is to take advantage of assets that are traded at incorrect prices to make a profit. This approach requires a high level of expertise from the portfolio manager, who must be well-informed about the financial market and each asset in the portfolio. However, it is becoming more difficult to spot these false prices as the market becomes more efficient. An efficient market is one where the quoted prices of assets reflect all available information, making it impossible to exploit false prices. This has led to the development of passive portfolio management, a quantitative approach that focuses more on the overall portfolio rather than individual assets. This approach has proven successful, leading to a shift towards passive portfolio management in recent years. In 2014, 18% of the total net asset value of all funds was passively managed, and this share has increased to 25% in 2018 [Derenzis, 2019]. The reason for this is complex, but passive management is characterized by many data-driven quantitative strategies, which automate much of the work and save costs by allowing fewer portfolio managers to manage more portfolios. Additionally, passively managed portfolios typically have less rebalancing, which keeps transaction costs lower. In addition to the lower cost, passive portfolios are also often less risky, which is of interest to many investors.

Building a cost-efficient, fully automated portfolio using a quantitative strategy can be challenging due to the complexity of the optimization problems that must be solved. To overcome these challenges, metaheuristics such as Particle Swarm Optimization (PSO) are often used. A metaheuristic is a higher-level optimization method that treats the optimization problem as a black box, only requiring the ability to evaluate the objective function. However, this approach does not provide any guarantee about the quality of the solution, and the runtime is heavily dependent on the time needed to evaluate the function. Thus, it is crucial to thoroughly test a metaheuristic before using it. Research from [Mercangöz, 2021] suggests that PSO is more effective than other metaheuristics in various areas, including portfolio optimization problems that cannot be solved by deterministic optimization methods. The goal of this thesis is to apply PSO to financial problems, develop a deep understanding of its behavior, and make

the code and data for this thesis freely available for readers to use and apply PSO themselves.

Another objective of this thesis is to provide readers with an introduction to the R programming language, which is particularly well-suited for the purpose of financial data analysis. R is a powerful tool that allows for the implementation of complex algorithms in a user-friendly and intuitive manner. Additionally, R is a widely adopted programming language in various practical domains, including portfolio management, and is frequently used for data analysis and modeling.

The following two chapters present an overview of the software and data utilized, with the aim of ensuring reproducibility. Subsequently, in chapter 4, the mathematical foundations and conventions are outlined. Chapter 5 conducts an analysis of active and passive portfolio management, highlighting the advantages of the latter. The subsequent chapter addresses two common objectives of passive portfolio management, namely the index tracking portfolio and the optimal risk return portfolio, which serve as test optimization problems for the application of the PSO algorithm. In order to be able to validate the PSO, a deterministic optimization method is explained in chapter 7, which can calculate the optimal solution in the continuous case of the two test problems. The core focus of the thesis, the standard PSO and its behavior, is presented in chapter 8. The test problems from the previous chapter are used to evaluate the performance of the standard PSO. Additionally, a discrete problem that cannot be solved by deterministic optimization methods is demonstrated and solved using the standard PSO. In chapter 9, various modifications of the PSO are evaluated by solving a discrete problem, with the standard PSO serving as a benchmark. The results indicate that the local PSO and the self-adaptive velocity PSO are most suitable for such problems. Finally, in chapter 10, the self-adaptive velocity PSO is applied to solve a real-world index tracking problem, considering transaction costs and a maximum rebalancing weight constraint. The results of the backtests suggest the potential utility of the PSO in solving such complex problems.

Chapter 2

Information about the used Software

All of the thesis was done in the R-Studio IDE using R Markdown extended with the bookdown package. R Markdown allows the user to execute R code in so-called code chunks that are located between the texts. These code chunks can be used, for example, to generate charts that can be displayed in the output instead of the code. Each chunk of code can be cached to avoid time-consuming repetition, and the caching can be disabled to recreate the entire thesis, including all the code and charts it contains. This is helpful primarily for the user to ensure stability of older code, and secondarily for the reader who has access to the R-Markdown files, as they can trace back any analysis or diagram displayed. R-Markdown can produce various formats, such as HTML or PDF output, both of which are used for this thesis. The HTML output is deposited in the corresponding GitHub repository at [Roth, 2022a]. The recommended output format of R-Markdown is an HTML file, which has all the capabilities of a web page with HTML, CSS, and Javascript. For this reason, some javascript code has been added to the HTML version to allow the reader to expand important blocks of code alongside the text. The bookdown package is used as a template that contains the necessary styles and formatting to write books with an HTML or PDF output including table of contents, references and more. Together with the possibilities of the programming language R, a reproducible and traceable thesis can be created.

2.1 R Version and Packages

The used packages and versions can be found in the table below:

```
R version 4.1.1 (2021-08-10)
Platform: x86_64-w64-mingw32/x64 (64-bit)
Running under: Windows 10 x64 (build 19044)

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods
[7] base
```

```
other attached packages:
 [1] ggpubr_0.5.0      metR_0.12.0       gganimate_1.0.7
 [4] quantmod_0.4.20   TTR_0.24.3        reactable_0.3.0
 [7] bookdown_0.31     webshot2_0.1.0    corrr_0.4.3
[10] data.table_1.14.2 pso_1.0.3         plotly_4.10.0
[13] matrixcalc_1.0-5  Matrix_1.5-3      quadprog_1.5-8
[16] xts_0.12.1        zoo_1.8-9         rvest_1.0.2
[19] forcats_0.5.1     stringr_1.4.0     purrr_0.3.4
[22] tidyr_1.2.0       tibble_3.1.8      ggplot2_3.4.0
[25] tidyverse_1.3.2   readr_2.1.2       dplyr_1.0.9
[28] lubridate_1.8.0

loaded via a namespace (and not attached):
 [1] fs_1.5.2            progress_1.2.2      httr_1.4.3
 [4] tools_4.1.1         backports_1.2.1     utf8_1.2.2
 [7] R6_2.5.1            DBI_1.1.3           lazyeval_0.2.2
[10] colorspace_2.0-2    withr_2.5.0         prettyunits_1.1.1
[13] tidyselect_1.1.2    processx_3.5.2      curl_4.3.2
[16] compiler_4.1.1      cli_3.3.0           xml2_1.3.3
[19] checkmate_2.0.0     scales_1.2.1        digest_0.6.29
[22] rmarkdown_2.19      pkgconfig_2.0.3     htmltools_0.5.3
[25] dbplyr_2.2.1        fastmap_1.1.0       htmlwidgets_1.5.3
[28] rlang_1.0.6         readxl_1.4.0        rstudioapi_0.13
[31] farver_2.1.0        generics_0.1.3      jsonlite_1.8.0
[34] car_3.1-1           googlesheets4_1.0.0 magrittr_2.0.3
[37] Rcpp_1.0.9          munsell_0.5.0       fansi_0.5.0
[40] abind_1.4-5         lifecycle_1.0.3     stringi_1.7.6
[43] yaml_2.3.5          carData_3.0-5       grid_4.1.1
[46] promises_1.2.0.1    crayon_1.5.1        lattice_0.20-44
[49] haven_2.5.0         chromote_0.1.0      hms_1.1.1
[52] knitr_1.41          ps_1.6.0            pillar_1.8.0
[55] ggsignif_0.6.4      reprex_2.0.1        glue_1.6.2
[58] evaluate_0.19       modelr_0.1.8        tweenr_1.0.2
[61] vctrs_0.5.1         tzdb_0.3.0          cellranger_1.1.0
[64] gtable_0.3.0        assertthat_0.2.1    xfun_0.35
[67] broom_1.0.0         rstatix_0.7.1       later_1.3.0
[70] googledrive_2.0.0   viridisLite_0.4.0   gargle_1.2.0
[73] websocket_1.4.0     ellipsis_0.3.2
```

2.2 Reproducibility

Everything in this thesis can be reproduced via the public GitHub repository at [Roth, 2022a]. All code sections that use random number generators start with seed 0 to ensure the credibility of the analyses and diagrams shown.

2.3 R Functions

The following functions were created to simplify working with R Markdown and to take advantage of the HTML version of this thesis.

2.3.1 `html_save()`

Converts HTML plots to images to display them in the PDF output. This allows the author to use the desired plot packages without having to worry about the different output formats. The disadvantage is, that it is not possible to define the font size within the charts and tables exactly, because the saved images are adjusted to the page width by latex.

2.3.2 javascript files

The javascript files are located in the `www/` folder of the attached GitHub repository and are used for the HTML version of this thesis to make code blocks expandable. This allows the reader to inspect the code used and to follow analyses in detail.

Chapter 3

Open Data Sources

To increase reproducibility, all data are free and can be loaded from the quantmod R package with the function `getSymbols()`. It is possible to choose between different data sources like yahoo-finance (default), alpha-vantage and others.

3.1 R Functions

The following functions were created to increase the ease of data collection with the quantmod R package, which can be found in the `R/` directory in the dedicated GitHub repository [Roth, 2022a].

3.1.1 `get_yf()`

This function is the main wrapper for collecting data with `getSymbols()` from yahoo-finance, and converts prices to returns with the `pri_to_ret()` function explained in section 4.6.2. The output is a list containing prices and returns as xts objects, which is the usual object in R for storing time series. The arguments that can be passed to `get_yf()` are:

- `tickers`: Vector of symbols (asset names, e.g. "APPL", "GOOG", ...)
- `from ="2018-01-01"`: R Date
- `to = "2019-12-31"`: R Date
- `price_type = "close"`: Type of prices to be recorded (e.g. "open", "high", "low", "close", "adjusted")
- `return_type = "adjusted"`: Type of prices used to calculate the returns (e.g. "open", "high", "low", "close", "adjusted")
- `print = F`: Should the function print the return of `getSymbols()`

3.1.2 `buffer()`

To make data reusable and reduce compilation time, the `buffer()` function stores the data collected with `get_yf()`. It receives an R expression, evaluates it and stores it in the `buffer_data/` directory under the specified name. If this name already exists, it loads the R object from the RData files without evaluating the expression. The evaluation and overwriting of the existing RData file can be forced with `force=T`.

Chapter 4

Mathematical Foundations

This chapter provides an overview of the mathematical calculations and conventions used in this thesis. It is important to note that most mathematical formulas are written in matrix notation and vectors are interpreted as one-dimensional matrices. In most cases, this will result in a direct translation to R code. All necessary assumptions required for the modeled return structure are listed in this chapter so that any reader can understand the formulas given. It is important to note that reality is too complex and can only be partially modeled. Simple, basic models are used that do not stand up to reality, but these models or variations of them are commonly used in the financial world and have proven to be helpful. The complexity of solving advanced and basic models does not differ for the PSO because the dimension of the objective function is based on the number of elements that can be selected, see chapter 6.

4.1 Basic Operators

The programming language R makes it possible to formulate almost all computations in a matrix notation, since the usual matrix operators and element-wise operators are already efficiently implemented in base R. Element-wise operators are not considered standard in mathematics, so they are defined in more detail in this section. The representation of calculations in a matrix notation helps the user to keep the overview, because the formulas are more compact and in addition, these operators are already implemented efficiently in base R, which makes them more performant than simple for-loops. The operators used here can be found in the table below and are intended to establish the relationship between the notation in the text, in the R code and their meaning:

Latex/Displayed	R Code	Meaning
×	%*%	Matrix product
⊗	%*%	Outer product (special case of the matrix product)
A^T	t(A)	Transpose of matrix or vector A
·	*	Scalar or element-wise matrix multiplication
/	/	Scalar or element-wise matrix division
+	+	Scalar or element-wise matrix addition
-	-	Scalar or element-wise matrix subtraction

Figure 4.1: List of used operators and their meanings

For a better understanding of the operators listed, the following examples are intended to illustrate the resulting dimensions and provide insight into the use of these operators.

The matrix product is defined by the following mapping:

$$\times: \quad \mathbb{R}^{x\times y} \times \mathbb{R}^{y\times z} \to \mathbb{R}^{x\times z},$$

with an example:

$$\begin{bmatrix} a_{1,1} & \cdots & a_{1,y} \\ \vdots & \ddots & \vdots \\ a_{x,1} & \cdots & a_{x,y} \end{bmatrix} \times \begin{bmatrix} b_{1,1} & \cdots & b_{1,z} \\ \vdots & \ddots & \vdots \\ b_{y,1} & \cdots & b_{y,z} \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{y} a_{1,i} \cdot b_{i,1} & \cdots & \sum_{i=1}^{y} a_{1,i} \cdot b_{i,z} \\ \vdots & \ddots & \vdots \\ \sum_{i=1}^{y} a_{x,i} \cdot b_{i,1} & \cdots & \sum_{i=1}^{y} a_{x,i} \cdot b_{i,z} \end{bmatrix}.$$

In the case of vectors, the matrix product has the following mapping:

$$\times: \quad \mathbb{R}^{1\times N} \times \mathbb{R}^{N\times 1} \to \mathbb{R},$$

with an example:

$$\begin{bmatrix} a_1 & \cdots & a_N \end{bmatrix} \times \begin{bmatrix} b_1 \\ \vdots \\ b_N \end{bmatrix} = \sum_{i=1}^{N} a_i \cdot b_i.$$

The outer product is defined by the following mapping:

$$\otimes: \quad \mathbb{R}^{x\times 1} \times \mathbb{R}^{1\times y} \to \mathbb{R}^{x\times y},$$

with an example:

$$\begin{bmatrix} a_1 \\ \vdots \\ a_x \end{bmatrix} \otimes \begin{bmatrix} b_1 & \cdots & b_y \end{bmatrix} = \begin{bmatrix} a_1 \cdot b_1 & \cdots & a_1 \cdot b_y \\ \vdots & \ddots & \vdots \\ a_x \cdot b_1 & \cdots & a_x \cdot b_y \end{bmatrix}.$$

This thesis is specified for the use of R, so element-wise operators are very important to make code comparable with formulas. In mathematics, these operators are not common. For this reason, they must be explicitly specified. All element-wise operators work in the same way. Assuming that $\square$ is one of the four element-wise operators, then the mapping is defined as follows:

$$\begin{aligned} \square: &\quad R^{x \times y} \times R^{x \times y} \to R^{x \times y} \\ \text{or } \square: &\quad R^{x} \times R^{x \times y} \to R^{x \times y} \\ \text{or } \square: &\quad R \times R^{x \times y} \to R^{x \times y}, \end{aligned}$$

with examples respectively:

$$\begin{bmatrix} a_{11} & \cdots & a_{1y} \\ \vdots & \ddots & \vdots \\ a_{x1} & \cdots & a_{xy} \end{bmatrix} \square \begin{bmatrix} b_{11} & \cdots & b_{1y} \\ \vdots & \ddots & \vdots \\ b_{x1} & \cdots & b_{xy} \end{bmatrix} = \begin{bmatrix} a_{11} \square b_{11} & \cdots & a_{1y} \square b_{1y} \\ \vdots & \ddots & \vdots \\ a_{x1} \square b_{x1} & \cdots & a_{xy} \square b_{xy} \end{bmatrix}$$

or

$$\begin{bmatrix} a_1 \\ \vdots \\ a_x \end{bmatrix} \square \begin{bmatrix} b_{11} & \cdots & b_{1y} \\ \vdots & \ddots & \vdots \\ b_{x1} & \cdots & b_{xy} \end{bmatrix} = \begin{bmatrix} a_1 \square b_{11} & \cdots & a_1 \square b_{1y} \\ \vdots & \ddots & \vdots \\ a_x \square b_{x1} & \cdots & a_x \square b_{xy} \end{bmatrix}$$

or

$$a \square \begin{bmatrix} b_{11} & \cdots & b_{1y} \\ \vdots & \ddots & \vdots \\ b_{x1} & \cdots & b_{xy} \end{bmatrix} = \begin{bmatrix} a \square b_{11} & \cdots & a \square b_{1y} \\ \vdots & \ddots & \vdots \\ a \square b_{x1} & \cdots & a \square b_{xy} \end{bmatrix}.$$

The operators on the right side of the above example equations corresponds to the common operators in scalar.

4.2 Formula Conventions

In mathematics, random variables are written in capital letters, which also applies to matrices. In order to allow an unambiguous assignment, the random variables are written in bold and in capital letters. For example, A should represent a matrix and $\boldsymbol{A}$ should represent a random variable.

4.3 Return Calculation

Any portfolio optimization strategy based on historical data must start with returns. These returns are calculated using adjusted closing prices, which show the percentage change over time. Adjusted closing prices reflect dividends and are adjusted for stock splits and rights offerings. These returns are essential for comparing assets and analyzing dependencies.

4.3.1 Simple Returns

The default time frame for all raw data in this thesis is a working day, prices are always measured in the evening, and only simple returns are used. Assuming that there is an asset with price p_{t_i} on working day t_i and price $p_{t_{i+1}}$ on the following working day t_{i+1}, the simple rate of return for t_{i+1} can be calculated as follows:

$$r_{t_{i+1}} = \frac{p_{t_{i+1}}}{p_{t_i}} - 1.$$

4.4 Markowitz Modern Portfolio Theory

In 1952, Harry Markowitz published his first seminal paper, which had a significant impact on modern finance, primarily by outlining the implications of diversification and efficient portfolios in [Markowitz, 1952]. He later published a monograph on a more in-depth analysis of his findings in [Markowitz, 1959]. The definition of an efficient portfolio is a portfolio that has either the maximum expected return for a given risk target or the minimum risk for a given expected return target. A simple quote to define diversification might be, "A portfolio has the same return but less variance than the sum of its parts." This is the case when assets are not perfectly correlated, as poor and good performances can offset each other, reducing the likelihood of extreme events. For more information, see [Maringer, 2005].

4.4.1 Assumptions of Markowitz Portfolio Theory

This thesis focuses on problems derived from Markowitz's portfolio theory, without closed form solutions, which are studied in [Maringer, 2005] by excluding short selling. The following list contains these types of Markowitz assumptions according to [Maringer, 2005] and [Markowitz, 1952]:

- Perfect market without taxes or transaction costs
- Assets are infinitely divisible
- Short sales are disallowed
- Expected returns, variances and covariances contain all information
- Investors are risk-averse, they will only accept greater risk if they are compensated with a higher expected return

The assumption that the returns are normally distributed is not required, but is assumed in this case to simplify the problem. It is obvious that these assumptions are unrealistic in reality. More details on the requirements for using other distributions can be found in [Maringer, 2005].

4.5 Portfolio Math

Proofs of the basic calculations required for portfolio optimization, as shown in [Zivot, 2021], are provided in this section. Returns are presented differently than in most sources, as this is the most common data format used in practice. Suppose there are N assets described by a return vector $\boldsymbol{R}$ of random variables,

and the relative contribution to the net asset value of the portfolio is described by the weight vector w, which typically satisfies $\sum w = 1$:

$$\boldsymbol{R} = \begin{bmatrix} \boldsymbol{R}_1 & \boldsymbol{R}_2 & \cdots & \boldsymbol{R}_N \end{bmatrix}, \quad w = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_N \end{bmatrix}.$$

In this thesis, each return is simplified as being normally distributed with $\boldsymbol{R}_i = \mathbb{N}(\mu_i, \sigma_i^2)$. As a result, linear combinations of normally distributed random variables are jointly normally distributed and have a mean, variance, and covariance that can be used to fully describe them.

4.5.1 Expected Returns

The following formula can be used to get the expected returns of a vector with normally distributed random variables $\boldsymbol{R} \in \mathbb{R}^{1\times N}$:

$$\begin{aligned} E[\boldsymbol{R}] &= \begin{bmatrix} E[\boldsymbol{R}_1] & E[\boldsymbol{R}_2] & \cdots & E[\boldsymbol{R}_N] \end{bmatrix} \\ &= \begin{bmatrix} \mu_1 & \mu_2 & \cdots & \mu_N \end{bmatrix} = \mu \end{aligned}$$

and μ_i can be estimated in R using historical data and the formula for the geometric mean of returns (also called compound returns). The function to calculate the geometric mean of returns from an xts object can be found in section 4.6.4.

4.5.2 Expected Portfolio Returns

The following equation can be used to obtain the expected portfolio return $E[\boldsymbol{R}_p]$ using the formulations from the section above and a weighting vector w (e.g. portfolio weights):

$$\begin{aligned} E[\boldsymbol{R}_p] &= E[\boldsymbol{R} \times w] = E[\boldsymbol{R}] \times w \\ &= \begin{bmatrix} E[\boldsymbol{R}_1] & E[\boldsymbol{R}_2] & \cdots & E[\boldsymbol{R}_N] \end{bmatrix} \times \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_N \end{bmatrix} \\ &= \begin{bmatrix} \mu_1 & \mu_2 & \cdots & \mu_N \end{bmatrix} \times \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_N \end{bmatrix} \\ &= \mu_1 \cdot w_1 + \mu_2 \cdot w_2 + \cdots + \mu_N \cdot w_N = \mu_P \end{aligned}$$

4.5.3 Covariance

The general formula of the covariance matrix $\sum$ of a random vector $\boldsymbol{R}$ with N normally distributed elements and $\sigma_{i,j}$ as correlation of two distinct values is described as follows:

$$
\begin{aligned}
Cov(\boldsymbol{R}) &= E[(\boldsymbol{R}-\mu)^T \otimes (\boldsymbol{R}-\mu)] \\
&= \begin{bmatrix} \sigma_1^2 & \sigma_{1,2} & \cdots & \sigma_{1,N} \\ \sigma_{2,1} & \sigma_2^2 & \cdots & \sigma_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ \sigma_{N,1} & \sigma_{N,2} & \cdots & \sigma_N^2 \end{bmatrix} \\
&= \sum
\end{aligned}
$$

and can be estimated in R using the base function `cov()` and historical data. The function `cov_()` explained in section 4.6.1 can be used to calculate the covariance using the geometric mean to estimate the expected returns.

