# Asset Allocation using Particle Swarm Optimization in R

Axel Roth

2022-11-27

# Contents

**Preface** **3**

**Abstract** **4**

**1 Software information and usage** **5**

1.1 R-Version and Packages . . . . . 5

1.2 Reproducibility . . . . . 7

1.3 R-functions . . . . . 7

**2 Open Data Sources** **8**

2.1 R-Functions . . . . . 8

**3 Mathematical Fundations** **10**

3.1 Basic Operators . . . . . 10

3.2 Formula Conventions . . . . . 12

3.3 Return Calculation . . . . . 12

3.4 Markowitz Modern Portfolio Theory (MPT) . . . . . 13

3.5 Portfolio Math . . . . . 13

3.6 R-Functions . . . . . 16

**4 Activ vs Passiv Portfolio Managment** **19**

**5 Challenges of Passiv Investing** **21**

5.1 Mean-Variance Portfolio (MVP) . . . . . 21

5.2 Index-Tracking Portfolio (ITP) . . . . . 24

**6 Analytic Solver for Quadratic Programming Problems 29**

6.1 Quadratic Programming (QP) . . . . . 29

6.2 QP-Solver from quadprog . . . . . 30

6.3 Example: Solving MVP Problem with `solve.QP()` . . . . . 30

6.4 Example: Solving ITP-MSTE with `solve.QP()` . . . . . 33

**7 Particle Swarm Optimization (PSO) 37**

7.1 The Algorithm . . . . . 37

7.2 `pso()` Function . . . . . 38

7.3 Animation 2-Dimensional . . . . . 41

7.4 Simple Constraint Handling . . . . . 42

7.5 Example MVP . . . . . 43

7.6 Example: ITP-MSTE . . . . . 44

7.7 Pros and Cons for Continuous Problems . . . . . 45

7.8 Discrete Problems . . . . . 45

7.9 Example: Discrete ITP-MSTE . . . . . 46

**8 PSO Variations 47**

8.1 Testproblem: Discrete ITP-MSTE . . . . . 47

8.2 Function Stretching . . . . . 48

8.3 Local PSO . . . . . 51

8.4 Preserving Feasibility . . . . . 53

8.5 Self-Adaptive Velocity . . . . . 55

8.6 PSO R-Package . . . . . 58

**9 Real Life ITP Example 60**

9.1 Transaction Costs . . . . . 60

9.2 Rebalancing Constraint . . . . . 62

9.3 Objective . . . . . 62

9.4 Complete ITP Example . . . . . 64

**10 Future Research 68**

**11 Conclusion 69**

# Preface

|||in progress|||
(soll vor dem TOC kommen denke ich)

# Abstract

|||in progress|||
(zusammenfassung: Vor dem TOC)

1. motivation
2. structure
3. results

# Chapter 1

# Software information and usage

All of the work was done in the amazing IDE of R-Studio using R-Markdown extended with the bookdown package. R-Markdown allows the user to write documents with encapsulated chunks of code alongside the text they are writing, and create plots to display in the output. Each codechunk can be cached to avoid time-consuming repetition, and it can be disabled to regenerate the entire work, including all the code and plots it contains. Primarily, it is helpful for the user to ensure the stability of older code, and secondarily, it is helpful for the reader who has access to the R-Markdown files, as they can trace back any analysis or plot that is displayed. R-Markdown can produce different formats such as HTML or PDF output, both of which are used for this work. The HTML output is deposited in the corresponding GitHub repository at (Roth, 2022). 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, allowing the reader to unfold important code blocks 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 immense capabilities of the R programming language, this is one of the best ways to write reproducible and good-looking works.

## 1.1 R-Version and Packages

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

```
R version 4.2.2 (2022-10-31 ucrt)
```

```
Platform: x86_64-w64-mingw32/x64 (64-bit)
Running under: Windows 10 x64 (build 19044)

Matrix products: default

locale:
[1] LC_COLLATE=German_Germany.utf8  LC_CTYPE=German_Germany.utf8
[3] LC_MONETARY=German_Germany.utf8 LC_NUMERIC=C
[5] LC_TIME=German_Germany.utf8

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

other attached packages:
 [1] ggpubr_0.5.0        metR_0.13.0         gganimate_1.0.8
 [4] quantmod_0.4.20     TTR_0.24.3          reactable_0.3.0
 [7] bookdown_0.30       webshot2_0.1.0      corrr_0.4.4
[10] data.table_1.14.4   pso_1.0.4           plotly_4.10.1
[13] matrixcalc_1.0-6    Matrix_1.5-1        quadprog_1.5-8
[16] xts_0.12.2          zoo_1.8-11          rvest_1.0.3
[19] forcats_0.5.2       stringr_1.4.1       purrr_0.3.5
[22] tidyr_1.2.1         tibble_3.1.8        ggplot2_3.4.0
[25] tidyverse_1.3.2     readr_2.1.3         dplyr_1.0.10
[28] alphavantager_0.1.2 lubridate_1.9.0     timechange_0.1.1

loaded via a namespace (and not attached):
 [1] fs_1.5.2            progress_1.2.2      httr_1.4.4
 [4] tools_4.2.2         backports_1.4.1     utf8_1.2.2
 [7] R6_2.5.1            DBI_1.1.3           lazyeval_0.2.2
[10] colorspace_2.0-3    withr_2.5.0         prettyunits_1.1.1
[13] tidyselect_1.2.0    processx_3.8.0      curl_4.3.3
[16] compiler_4.2.2      cli_3.4.1           xml2_1.3.3
[19] checkmate_2.1.0     scales_1.2.1        digest_0.6.30
[22] rmarkdown_2.18      pkgconfig_2.0.3     htmltools_0.5.3
[25] dbplyr_2.2.1        fastmap_1.1.0       htmlwidgets_1.5.4
[28] rlang_1.0.6         readxl_1.4.1        rstudioapi_0.14
[31] farver_2.1.1        generics_0.1.3      jsonlite_1.8.3
[34] car_3.1-1           googlesheets4_1.0.1 magrittr_2.0.3
[37] Rcpp_1.0.9          munsell_0.5.0       fansi_1.0.3
[40] abind_1.4-5         lifecycle_1.0.3     stringi_1.7.8
[43] yaml_2.3.6          carData_3.0-5       grid_4.2.2
[46] promises_1.2.0.1    crayon_1.5.2        lattice_0.20-45
[49] haven_2.5.1         chromote_0.1.1      hms_1.1.2
[52] knitr_1.41          ps_1.7.2            pillar_1.8.1
[55] ggsignif_0.6.4      reprex_2.0.2        glue_1.6.2
[58] evaluate_0.18       modelr_0.1.10       tweenr_2.0.2
```

```
[61] vctrs_0.5.0         tzdb_0.3.0          cellranger_1.1.0
[64] gtable_0.3.1        assertthat_0.2.1    cachem_1.0.6
[67] xfun_0.34           broom_1.0.1         rstatix_0.7.1
[70] later_1.3.0         googledrive_2.0.0   viridisLite_0.4.1
[73] gargle_1.2.1        websocket_1.4.1     memoise_2.0.1
[76] ellipsis_0.3.2
```