4.5.4 Portfolio Variance

Let $\boldsymbol{R}$ be a random vector with N normally distributed elements and w a weight vector. Assuming that the covariance matrix $\sum$ of $\boldsymbol{R}$ is known, the variance of the linear combination of $\boldsymbol{R}$ can be calculated as follows:

$$
\begin{aligned}
Var(\boldsymbol{R} \times w) &= E[(\boldsymbol{R} \times w - \mu \times w)^2] \\
&= E[((\boldsymbol{R}-\mu) \times w)^2]
\end{aligned}
$$

Since $(\boldsymbol{R}-\mu) \times w$ is a scalar, it can be transformed from $((\boldsymbol{R}-\mu) \times w)^2$ to $((\boldsymbol{R}-\mu) \times w)^T \cdot ((\boldsymbol{R}-\mu) \times w)$ and results in:

$$
\begin{aligned}
Var(\boldsymbol{R} \times w) &= E[((\boldsymbol{R}-\mu) \times w)^T \cdot ((\boldsymbol{R}-\mu) \times w)] \\
&= E[(w^T \times (\boldsymbol{R}-\mu)^T) \cdot ((\boldsymbol{R}-\mu) \times w)] \\
&= w^T \times E[(\boldsymbol{R}-\mu)^T \otimes (\boldsymbol{R}-\mu)] \times w \\
&= w^T \times Cov(\boldsymbol{R}) \times w \\
&= w^T \times \sum \times w.
\end{aligned}
$$

The same is true for an estimate of $Var(\boldsymbol{R} \times w)$ by using the estimate of $\sum$.

4.5.5 Portfolio Returns

Suppose there are N assets that form a portfolio with weights w_{t_0} at time step t_0, and the portfolio is to go through several time steps until t_T without rebalancing. What are the portfolio returns at each time step t_i? Clearly, assets with higher performance at the current time step will have a higher weight at the next time step. This can be done by adjusting the weights after each time step as a function of returns. Suppose we have a complete portfolio $\sum w_{t_0} = 1$ with a return matrix $R \in \mathbb{R}^{T \times N}$ and we want to calculate the portfolio returns $R_{t_1}^P$. This can be done with the following two steps:

$$
\begin{aligned}
Z_{t_1} &= (1 + R_{t_1}) \cdot w_{t_0}^T \\
R_{t_1}^P &= \sum_{n=1}^{N} Z_{t_1,n} - 1.
\end{aligned}
$$

And since the portfolio was full in t_0, the adjusted weights in t_1 are $w_{t_1} = Z_{t_1} / \sum_{n=1}^{N} Z_{t_1,n}$ (element-wise division). These new weights can be used in the next time step to replace the weights w_{t_0} in the above formula. The recursive

formula for holding a portfolio with weights $\sum w_{t_0} = 1$ in t_0 and return matrix $R \in \mathbb{R}^{T \times N}$, has portfolio return $R^P_{t_i} = \sum_{n=1}^{N} Z_{t_i,n} - 1$ in t_i with $i = 1, 2, \cdots, T$ for:

$$Z_{t_i} = \begin{cases} (1 + R_{t_i}) \cdot w^T_{t_0} & \text{if } i = 1 \\ (1 + R_{t_i}) \cdot \frac{Z_{t_{i-1}}}{\sum_{n=1}^{N} Z_{t_{i-1},n}} & \text{if } i > 1. \end{cases}$$

The requirement that the portfolio is full can be achieved by adding an additional weight to w_{t_0}, which includes the residual $1 - \sum w_{t_0}$ and an additional zero return vector to R. This calculation of portfolio returns is implemented in the function `calc_portfolio_returns()` below.

4.6 R Functions

The following functions serve the purpose to encapsulate mathematical calculations and to make the calculations reusable. All functions are located in the dedicated GitHub repository inside the `R/` folder.

4.6.1 `cov_()`

This function extends the base R function `cov()` by allowing the user to pass an expected value vector. This is used to calculate the covariance with expected returns based on the geometric mean instead of the arithmetic mean.

4.6.2 `pri_to_ret()`

This function calculates the simple returns of a given xts object containing prices. Before that, missing prices are replaced by the previous value in the respective column.

4.6.3 `ret_to_cumret()`

This function is used to calculate cumulative returns normalized to 100 from a given xts object containing returns. This function is often used before plotting return time series.

4.6.4 `ret_to_geomeanret()`

The geometric mean of returns is a better estimator than the arithmetic mean of returns because it captures the exact mean price changes over a period of time. The variance estimated from the daily returns is a daily variance, so the returns must have the same time base. This can be done by calculating the geometric mean of the returns from multiple daily returns. Assuming there is an asset with returns $r_1 = 0.01$, $r_2 = -0.03$, and $r_3 = 0.02$, it follows that the geometric mean return r^{geom} can be calculated as:

$$r^{geom} = ((1 + r_1) \cdot (1 + r_2) \cdot (1 + r_3))^{1/3} - 1 = -0.0002353887$$

And the advantage is that it is a daily average return that gives exactly the same result as the real returns, that is:

$$(1 + r^{geom})^3 = (1 + r_1) \cdot (1 + r_2) \cdot (1 + r_3)$$

This is not the case with the arithmetic mean of the returns. The general formula for calculating the geometric mean return of n days is:

$$r^{geom} = (\prod_{i=1}^{n}(1 + r_i))^{\frac{1}{n}} - 1$$

and as R code:

```
ret_to_geomeanret <- function(xts_ret){
  sapply((1+xts_ret), prod)^(1/nrow(xts_ret))-1
}
```

4.6.5 `calc_portfolio_returns()`

This is the implementation of a vectorial calculation of portfolio returns over multiple periods with a weighting vector `weights` at t_0 and no re-balancing:

```
calc_portfolio_returns <-
  function(xts_returns, weights, name="portfolio"){
  if(sum(weights)!=1){
    xts_returns$temp___X1 <- 0
    weights <- c(weights, 1-sum(weights))
  }
  res <- cumprod((1+xts_returns)) * matrix(
    rep(weights, nrow(xts_returns)), ncol=length(weights), byrow=T)
  res <- xts(
    rowSums(res/c(1, rowSums(res[-nrow(xts_returns),]))-1,
    order.by=index(xts_returns)) %>%
    setNames(., name)
  return(res)
}
```

This function has the same results as the `Return.portfolio()` function from the `PortfolioAnalytics` package.

Chapter 5

Active vs. Passive Portfolio Management

An active portfolio manager seeks to achieve positive alpha, i.e., excess returns relative to the market, by applying his knowledge, experience, and in-depth analysis of individual assets. Therefore, the manager assumes that the market is not efficient and tries to select mispriced assets to generate excess returns. The passive portfolio manager, on the other hand, assumes that the market is efficient, that is, that prices reflect all available information, and attempts to replicate the average market return by building a diversified portfolio. He knows that stock movements follow a "random walk" and are therefore unpredictable for any individual stock. The passive portfolio manager achieves his goal with a quantitative strategy and assumes that the results will be stable over time. ETFs that aim to track indices are classified as passively managed in this thesis.

The question is which of the two types of portfolio management is preferable.

5.1 Results of the Literature

Researchers Fama and French studied the returns of active and passive portfolio management by designing factor models that led to a wide range of models commonly used today to analyze the causes of returns. Their theory states that passive investors earn passive returns that have an alpha of zero before costs. This implies that active investors also collectively have an alpha of zero before costs. This means that if some active investors have a positive alpha, other active investors have achieved a negative alpha. Indeed, Fama and French analyzed this behavior in more detail in [Fama and French, 2010], finding that value-weighted professionally managed mutual funds that invest primarily in the U.S. equity market have a slightly positive alpha before costs at the expense of active investors outside of professionally managed mutual funds. In addition, Fama and French attempted to use regression models to distinguish between luck and skill in active portfolio management. Their results suggest that some active managers in the top percentiles have sufficient skill to cover costs over the long run. Nevertheless, actively managed funds in the top percentiles have

an estimated alpha after costs that is close to zero, which is also true for large, efficiently managed passive funds.

Similar results were obtained in [Pace et al., 2016b], which analyzed European and U.S. active and passive funds. The results show that none of them is superior by cost. They suggest comparing active and passive managed funds on a case-by-case basis by considering all expenses. When the tax advantages of passively managed funds are taken into account, they can be slightly advantageous for investors with a long investment horizon.

A different viewpoint was taken in [Shukla, 2004], analyzing the value of active portfolio management. The results show that some actively managed funds generate large positive excess returns. These funds often had a small number of assets, which could be due to the fact that portfolio managers had more capacity to analyze each asset in detail. This suggests that experienced portfolio managers can indeed generate stable excess returns over time. But in contrast, these same funds charge increased fees that negate the benefits to clients.

The last source analyzed passive and active EU equity funds in [Derenzis, 2019]. It was observed that the top 25% of actively managed funds outperform passively managed funds, but the group of actively managed funds changes over time. This leads to increased risks for investors as they need to buy and sell funds at the right time. They also analyzed the increasing popularity of passively managed funds, whose market capitalization increased by more than 61% between 2014 and 2018. On the other hand, the market capitalization of actively managed funds increased by only 16% overall, indicating a significant shift toward passively managed funds. After comparing the aggregated performance after costs between 2009 and 2018, it was found that the group of actively managed funds with smaller market capitalization lagged behind the larger ones and the larger ones lagged behind the passively managed funds.

5.2 Aggregation of the Literature

In summary, it can be said that both strategies have their justification. On the one hand, it was shown that there are actively managed funds that can constantly generate positive alpha and share this in part with the investor. But this refers only to a small part of the actively managed funds. Passively managed funds cannot be expected to generate excess returns, but on average they often meet their benchmark tracking target. Since actively managed funds often incur higher costs, they must constantly beat the market, which carries an implicit risk. In particular, it was shown that many of the active funds with positive performance only managed to do so within a certain period of time before underperforming again. From these results, one could conclude that passively managed funds carry less risk for investors in the long run. Especially if external factors such as tax relief also apply. As passive management has steadily gaine popularity in recent years, it is also more future-oriented to develop and improve these strategies.

Chapter 6

Challenges of Passive Portfolio Management

In this chapter, two common challenges of passive portfolio management are analyzed to create simple use cases for testing the PSO. The first challenge is the mean-variance portfolio (MVP) from Markowitz's modern portfolio theory, which, simply put, is an optimal allocation of assets in terms of risk and return. The second challenge is the index tracking problem, which attempts to construct a portfolio with minimal tracking error to a given benchmark.

6.1 Mean-Variance Portfolio (MVP)

Markowitz showed that diversifying risk across multiple assets reduces overall portfolio risk. This result was the beginning of the widely used modern portfolio theory, which uses mathematical models to create portfolios with minimal variance for a given return target. All such optimal portfolios for a given return target are called efficient and constitute the efficient frontier. The problem behind Markowitz's original MVP without constraints can be solved in a closed form, which is explained in [Zivot, 2021]. This type of MVP has no practical use, so only MVP problems with constraints and without closed forms are of interest in this thesis.

6.1.1 MVP Problem

Let there be N assets and their returns on T different days, creating a return matrix $R \in \mathbb{R}^{T \times N}$. Each element $R_{t,i}$ contains the return of the i-th asset on day t. The estimated covariance matrix of the returns is $\sum \in \mathbb{R}^{N \times N}$ and the estimation of the expected returns are $\mu \in \mathbb{R}^N$. The MVP problem with the risk aversion parameter $\lambda \in [0, 1]$, as shown in [Maringer, 2005], can be formalized as follows:

$$\min_{w} \quad \lambda \, w^T \sum w - (1 - \lambda) \, \mu^T w. \tag{6.1}$$

The risk aversion parameter λ defines the tradeoff between risk and return. With $\lambda = 1$, the minimization problem contains only the variance term, leading

to a minimum variance portfolio, and $\lambda = 0$ transforms the problem into a minimization of negative expected returns, leading to a maximum return portfolio. All possible portfolios created by $\lambda \in [0, 1]$ define the efficient frontier.

6.1.2 Example MVP

All possible MVP's together define the efficient frontier, which is analyzed in this section without going into the details of its calculation. This example uses three assets (stocks: IBM, Google, Apple) and calculates the MVP for each λ. First, the daily returns of these three assets from 2018-01-01 to 2019-12-31 are loaded.

The cumulative daily returns are:

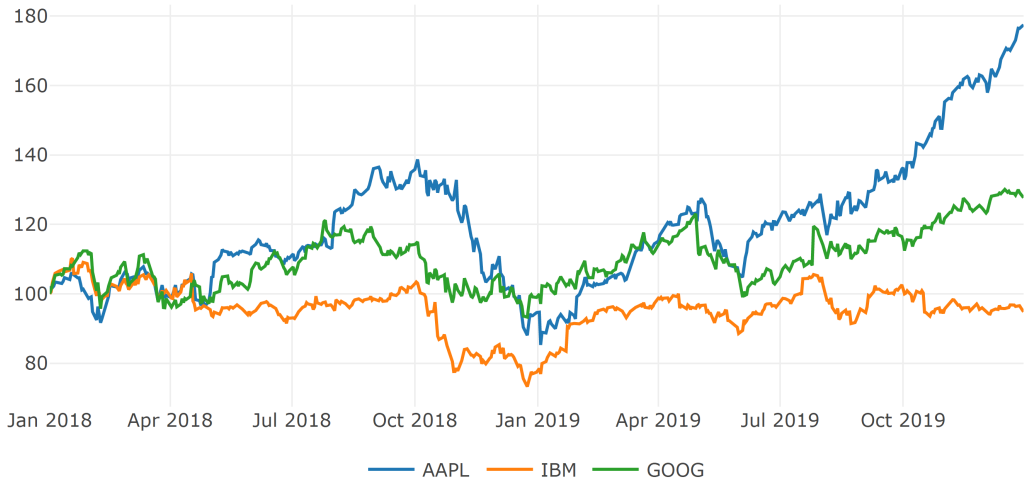

Figure 6.1: Cumulative daily returns of IBM, Apple and Google

The expected daily returns and the covariance matrix for the three assets can be estimated using the formulas from chapter 4:

```
estimation of expected daily returns:
          AAPL           IBM          GOOG
 0.0011434117 -0.0001059161  0.0004870292

estimation of positiv definite covariance matrix:
             AAPL          IBM          GOOG
AAPL 0.0003012226 0.0001177830 0.0001799099
IBM  0.0001177830 0.0002047607 0.0001158736
GOOG 0.0001799099 0.0001158736 0.0002728911
```

This is all of the data needed to solve the MVP problem with $\lambda \in \{0.01, 0.02, ..., 0.99, 1\}$. All 100 portfolios are computed by solving a quadratic problem with constraints long only ($w_i \geq 0 \; \forall \; i$) and the weights should sum to one.

The resulting portfolios are plotted in the daily μ-σ diagram to create the efficient frontier:

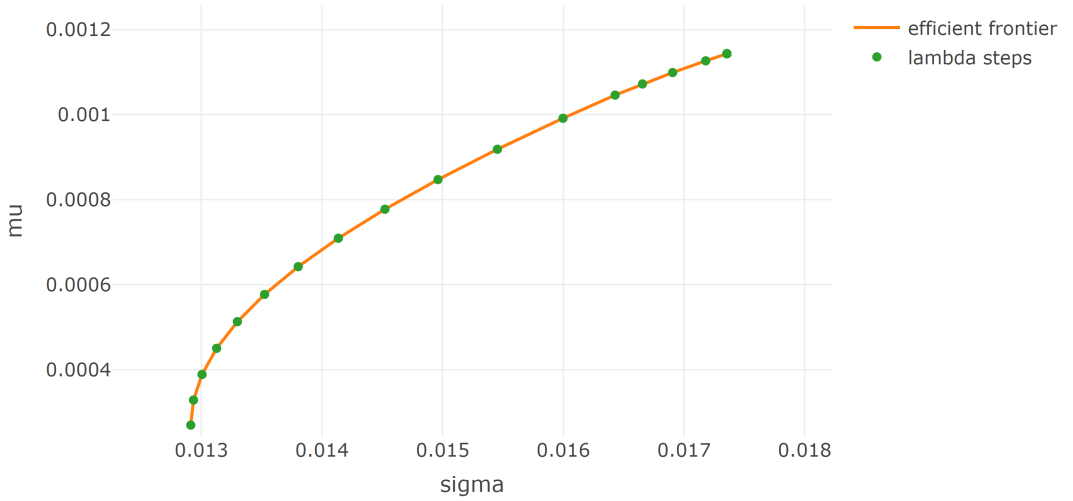

Figure 6.2: Optimal MVP for each lambda and interpolated efficient frontier

The portfolio compositions for each λ are:

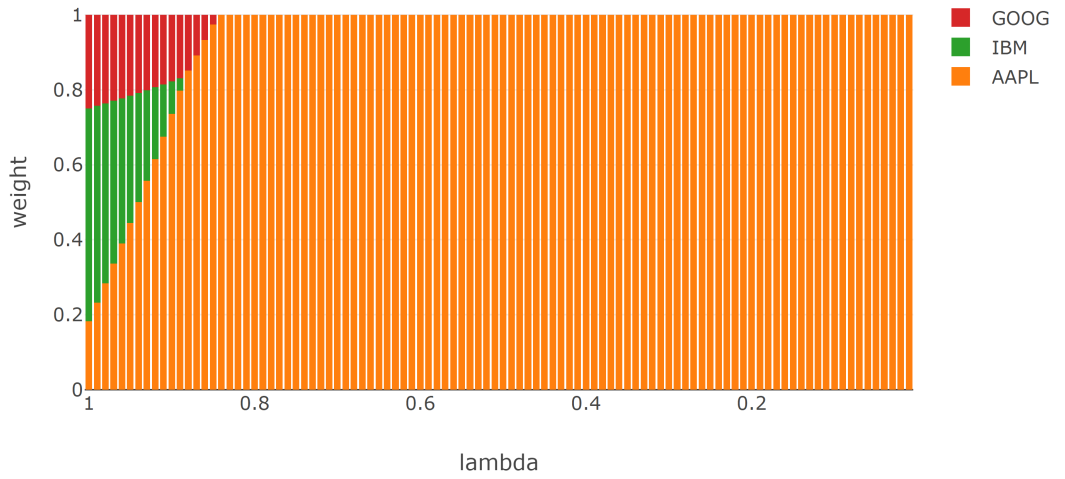

Figure 6.3: Portfolio composition of MVP's in respect of each lambda

It can be observed that the portfolio with the lowest variance was obtained with a diversified composition of the three assets. With gradually decreasing λ, the minimization problem starts to ignore the variance, which leads to a portfolio investing more in the riskiest asset with the highest return.

6.1.3 MVP Compare Estimators

The above solution for the MVP problem was performed using a geometric mean to estimate the expected returns μ and was also used in the estimation of the covariance $\sum$. This raises the question of whether the result is different from the classical approach of estimating these parameters using the arithmetic mean. The following plot illustrates the efficient frontier created from the MVP's as a function of λ using the arithmetic mean versus the geometric mean to estimate μ. The weights of the MVP's using the geometric mean are used to calculate μ and σ as a function of the arithmetic mean to make both efficient frontiers comparable:

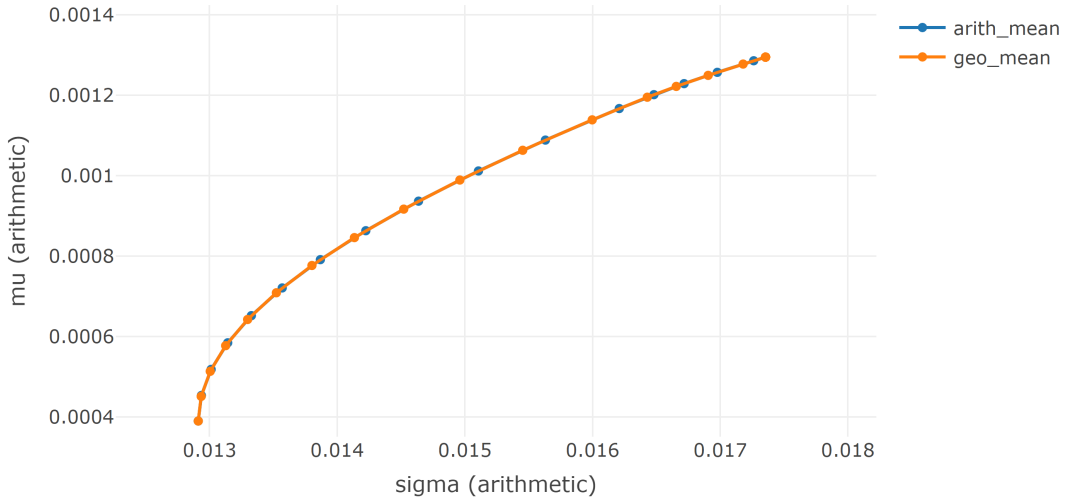

Figure 6.4: Comparison of the use of the arithmetic and geometric means in the calculation of MVP's for each lambda. The plot of MVP's in terms of mu - sigma is calculated with the arithmetic mean using the resulting portfolio weights

It can be seen, that both estimators produce portfolios on the same efficient frontier. If the aim of the MVP is to generate a portfolio with minimal variance for a given return target, the type of return needs to be specified to use the geometric or arithmetic mean. This specification will determine the type of estimator needed for the MVP. This analogy is not needed in the scope of this thesis, because the more generic portfolios specified with λ are sufficient to create test-cases for the PSO. The later examples with the MVP will always use the geometric mean of returns as estimation for the expected returns.

6.2 Index-Tracking Portfolio (ITP)

Indices are baskets of assets that are used to track the performance of a particular asset group. For example, the well-known Standard and Poor's 500 Index (abbreviated S&P 500) tracks the 500 largest companies in the United States. Indices are not for sale and serve only to visualize the performance of a particular asset group, without incurring transaction costs. Such indices, or a combination of indices, are used by portfolio managers as benchmarks to compare the performance of their funds. Each fund has its own benchmark, which contains roughly the same assets that the manager might buy. If the fund underperforms its benchmark, it may indicate that the fund manager has made poor decisions. Therefore, fund managers strive to outperform their benchmarks through carefully selected investments. Past experience has shown that this is rarely achieved with active management by cost [Pace et al., 2016a]. This has led to the growing popularity of passively managed funds whose goal is to track their benchmarks as closely as possible. This can be achieved through either full or sparse replication. Full replication is a portfolio that contains all the assets in the benchmark with the same weightings. The resulting performance equals the performance of the benchmark when transaction costs are neglected. The first problem is that a benchmark may contain assets that are not liquid or cannot be purchased. The second problem is the weighting scheme of the indices, because

they are often weighted by their market capitalization, which changes daily. This would result in the need to rebalance daily and increase transaction costs to replicate the performance of the benchmark as closely as possible. To avoid this, sparse replications are used that contain only a fraction of the benchmark's assets. To do so, the portfolio manager must define his benchmark, which serves as the investment universe for his fund. He then reduces this universe, taking into account investor constraints and availability, to create a pool of possible assets. For example, a pool that replicates the S&P 500 might consist of the one hundred highest-weighted assets in the S&P 500. The ITP can be modeled in two ways analysed in [Gavriushina et al., 2019].

6.2.1 ITP with TEV objective (ITP-TEV)

The classic and widely used model tries to reduce the tracking error variance (TEV) with the following formula:

$$\min \quad Var(\boldsymbol{TE}) = Var(\boldsymbol{R}_p - \boldsymbol{R}_{bm}),$$

where the random tracking portfolio return is $\boldsymbol{R}_p$ and the random benchmark return is $\boldsymbol{R}_{bm}$. To obtain the portfolio weights w, one needs to substitute the tracking portfolio return $\boldsymbol{R}_p$ as follows:

$$\boldsymbol{R}_p = \boldsymbol{R} \times w,$$

where $\boldsymbol{R}$ is the random return vector containing the random return of each asset. The variance is then solved until a quadratic problem is presented as a function of portfolio weights w:

$$\begin{aligned} Var(\boldsymbol{R}_p - \boldsymbol{R}_{bm}) &= Var(\boldsymbol{R} \times w - \boldsymbol{R}_{bm}) \\ &= Var(\boldsymbol{R} \times w) + Var(\boldsymbol{R}_{bm}) - 2 \cdot Cov(\boldsymbol{R} \times w, \boldsymbol{R}_{bm}). \end{aligned}$$

Now the three terms can be solved, starting with the simplest one.

$$Var(\boldsymbol{R}_{bm}) = \sigma_{bm}^2$$

The variance of the portfolio can be solved with section 4.5.4:

$$Var(\boldsymbol{R} \times w) = w^T \times Cov(\boldsymbol{R}) \times w.$$

And the last term can be solved in the same way as in [Zivot, 2021]:

$$\begin{aligned} Cov(\boldsymbol{R} \times w, \boldsymbol{R}_{bm}) &= Cov(\boldsymbol{R}_{bm}, \boldsymbol{R} \times w) \\ &= E[(\boldsymbol{R}_{bm} - \mu_{bm}) \cdot (\boldsymbol{R} \times w - \mu_{\boldsymbol{R}} \times w)] \\ &= E[(\boldsymbol{R}_{bm} - \mu_{bm}) \cdot (\boldsymbol{R} - \mu_{\boldsymbol{R}}) \times w] \\ &= E[(\boldsymbol{R}_{bm} - \mu_{bm}) \cdot (\boldsymbol{R} - \mu_{\boldsymbol{R}})] \times w \\ &= Cov(\boldsymbol{R}_{bm}, \boldsymbol{R}) \times w. \end{aligned}$$

This results in the final formulation of the ITP:

$$\begin{aligned} Var(\boldsymbol{R}_p - \boldsymbol{R}_{bm}) &= Var(\boldsymbol{R} \times w - \boldsymbol{R}_{bm}) \\ &= Var(\boldsymbol{R} \times w) - 2 \cdot Cov(\boldsymbol{R} \times w, \boldsymbol{R}_{bm}) + Var(\boldsymbol{R}_{bm}) \\ &= w^T \times Cov(\boldsymbol{R}) \times w - 2 \cdot Cov(\boldsymbol{R}_{bm}, \boldsymbol{R}) \times w + \sigma_{bm}^2. \end{aligned}$$

The above problem can be estimated using the formulas and functions created in chapter 4 and historical returns R and r_{bm}. The minimization problem of the ITP in the general structure required by many optimizers is:

$$\min_w \quad \frac{1}{2} \cdot w^T \times D \times w - d^T \times w.$$

Minimization problems can ignore constant terms and global stretch coefficients and still find the same minimum. This leads to a general substitution of the ITP with TEV objective as follows:

$$D = Cov(R)$$

and

$$d = Cov(r_{bm}, R)^T.$$

It is possible to add some basic constraints, such as in the MVP problem where the sum of the weights should be one and all assets are long only. Despite the fact that this model is often used, it has a big disadvantage in that it cannot detect constant deviations in the returns. For this reason, the following model exists, which focuses on the mean square tracking error of returns.

6.2.2 ITP with MSTE objective (ITP-MSTE)

A good explanation of the ITP with MSTE objective can be found in [Badary, 2017]. The objective is to minimize the mean square tracking error (MSTE) of daily portfolio returns $r_{t,p}$ and daily benchmark returns $r_{t,bm}$ on T historical days:

$$\frac{1}{T} \sum_{t=1}^{T} (r_{t,p} - r_{t,bm})^2.$$

The formula can be rewritten as vector norm:

$$\frac{1}{T} \left\| r_p - r_{bm} \right\|_2^2.$$

This results in the following minimization with neglected stretching coefficient:

$$min \quad \left\| r_p - r_{bm} \right\|_2^2.$$

The portfolio returns r_p needs to be substituted to contain the portfolio weights w like in the TEV objective above. This results in the below transformation of the problem:

$$\begin{aligned}\|r_p - r_{bm}\|_2^2 &= \|R \times w - r_{bm}\|_2^2 \\ &= (R \times w - r_{bm})^T \times (R \times w - r_{bm}) \\ &= (w^T \times R^T - r_{bm}^T) \times (R \times w - r_{bm}) \\ &= w^T \times R^T \times R \times w - w^T \times R^T \times r_{bm} - r_{bm}^T \times R \times w + r_{bm}^T \times r_{bm}.\end{aligned}$$

The minimization and the fact that the scalars $w^T \times R^T \times r_{bm}$ and $r_{bm}^T \times R \times w$ are equal, transforms the problem to:

$$\min_w \quad \|r_p - r_{bm}\|_2^2 = w^T \times R^T \times R \times w - 2 \cdot r_{bm}^T \times R \times w.$$

This leads to the equivalent general representation of the ITP with MSTE objective as follows:

$$\min_w \quad \frac{1}{2} \cdot w^T \times D \times w - d^T \times w$$

with

$$D = R^T \times R$$

and

$$d = R^T \times r_{bm}.$$

6.2.3 Example ITP

This example shows the results of tracking the S&P 500 with a tracking portfolio that can only invest in IBM, Apple and Google. Because the returns are calculated from adjusted closing prices, the index needs to represent dividends, stock splits and rights offerings too. This can be achieved by taking the S&P 500 Total Return Index (abbreviated SP500TR). The time frame ranges from 2018-01-01 till 2019-12-31 and the goal is to minimize the difference in returns between the portfolio and the benchmark. The fitted performance of the ITP-TEV and ITP-MSTE are:

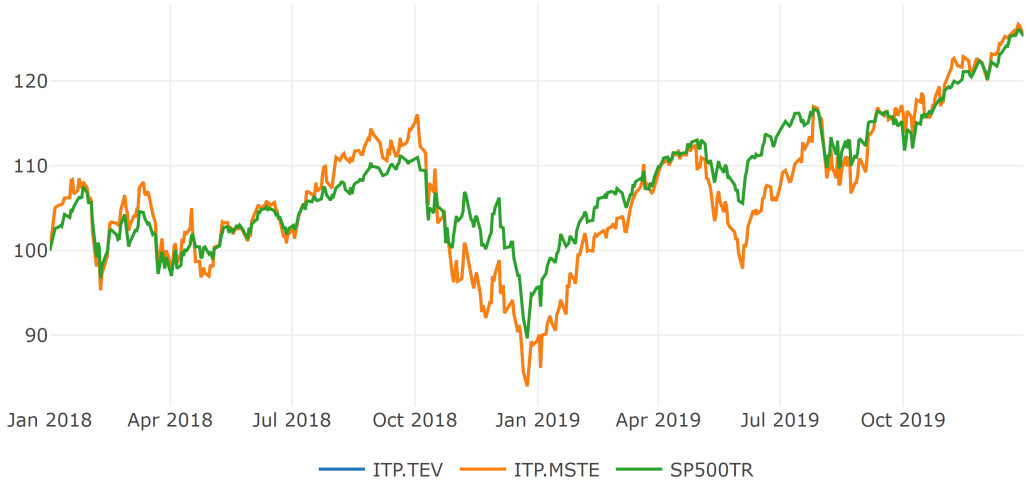

Figure 6.5: Comparison of ITP-TEV and ITP-MSTE objectives, presented as cumulative daily returns

The ITP-TEV and the ITP-MSTE produced almost the same results, which can be seen in the compositions below:

```
      type       AAPL       IBM      GOOG
1  ITP-TEV 0.2588843 0.4163274 0.3247883
2 ITP-MSTE 0.2586716 0.4165149 0.3248135
```

Chapter 7

Deterministic Optimization Methods for Quadratic Programming Problems

In this chapter, the advantages and disadvantages of deterministic optimization methods for quadratic programming problems are discussed. For simplicity, this thesis refers to the implementation of a deterministic optimization method as a solver. It is beyond the scope of this thesis to explain the underlying mathematical principles of how a solver handles quadratic problems; only the applications and analysis are discussed. The main reason for dealing with solvers for quadratic programming problems is to use them as a benchmark for the PSO.

7.1 Quadratic Programming (QP)

A quadratic program is a minimization problem of a function that returns a scalar and consists of a quadratic term and a linear term that depend on the variable of interest. In addition, the problem may be constrained by several linear inequalities that bound the solution. The general formulation used is to find x that minimizes the following problem:

$$\min_x \ \frac{1}{2} \cdot x^T \times D \times x - d^T \times x$$

and is valid under the linear constraints:

$$A^T \times x \geq b_0.$$

Some other sources notate the problem with different signs or coefficients, all of which are interchangeable with the above problem. In addition, the above problem has the same notation as in the R package `quadprog`, which reduces the substitution overhead. All modern programming languages have many solvers for quadratic problems. They differ mainly in the computation time for certain problems and the requirements. Some commercial QP solvers additionally accept more complex constraints, such as absolute (e.g., $|A^T \times x| \geq a_0$) or mixed-integer

(e.g., $x \in \mathbb{N}$). Especially the mixed-integer constraint problems lead to a huge increase in memory requirements.

7.2 QP Solver from quadprog

The common free QP solver used in R comes from the package quadprog, which consists of a single function called `solve.QP`. Its implementation routine is the dual method of Goldfarb and Idnani published in [Goldfarb and Idnani, 1982] and [Goldfarb and Idnani, 1983]. It uses the above QP with the condition that D has to be a symmetric positive definite matrix. This means that $D \in \mathbb{R}^{N \times N}$ and $x^T Dx > 0 \; \forall \; x \in \mathbb{R}^N \setminus \{\vec{0}\}$, which is equivalent to all eigenvalues being greater than zero. In most cases this is not achieved by estimating the covariance matrix $\sum$, but it is possible to find the nearest positive definite matrix of $\sum$ using the function `nearPD()` from the matrix R package. The error encountered often does not exceed a percent change in elements greater than $10^{-15}\%$, which is negligible for the context of this work. The function `solve.QP` for an N dimensional vector of interest, has the following arguments, which are also found in the above formulation of a QP:

- `Dmat`: Symmetric positive definite matrix $D \in \mathbb{R}^{N \times N}$ of the quadratic term
- `dvec`: Vector $d \in \mathbb{R}^N$ of the linear term
- `Amat`: Constraint matrix A
- `bvec`: Constraint vector b_0
- `meq = 1`: means that the first `meq` columns of A are treated as an equality constraint

The return of `solve.QP` is a list and contains, among others, the following attributes of interest:

- `solution`: Vector containing the solution x of the quadratic programming problem (e.g. portfolio weights)
- `value`: Scalar, the value of the quadratic function at the solution

7.3 Solving MVP Problem with `solve.QP`

This section provides insights into the effects of diversification and the use of `solve.QP` by creating ten different efficient frontiers from a pool of ten assets. Each efficient frontier $i \in \{1, 2, \cdots, 10\}$ consists of $N_i = i$ assets and is created by adding the asset with the next smallest variance first. After loading the returns for ten of the largest stocks in the U.S. market, the variance is calculated to rank all columns in ascending order of variance, as shown in the code below:

```
returns_raw <- buffer(
  get_yf(
    tickers = c("IBM", "GOOG", "AAPL", "MSFT", "AMZN",
                "NVDA", "JPM", "META", "V", "WMT"),
    from = "2018-01-01",
    to = "2019-12-31"
  )$returns,
  "AS_10_assets"
)
```

```
# re-arrange: low var first
vars <- sapply(returns_raw, var)
returns_raw <- returns_raw[, order(vars, decreasing = F)]
```

The next step is to create a function `mvp()` that has the arguments `return` and `lambda`. It computes the expected returns `mu` and the estimated positive definite covariance `cov`. It then solves an MVP problem with constraints $\sum w_i = 1$ and $w_i \geq 0$, which yields the key features `mu`, `var` and `composition` of the portfolio.

```
mvp <- function(returns, lambda){
  tc <- tryCatch({
    mu <- ret_to_geomeanret(returns)

    cov <- as.matrix(nearPD(cov_(returns, mu))$mat)

    mat <- list(
      Dmat = lambda * cov,
      dvec = (1-lambda) * mu,
      Amat = t(rbind(
        rep(1, ncol(returns)), # sum up to 1
        diag(
          1, nrow=ncol(returns), ncol=ncol(returns)
        ) # long only
      )),
      bvec = c(
        1, # sum up to 1
        rep(0, ncol(returns)) # long only
      ),
      meq = 1
    )

    qp <- solve.QP(
      Dmat = mat$Dmat, dvec = mat$dvec,
      Amat = mat$Amat, bvec = mat$bvec, meq = mat$meq
    )

    res <- list(
      "mu" = mu %*% qp$solution,
      "var" = t(qp$solution) %*% cov %*% qp$solution,
      "composition" = setNames(qp$solution, colnames(returns))
    )
    TRUE
  }, error = function(e){FALSE})


  if(tc){
    return(res)
  }else{
    return(list(
      "mu" = NA,
      "var" = NA,
      "composition" = NA
    ))
  }
}
```

Each $\lambda \in \{0.01, 0.02, \cdots, 1\}$ and each combination of ascending number of assets results in a portfolio that can be created with two for loops.

```
df <- data.frame(
  "index"=1,
  "var"=as.numeric(var(returns_raw[, 1])),
  "return" = as.numeric(ret_to_geomeanret(returns_raw[, 1])),
  row.names=NULL
)
for(i in 2:ncol(returns_raw)){
  returns <- returns_raw[, 1:i]
  for(lambda in seq(0.01, 1, 0.01)){
    res <- mvp(returns, lambda)

    df <- rbind(
      df,
      data.frame("index"=i, "var"=res$var, "return" = res$mu)
    )
  }
}
```

The result is filtered and names are added to represent the number of assets. Now the μ-σ diagram can be created:

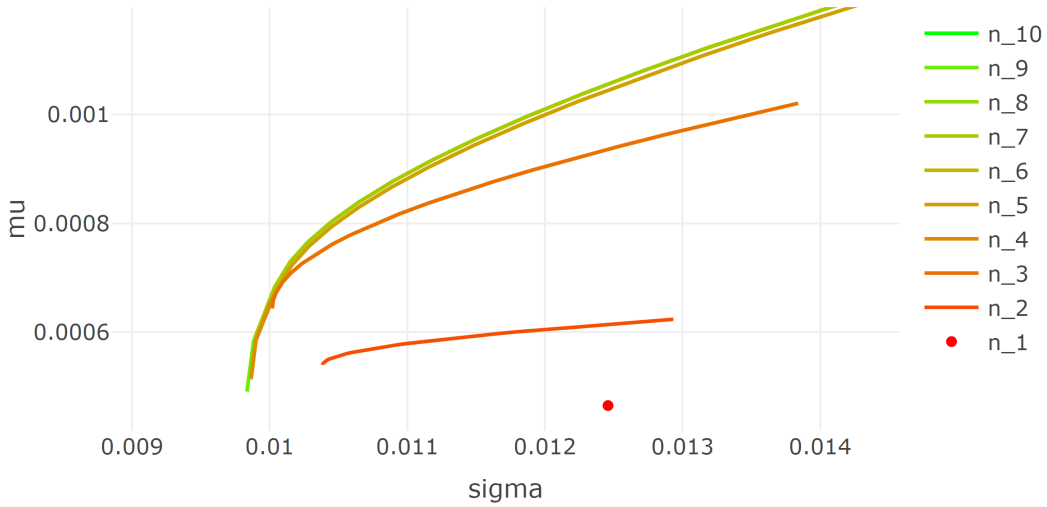

Figure 7.1: Comparison of efficient frontiers generated from an ascending pool of assets, starting with the asset that has the lowest variance and sequentially adding the asset with the next lowest variance to the pool. Displayed in a mu - sigma diagramm

It can be seen that each added asset leads to a minimum variance portfolio with a smaller or equal standard deviation. Despite starting with the asset that has the smallest standard deviation of 0.012459. This is the effect of diversification mentioned by Markowitz.

7.4 Solving ITP-MSTE with `solve.QP`

This example analyzes how many assets are needed to minimize the mean square error between the replication and historical returns of the SP500TR from 2018-01-01 to 2019-12-31. The constraints are set to be long only and the weights should sum to one. To gradually reduce the number of assets, the five assets with the lowest weights are discarded, and the remaining assets serve as the new asset pool for the next replication until only five assets remain. First, the required data can be downloaded from the `R/` directory using existing functions. The function `get_spx_composition()` uses web scraping to read the components of wikipedia and converts them into monthly compositions of the SP500TR. The pool is formed from all assets present in the last month of the time frame, reduced by assets with missing values. The code below loads the returns of all assets in the pool and the SP500TR:

```
from <- "2018-01-01"
to <- "2019-12-31"

spx_composition <- buffer(
  get_spx_composition(),
  "AS_spx_composition"
)


pool_returns_raw <- buffer(
  get_yf(
    tickers = spx_composition %>%
      filter(Date<=to) %>%
      filter(Date==max(Date)) %>%
      pull(Ticker),
    from = from,
    to = to
  )$returns,
  "AS_sp500_assets"
)
pool_returns_raw <-
  pool_returns_raw[, colSums(is.na(pool_returns_raw))==0]


bm_returns <- buffer(
  get_yf(tickers = "^SP500TR", from = from, to = to)$returns,
  "AS_sp500tr"
) %>% setNames(., "SP500TR")
```

The required data is now available and the function for the ITP-MSTE can be created. It requires `pool_returns` with variable number of columns and the single-column matrix `bm_returns`.

```
itp <- function(pool_returns, bm_returns){
  mat <- list(
    Dmat = t(pool_returns) %*% pool_returns,
    dvec = t(pool_returns) %*% bm_returns,
    Amat = t(rbind(
      rep(1, ncol(pool_returns)), # sum up to 1
      diag(1,
```

```
            nrow=ncol(pool_returns),
            ncol=ncol(pool_returns)) # long only
    )),
    bvec = c(
      1, # sum up to 1
      rep(0, ncol(pool_returns)) # long only
    ),
    meq = 1
  )

  qp <- solve.QP(
    Dmat = mat$Dmat, dvec = mat$dvec,
    Amat = mat$Amat, bvec = mat$bvec, meq = mat$meq
  )

  res <- list(
    "var" = as.numeric(
      var(pool_returns %*% qp$solution - bm_returns)),
    "solution" = setNames(qp$solution, colnames(pool_returns))
  )
}
```

The successive removal of assets can begin and the results are stored in `res`.

```
res <- NULL
asset_pool <- NULL
n_assets <- rev(seq(5, ncol(pool_returns_raw), 5))
for(i in n_assets){
  temp <- if(i==max(n_assets)){
    itp(pool_returns_raw, bm_returns)
  }else{
    asset_pool <- names(sort(temp$solution, decreasing = T)[1:i])
    itp(
      pool_returns_raw[, asset_pool],
      bm_returns
    )
  }
  res <- rbind(
    res,
    data.frame("N"=i, "var"=temp$var, "sd"=sqrt(temp$var), row.names =
    ↪  NULL)
  )

  # for later analysis
  if(length(asset_pool)==100){
    assets_pool_100 <- asset_pool
    save(assets_pool_100, file="data/assets_pool_100.rdata")
  }
  if(length(asset_pool)==50){
    assets_pool_50 <- asset_pool
    save(assets_pool_50, file="data/assets_pool_50.rdata")
  }
}
```

The following chart illustrates the increase in standard deviation to track the historical performance of the SP500TR as the number of assets N in the asset pool decreases:

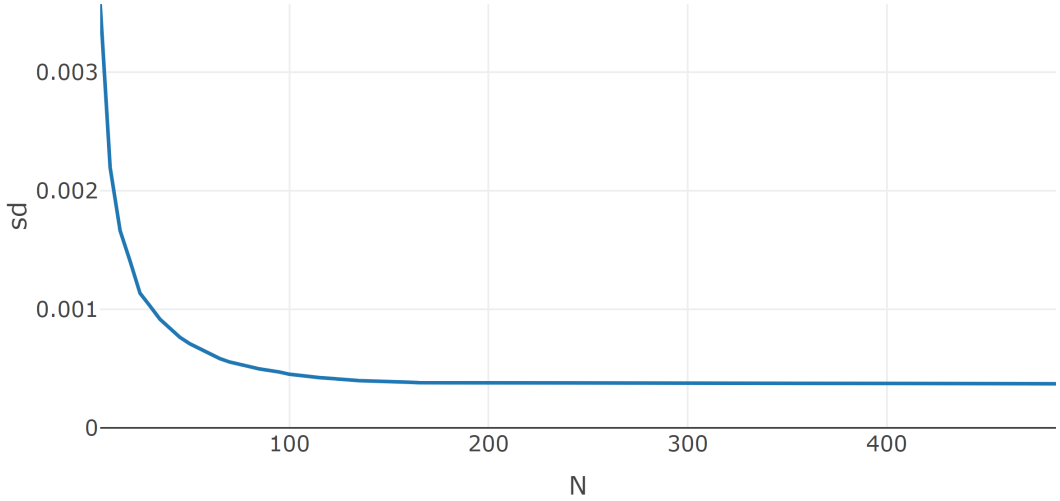

Figure 7.2: Standard deviation between the performance of the SP500TR and the tracking portfolio over the fitting period for each asset pool of size N. The asset pool was created successively by discarding the five assets with the lowest weights to create the next pool

It can be seen that the standard deviation stagnates at about $N = 200$. This leads to the conclusion that a sparse replication with two hundred assets is sufficient in this particular case to track the historical performance of the SP500TR over this period.