## 1.2 Reproducibility

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

## 1.3 R-functions

The following functions were created to simplify working with R-Markdown.

### 1.3.1 `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.

### 1.3.2 javascript files

These files are located in the `www/` folder and are used for the HTML version to make chunks expandable.

# Chapter 2

# 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, google and others.

## 2.1 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 attached (Roth, 2022).

### 2.1.1 `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 3.6.2. The output is a list containing prices and returns as xts objects. 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", "closed", "adjusted")
- `return_type = "adjusted"`: Type of return to be recorded (e.g. "open", "high", "low", "closed", "adjusted")
- `print = F`: Should the function print the return of `getSymbols()`

### 2.1.2 `buffer()`

To make data reusable and reduce compilation time, this 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 3

# Mathematical Fundations

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. 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 5.

## 3.1 Basic Operators

The table below compares frequently used mathematical symbols with R code and their meaning:

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

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.

Matrix product:

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

with an example:

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

and for a vector

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

with an example:

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

Outer product:

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

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. Suppose $\square$ is one of the four element-wise operators, then this 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

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

## 3.2 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.

## 3.3 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.

### 3.3.1 Simple Returns

The default time frame for all raw data in this thesis is one working day and only simple rates of return 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:

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

## 3.4 Markowitz Modern Portfolio Theory (MPT)

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. 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).

### 3.4.1 Assumptions of Markowitz Portfolio Theory

This thesis focuses on problems derived from Markowitz's portfolio theory (Markowitz, 1959), 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):

- 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).

## 3.5 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 a portfolio weight vector $w$, respectively:

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

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.

### 3.5.1 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 $\mu_i$ 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 3.6.4.

### 3.5.2 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):

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

### 3.5.3 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 unique 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 basis function `cov()` and historical data. The function `cov_()` in 3.6.1 can be used to calculate the covariance using the geometric mean to estimate the expected returns.

### 3.5.4 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:

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

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

### 3.5.5 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 $R^P_{t_1}$. This can be done with the following two steps:

$$Z_{t_1} = (1 + R_{t_1}) \cdot t(w_{t_0})$$
$$R^P_{t_1} = \sum_{n=1}^{N} Z_{t_1,n} - 1$$

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:

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

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.

## 3.6 R-Functions

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

### 3.6.1 `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.

### 3.6.2 `pri_to_ret()`

This function calculates the simple returns of a given xts object and replaces missing values with the previous value of this column.