Chapter 8

Particle Swarm Optimization

The PSO was developed by J. Kennedy as a global optimization method based on swarm intelligence and presented to the public in 1995 by Eberhart and Kennedy [Kennedy and Eberhart, 1995]. The original PSO was intended to resemble a flock of birds flying through the sky without collisions. Therefore, its first applications were found in particle physics to analyze moving particles in high-dimensional spaces, which the name Particle recalls. Later, it was adapted in Evolutionary Computation to exploit a set of potential solutions in high dimensions and to find the optima by cooperating with other particles in the swarm [Parsopoulos and Vrahatis, 2002]. Since it does not require gradient information, it is easier to apply than other global optimization methods. It can find the optimum by considering only the result of the function to be optimized. This means that the function can be arbitrarily complex and it is still possible to reach the global optimum. Other advantages are extensibility, simplicity, and low computational costs, since only basic mathematical operators are used.

8.1 The Algorithm

Each particle d with position x_d moves in the search space $\mathbb{R}^N$ and has its own velocity v_d and remembers its previous best position $p_{p,d}$. After each iteration, the velocity changes in the direction of the intrinsic velocity, the best previous position, and the global best position p_g of all particles. A position change from i to $i+1$ can be calculated by the following two equations [Parsopoulos and Vrahatis, 2002]:

$$v_d^{i+1} = w v_d^i + c_p r_1^{i(d)} (p_{p,d}^i - x_d^i) + c_g r_2^{i(d)} (p_g^i - x_d^i)$$
$$x_d^{i+1} = x_d^i + v_d^{i+1}.$$

Where $r_1^{i(d)}$ and $r_2^{i(d)}$ are uniformly distributed random numbers in $[0, 1]$. The cognitive parameter c_p acts as a weighting of the direction to its previous best position of the particle. This contrasts with the social parameter c_g, which is a weighting of the direction to the global best position. The inertia weight w is crucial for the convergence behavior by remembering part of its previous trajectory. A study reviewed in [Parsopoulos and Vrahatis, 2002] showed that these parameters can be set to $c_p = c_g = 0.5$ and w should decrease from 1.2 to 0.

However, some problems benefit from a more precise tuning of these parameters. To allow effortless translation to code, the above formula for $d = 1, 2, \cdots, D$ particles can be given in the following matrix notation:

$$V^{i+1} = w \cdot V^i + c_p \cdot (\vec{r}_1^{\,i} \cdot (P^i - X^i)^T)^T + c_g \cdot (\vec{r}_2^{\,i} \cdot (p_g^i - X^i)^T)^T$$
$$X^{i+1} = X^i + V^{i+1}$$

with current positions $X \in \mathbb{R}^{N \times D}$, current velocities $V \in \mathbb{R}^{N \times D}$, previous best positions $P \in \mathbb{R}^{N \times D}$, and global best position $p_g \in \mathbb{R}^N$. The parameters w, c_p and c_g are stile scalars. The random numbers r_1 and r_2 are replaced by the vectors $\vec{r}_1$ and $\vec{r}_2$, in which each element is a uniformly distributed random number generated in $[0, 1]$. The first transpose is needed to multiply each random number element-wise with each column and the second transpose transforms it back to the format of V and X.

The algorithm mentioned above is the first published variant of PSO and is therefore referred to as standard PSO in this thesis. There are also other names like global PSO, because the information is distributed globally over all particles in the swarm and some also call it the basic or original PSO.

8.2 `pso()` Function

In this section, a general PSO function is created that follows the structure of other optimization heuristics in R, in particular the existing PSO implementation from the R package `pso`. The key component of the problem is a objective function called `fn()`, which needs a vector that describes the position of one particle (e.g. weights) and returns a scalar that needs to be minimized. The other main parameter for the PSO function is `par`, which is a position of a particle used to derive the dimension of the problem and used as the initial position of one particle. The vector can contain only `NA`'s, resulting in completely random starting positions. The last two arguments are `lower` and `upper` bounds (e.g. weights greater than 0 and less than 1). All other parameters have default values that can be overridden by passing a list called `control`. The resulting structure is:

```
pso <- function(
    par,
    fn,
    lower,
    upper,
    control = list()
  ){}
```

Before the main data structure can be initialized, some sample inputs must be created for the `pso()` function as described below:

```
par <- rep(NA, 2)
fn <- function(x){return(sum(abs(x)))}
lower <- -10
upper <- 10
control = list(
```

```
  s = 10, # swarm size
  c.p = 0.5, # inherit best
  c.g = 0.5, # global best
  maxiter = 100, # iterations
  w0 = 1.2, # starting inertia weight
  wN = 0, # ending inertia weight
  save_traces = F # save more information
)
```

Now it is time to initialize the random positions `X`, their fitness `X_fit` and their random velocities `V` with the created function `mrunif()` which produces a matrix of uniformly distributed random numbers between `lower` and `upper`:

```
set.seed(0)
X <- mrunif(
  nr = length(par), nc=control$s, lower=lower, upper=upper
)
if(all(!is.na(par))){
  X[, 1] <- par
}
X_fit <- apply(X, 2, fn)
V <- mrunif(
  nr = length(par), nc=control$s,
  lower=-(upper-lower), upper=(upper-lower)
)/10
```

The velocities are compressed by a factor of 10 to start with a maximum movement of one tenth of the space in each axis. The personal best positions `P` are the same as `X` and the global best position is the position with the smallest fitness:

```
P <- X
P_fit <- X_fit
p_g <- P[, which.min(P_fit)]
p_g_fit <- min(P_fit)
```

The required data structure is available and the optimization can start with the calculation of the new velocities and the transformation of the old positions. When particles have left the valid space of an axis, they are pushed back to the edge and the velocity on this axis is set to zero. Then the fitness is calculated and the personal best and global best positions are saved if they have improved.

```
trace_data <- NULL
for(i in 1:control$maxiter){
  # move particles
  V <-
    (control$w0-(control$w0-control$wN)*i/control$maxiter) * V +
      control$c.p * t(runif(ncol(X)) * t(P-X)) +
      control$c.g * t(runif(ncol(X)) * t(p_g-X))
  X <- X + V

  # set velocity to zeros if not in valid space
  V[X > upper] <- 0
  V[X < lower] <- 0

  # move into valid space
```

```
  X[X > upper] <- upper
  X[X < lower] <- lower

  # evaluate objective function
  X_fit <- apply(X, 2, fn)

  # save new previews best
  P[, P_fit > X_fit] <- X[, P_fit > X_fit]
  P_fit[P_fit > X_fit] <- X_fit[P_fit > X_fit]

  # save new global best
  if(any(P_fit < p_g_fit)){
    p_g <- P[, which.min(P_fit)]
    p_g_fit <- min(P_fit)
  }
}
```

The global minimum is located at $[0, 0]$ which has a fitness of 0 and the best position found from the PSO after 100 iterations is located at [0.000007596889, 0.000001662703] and has a fitness of 0.0000093.

8.3 Animation 2-Dimensional

This section provides insights into the behavior of the PSO by visualizing multiple iterations in a GIF. The GIF works in Adobe Acrobat DC or in the Markdown/HTML version of this thesis. The animation template and the objective function is inspired by [Roy, 2021]. The PSO core from the above chapter was used to complete the `pso()` function and is tested here with seed 0. The objective is to minimize the following function ($f : \mathbb{R}^2 \to \mathbb{R}$):

$$f(x, y) = -20 \cdot e^{-0.2 \cdot \sqrt{0.5 \cdot ((x-1)^2 + (y-1)^2)}} - e^{0.5 \cdot (cos(2 \cdot \pi \cdot x) + cos(2 \cdot \pi \cdot y))} + e + 20.$$

The following code runs the PSO and tries to minimize the objective function:

```
set.seed(0)

f <- function(pos){
  -20 * exp(-0.2 * sqrt(0.5 *((pos[1]-1)^2 + (pos[2]-1)^2))) -
  exp(0.5*(cos(2*pi*pos[1]) + cos(2*pi*pos[2]))) +
  exp(1) + 20
}

res <- pso(
  par = rep(NA, 2),
  fn = f,
  lower = -10,
  upper = 10,
  control = list(
    s = 10,
    maxiter = 30,
    w0 = 1.2,
    save_traces = T
  )
)
```

The function `f` has many local minima and a global minimum at $(1, 1)$ with the value 0. The background color scale ranges from 0 as purple to 20 as yellow. The PSO has 10 particles, iterated 30 times with an inertia weight decreasing from 0.8 to 0. The iterations are visualized in the following GIF:

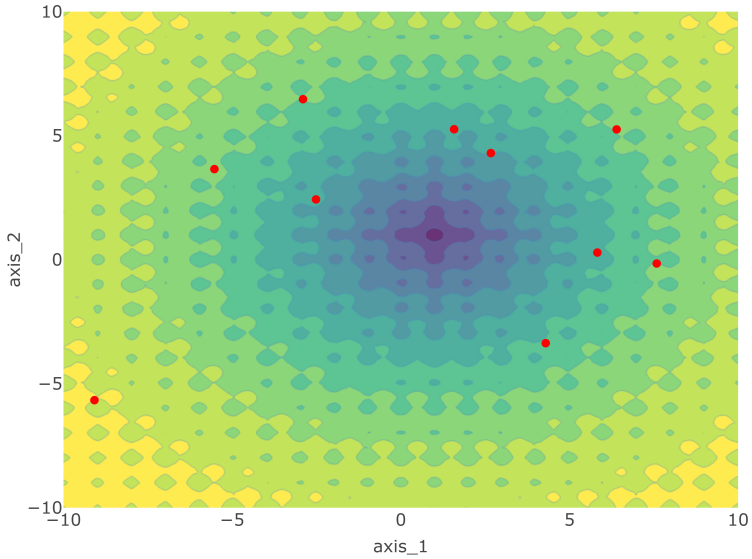

Figure 8.1: Visualization of the behavior of the standard PSO in a GIF.

8.4 Animation 2-Dimensional App

To gain an even better understanding of the behavior of a PSO, a WebApp was developed that allows the user to minimize any two-dimensional functions with constraints. Three variants of the PSO were implemented, which will be analyzed in detail in the later chapters of this thesis. The app can also be used to study the effect of hyperparameters on the behavior of the PSO in more detail. Within the app it is possible to display each step of the iterations and this even with all the direction vectors that generate the resulting motion of each particle. The app is hosted at [Roth, 2022c] and the code for it is also freely available at [Roth, 2022b]. The hosted app may be used for educational purposes and using, copying, modifying, and sharing the code is permitted without restrictions.

8.5 Simple Constraint Handling

The simplest method for dealing with constraints is the penalty method, which takes into account the intensity of constraint breaks by increasing the objective value of a minimization problem. The two common problems studied in the last two chapters are quadratic problems with the same structure of constraints. This can be used to create a generic constraint handling function for these particular QP's. Both problems has to satisfy the following equation:

$$A^T \times x \geq b_0$$

To calculate a value for the intensity of constraint breaks, the above inequality is subtracted by b_0 and the left-hand side is defined as follows:

$$c := A^T \times x - b_0.$$

All negative elements in the vector c represent constraint breaks that are squared and summed to extract a value that describes the intensity of constraint breaks like follows:

$$c_{break} = \sum p(c_i)^2$$

with

$$p(x) = \begin{cases} 0 & \text{if } x \geq 0 \\ x & \text{if } x < 0. \end{cases}$$

By following the name conventions of `solve.QP`, a list named `mat` is created in the parent environment, that contains the necessary inputs. The generic R function to calculate the constraint breaks can be defined as follows:

```
calc_const <- function(x){
  const <- t(mat$Amat) %*% x - mat$bvec
  sum(pmin(0, const)^2)
}
```

In contrast to the `solve.QP`, it's difficult for the PSO to find a feasible point, if equality constraints are used, which is why the equality constraint $\sum w_i = 1$ is reduced to $0.99 \leq \sum w_i$ and $\sum w_i \leq 1$.

The new objective function `fn()` consists of two parts. The first part is to evaluate the unconstrained objective of the QP with the following function:

```
calc_fit <- function(x){
  0.5 * t(x) %*% mat$Dmat %*% x - t(mat$dvec) %*% x
}
```

The second part is the function `calc_const()`. Since breaking constraints is much worse than losing fitness, it must have a higher intensity (e.g. 10) which must be fine-tuned. This results in the final `fn()` function composition:

```
fn <- function(x){
  fitness <- calc_fit(x)
  constraints <- calc_const(x)
  return(fitness + 10 * constraints)
}
```

This approach to handle constraints is called the penalization method and is definitely the most straightforward approach. Its disadvantage is the fact that the PSO has to find a balance between the violation of constraints and the goal. As explained in [Innocente and Sienz, 2021], there are three other constraint handling methods, but the results show that none of them is superior. The treatment of constraints should be chosen appropriately for the given problem. For example, it may be useful to use the feasibility preservation technique to obtain a solution that is guaranteed not to break any constraints. The disadvantages here are

longer computation time and less exploration of particles, since only feasible solutions can be stored as personal or global best solutions.

8.6 Convergence Analysis of the Standard PSO

This section summarizes the convergence analysis of the standard PSO with constant inertia weight w performed in [Van den Bergh and Engelbrecht, 2010] and in [Van Den Bergh et al., 2007], whose main goal is to prove whether the PSO converges to a local or even a global minimum. The results from the sources are presented in a simplified form so that the full scope can be understood. No mathematical foundations are proven, as this would otherwise be beyond the scope of this thesis. It is recommended to read the detailed proofs in the referenced sources to gain a deeper understanding. The structure of this section begins by gathering conditions that have to be met for a general stochastic search algorithm to be classified as a local or global search algorithm. Then, these conditions are used to classify the standard PSO with constant inertia weight w. It follows that certain conditions have to be imposed on the hyperparameters and the PSO has to be modified to be classified as a global or local search algorithm. Finally, these findings are applied to the practical implementation of PSO to derive relevant conclusions.

8.6.1 Convergence of a General Stochastic Search Algorithm

We start by defining a general problem where a measurable function $f : \mathbb{R}^n \to \mathbb{R}$ has to be minimized on the measurable search space $S \subseteq \mathbb{R}^n$. To do so, we use a procedure that starts with a position generated from the search space S and in each iteration compares its old position with a new one and keeps one of the two. More precisely, we assume that in each iteration of the procedure a random vector ξ is selected according to a probability measure μ_i (for iteration i) and then compared to the current position. This comparison of positions and the return of one of them is denoted by the mapping D. For the procedure to be classified as an algorithm, it is sufficient to show that the following *algorithm condition* is satisfied:

Algorithm condition: The mapping $D : S \times \mathbb{R}^n \to S$ should satisfy $f(D(z, \xi)) \leq f(z)$ and if $\xi \in S$, then $f(D(z, \xi)) \leq f(\xi)$.

Thus, the procedure is an algorithm if after each iteration the position is returned that has the lower fitness value and is also in the search space S.

Next, we need to adapt the definition of the global minimum to a set based perspective. Since the probability of generating an exact point in a search space S with a stochastic algorithm is zero, we define an optimality region R_ϵ that has a non-zero volume:

$$R_\epsilon = \{z \in S | f(z) < \psi + \epsilon\}$$

where ψ denotes the essential infimum of f on S, with $\epsilon > 0$. Here the essential infimum of f on S is defined as

$$\psi := \text{ess inf}(f) := \inf\{b \in R \mid L(\{x \in S \mid f(x) < b\}) > 0\}$$

and L denotes the Lebesgue measure.

We are now interested in the conditions under which the algorithm is guaranteed to converge to the optimality region. For this purpose, we first consider the simpler case of global convergence, for which the following convergence condition has to be satisfied:

Convergence condition (for global search): Sufficient condition for convergence to a global minimum: For any (Borel) subset A of S with a positive Lebesgue measure $L(A) > 0$,

$$\prod_{i=0}^{\infty}(1 - \mu_i(A)) = 0,$$

where $\mu_i(A)$ is the probability of the algorithm, in iteration i, to generate a new point in A.

This means that it is sufficient to show that the generation of new positions in the algorithm have a probability of zero to miss any subset of S with a positive volume infinitely often. For example, an algorithm that generates new positions uniformly distributed on S would satisfy this condition.

In summary, an algorithm that satisfies the *algorithm condition* and the *convergence condition (for global search)* is guaranteed to converge to the optimality region, which is stated in the following theorem:

Theorem 8.1 (Global search algorithm)**.** *Assume that f is a measurable function, S is a measurable subset of $\mathbb{R}^n$ and both the algorithm condition and the convergence condition (for global search) are satisfied. If $\{x_i\}_{i=0}^{\infty}$ is a sequence generated by the algorithm, D, then*

$$\lim_{i\to\infty} P(x_i \in R_\epsilon) = 1$$

Now we address the local convergence of the algorithm, which in addition to the *algorithm condition* has to satisfy the following condition:

Convergence condition (for local search): Sufficient condition for convergence to at least a local minimum: The function f is unimodal on a compact set S and for any $x_i \in S$ there exists a $\gamma > 0$ and an $0 < \eta \leq 1$ such that

$$\mu_i(\text{dist}(x_{i+1}, R_\epsilon) \leq \text{dist}(x_i, R_\epsilon) - \gamma \;\text{ or }\; x_i \in R_\epsilon) \geq \eta$$

for all probability measures μ_i and a distance measure dist() where $x_{i+1} = D(x_i, \xi)$.

This means that the algorithm has to be able to get closer to the optimality region by at least γ in each iteration with a non-zero probability, and the fitness function f on a compact set S has to be unimodal, which means that it has exactly one minimum, which is also the global minimum. Later, we will explain why this condition, extended to a larger set of functions, leads to convergence to a local minimum for the PSO.

Thus, for a stochastic search algorithm that satisfies the *algorithm condition* and the *convergence condition (for local search)*, the following theorem applies:

Theorem 8.2 (Local search algorithm)**.** *Assume that f is a measurable function, S is a measurable subset of $\mathbb{R}^n$ and both the algorithm condition and the convergence condition (for local search) are satisfied. Let $\{x_i\}_{i=0}^{\infty}$ be a sequence generated by the algorithm, D. Then,*

$$\lim_{i\to\infty} P(x_i \in R_\epsilon) = 1$$

It is difficult to prove that an algorithm satisfies the *convergence condition (for local search)*, so one may ask why this is important at all. The biggest advantage of local search algorithms is, that they usually converge much faster than global search algorithms. Even if it is in local minima, restarting can often increase the chance of finding the global minimum at some point. In case it is sufficient to find a good local minimum, a local search algorithm is preferable in most cases.

8.6.2 Convergence of the PSO

First of all it has to be ensured that the PSO satisfies the *algorithm condition*. That is, the *algorithm condition* is satisfied if the algorithm has a mapping D that compares two positions and returns the position with the lowest objective value that is also in the domain of definition. In terms of the PSO, we are concerned with the behavior when the global best position is changed. If a new personal best position ξ is found, it is guaranteed to be in the search space, since all calculated positions are pushed back to their edge if they are outside the search space. Then, the PSO compares the objective value of ξ with the objective value of the previous global best position z and stores the position with the lowest objective value as the global best position. Thus, the PSO satisfies the *algorithm condition*.

Before we can begin to analyze whether the PSO converges to a local or global minimum, conditions has to be placed on the hyperparameters to ensure that the entire swarm converges to a fixed point. In [Van den Bergh and Engelbrecht, 2010], results were referenced from [Poli, 2007], which analyzed the convergence property of a deterministic PSO, i.e., when the uniformly random numbers are replaced by their upper bounds. Subsequently, the results were verified by simulations of the stochastic standard PSO. This work was subsequently extended by [Jiang et al., 2007] to deal directly with the stochastic version of the standard PSO.

In [Jiang et al., 2007], the trajectory of each particle d is described as a sequence of random variables $\{\boldsymbol{X}_i^d\}$, with particle positions $\boldsymbol{X}_i^d$ in iteration i. This is used to determine the following three properties that govern the convergence behavior of the PSO: Property 1 is satisfied if the expected value of particle positions $\boldsymbol{X}_i^d$ converges to a fixed point for each particle, which can be summarized as $\lim_{i\to\infty} E[\boldsymbol{X}_i^d] = p^d \ \forall d$. Property 2 is satisfied if the variance of the positions for each particle tends to zero, which can be summarized as $\lim_{i\to\infty} Var[\boldsymbol{X}_i^d] = 0 \ \forall d$. Property 3 is satisfied if the entire particle swarm converges to a fixed point in

the mean square, i.e., all particle positions converge to a single point $\hat{p}$, which can be summarized as $\lim_{i\to\infty} E[(\boldsymbol{X}_i^d - \hat{p})^2] = 0 \ \forall d$.