### 3.6.3 `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 time series.

### 3.6.4 `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 return, 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
}
```

### 3.6.5 `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 4

# Activ vs Passiv Portfolio Managment

An active portfolio manager seeks to achieve positive alpha, i.e., excess returns relative to the market, by applying its 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 manager achieves his goal with a quantitative strategy and assumes that the results will be stable over time.

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

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.

Fama and French attempted to distinguish between luck and chance using regression models. Their results suggest that some active managers in the top percentiles have sufficient skill to cover costs over the long run. Nonetheless, aggregate actively managed funds in the top percentiles have estimated alpha

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

Similar results were obtained in (Desmond Pace and Grima, 2016a), which analyzed European and U.S. active and passive funds. The results show that none of them is superior by cost. They suggest comparing actively and passively 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 may have a slight advantage for investors with a long investment horizon.

# Chapter 5

# Challenges of Passiv Investing

In this chapter, two common challenges of passive investing 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.

## 5.1 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.

### 5.1.1 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:

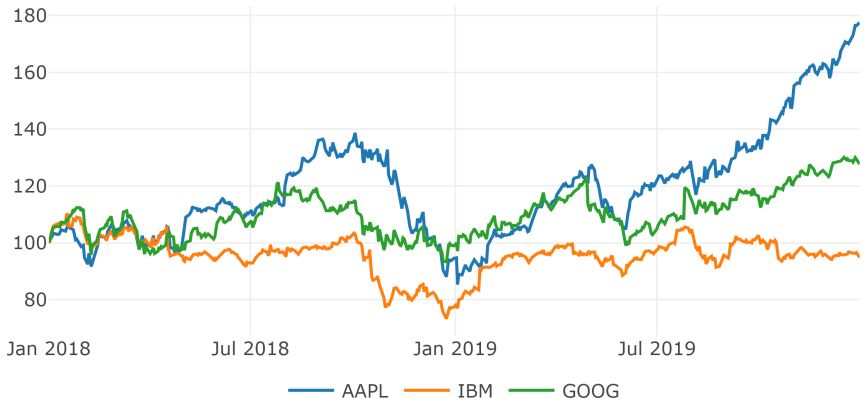
$$\min_{w} \quad \lambda \, w^T \sum w - (1 - \lambda) \, \mu^T w \tag{5.1}$$

The risk aversion parameter $\lambda$ 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.

### 5.1.2 MVP: Example

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 $\lambda$. First, the daily returns of these three assets from 2018-01-01 to 2019-12-31 are loaded.

The cumulative daily returns are:

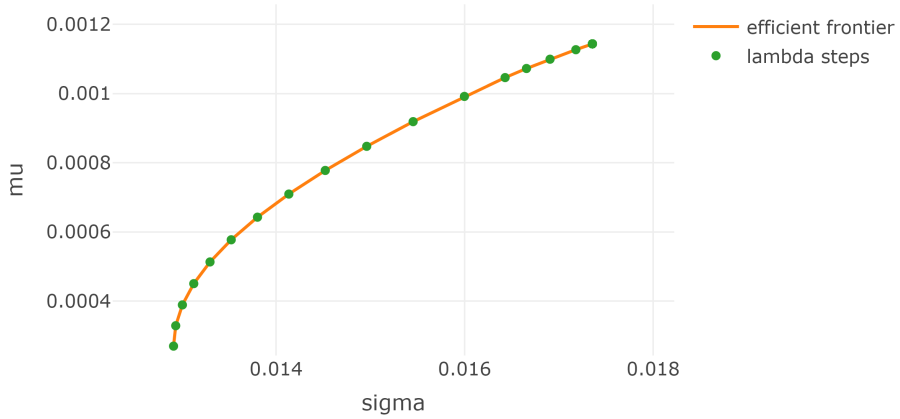



The expected daily returns and the covariance matrix for the three assets can be estimated using the formulas from chapter 3:

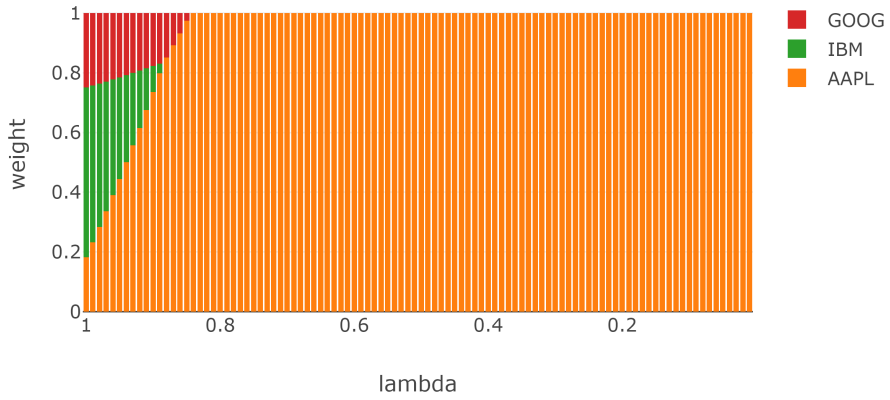
```
estimation of expected daily returns:
         AAPL           IBM          GOOG
 0.0011434115 -0.0001059164  0.0004870292

estimation of positiv definite covariance matrix:
              AAPL          IBM         GOOG
AAPL 0.0003012226 0.0001177826 0.0001799097
IBM  0.0001177826 0.0002047608 0.0001158735
GOOG 0.0001799097 0.0001158735 0.0002728911
```

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

The resulting portfolios are plotted in the daily $\mu$-$\sigma$ diagram to create the efficient frontier:



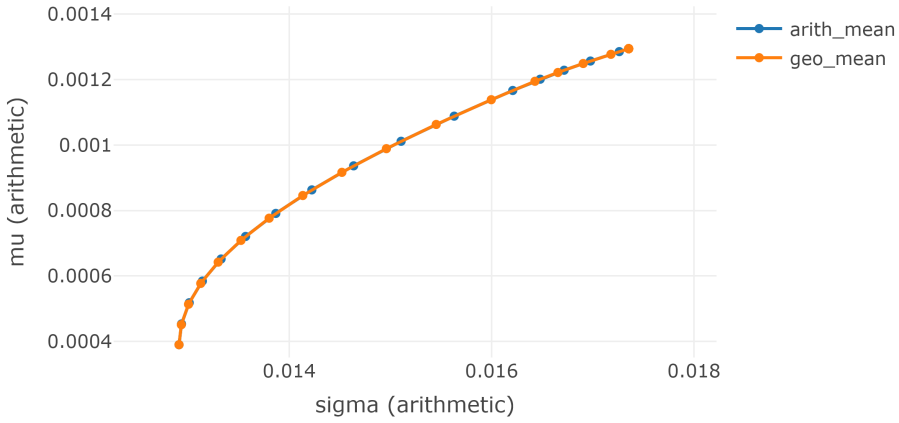

The portfolio compositions for each $\lambda$ are:




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

### 5.1.3 MVP: Compare Estimators

The above solution for the MVP problem was performed using a geometric mean to estimate the expected returns $\mu$ 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 $\lambda$ using the arithmetic mean versus the geometric mean to estimate $\mu$. The weights of the MVP's using the geometric mean are used to calculate $\mu$ and $\sigma$ as a function of the arithmetic mean to make both efficient frontiers comparable:




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 $\lambda$ are sufficient to create test-cases for the PSO. The later examples with the MVP will always use the geometric mean returns as estimation for the expected returns.

## 5.2 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 (S&P 500 for short) 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 asset 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 (Desmond Pace and Grima, 2016b). 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 reblanance 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 should overlap with the investment universe of 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 (Iuliia Gavriushina, 2019).

### 5.2.1 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 3.5.4:

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

And the last term can be solved in the same way as in (Zivot, 2021):

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

This results in the final formulation of the ITP:

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

The above problem can be estimated using the formulas and functions created in chapter 3 and historical data $R$ 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

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

It is possible to add some basic constraints, as in the MVP to sum the weights to 1 and be 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 (MSTE).

### 5.2.2 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:

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

Which results in the following minimization with neglected stretching factor:

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

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

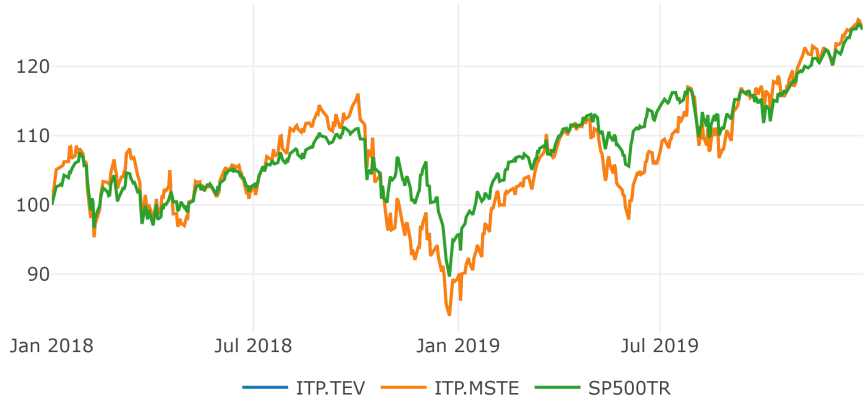
$$D = R^T \times R$$

and

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

### 5.2.3 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 (SP500TR in short). 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 return changes of the ITP-TEV and ITP-MSTE are:




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

```
      type      AAPL       IBM      GOOG
1  ITP-TEV 0.2588844 0.4163274 0.3247882
2 ITP-MSTE 0.2586717 0.4165150 0.3248133
```

# Chapter 6

# Analytic Solver for Quadratic Programming Problems

The advantages and disadvantages of analytical solvers for quadratic programming problems are discussed in this chapter. It is beyond the scope of this thesis to explain the underlying mathematical principles of how a solver solves quadratic problems; only the applications and analysis are discussed. The main reason for dealing with analytic solvers for quadratic programming problems is to use them as a benchmark for PSO.

## 6.1 Quadratic Programming (QP)

A quadratic program is a minimization problem of a function that returns a scalar value 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 used 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.

## 6.2 QP-Solver from quadprog

The most 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 $D$ must 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 percentage change in elements over $10^{-15}\%$, 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

## 6.3 Example: 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
    )
```

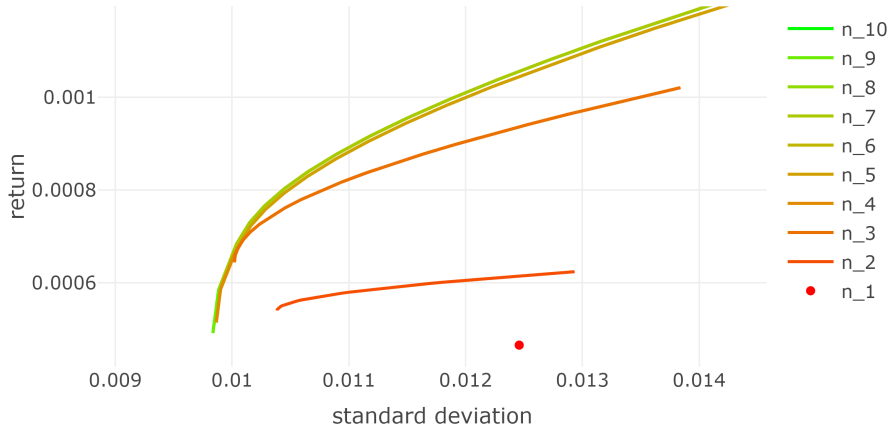
```
    qp <- solve.QP(
      Dmat = mat$Dmat, dvec = mat$dvec,
      Amat = mat$Amat, bvec = mat$bvec, meq = mat$meq
    )

    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 diagram can be created:




It can be seen, that each asset added results in a minimum variance portfolio with smaller or equal standard deviation. Nevertheless, we started with the asset that has the smallest standard deviation of 0.012459. This is the effect of diversification mentioned by Markowitz.

## 6.4 Example: 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`.

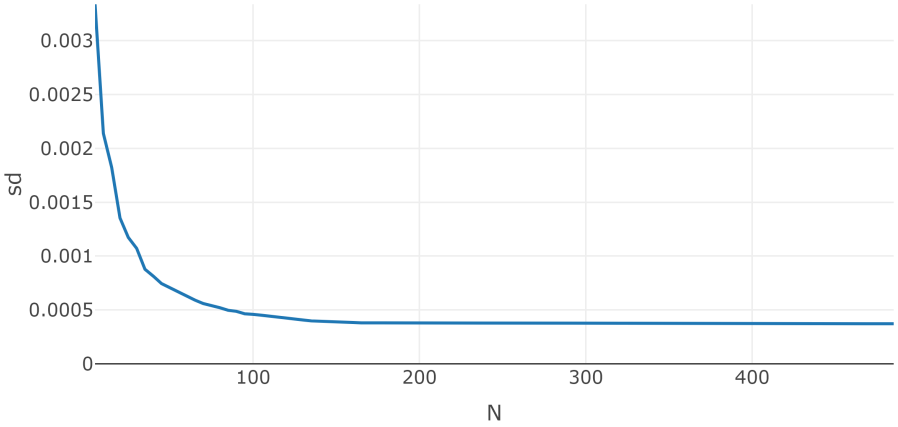
```
itp <- function(pool_returns, bm_returns){
  mat <- list(
    Dmat = t(pool_returns) %*% pool_returns,
    dvec = t(pool_returns) %*% bm_returns,
    Amat = t(rbind(
      rep(1, ncol(pool_returns)), # sum up to 1
      diag(1,
           nrow=ncol(pool_returns),
           ncol=ncol(pool_returns))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

```
  }
}
```