Properties 1-3 depend on the choices of c_p, c_g and w, which can be found in [Jiang et al., 2007]. The proofs and deeper explanations of the constraints on parameters are beyond the scope of this thesis. In the following, the regions for each property are visualized and the blue region is denoted as the convergent parameter region with bounds $c_p{=}c_g{=}c > 0$ and $0 < w < 1$:

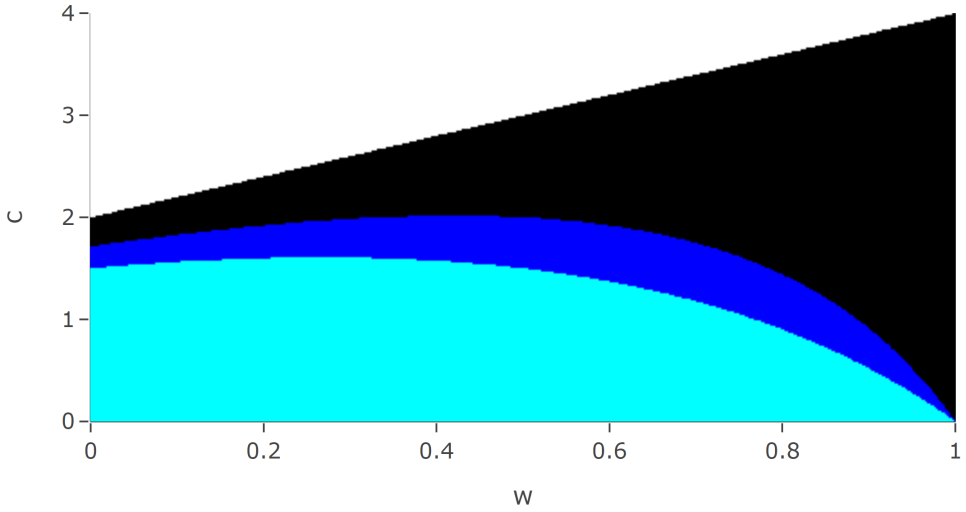

Figure 8.2: Relationship between w and c when $c_p{=}c_g{=}c$ to distinguish between different convergence behaviors. The black area shows the convergence condition for the sequence of expected positions satisfying property 1. The cyan area shows the convergence condition for the sequence of expected positions and the variances tending to zero, which satisfies properties 1 and 2. And the blue area shows the convergence condition for the sequence of expected positions with the variances tending to zero and the entire swarm converging to a single point, which satisfies properties 1, 2, and 3. This is a replication of the visualization from [Jiang et al., 2007]

In the HTML version of this thesis, the diagram can be displayed with additional information as a mouse-hover, which facilitates the precise classification of a parameter pair.

Furthermore, it is assumed that the choice of parameters w, c_p, and c_g lies within the convergent parameter region, leading the PSO algorithm to converge to a point $\hat{p}$ with all particles. The point $\hat{p}$ is not necessarily a local minimum. To see this, we examine the *convergence condition (for local search)* for the PSO. This would mean that the PSO has to be able to reduce the distance to the optimality region in each iteration with a downward bounded non-zero probability. Unfortunately, this is not the case, as can be seen in a simple example. Suppose the PSO is to minimize a 2-dimensional objective function f with only two particles and the initial positions are $x_1 = (k_1, k_2)$, $x_2 = (k_1, k_2) + a_1 \cdot (1, 0)$ and the initial velocities are $v_1 = a_2 \cdot (1, 0)$ and $v_2 = a_3 \cdot (1, 0)$ with constants k_1, k_2, a_1, a_2, and a_3. It follows that all particles and saved positions have k_2 in the second coordinate, resulting in a zero in the second coordinate of the

velocity update in all iterations. Therefore, the second coordinate with k_2 is not guaranteed to lead to the optimality region and the PSO cannot improve in this coordinate, which contradicts with the *convergence condition (for local search)* and it follows that the PSO is not a local search algorithm.

The possibility of this premature stagnation in a coordinate decreases significantly with increasing particle number, which is why it is very unlikely in practice. In order for the PSO to still be classified as a local search algorithm, a variant of the PSO was developed in [Van den Bergh and Engelbrecht, 2010], called the guaranteed convergence PSO (GCPSO). In this variant, it is guaranteed that it is at least possible to randomly shift the position of the global best particle in each coordinate by a uniformly generated value from the interval $[-\rho_i, \rho_i]$ in iteration i. Therefore, the GCPSO does not stagnate in the above example and it is possible to prove that the *convergence condition (for local search)* is satisfied. The exact implementation of the value ρ_i and how it is changed per iteration is quite technical and is not exactly reproduced here. Further, it can be imagined that ρ_0 is chosen large enough and doubles per iteration if the global best position is successfully changed and halves if the global best particle is not changed. This behavior of the GCPSO is necessary to explain why the GCPSO can converge to a local minimum for multimodal functions.

Assume that the function f is multimodal and therefore has several local minima on S. But it is possible to divide the search space S and all k minima into compact subsets $\hat{S}_k \subseteq S$ with $\hat{S}_j \cap \hat{S}_t = \emptyset \ \forall j \neq t$ on which f is unimodal and therefore has exactly one minimum. It follows that if the GCPSO with hyperparameters chosen from the convergent parameter region is applied to this problem, it will initially jump around between the subsets $\hat{S}_k$. However, due to the choice of parameters, the swarm will eventually contract and converge towards the global best particle. From this point on, the GCPSO stagnates, which is why the ρ_i becomes smaller and smaller. After a certain iteration, it is no longer possible for the GCPSO to leave the current subset $\hat{S}_{k^*}$. Subsequently, all requirements for the *convergence condition (for local search)* in the respective subset $\hat{S}_{k^*}$ hold, which is why the swarm converges with probability 1 towards the minimum in this subset. It cannot be guaranteed in which subset $\hat{S}_k$ this happens, so it can only be said that the GCPSO converges to at least a local minimum if the function f can be divided into compact subsets on which f is unimodal.

It is clear that the standard PSO and also the GCPSO do not satisfy the *convergence condition (for global search)*, since it cannot be guaranteed that every region in S is searched. However, this can be enforced by rather trivial changes to the algorithm. For example, by introducing a particle that is randomly generated every k iterations. Another approach is to restart the PSO when the majority of particles are in a small region. Other approaches and their advantages are discussed in [Van den Bergh and Engelbrecht, 2010].

In this theoretical approach, the behavior of the PSO is evaluated as the number of iterations approaches infinity. In practical applications, PSO is frequently utilized to tackle complex problems and a local minimum with an acceptable objective value is deemed sufficient as speed is often prioritized. To expedite convergence to minima, the decreasing inertia weight w is employed. To enhance

the likelihood of finding the global optimum or an acceptable local minimum, the PSO is restarted after a predetermined maximum number of iterations. This process is repeatedly carried out until a specified time or number of restarts has been reached.

8.7 Example MVP

This example uses the `solve.QP` approach from section 7.3 with ten assets as the benchmark. Briefly, the goal is to create an MVP from ten of the largest U.S. stocks between 2018-01-01 and 2019-12-31 for each possible λ. The constraints are long only and the weights should satisfy $0.99 \leq \sum w_i \leq 1$. The PSO has 300 particles and 200 iterations for each lambda. The starting position is the equally weighted vector v with $\sum v_i = 1$, the inertia weight starts at 1.2 and decreases to zero and the coefficients are $c_p = c_g = 0.5$. The main characteristics of all portfolios created with the `solve.QP` compared to the PSO are shown below:

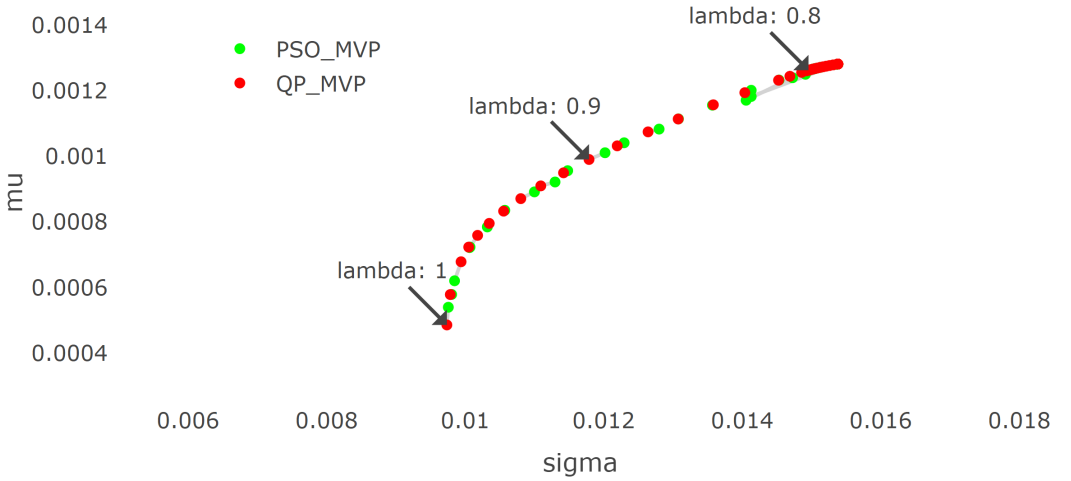

Figure 8.3: Comparison of the MVP's generated with solve.QP and the PSO. The gray lines show the difference of the solutions of both approaches for each lambda

The corresponding portfolios for each λ are connected with a grey line to visualize the error of the PSO. It turns out that it is possible to solve MVP problems with a PSO approach. It is noticable that some PSO runs were not able to reach the global minimum and thus show a deviation from the `solve.QP` approach, which can often be fixed by repeated runs.

8.8 Example ITP-MSTE

A similar ITP-MSTE solved with `solve.QP` in section 7.4 is used as a benchmark for the PSO. In summary, the goal is to create a portfolio that minimizes the mean square error of the returns of itself and the SP500TR between 2018-01-01 and 2019-12-31. The pool of assets includes all assets that are present in 2019-12-31 and have no missing values. The constraints are long only and the weights should satisfy $0.97 \leq \sum w_i \leq 1$. The parameters for the PSO are a swarm size

of 200, 300 iterations, the inertia weight starts at 1.2 and decreases to zero, the coefficients are $c_p = c_g = 0.5$, the maximum weight of each asset is 5% and the starting position is the equally weighted vector v with $\sum v_i = 1$. The PSO was run ten times, and the aggregated best and mean runs are compared to the `solve.QP` approach for seed 0 in the table below:

type	sd	fitness	constraint break	time
ITP-MSTE_QP	0,00037	-0.0223073	0,000000000	0.7
ITP-MSTE_PSO_best	0,00109	-0.0220467	0,000000000	158.7
ITP-MSTE_PSO_mean	0,00123	-0.0219651	0,000000000	158.6

Figure 8.4: Comparison of solving ITP-MSTE with solve.QP and the PSO

It can be seen that in all PSO runs, sufficient fitness was achieved with negligible constraint breaks, but much more computation time was required.

8.9 Pros and Cons for Continuous Problems

A PSO approach has advantages and disadvantages, since on the one hand any problem can theoretically be solved, but it cannot be guaranteed that the solution is also optimal. In addition, the calculations take much longer than with the `solve.QP` approach, which raises the question why a PSO approach should have any benefit at all. This is exactly the case, if the solution of the problem is no longer possible by the `solve.QP` alone, as it is for example the case with mixed-integer-quadratic-problems. In this type of problems, the condition is that the variable of interest x has to be an integer vector, which can only be solved continuously using `solve.QP` and then rounded. However, this rounding error can become arbitrarily large, which is why the chances of the PSO approach to achieve a better solution are greater than with the `solve.QP` approach.

8.10 Discrete Problems

A continuous solution for a portfolio is not sufficient for practical purposes, since usually only integer amounts of assets can be purchased. It's even worse if lot sizes are needed, because these can only be bought in minimum denomination of e.g. ten thousand. Lot sizes are often used in fixed income products. The biggest drawbacks of rounding a continuous solution are the disregarding of constraints and the difference in the objective value, which often can't reach the new optimum. A solution with broken conditions is not acceptable in practice and a `solve.QP` approach only produces one solution, which is why its insecure to hope for a sufficient solution after rounding. The PSO doesn't have these drawbacks and can be easily used for discrete problems by rounding the input of the objective function `fn()`. In a portfolio with net asset value (`nav`) consisting of only American stocks with weights w_i and closing prices p_i can be discretized to w_i^d by the following formula:

$$w_i^d = \text{round}(w_i \cdot \frac{\text{nav}}{p_i}) \cdot \frac{p_i}{\text{nav}}.$$

8.11 Example Discrete ITP-MSTE

This example analyses the error of rounding a solution with the `solve.QP` approach and compares it to a discrete PSO. A second discrete PSO is added, that takes the continuous solution of the `solve.QP` and uses it as starting position of one particle. The ITP-MSTE focuses on replicating the SP500TR with its top 100 assets derived from the example with discarding in section 7.4 and tries to construct a tracking portfolio with the constraints long only, $0.99 \leq \sum w_i \leq 1$ and nav = 10000 in the time frame from 2018-01-01 to 2019-12-31. The prices used to discretize are the closing prices on 2019-12-31 and both PSO's have 100 particles, 300 iterations, coefficients $c_p = c_g = 0.5$ and the inertia weight starts at 1.2 and decreases to zero. The results can be observed in the table below:

type	fitness	const_break	sum_wgt	time
solve.QP discrete	-0,02163	0,021139689	0,845	0,010
PSO	-0,02158	0,000000000	0,991	9,590
PSO with solve.QP as init	-0,02168	0,000000000	0,997	11,270

Figure 8.5: Comparison of solving a discrete ITP-MSTE with PSO and the solve.QP with rounding afterwards

It can be seen that the rounded `solve.QP` solution still has a good fitness but the constraints are not satisfied. The PSO has no constraint breaks and better fitness than the rounded `solve.QP`. The PSO with `solve.QP` solution as starting position has beaten both approaches. This indicates that a hybrid approach consisting of both the `solve.QP` and afterwards the PSO for intelligent rounding with observed constraints would be a good heuristic for such problems in practice.

Chapter 9

PSO Variations

The standard PSO analyzed in the previous chapter is capable of solving a wide range of problems, but often gets stuck in local minima. In this chapter, different variants of the standard PSO are analyzed using a problem from the financial domain. The first variant is the PSO with function stretching, which is designed to allow the PSO to escape from local minima if they are discovered. The second variant is the local PSO, which is designed to reduce the probability of getting stuck in local minima by limiting the spread of information in the swarm. The third variant, the PSO with feasibility preservation, tries to optimize within the feasibility space and therefore provide only feasible solutions. The last variant is the PSO with self-adaptive velocity, which tries to adjust the control parameters according to certain rules and randomness.

9.1 Testproblem Discrete ITP-MSTE

All variants are tested on a discrete ITP-MSTE to replicate the SP500TR with a tracking portfolio consisting of the top 50 assets in the S&P 500 derived from the example with discarding in section 7.4. The daily data used to solve the ITP ranges from 2018-01-01 to 2019-12-31, and the assets must be in the SP500TR at the end of the time frame and have no missing values. The tracking portfolio is discrete and has a net asset value of twenty thousand USD. The tracking portfolio is discretized using closing prices on 2019-12-31, and returns are calculated as simple returns using the adjusted closing prices. The maximum weighting for each asset is 10% to reduce the dimension of the search space. Additional constraints are long only and portfolio weights w should satisfy $0.99 \leq \sum w_i \leq 1$. All variants are run 100 times and compared to 100 runs of the standard PSO created in the previous chapter. The swarm size for the PSO and all variants is 50 and the iterations are set to 400 with coefficients $c_p = c_g = 0.5$ and the inertia weight starts at 1.2 and decreases to zero. All PSO's start with the zero vector as the initial particle position to test the ability to find the feasible space.

The next plot analyzes the behavior of the 100 standard PSO runs in each iteration by plotting the median of the best fitness achieved in each iteration. The confidence bands for the 95% and 5% quantiles of the best fitness values are plotted in the same color as the median, with less transparency:

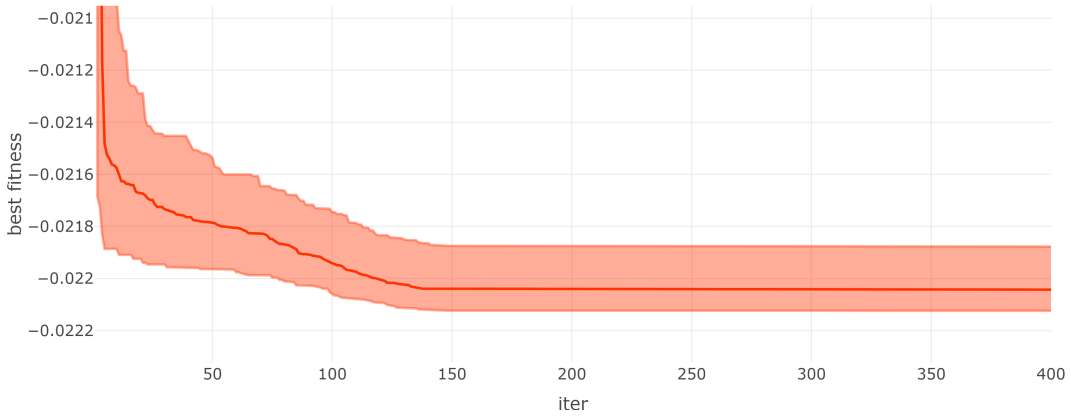

Figure 9.1: Fitness of the standard PSO for comparison with later variants

The aggregate statistics of the last iterations of all 100 runs can be found in the table below:

iter	type	time_mean	const_break_mean	best_fit_q1	best_fit_q3	best_fit_mean	best_fit_median
400	PSO	4,40	0,000000	-0,022124	-0,021877	-0,022024	-0,022043

Figure 9.2: Statistics of the standard PSO for comparison with later variants

9.2 Function Stretching

PSO often gets stuck in local minima, i.e. when the current global best position is a local minimum with a surrounding region, with only higher fitnesses. In such situations, it is difficult for the PSO to escape and find the global minimum. Function stretching attempts to allow the PSO to escape such local minima by transforming the fitness function in the same way as described in [Parsopoulos and Vrahatis, 2002]. After finding a local minimum, a two-stage transformation proposed by Vrahatis in 1996 can be used to stretch the original function such that the discovered local minimum is transformed into a maximum and any position with less fitness remains unchanged. The two stages of the transformation with a discovered local minimum $\bar{x}$ are:

$$G(x) = f(x) + \gamma_1 \cdot \|x - \bar{x}\| \cdot (\text{sign}(f(x) - f(\bar{x})) + 1) \tag{9.1}$$

and

$$H(x) = G(x) + \gamma_2 \cdot \frac{\text{sign}\Big(f(x) - f(\bar{x})\Big) + 1}{\tanh\Big(\mu \cdot (G(x) - G(\bar{x}))\Big)}. \tag{9.2}$$

The value $G(\bar{x})$ can be simplified to $f(\bar{x})$, the norm used in $G(x)$ is the Manhattan norm and the sign() function is defined as follows:

$$\text{sign}(x) = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{if } x = 0 \\ -1, & \text{if } x < 0. \end{cases}$$

In the source it is suggested to select the following parameter values as default:

$$\gamma_1 = 5000$$
$$\gamma_2 = 0.5$$
$$\mu = 10^{-10}.$$

It is difficult to interpret both transformations accurately, especially in higher dimensions, but some concepts can be recognized by looking only at the most important parts. The first transformation $G(x)$ uplifts all values greater than or equal to the local minimum and increases the uplift as a function of distance from the local minimum. The second function $H(x)$ also does not change any values below the local minimum and otherwise focuses on all values near the local minimum, stretching it to infinity and dropping steeply to repel particles.

To better understand the transformation, it is used to stretch a simple function in $\mathbb{R}^1$ defined as follows:

$$f(x) = cos(x) + \frac{1}{10} \cdot x$$

and translated to the objective function:

```
fn <- function(pos){
  cos(pos) + 1/10 * pos
}
```

The domain of definition is chosen as $x \in [-20, 20]$. Suppose the PSO gets stuck in the local minimum at $\bar{x} = \pi - \arcsin(\frac{1}{10}) \approx 3.04$. The original function `fn` and the transformed function `fn_stretched`, which matches $H(x)$ in equation (9.2), are shown in the following graph:

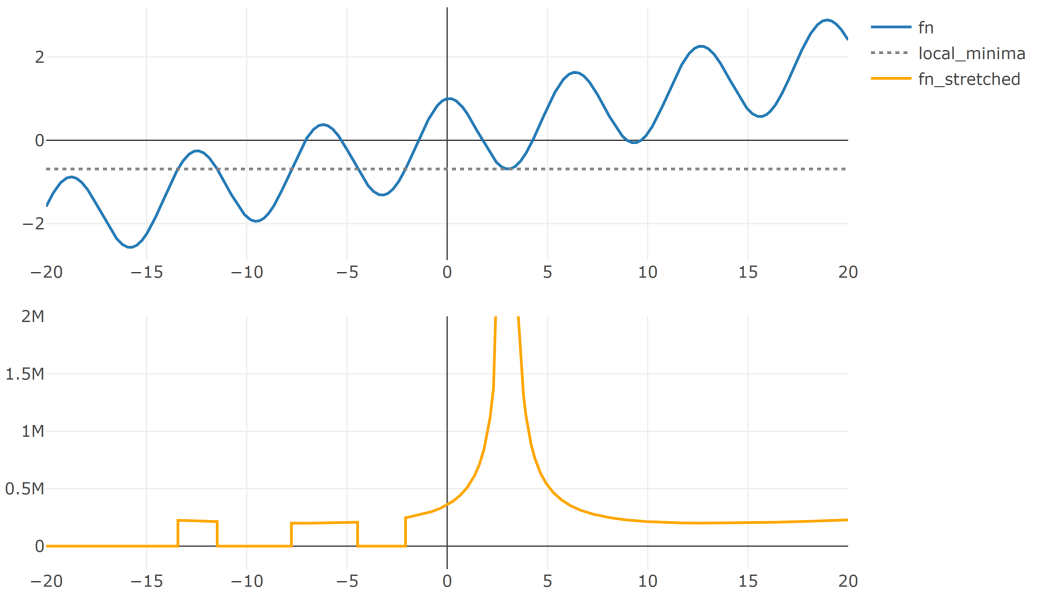

Figure 9.3: The effect of function stretching in a 1D example

It can be seen that the fitness is stretched upward around the local minimum $\bar{x}$, making it much easier for the PSO to go down the hill and fall into new minima with lower fitness. It can be observed that on the right hand side there was only

higher fitness before and after the transformation a local minimum has emerged which has a large area where only higher fitness values are located. In the worst case this can lead to a new local minimum from which it is even more difficult to escape. All regions with lower fitness remain unchanged, as can be seen in the zoomed version of the bottom diagram from above:

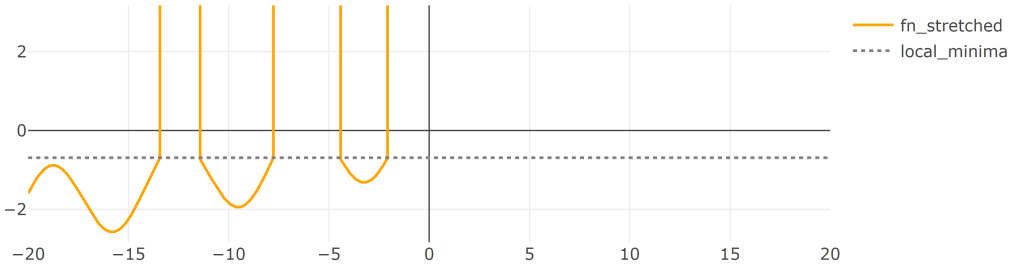

Figure 9.4: Zoomed version of the lower graphic from the figure above

9.2.1 Implementation

Since it is not possible to know if the PSO is stuck in a local minimum, a stagnation value was added that increases by one if the global best particle does not change. After ten iterations with no change, a local minimum is assumed and the transformation of the original objective function $f(x)$ takes place. After that, all personal best fitness values must be re-evaluated to work with the evaluated space and the stagnation value is set to zero. To prevent transformation just at the end of all iterations, the current iteration must be less than the maximum iteration minus twenty to allow transformation to occur. The transformed function $H(x)$ remains the objective function until the next local minimum is suspected, which again leads to a transformation of the original function $f(x)$.

9.2.2 Test PSO with Function Stretching

The PSO with function stretching is called `PSO-fnS` and is evaluated on the test problem with $\gamma_1 = 5000$, $\gamma_2 = 0.5$ and $\mu = 10^{-10}$:

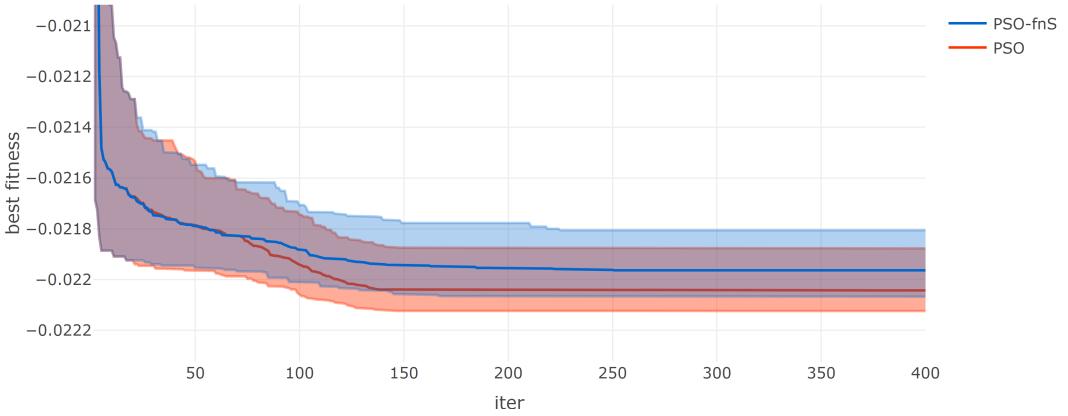

Figure 9.5: Comparison of fitness between the PSO with functional stretching and the standard PSO

The aggregate statistics of the last iterations of all 100 runs can be found in the table below:

iter	type	time_mean	const_break_mean	best_fit_q1	best_fit_q3	best_fit_mean	best_fit_median
400	PSO	4,40	0,000000	-0,022124	-0,021877	-0,022024	-0,022043
400	PSO-fnS	4,11	0,000000	-0,022067	-0,021806	-0,021952	-0,021964

Figure 9.6: Comparison of statistics between the PSO with functional stretching and the standard PSO

It turns out that function stretching does not provide any benefits in this particular problem. Perhaps it is due to the unfavorable values for γ_1, γ_2 and μ, which lead to too strong repulsion of the particles. Another reason could be that the stretching of the function generates new local minima which are even more difficult to overcome than the original ones, especially in higher dimensions.

9.3 Local PSO

The standard PSO, also referred to as global PSO is a special case of the local PSO. The main difference lies in how the best particle is selected. In the local PSO, after the particles $x_1, x_2, \cdots, x_N$ are initialized, each particle x_i is assigned a neighborhood $N(x_i, \bar{k})$. The best particle within this neighborhood is referred to as the local best particle for particle x_i, and it replaces the global best particle in the velocity update. If the neighborhood is chosen to be large enough to include all particles, it becomes equivalent to the standard PSO. A simple way to define the neighborhood with k neighbors for particle x_i is given in [Engelbrecht, 2013]:

$$N(x_i, k) = \{x_{i-\bar{k}}, x_{i-(\bar{k}-1)}, x_{i-(\bar{k}-2)}, \cdots, x_i, \cdots, x_{i+(\bar{k}-2)}, x_{i+(\bar{k}-1)}, x_{i+\bar{k}}\}$$

with

$$\bar{k} = floor(\frac{k}{2}) = \lfloor \frac{k}{2} \rfloor.$$

To illustrate this, the following figure defines the neighborhoods $N(x_3, 4)$ and $N(x_1, 4)$:

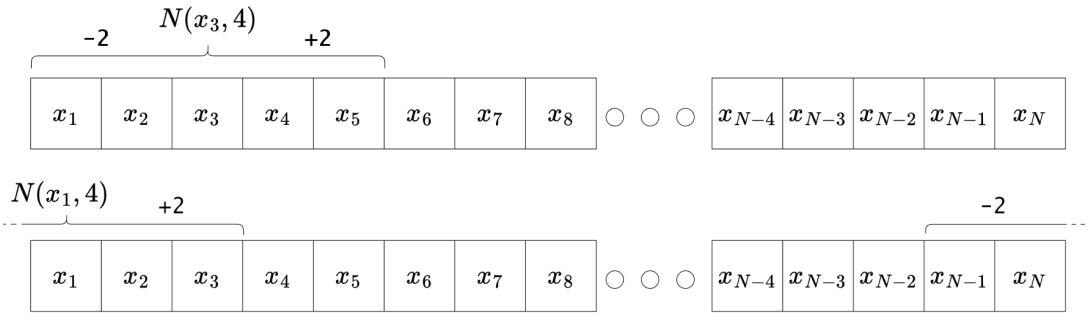

Figure 9.7: Visualization of the simple neighborhood topology.

In the latter case, it can be seen that the overflowing boundary will continue on the opposite side of the arranged particles.

9.3.1 Implementation

First, the neighbors for each particle are stored in a suitable data structure before the main part of the PSO is executed. In the local PSO, the global best particle is replaced by the local best particle, which must be determined for each particle in every iteration.

9.3.2 Test Local PSO

The PSO with particle neighborhoods is called `PSO-local` and is evaluated on the test problem with $k = 10$:

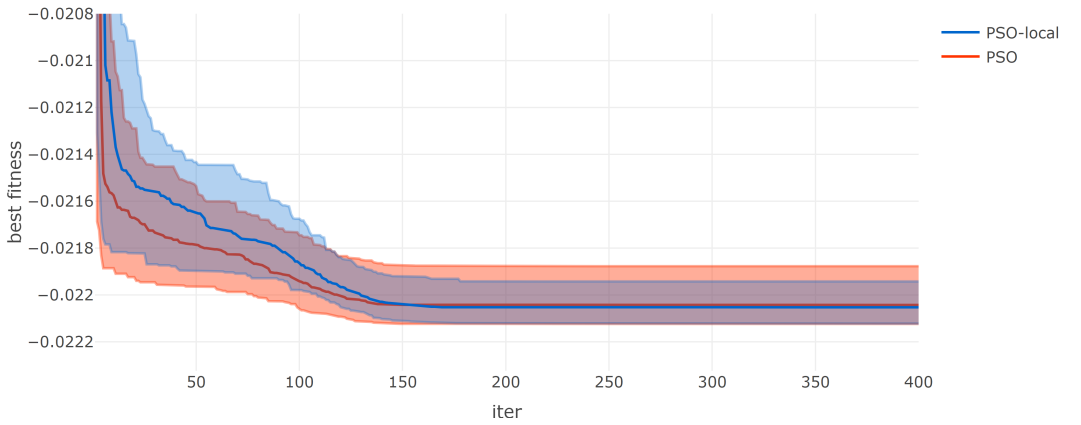

Figure 9.8: Comparison of fitness between the local PSO and the standard PSO

The aggregate statistics of the last iterations of all 100 runs can be found in the table below:

iter	type	time_mean	const_break_mean	best_fit_q1	best_fit_q3	best_fit_mean	best_fit_median
400	PSO	4,40	0,000000	-0,022124	-0,021877	-0,022024	-0,022043
400	PSO-local	3,89	0,000000	-0,022122	-0,021942	-0,022047	-0,022053

Figure 9.9: Comparison of statistics between the local PSO and the standard PSO

It can be seen that it is superior to the standard PSO in this case. Especially in preventing stagnation in local minima, which can be seen in the narrower quantile bands at the end.

9.4 Preserving Feasibility

Other variants of PSO often provide solutions that are infeasible, resulting in the need to run them multiple times. To ensure that each solution is feasible, a new variant was created in [Hu et al., 2003] that preserves the feasibility of the solutions. To be precise, this is not a variant of its own, but rather a different method for handling constraints instead of the commonly used penalty method. Nevertheless, it must change the core of the PSO implementation, which is why it is classified as its own variant in this work. The difference to the standard PSO is that the initialization of the particles is repeated until all positions are

feasible. After that, only feasible solutions are stored as global or personal best positions, resulting in a guaranteed feasible final solution. Even the first step is the most difficult to achieve in practice. To illustrate this, the result of trying to find a feasible position among a million randomly generated positions of the test problem looks like this:

```
[1] "Feasable positions: 0"

[1] "Elapsed time: 153.4 seconds"
```

It can be seen that after a million randomly generated positions, there was not a single feasible position. For this reason, the first step has been modified to start with only one feasible position, randomly selected from one of the final solutions of the standard PSO. The standard PSO was also repeated with the same starting positions to allow comparison.

9.4.1 Test Preserving Feasibility PSO

This section compares the PSO with feasibility preservation and the standard PSO. Both PSOs use the same feasible solutions as starting positions:

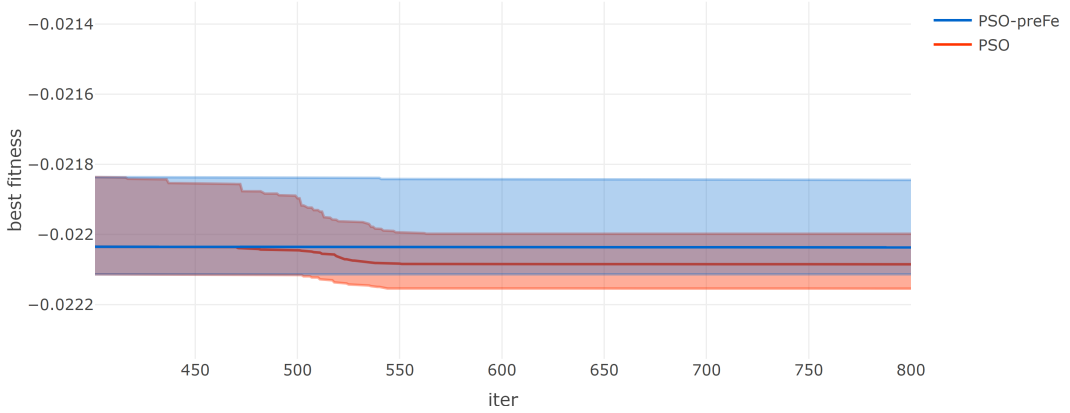

Figure 9.10: Comparison of fitness between the PSO with feasibility preservation and the standard PSO, starting from the same randomly selected initial particles obtained from the last iterations of the standard PSO solutions

The aggregate statistics of the last iterations of all 100 runs can be found in the table below:

iter	type	time_mean	const_break_mean	best_fit_q1	best_fit_q3	best_fit_mean	best_fit_median
800	PSO	3,85	0,000000	-0,022154	-0,021998	-0,022078	-0,022085
800	PSO-preFe	7,05	0,000000	-0,022113	-0,021844	-0,022018	-0,022037

Figure 9.11: Comparison of statistics between the PSO with feasibility preservation and the standard PSO, starting from the same randomly selected initial particles obtained from the last iterations of the standard PSO solutions

The results indicate that the PSO with feasibility preservation is not able to solve finance-related problems that have a very small feasible space. It is more efficient to repeat the standard PSO with the previous best position until the solution is feasible.

9.5 Self-Adaptive Velocity

A PSO approach with self-adaptive velocity that attempts to reduce hyperparameters was analyzed in [Fan and Yan, 2014]. The self-adaptive velocity is enabled by multiple velocity update schemes that are used randomly. Moreover, all hyperparameters are self-adaptive, since each particle has its own coefficients c_g, c_p, and w, which change after each iteration depending on the fitness and other factors. The resulting PSO has no real hyperparameters that need to be tuned, and thus can be used as a general-purpose PSO.

9.5.1 Implementation

The process of this PSO variant is significantly different from the standard PSO, so all changes are summarized in steps.

1) Initialize: Each particle d must initialize its own inertial weight $w_d^0 = 0.5$ and acceleration coefficients $c_{p,d}^0 = c_{g,d}^0 = 2$.

2) Velocity and positions: Update the velocity of each particle d with the following switch-case for a uniform random number $r = \text{Unif}(0, 1)$ and maximal iterations i_{max}, in iteration $i + 1$:

$$v_d^{i+1} = w_d^i \cdot v_d^i + c_{p,d}^i \cdot Z \cdot (p_{p,d}^i - x_d^i) + c_{g,d}^i \cdot Z \cdot (p_g^i - x_d^i)$$

$$Z = \begin{cases} \text{Unif}(0,1), & \text{if } r > 0.8 \\ \text{Cauchy}(\mu_1, \sigma_1), & \text{if } 0.8 \geq r > 0.4 \\ \text{Cauchy}(\mu_2, \sigma_2), & \text{if } 0.4 \geq r \end{cases}$$

with

$$\mu_1 = 0.1 \cdot (1 - (\frac{i}{i_{max}})^2) + 0.3$$
$$\sigma_1 = 0.1$$
$$\mu_2 = 0.4 \cdot (1 - (\frac{i}{i_{max}})^2) + 0.2$$
$$\sigma_2 = 0.4$$

and $\text{Cauchy}(\mu, \sigma)$ is a random number generated from the Cauchy distribution obtained with `rcauchy()` in R. The position update is the same as for the standard PSO. When a particle d has left the search space in its coordinate z, it is moved back with the following switch-case for $r = \text{Unif}(0, 1)$:

$$x_{z,d} = \begin{cases} \text{generate uniform in the search space,} & \text{if } r > 0.7 \\ \text{push back to boundary,} & \text{otherwise.} \end{cases}$$

3) Fitness evaluation: In the same way as for the standard PSO.

4) Self-adaptive control parameters: For an objective function $f()$ and the maximum fitness of all particles $f_{max} = \max(f(X^{i+1}))$, the parameters w_d^i, $c_{p,d}^i$ and $c_{g,d}^i$ are adjusted for each particle d as follows:

$$
\begin{aligned}
W_d^i &= \frac{\left|f(x_d^{i+1}) - f_{max}\right|}{\sum_d \left|f(x_d^{i+1}) - f_{max}\right|} \\
w_d^{i+1} &= \text{Cauchy}(\sum_d W_d^i \cdot w_d^i, 0.2) \\
c_{p,d}^{i+1} &= \text{Cauchy}(\sum_d W_d^i \cdot c_{p,d}^i, 0.3) \\
c_{g,d}^{i+1} &= \text{Cauchy}(\sum_d W_d^i \cdot c_{g,d}^i, 0.3).
\end{aligned}
$$

Then, the parameters are adjusted to their limits using the following formulas:

$$
w_d^{i+1} = \begin{cases} \text{Unif}(0,1), & \text{if } w_d^{i+1} > 1 \\ \text{Unif}(0,0.1), & \text{if } w_d^{i+1} < 0 \\ w_d^{i+1}, & \text{otherwise} \end{cases}
$$

$$
c_{p,d}^{i+1} = \begin{cases} \text{Unif}(0,1) \cdot 4, & \text{if } c_{p,d}^{i+1} > 4 \\ \text{Unif}(0,1), & \text{if } c_{p,d}^{i+1} < 0 \\ c_{p,d}^{i+1}, & \text{otherwise} \end{cases}
$$

$$
c_{g,d}^{i+1} = \begin{cases} \text{Unif}(0,1) \cdot 4, & \text{if } c_{g,d}^{i+1} > 4 \\ \text{Unif}(0,1), & \text{if } c_{g,d}^{i+1} < 0 \\ c_{g,d}^{i+1}, & \text{otherwise.} \end{cases}
$$

5) Update the best positions: Update the personal best positions $p_{p,d}$ and global best position p_g as in the standard PSO.

6) Repeat: Steps 2 to 5 are repeated until the maximum iteration i_{max} is reached.

9.5.2 Analyse Implementation

The random use of the distributions for the velocity update increases the diversity of the swarm. The coefficients in iteration i with 100 maximum iterations are distributed as follows:

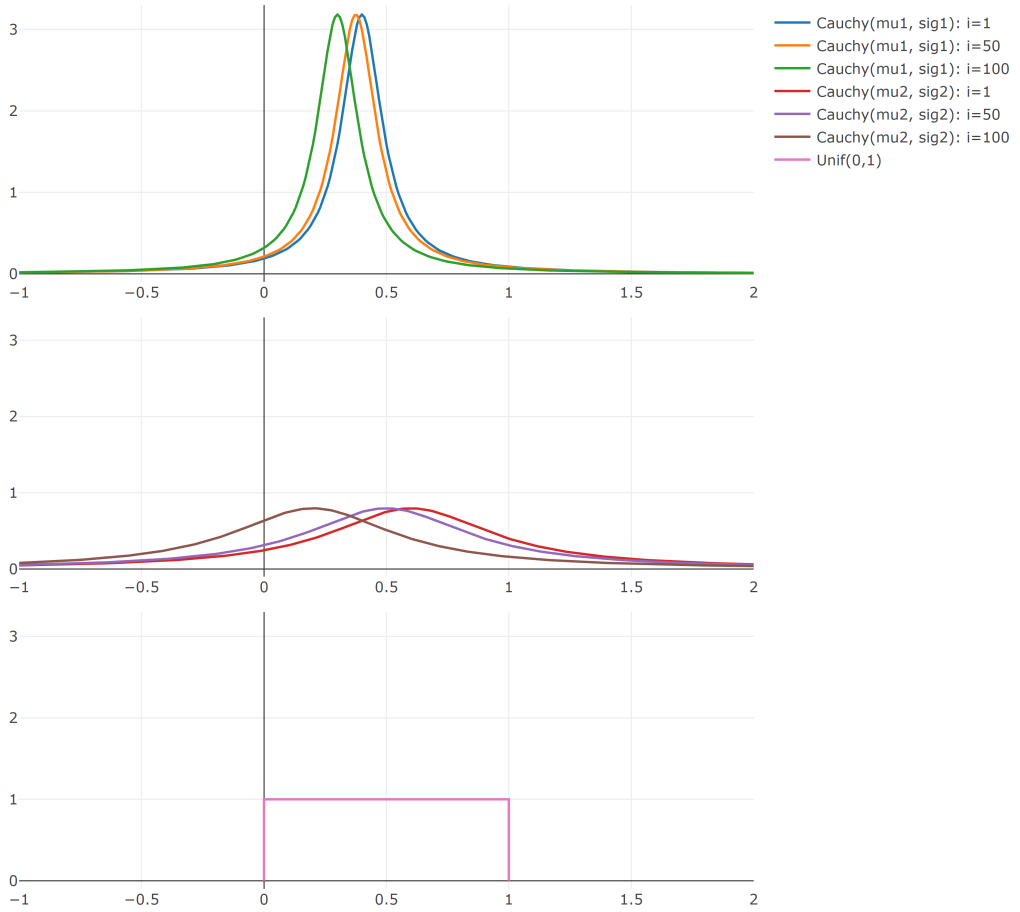

Figure 9.12: Comparison of the different distributions used in the velocity update of the PSO with self-adapting velocity depending on the iteration

It can be seen that the randomness of the motion increases compared to the uniform distribution and the center of the Cauchy distributions slowly decreases towards the absolute term. In addition, the two Cauchy distributions differ in explorability and exploitability, indicated by probabilities outside $[0, 1]$.

Even more difficult to interpret is the adjusting of the control parameters. The value W_d^i is a weighting of the distances to the worst fitness, resulting in a higher weighting of the particles with good fitness. Later, the control parameters are adjusted using the Cauchy distribution with a weighted value of the previous control parameters as the center, giving higher weights to the control parameters that produced better fitness. This results in random control parameters distributed around the best previous control parameters. The resulting behavior can be described with a small quote, "If exploration is beneficial, more exploration is done. If not, more is exploited."

9.5.3 Test PSO with Self-Adaptive Velocity

The PSO with self-adaptive velocity is called `PSO-SAvel` and is evaluated for the test problem with the constants used in the implementation section:

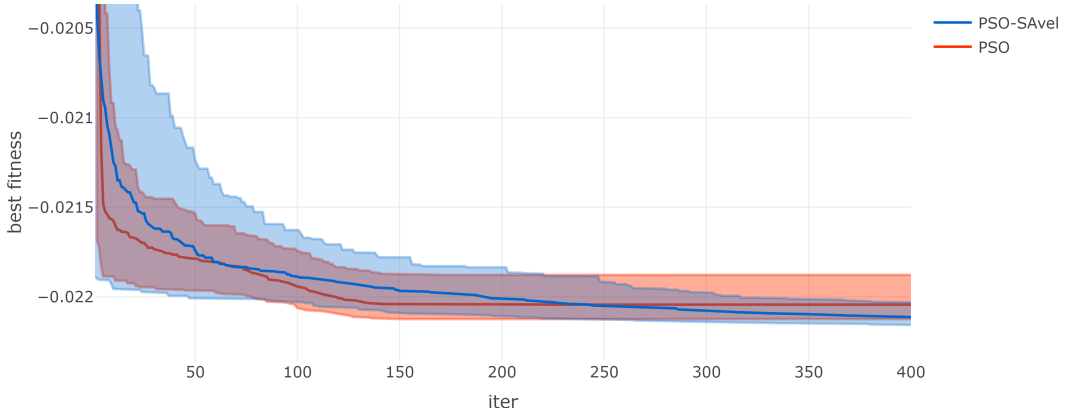

Figure 9.13: Comparison of fitness between the PSO with self-adaptive velocity and the standard PSO

The aggregate statistics of the last iterations of all 100 runs can be found in the table below:

iter	type	time_mean	const_break_mean	best_fit_q1	best_fit_q3	best_fit_mean	best_fit_median
400	PSO	4,40	0,000000	-0,022124	-0,021877	-0,022024	-0,022043
400	PSO-SAvel	4,33	0,000000	-0,022157	-0,022030	-0,022108	-0,022112

Figure 9.14: Comparison of statistics between the PSO with self-adaptive velocity and the standard PSO

The results look very promising, and with fewer hyperparameters to fine-tune, this may be one of the best variants for general use of a PSO.