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 7

# Particle Swarm Optimization (PSO)

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 (James Kennedy, 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 (Konstantinos Parsopoulos, 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 the low computational costs, since only basic mathematical operators are used, the extensibility and the simplicity.

## 7.1 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 $P_d$. 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 (Konstantinos Parsopoulos,

2002):

$$v_d^{i+1} = wv_d^i + c_p r_1^{i(d)}(P_d^i - x_d^i) + c_g r_2^{i(d)}(p_g^i - x_d^i)$$
$$x_d^{i+1} = x_d^i + v_d^{i+1}$$

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 inertial weight $w$ is crucial for the convergence behavior by remembering part of its previous trajectory. A study reviewed in (Konstantinos Parsopoulos, 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 $X$.

## 7.2 `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 returns a scalar that needs to be minimized. The objective function mainly needs a vector `pos` that describes the position of one particle (e.g. weights). 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 function `mrunif()` which produces a matrix of uniformly distributed random numbers between `lower` and `upper`:

```
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 a axis, they are pushed back to the edge and the velocities 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 best fitness after 100 iterations is 0.0000001 and the best possible solution is 0.

## 7.3 Animation 2-Dimensional

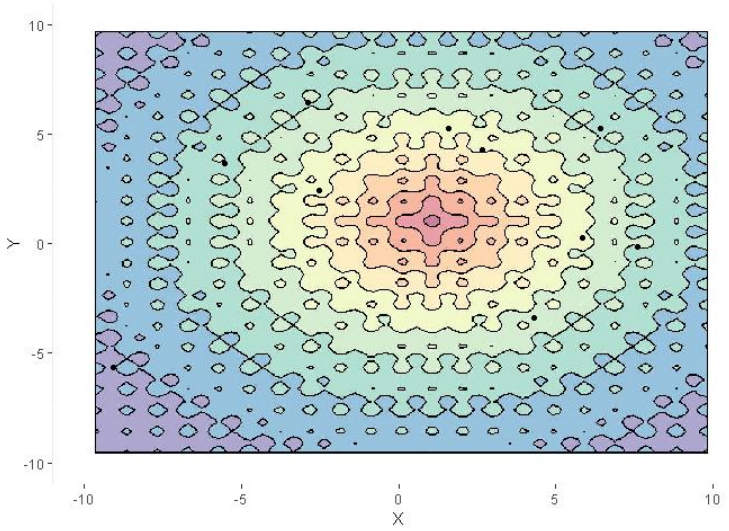
This section provides insights into the behavior of the PSO by visualizing multiple iterations in a GIF. The GIF works in Adobe Acrobat DC or in the Markdown/HTML version of this thesis. The amazing animation template and the objective function is inspired by (R'tichoke, 2021). The PSO core from the 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 minima at $(1, 1)$ with the value 0. The background color scale ranges from 0 as red to 20 as purple. 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:




## 7.4 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. This can be used to create a generic constraint handling function for these particular QP's. Both problems must satisfy the following equation:

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

To calculate a value for the intensity of constraint breaks, the above inequality gets subtracted by $b_0$ and defines:

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

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:

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

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:

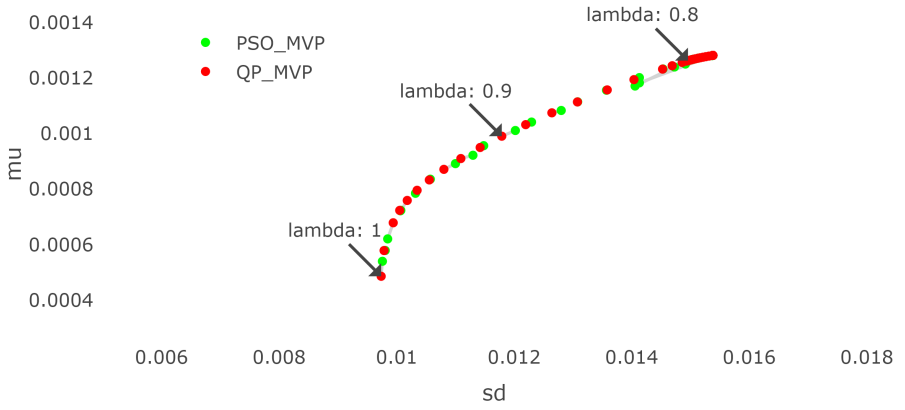
```
fn <- function(x){
  fitness <- calc_fit(x)
  constraints <- calc_const(x)
  return(fitness + 10 * constraints)
}
```

This approach to dealing with 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 (Mauro S. Innocente, 2008), 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.

## 7.5 Example MVP

This example uses the `solve.QP()` approach from section 6.3 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 $\lambda$. 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 main characteristics of all portfolios created with the `solve.QP()` compared to the PSO are shown below:




The corresponding portfolios for each $\lambda$ 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.

## 7.6 Example: ITP-MSTE

The same ITP-MSTE solved with `solve.QP()` in 6.4 is used as the 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 sum to one. The parameters for the PSO are a swarm size of 100, 100 iterations, the inertia weight starts at 1.2 and decreases to zero, the upper bound is 0.05, and a 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 | 0.4 |
| ITP-MSTE_PSO_best | 0,00160 | -0.0217001 | 9.20462883e-9 | 43.9 |
| ITP-MSTE_PSO_mean | 0,00165 | -0.0216649 | 6.245488711e-9 | 44 |

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

## 7.7 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 these types of problems, the condition for the variable of interest $x$ is to be a integer vector. These kind of problems could be solved by the `solve.QP()` approach only continuously 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.

## 7.8 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 conditions 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:

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

## 7.9 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 6.4 and tries to construct a 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 used prices are closing prices and both PSO's have 200 particles and 200 iterations. The results can be observed in the table below:

| type | fitness | const_break | sum_wgt | time |
|---|---|---|---|---|
| solve.QP discrete | -0,02143 | 0,0216 | 0,843 | 0,010 |
| PSO | -0,02179 | 0,0000 | 0,992 | 14,270 |
| PSO with solve.QP as init solution | -0,02187 | 0,0000 | 0,994 | 12,230 |

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 still reached a fitness close to 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 problems in practice.

# Chapter 8

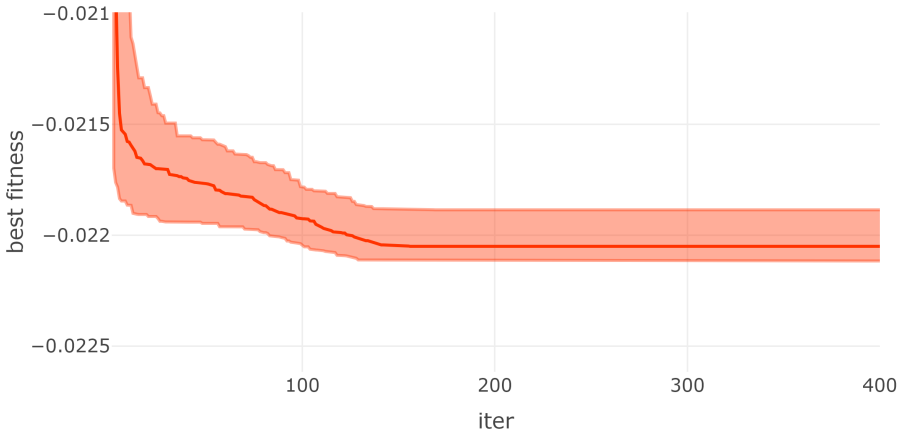
# PSO Variations

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.

## 8.1 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 6.4. 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 space of the problem. 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 function created in the previous chapter. The swarm size for the PSO and all variants is 50 and the iterations are set to 400. 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:




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 | 5,40 | 0,000000 | -0,022172 | -0,022108 | -0,022139 | -0,022140 |

## 8.2 Function Stretching

PSO often gets stuck in local minima, i.e., if the current best global position is a local minima with a larger environment around it, with only higher fitness, it is hard for the PSO to escape and find the global minima. Function stretching tries to make the PSO escape from such local minima by transforming the fitness function in a way described in (Konstantinos Parsopoulos, 2002). It states that after finding a local minimum, a two-stage transformation proposed by Vrahatis in 1996 can be used to stretch the original function so that the discovered local minimum is transformed into a maximum, but any position with less fitness remains unchanged. The two stages of the transformation with a discovered local minimum $\bar{x}$ are:

$$G(x) = f(x) + \gamma_1 \cdot \|x - \bar{x}\| \cdot (\text{sign}(f(x) - f(\bar{x})) + 1) \tag{8.1}$$

and

$$H(x) = G(x) + \gamma_2 \cdot \frac{\text{sign}\Big(f(x) - f(\bar{x})\Big) + 1}{\tanh\Big(\mu \cdot (G(x) - G(\bar{x}))\Big)} \tag{8.2}$$

The function $G(\bar{x})$ can be simplified to $f(\bar{x})$ 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:

$$\begin{aligned} \gamma_1 &= 5000 \\ \gamma_2 &= 0.5 \\ \mu &= 10^{-10} \end{aligned}$$