9.6 PSO R Package

In this section, the existing package `pso` is used to compare the results with the standard PSO. The PSO from the existing package is called `PSO-pkg` and has been reconfigured to have the same hyperparameters as the standard PSO. It must be said that the `PSO-pkg` differs in the initialization of the velocity and does not use the compression coefficient of one tenth of the initialized velocity. The following diagram compares the fitness of the standard PSO and the `PSO-pkg` based on the test problem:

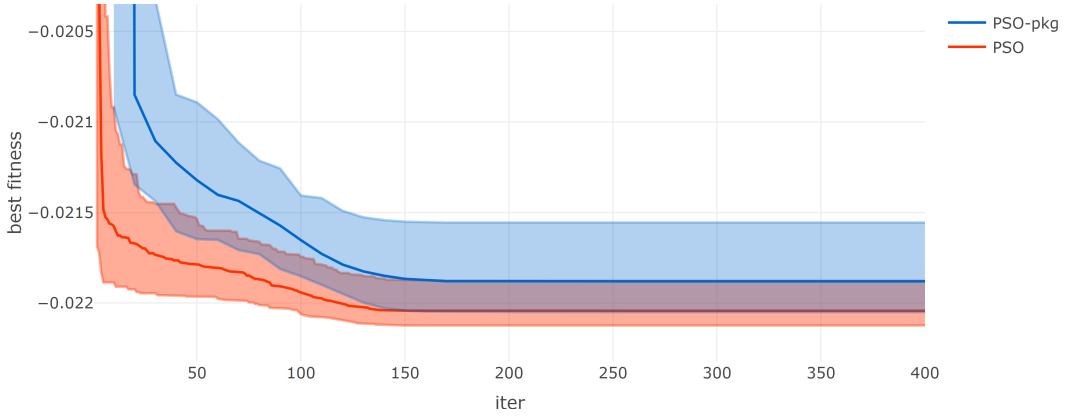

Figure 9.15: Comparison of fitness between the PSO R package with standard PSO settings and the standard PSO

The aggregate statistics of the last iterations of all 100 runs can be found in the table below:

iter	type	time_mean	const_break_mean	best_fit_q1	best_fit_q3	best_fit_mean	best_fit_median
400	PSO	4,40	0,000000	-0,022124	-0,021877	-0,022024	-0,022043
400	PSO-pkg	4,05	0,000000	-0,022050	-0,021555	-0,021863	-0,021880

Figure 9.16: Comparison of statistics between the PSO R package with standard PSO settings and the standard PSO

The `PSO-pkg` is slightly different from the standard PSO, which is most likely due to the different velocity initialization. It is important to note that the package `pso` has many features and variants that are disabled for this comparison.

9.7 Comparison with other Metaheuristics

In addition to the PSO, there are other metaheuristics that also treat the optimization problem as a black box and can thus be brought to a generic form. This is exactly what has been done in the R package `metaheuristicOpt` by implementing 19 more metaheuristics besides the PSO. Therefore, this package can be used to compare which metaheuristic is suitable for a specific type of optimization problem. The implemented metaheuristics are the standard versions, which is why no statement about variants of these metaheuristics can be given. Likewise, it is not guaranteed how performant the respective metaheuristics have been implemented. Nevertheless, this package is a great help to quickly and easily test different metaheuristics for an explicit problem. The metaheuristics implemented in the R package `metaheuristicOpt` are the following:

abbreviation	full name
ABC	Artificial Bee Colony Algorithm
ALO	Ant Lion Optimizer Algorithm
BA	Bat Algorithm
BHO	Black-Hole based Optimization Algorithm
CLONALG	Clonal Selection Algorithm
CS	Cuckoo search Algorithm
CSO	Cat Swarm Optimization Algorithm
DA	Dragonfly Algorithm
DE	Differential Evolution Algorithm
FFA	Firefly Algorithm
GA	Genetic Algorithm
GOA	Grasshopper Algorithm
GWO	Grey Wolf Optimizer Algorithm
HS	Harmony Search Algorithm
KH	Krill-Herd Algorithm
MFO	Moth Flame Optimization Algorithm
SCA	Sine Cosine Algorithm
SFL	Shuffled Frog Leaping Algorithm
WOA	Whale Optimization Algorithm

Figure 9.17: List of metaheuristics from the R package metaheuristicOpt

These metaheuristics are used to solve the test problem defined at the beginning of this chapter. The same settings are used as for the PSO variants, i.e. each run has a population size of 50, which corresponds to the number of particles in the swarm. The maximum iteration is 400 and each metaheuristic is run 100 times. Unlike the analysis above, only the last iteration of each run is recorded and compared. The data collected from each run of a metaheuristic are the fitness, the constraint breaks, and the run time. Aggregated for the respective metaheuristic, the 5% and 95% quantiles of the fitness and constraint breaks are calculated for the 100 runs, as well as the mean fitness, mean constraint breaks and mean runtime. These metrics are used to compare the respective metaheuristics in a ranking. The score calculated for this purpose is computed using the ranking in the respective category, which takes values from 1 to 24. The resulting score is a weighted sum of these individual rankings as follows:

$$\begin{aligned}\text{score} = {} & 3 \cdot \text{q5\_fitness\_rnk} + 2 \cdot \text{mean\_fitness\_rnk} + \text{q95\_fitness\_rnk} \\ & + \text{q5\_const\_break\_rnk} + 0.5 \cdot \text{mean\_const\_break\_rnk} \\ & + 0.5 \cdot \text{q95\_const\_break\_rnk} + 3 \cdot \text{mean\_runtime\_rnk}\end{aligned}$$

The score is to be interpreted in such a way that a smaller number is better than a larger one. It can be seen that the highest weight is on the 5% quantile of fitness, as this is representative of the best solutions after repeating the metaheuristic several times. Overall, this is a subjective ranking to sort the metaheuristics and compare them with a single number. In the following chart should therefore be all the necessary information to allow the reader to form his or here own opinion:

type	fitness_q5	fitness_mean	fitness_q95	const_break_q5	const_break_mean	const_break_q95	time_mean	score
PSO-SAvel	-0,022157	-0,022108	-0,022030	0,000000	0,000000	0,000000	4,33	33,0
PSO-local	-0,022122	-0,022047	-0,021942	0,000000	0,000000	0,000000	3,89	33,8
PSO	-0,022124	-0,022024	-0,021877	0,000000	0,000000	0,000000	4,40	46,8
PSO-fnS	-0,022067	-0,021952	-0,021806	0,000000	0,000000	0,000000	4,11	57,8
PSO-pkg	-0,022050	-0,021863	-0,021555	0,000000	0,000000	0,000000	4,05	72,5
CLONALG	-0,022115	-0,022076	-0,022026	0,000000	0,000000	0,000000	12,94	77,3
GA	-0,021996	-0,021860	-0,021667	0,000000	0,000000	0,000000	6,80	92,5
ALO	-0,022113	-0,022043	-0,021939	0,000000	0,000000	0,000000	42,86	94,3
MFO	-0,021904	-0,021710	-0,021467	0,000000	0,000000	0,000000	6,58	103,0
GWO	-0,022064	-0,021946	-0,021808	0,000000	0,000000	0,000000	17,95	108,5
HS	-0,021812	-0,021717	-0,021617	0,000000	0,000000	0,000000	11,90	124,5
BA	-0,021965	-0,021744	-0,021435	0,000000	0,000000	0,000000	14,67	126,8
DE	-0,021758	-0,021553	-0,021286	0,000000	0,000000	0,000000	10,37	134,0
ABC	-0,021736	-0,021638	-0,021526	0,000000	0,000000	0,000000	11,97	139,5
SCA	-0,021064	-0,020667	-0,020169	0,000000	0,000000	0,000000	10,65	145,0
BHO	0,258507	0,379743	0,500176	0,274948	0,393875	0,511849	10,65	189,0
CS	0,372524	0,519752	0,641149	0,386463	0,530286	0,650912	7,47	193,0
KH	-0,020878	0,183369	0,504408	0,000621	0,199960	0,515100	44,86	194,5
DA	-0,020186	0,404602	0,803477	0,000000	0,416484	0,807428	24,25	204,0
CSO	0,065864	0,090774	0,112772	0,085324	0,109829	0,131125	122,47	209,0
FFA	2,533888	3,554970	4,628854	2,495630	3,501443	4,559526	9,31	211,0
GOA	-0,015431	0,279220	0,707806	0,005185	0,293978	0,712788	120,33	214,5
SFL	0,391785	0,511263	0,633258	0,405184	0,523231	0,642968	89,82	232,0

Figure 9.18: Comparison of statistics between metaheuristics

To visualize the detailed runs, a boxplot of the important metrics fitness, constraint break and runtime was also created. In this boxplot, the individual points of each run are displayed under the respective boxes. If there are no points or boxes, it is because they have taken on too large values:

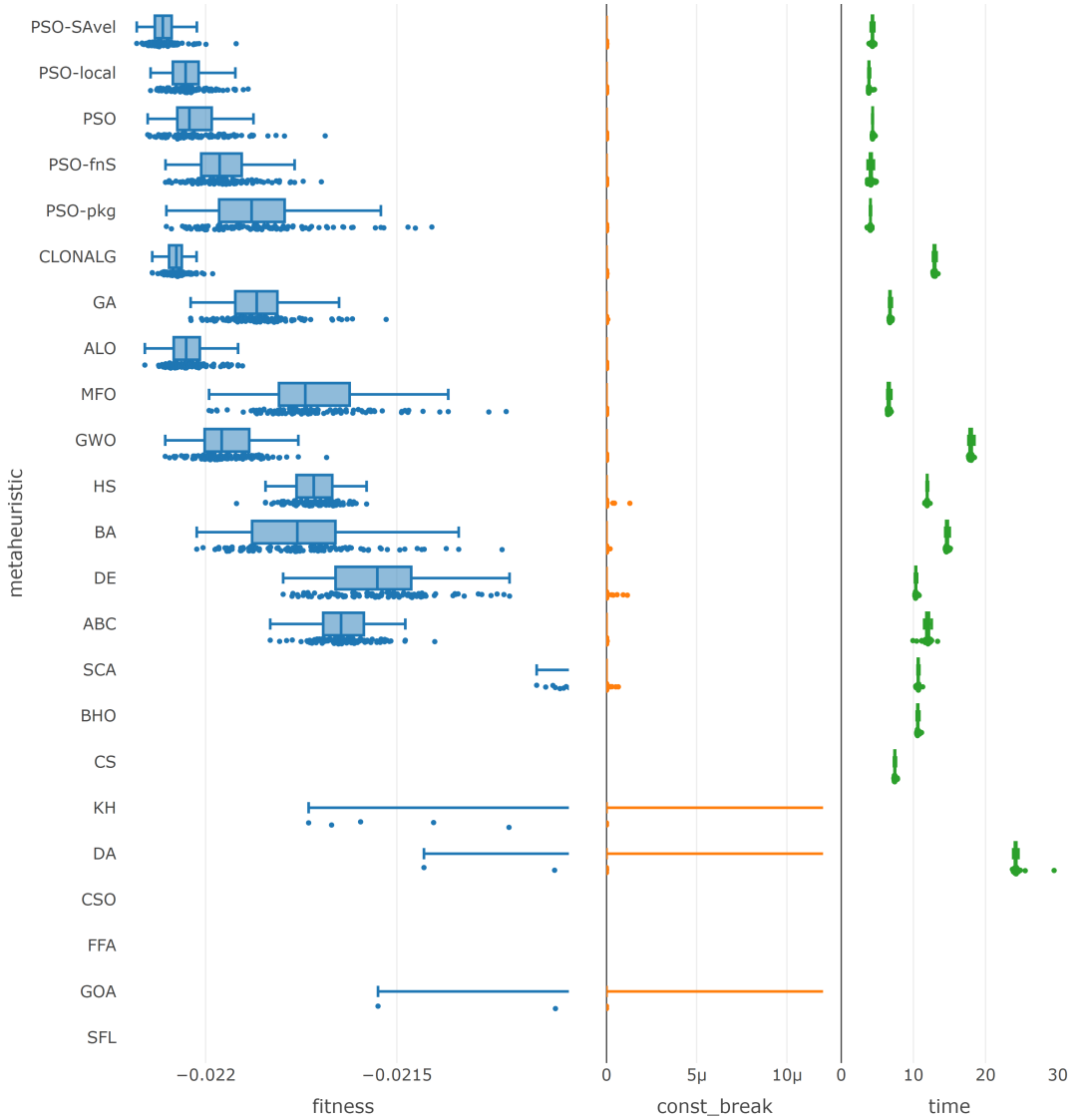

Figure 9.19: Comparison of statistics between metaheuristics with boxplots

It can be observed that the metaheuristics from the R package `metaheuristicOpt`, consistently have a longer runtime, which is partly due to the implementation, since it does not make use of vectorial operations. Additionally it can be seen that beside the PSO and its variants also the Clonal Selection Algorithm (CLONALG) and the Ant Lion Optimizer Algorithm (ALO) reached a satisfying fitness, but had a significantly longer runtime. Overall, the self-adaptive velocity PSO (PSO-SAvel) had the best ranking, which is also confirmed by the key statistics. In addition, the PSO-SAvel has fewer hyperparameters, which is why it will be used for the backtests in the next chapter.

Chapter 10

Real Life ITP Example

In the previous chapters, the capabilities of the PSO and the quality of its results were analyzed based on the solution of a problem at a single point in time. In practice, the stability of future outcomes at multiple points in time is of greater interest. Therefore, the next sections provide additional constraints needed to simulate real portfolios over multiple rebalancing dates, first by adding transaction costs to the problem. For the first rebalancing date, a problem is defined that simulates a portfolio manager who has a certain amount of cash and attempts to construct a portfolio from it, as described in the last section. After the first iteration, the portfolio manager must sell old assets and buy new ones. This, of course, incurs additional transaction costs and effort, so most portfolio managers consider a maximum rebalancing constraint that attempts to limit the amount of assets sold and purchased. The simulation of multiple rebalancing dates is called a backtest, which attempts to simulate the performance of a portfolio as a function of the previous portfolio and historical data for each rebalancing date. Later, a full backtest of an ITP-MSTE is evaluated and analyzed in a real-world environment.

10.1 Transaction Costs

The costs of buying or selling assets must be taken into account, as they can have a significant impact after several years of investment. There are many different costs that can be incurred depending on the concepts of the broker, the liquidity of the assets and the type of assets. For more information, see [Ganti, 2022] or [nyse.com, 2022]. For simplicity, we focus on the situation of a retail investor using an online broker that charges a fixed fee per transaction for the U.S. stocks included in the SP500TR. Each transaction consists of one or more shares of exactly one asset, and a transaction can be either a sale or a purchase. The fixed transaction fee is set at 1 USD, as is done by the online broker Trade Republic. The PSO can account for the transaction costs by increasing the objective value, but it is difficult to make the intensity of the transaction costs value comparable to the objective value. The objective value v^o of the ITP with MSTE approach, as in section 6.2.2, is defined as:

$$v^o = \|r_p - r_{bm}\|_2^2 = \sum_{t=1}^{T} (r_{t,p} - r_{t,bm})^2.$$

The objective value v^o is the squared tracking error or, more precisely, the squared difference of the portfolio returns r_p and the benchmark returns r_{bm}. To create a comparable value v^{tc} for transaction costs, we attempt to interpret the absolute loss due to transactions as the absolute error of return r_{tc} incurred in t_0 (before the first data point in t_1). This absolute error r_{tc} can be calculated by counting the required transactions divided by the net asset value nav. The required transactions can be calculated by comparing the shares vector of the previous portfolio s^{prev} and the shares vector of the rebalanced portfolio s^{reba}. This results in the following formula for the absolute error of return r_{tc}:

$$r_{tc} = \frac{1 \cdot \sum_{n=1}^{N} g(s_n^{prev} - s_n^{reba})}{nav}$$

with

$$g(x) = \begin{cases} 0 & \text{, if } \ x = 0 \\ 1 & \text{, otherwise,} \end{cases}$$

This results in the following transaction costs value v^{tc}:

$$v^{tc} = \|r_{tc}\|_2^2 = r_{tc}^2.$$

The idea is to use the v^{tc} value and increase the objective value v^o of the ITP with MSTE approach, but these values are still not the same. The v^o is the sum of squared positive or negative errors and v^{tc} is a squared negative error. To increase the impact of the transaction costs, a coefficient k should increase the intensity, which leads to the following minimization problem:

$$min \ \ v^o + k \cdot v^{tc}.$$

A suitable value for k could be calculated by dividing the number of training days by the number of days in the holding period increased by a factor of 2.5. This can be roughly interpreted as weighting the transaction costs error as the 2.5-day error in the test period. For example, a 4-month training period with 96 working days and a holding period of 1 month with 24 working days yields the following value:

$$k = \frac{96}{24} \cdot 2.5 = 10.$$

When the holding period is shortened, the intensity coefficient increases, which is a suitable behavior. Nevertheless, it should be analyzed and fine-tuned more.

10.2 Rebalancing Constraint

The rebalancing constraint restricts the changes in weights by comparing the previous portfolio weight vector w^{prev} and the rebalanced portfolio weight vector

w^{reba}. The value of the rebalancing should account for the weights shifted between assets and the additional weights added. Example: the previous portfolio had a weight vector $w^{prev} = [0.5, 0.4]$ at the time before rebalancing, and the rebalanced portfolio has a weight vector $w^{reba} = [0.8, 0.2]$. The shifted weight from the second asset to the first asset is 0.2 and the additional weight added is 0.1, resulting in a rebalancing of $0.2 + 0.1 = 0.3$ weight. Below is the formula for calculating the rebalancing weight w^{rb} in the general case:

$$w^{rb} := \frac{\left\| w^{prev} - w^{reba} \right\|_1 - |\sum w^{prev} - \sum w^{reba}|}{2} + |\sum w^{prev} - \sum w^{reba}|$$

and with a rebalancing constraint of, say, 30%, the rebalanced portfolio is feasible if:

$$w^{rb} \leq 0.3.$$

10.3 Objective

The goal is to simulate a tracking portfolio that tracks the SP500TR with a pool of 100 assets included in the SP500TR over multiple rebalancing dates between 2016-05-01 and 2022-10-27 with one-month intervals. The pool of assets is created for each rebalancing date using the solve.QP approach with continuously discarding of assets as in section 7.4, with a maximum of 10 assets changing on each rebalancing date to reduce forced rebalancing. All considered assets have no missing values in the training period. In addition, the solve.QP approach serves as a benchmark and the continuous solution is used as a particle position for the first rebalancing date. The tracking portfolio is solved using the self-adaptive velocity PSO from the last chapter, which yielded stable results. The tracking portfolio has the following constraints: discrete number of stocks, long only, maximum weight of 10% for each asset, $0.96 \leq \sum w_i \leq 0.995$, rebalancing under 30% weight, considering transaction costs with diffrent values for k, net asset value of 20000 USD, length of training period of four months and testing period of one month.

Each PSO run uses 100 particles and iterates 100 times. The PSO is repeated until the constraints are satisfied. Then it is run four more times to improve the quality of the feasible solution. Each rebalanced portfolio is simulated with the portfolio return function from section 4.5.5 until the next rebalancing date. In each step the weights and shares of the tracking portfolio are calculated and the shares are used to calculate the weights at the next rebalancing date. If assets are missing in the next asset pool, they are sold and reduce the net asset value due to transaction costs. The same is done by buying or selling any remaining assets. A rough illustration of the process can be found in the figure on the next page.

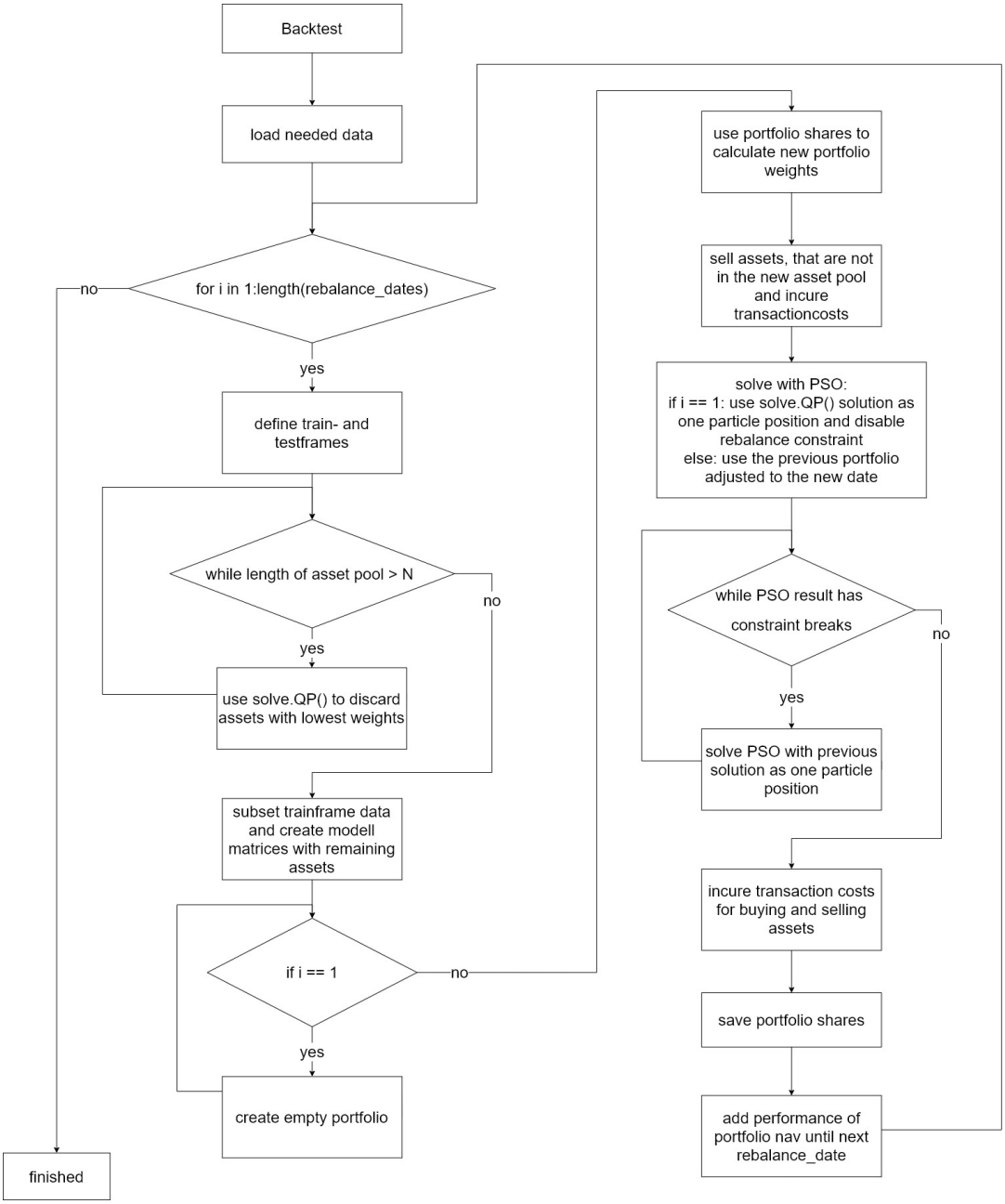

Figure 10.1: Backtest Process: visualized as a flowchart

10.4 Complete ITP Example

The following charts visualize the test period of the whole backtests. The `QP_MSTE_cont` line represents the performance of the continuous solution of the ITP-MSTE objective solved with solve.QP and stays the same for all backtests. The discretized solution using the PSO is named `PSO_MSTE_disc` and the `PSO_MSTE_disc_TR` considers all transaction costs in its performance. Everything is compared to the SP500TR which is the objective to track. Furthermore, different values for the transaction costs intensity $k \in \{0, 10, 20, 30\}$ of the individual backtests are used, indicated in the legends as `0tc`, `10tc`, `20tc` and `30tc`.

Backtest for $k = 0$:

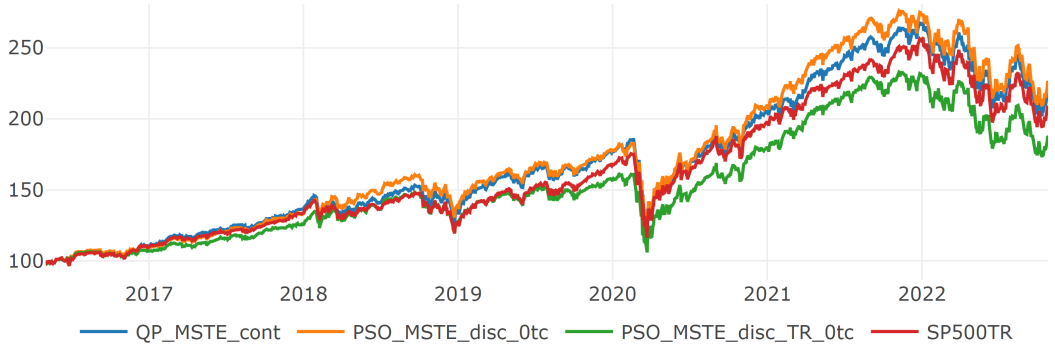

Figure 10.2: Comparison of the cumulative daily returns of the backtests with the transaction cost value k=0

and the statistics of the discretized PSO with considered transaction costs are:

calculation type	fitness	constraint break	rebalance constraint	transaction cost constraint	sum of weights	count of assets	transaction costs (in USD)
quantil 95%	0,000988	0,000	0,000	0,000	0,994	89,150	83,000
mean	0,000332	0,000	0,000	0,000	0,980	77,128	70,282
quantil 5%	0,000052	0,000	0,000	0,000	0,963	60,000	55,250

Figure 10.3: Comparison of the statistics of the backtests with the transaction cost value k=0

Backtest for $k = 10$:

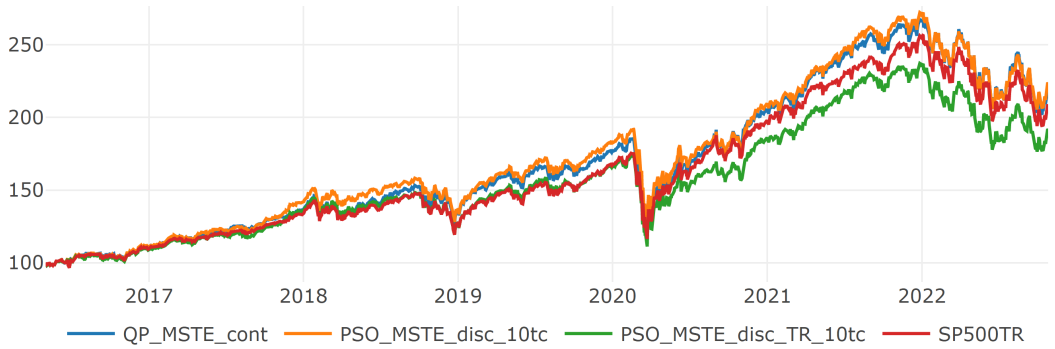

Figure 10.4: Comparison of the cumulative daily returns of the backtests with the transaction cost value k=10

and the statistics of the discretized PSO with considered transaction costs are:

calculation type	fitness	constraint break	rebalance constraint	transaction cost constraint	sum of weights	count of assets	transaction costs (in USD)
quantil 95%	0,000774	0,000	0,000	0,000	0,995	88,000	75,150
mean	0,000311	0,000	0,028	0,000	0,985	73,962	60,244
quantil 5%	0,000069	0,000	0,000	0,000	0,966	60,000	31,700

Figure 10.5: Comparison of the statistics of the backtests with the transaction cost value k=10

Backtest for $k = 20$:

Figure 10.6: Comparison of the cumulative daily returns of the backtests with the transaction cost value k=20

and the statistics of the discretized PSO with considered transaction costs are:

calculation type	fitness	constraint break	rebalance constraint	transaction cost constraint	sum of weights	count of assets	transaction costs (in USD)
quantil 95%	0,002837	0,000	0,000	0,000	0,995	78,150	69,300
mean	0,000752	0,000	0,000	0,000	0,984	65,744	48,308
quantil 5%	0,000138	0,000	0,000	0,000	0,963	48,000	5,700

Figure 10.7: Comparison of the statistics of the backtests with the transaction cost value k=20

Backtest for $k = 30$:

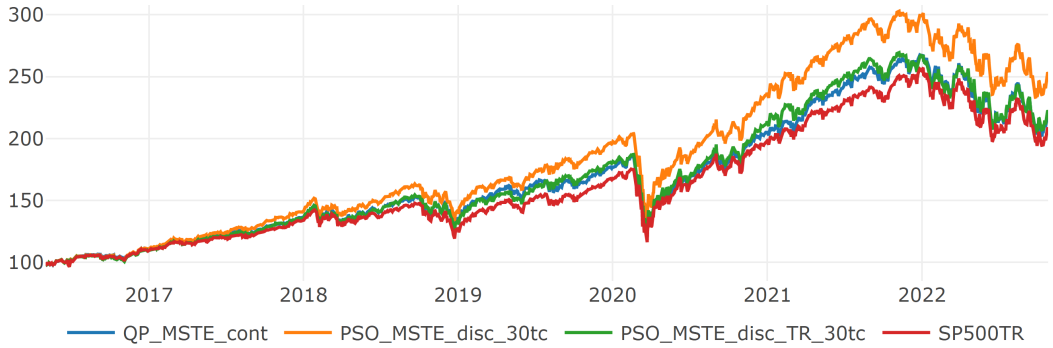

Figure 10.8: Comparison of the cumulative daily returns of the backtests with the transaction cost value k=30

and the statistics of the discretized PSO with considered transaction costs are:

calculation type	fitness	constraint break	rebalance constraint	transaction cost constraint	sum of weights	count of assets	transaction costs (in USD)
quantil 95%	0,001084	0,000	0,000	0,000	0,995	83,000	71,300
mean	0,000430	0,000	0,000	0,000	0,984	72,795	53,936
quantil 5%	0,000105	0,000	0,000	0,000	0,966	61,850	19,700

Figure 10.9: Comparison of the statistics of the backtests with the transaction cost value k=30

Since each backtest depends on different previous portfolios, further backtests are needed to draw reliable inferences about the observed phenomena. That being said, it can be observed in this particular case that an increase in k tends to lower the average transaction costs. This can be seen in the comparison of the statistics of the individual backtests, since there, except for $k = 30$, a higher k leads to lower average transaction costs. To analyze this in more detail, the loss due to transaction costs of each backtest is visualized in the following chart:

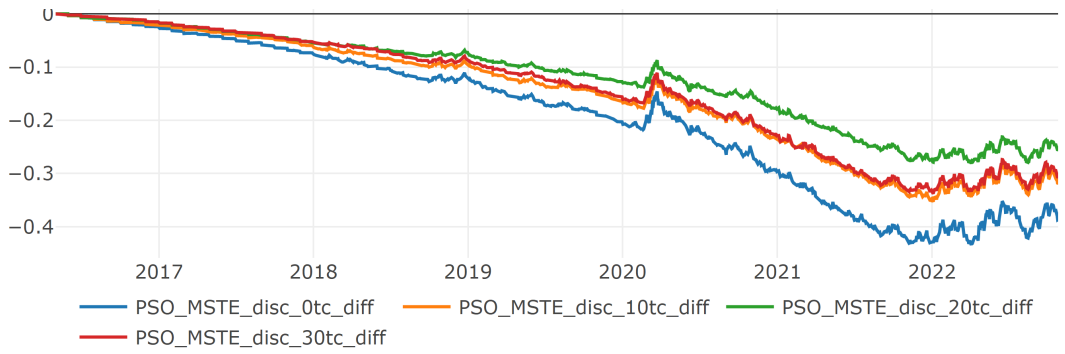

Figure 10.10: Cumulative daily return loss of the discrete PSO backtests due to transaction costs with respect to the different values of k

Another interesting result is that the continuous `solve.QP` approach with a pool of 100 assets had a slightly higher performance than the SP500TR, suggesting that the asset pool in that particular period consists of assets that performed relatively well. This can be inspected in the following performance chart, comparing only the `solve.QP` approach and the SP500TR:

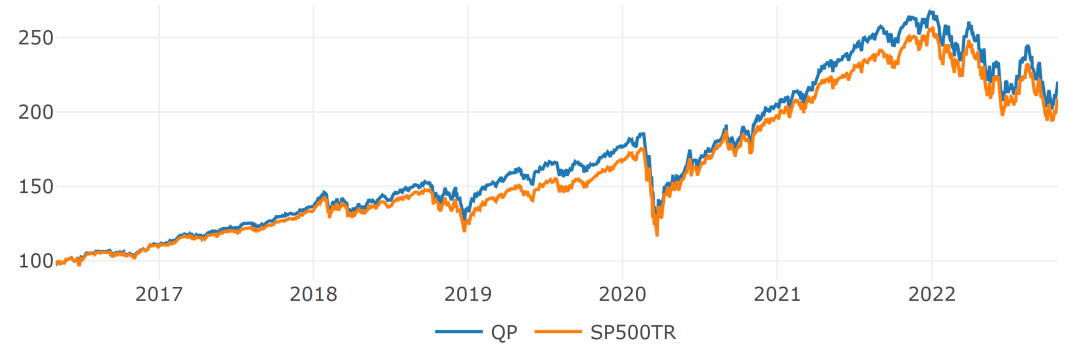

Figure 10.11: Comparison of cumulative daily returns of the continuous backtest performed with solve.QP and the SP500TR

Chapter 11

Future Research

The variants of the Particle Swarm Optimization were examined for their efficacy in solving the index tracking problem. The local PSO and the self-adaptive velocity PSO were found to be particularly effective, as they increase the diversity within the swarm and prevent premature convergence in local minima. Additionally, the self-adaptive velocity PSO offers the advantage of reducing the number of hyperparameters, making it capable of solving a wide range of problems without the need for extensive fine-tuning. The implementation of the self-adaptive velocity PSO, as described in [Fan and Yan, 2014], was found to be effective, but further research is needed to determine whether it can be improved by combining it with the local variant.

A backtesting study was conducted in the final chapter, evaluating the practical application of the index tracking problem for retail investors. The results were promising, but further evaluation is necessary to confirm the stability of the results, given the path-dependent nature of the portfolios.

Chapter 12

Conclusion

This thesis has been concerned with the analysis and testing of the PSO. The focus has been on optimization problems in the field of quantitative portfolio management. In practice, it is relevant to solve these kind of optimization problems with the same complexity as in real life, which is made possible by the PSO. It has been shown that even complex optimization problems can be solved in a stable manner.

It was shown in chapter 5 that more and more investors invest in passive managed funds, as they are more stable over time and have lower costs. To ensure competitiveness, it is important that quantitative strategies are automated as much as possible. The use of PSO in solving optimization problems in quantitative portfolio management is particularly relevant in this context, as it allows for the efficient and effective solution of these complex problems.

In chapter 6, various optimization problems commonly encountered in the realm of quantitative investment strategies within passive investment were examined and presented through the use of examples. One of the issues analyzed was the mean-variance portfolio (MVP) problem, the solution of which yields an optimal portfolio in terms of risk and return. Additionally, the topic of index tracking problem (ITP) was introduced, which aims to create a tracking portfolio that closely follows its benchmark. It has been demonstrated that it is beneficial to achieve the ITP by minimizing the mean square tracking error, which was abbreviated as ITP-MSTE.

In chapter 7, a quadratic optimizer called solve.QP was used to solve the quadratic problems with their constraints in the continuous case of the MVP problem and the ITP-MSTE. The solve.QP was also used as a benchmark to test the quality of the PSO in the following chapter 8. It was found that although the PSO requires significantly more computation time, the results are often close to the solve.QP solution. In addition, the standard PSO was explained in detail and illustrated using visualizations and WebApps. It was shown how the penalty method works to take constraints into account during optimization. Furthermore, the convergence behavior of the standard PSO was analyzed, leading to the conclusion that the standard PSO is not a local search algorithm, but can be transformed into a local and also a global search algorithm by applying small

modifications. It has been shown that PSO leads to stable results in the chosen financial optimization problems, provided that sufficient computing time is used.

Since in reality a portfolio consists of integer numbers of assets, it was shown that quadratic optimizers and subsequent rounding are not sufficient for discrete problems. For the PSO, it was shown how to deal with such discrete problems and then the discrete ITP-MSTE was solved, yielding stable results. This is particularly important in the context of quantitative portfolio management, as many real-world problems involve discrete variables.

In the next chapter 9 different variants of the PSO were analyzed and compared with the standard PSO. It was found that the local PSO and the self-adaptive velocity PSO are the best variants to solve the discrete ITP-MSTE. In addition, the created PSO variants were compared with other metaheuristics applied to the same problem. It was found that the local PSO and the self-adaptive velocity PSO are better than the other metaheuristics. Since the self-adaptive velocity PSO has no hyperparameters that need to be adjusted, it was tested extensively in the following chapter.

In the final chapter of this thesis, the self-adaptive velocity PSO algorithm was applied in a practical scenario. A simulation was conducted in which the optimization problem of a private investor who regularly rebalances their portfolio on a monthly basis was considered. To accurately simulate the conditions of a private investor who adjusts their portfolio on a monthly basis, the ITP-MSTE model was designed to consider the transaction costs and the maximum limit of rebalancing allowed in a single month. The self-adaptive velocity PSO was then utilized to solve this optimization problem and was repeatedly executed until no constraints were violated, which was achieved in all cases. The results of this simulation indicate that the model has been solved effectively. However, further extensive testing is required to accurately assess the stability of the proposed model, as the portfolios in the backtests are path dependent.

Overall, it has been demonstrated that PSO can effectively yield stable results in financial optimization problems when given sufficient computational resources. The simplicity of formulating even complex problems is a key advantage of this method. As the financial industry continues to evolve and place increasing emphasis on automation of quantitative strategies, it is likely that optimization techniques like PSO will become increasingly prevalent.

Bibliography

Ahmad Badary. How to build a basic particle swarm optimiser from scratch in r. https://ahmedbadary.github.io/work_files/research/conv_opt/hw/iftp, 2017. Accessed: 2022-10-24.

Tania Derenzis. Net performance of active and passive equity ucits. https://www.esma.europa.eu/sites/default/files/trv_2019_2-net_performance_of_active_and_passive_equity_ucits.pdf, 2019. Accessed: 2022-10-20.

Andries Petrus Engelbrecht. Particle swarm optimization: Global best or local best? In *2013 BRICS congress on computational intelligence and 11th Brazilian congress on computational intelligence*, pages 124–135. IEEE, 2013.

Eugene F Fama and Kenneth R French. Luck versus skill in the cross-section of mutual fund returns. *The journal of finance*, 65(5):1915–1947, 2010.

Qinqin Fan and Xuefeng Yan. Self-adaptive particle swarm optimization with multiple velocity strategies and its application for p-xylene oxidation reaction process optimization. *Chemometrics and Intelligent Laboratory Systems*, 139: 15–25, 2014.

Akhilesh Ganti. What is a brokerage fee? how fees work, types, and expense. https://www.investopedia.com/terms/b/brokerage-fee.asp#:~:text=Realtors%20and%20real%20estate%20brokers,offer%20a%20fixed%2Dfee%20service., 2022. Accessed: 2022-10-24.

Iuliia Gavriushina, Oliver Sampson, Michael R Berthold, Winfried Pohlmeier, and Christian Borgelt. Widened learning of index tracking portfolios. In *2019 18th IEEE International Conference On Machine Learning And Applications (ICMLA)*, pages 1800–1805. IEEE, 2019.

Donald Goldfarb and Ashok Idnani. Dual and primal-dual methods for solving strictly convex quadratic programs. *Numerical analysis*, pages 226–239, 1982.

Donald Goldfarb and Ashok Idnani. A numerically stable dual method for solving strictly convex quadratic programs. *Mathematical Programming, 27, 1–33*, 1983.

Xiaohui Hu, Russell C Eberhart, and Yuhui Shi. Engineering optimization with particle swarm. In *Proceedings of the 2003 IEEE Swarm Intelligence Symposium. SIS'03 (Cat. No. 03EX706)*, pages 53–57. IEEE, 2003.

Mauro S Innocente and Johann Sienz. Constraint-handling techniques for particle swarm optimization algorithms. *arXiv preprint arXiv:2101.10933*, 2021.

Ming Jiang, Yupin P. Luo, and Shiyuan Y. Yang. Stochastic convergence analysis and parameter selection of the standard particle swarm optimization algorithm. *Information processing letters*, 102(1):8–16, 2007.

James Kennedy and Russell Eberhart. Particle swarm optimization. In *Proceedings of ICNN'95-international conference on neural networks*, volume 4, pages 1942–1948. IEEE, 1995.

Dietmar Maringer. *Portfolio management with heuristic optimization*, volume 8. Springer Science & Business Media, 2005.

Harry Markowitz. Portfolio Selection. *Journal of Finance*, 7(1):77–91, March 1952. doi: j.1540-6261.1952.tb01525.

Harry Markowitz. *Portfolio Selection: Efficient Diversification of Investments*. Yale University Press, 1959.

Burcu Adıgüzel Mercangöz. *Applying Particle Swarm Optimization*. Springer, 2021.

nyse.com. New york stock exchange price list 2022. https://www.nyse.com/publicdocs/nyse/markets/nyse/NYSE_Price_List.pdf, 2022. Accessed: 2022-10-24.

Desmond Pace, Jana Hili, and Simon Grima. Active versus passive investing: An empirical study on the us and european mutual funds and etfs. In *Contemporary Issues in Bank Financial Management*. Emerald Group Publishing Limited, 2016a.

Desmond Pace, Jana Hili, and Simon Grima. Active versus passive investing: An empirical study on the us and european mutual funds and etfs. In *Contemporary Issues in Bank Financial Management*. Emerald Group Publishing Limited, 2016b.

Konstantinos E Parsopoulos and Michael N. Vrahatis. Recent approaches to global optimization problems through particle swarm optimization. *Natural computing*, 1(2):235–306, 2002.

Riccardo Poli. Csm-465: The sampling distribution of particle swarm optimisers and their stability. 2007.

Axel Roth. Asset allocation using particle swarm optimization in r. https://github.com/AxelCode-R/Asset-Allocation-using-Particle-Swarm-Optimization-in-R, 2022a. Accessed: 2022-11-27.

Axel Roth. Github:pso-app. https://github.com/AxelCode-R/PSO-App, 2022b. Accessed: 2022-12-01.

Axel Roth. Pso-app. https://edjut-all.shinyapps.io/PSO-App/, 2022c. Accessed: 2022-12-01.

Riddhiman Roy. How to build a basic particle swarm optimiser from scratch in r. https://www.r-bloggers.com/2021/10/how-to-build-a-basic-particle-swarm-optimiser-from-scratch-in-r/, 2021. Accessed: 2022-10-20.

Ravi Shukla. The value of active portfolio management. *Journal of economics and business*, 56(4):331–346, 2004.

Frans Van den Bergh and Andries Petrus Engelbrecht. A convergence proof for the particle swarm optimiser. *Fundamenta Informaticae*, 105(4):341–374, 2010.

Frans Van Den Bergh et al. *An analysis of particle swarm optimizers*. PhD thesis, University of Pretoria, 2007.

Eric Zivot. Introduction to computational finance and financial econometrics with r. https://bookdown.org/compfinezbook/introcompfinr/, 2021. Accessed: 2022-10-25.

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Abschlussarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Alle Stellen, die wörtlich oder sinngemäß aus den Quellen entnommen wurden, sind als solche kenntlich gemacht. Weiterhin erkläre ich, dass die Arbeit nicht anderweitig veröffentlicht oder an anderer Stelle als Prüfungsleistung vorgelegt wurde.

Stuttgart, den	Unterschrift