It is difficult to interpret both transformations exactly, 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.

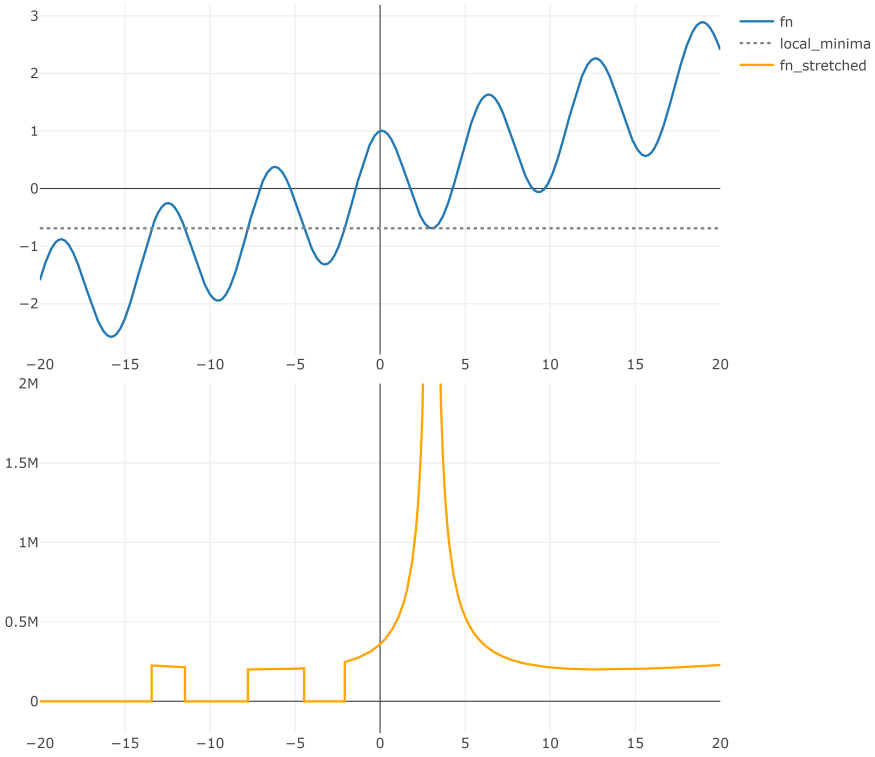
To better understand the transformation, it is used to stretch a simple function in $\mathbb{R}^1$ defined as follows:

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

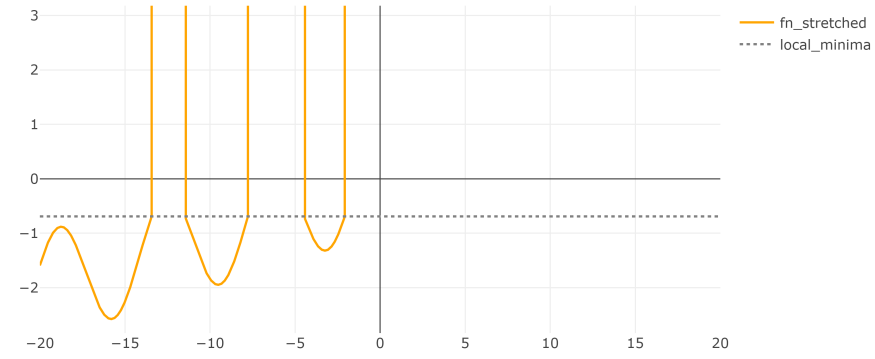
translated to the objective function:

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

and 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 (8.2), are shown in the following graph:




It can be seen that the fitness is stretched upward around the local minima $\bar{x}$, making it much easier for the PSO to move down the hill and fall into new minima with lower fitness. All the lower fitness regions remain unchanged, as can be seen in the zoomed version of the bottom diagram from above:

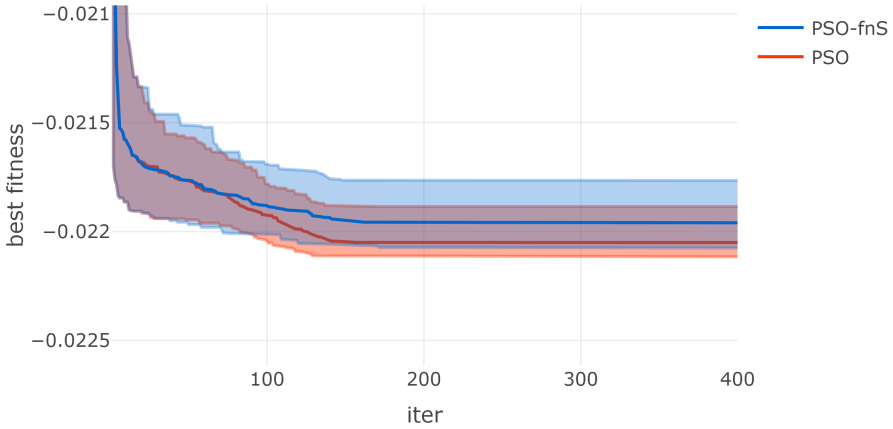


### 8.2.1 Implementation

Since it is not possible to know if the PSO is stuck in a local minima, 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 minima is assumed and the transformation of the objective function 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.

### 8.2.2 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 $\mu = 10^{-10}$:




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 | 5,40 | 0,000000 | -0,022172 | -0,022108 | -0,022139 | -0,022140 |
| 400 | PSO-fnS | 5,88 | 0,000000 | -0,022140 | -0,022036 | -0,022088 | -0,022089 |

## 8.3 Local PSO

A local PSO is a more general case of the global PSO, which is called the standard PSO. The only difference is the selection of the global best particle by defining

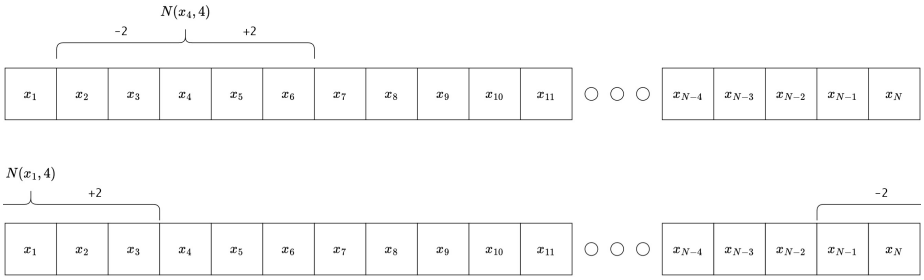
a neighborhood. Each particle $x_i$ has a neighborhood $N(x_i, \bar{k})$, and the global best particle in its neighborhood is called the local best particle of $x_i$. If the neighborhood is chosen large enough to contain all particles, it corresponds to the standard PSO (global PSO). A simple definition of a neighborhood with $k$ neighbors for particles $x_i$ given in (Engelbrecht, 2013) would be:

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

with

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

To illustrate this, the following figure defines the neighborhoods $N(x_4, 4)$ and $N(x_1, 4)$:



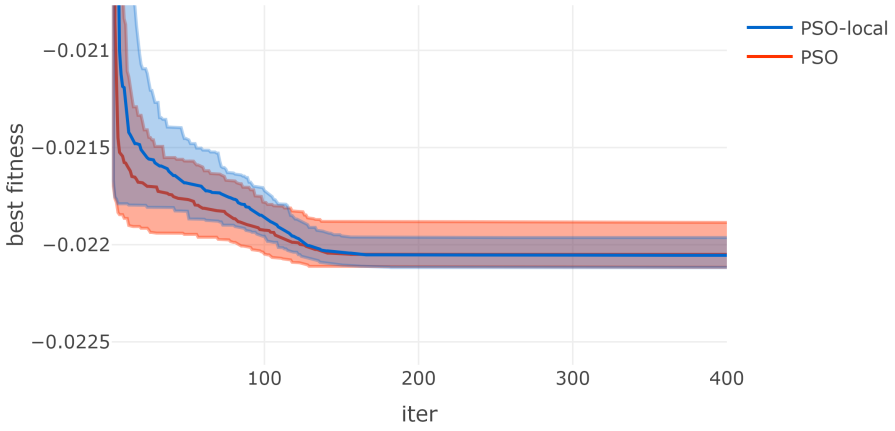

In the latter case, it can be seen that the overflowing boundary will continue on the opposite side of the arranged particles.

### 8.3.1 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 version there is no global best particle, instead the global best particle for the neighborhood of each particle has to be calculated in each step.

### 8.3.2 Test Local PSO

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




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 | 5,40 | 0,000000 | -0,022172 | -0,022108 | -0,022139 | -0,022140 |
| 400 | PSO-local | 5,39 | 0,000000 | -0,022110 | -0,022056 | -0,022083 | -0,022082 |

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

## 8.4 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, one of the inventors of PSO, Russel Eberhardt, in (Hu and Eberhardt, 2002) explored a variant that preserves the feasibility of the solutions. To be precise, this is not a variant of its own, but 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, the result of trying to find a feasible position among a million randomly generated positions is as follows:

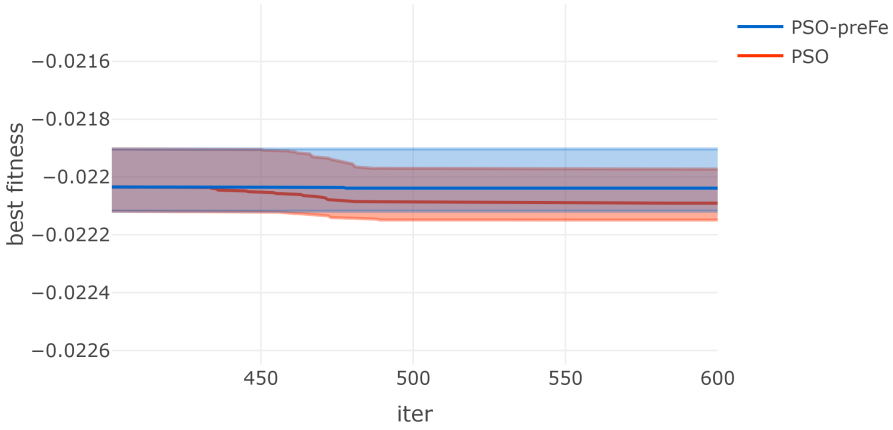
```
[1] "Feasable positions: 0"
```

```
[1] "Elapsed time: 297.01 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 was modified to start with only one feasible position, which was randomly determined from one of the final solutions of the standard PSO. To allow comparison with the standard PSO, the procedure was run again with the same starting positions.

### 8.4.1 Test Preserving Feasibility PSO

In this section, we compare the PSO with feasibility preservation and the standard PSO. Both PSOs use the same feasible solutions as starting positions:




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 |
|---|---|---|---|---|---|---|---|
| 600 | PSO | 2,60 | 0,000000 | -0,022183 | -0,022142 | -0,022166 | -0,022168 |
| 600 | PSO-preFe | 4,86 | 0,000000 | -0,022171 | -0,022113 | -0,022141 | -0,022142 |

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 PSO with the previous best position until the solution is feasible.

## 8.5 Self-Adaptive Velocity

A self-adaptive velocity PSO approach that attempts to reduce hyperparameters was analyzed in (Qinqin Fan, 2014). The self-adaptive velocity is enabled by multiple velocity update schemes that are used randomly. In addition, all hyperparameters are self-adaptive in the way that each particle has its own coefficients $c_g$, $c_p$, and $w$, which change after each iteration depending on the distance to maximum fitness, among other factors. The resulting PSO has no real hyperparameters to adjust, which allows it to be used as a general-purpose PSO.

### 8.5.1 Implementation

The process of this PSO variant is too different from the standard PSO, so all changes are combined 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 $r = \text{Unif}(0, 1)$ in iteration $i + 1$:

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

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

   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 feasible search space in its coordinate $z$, it is moved back with the following switch-case for $r = \text{Unif}(0, 1)$:

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

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:

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

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

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

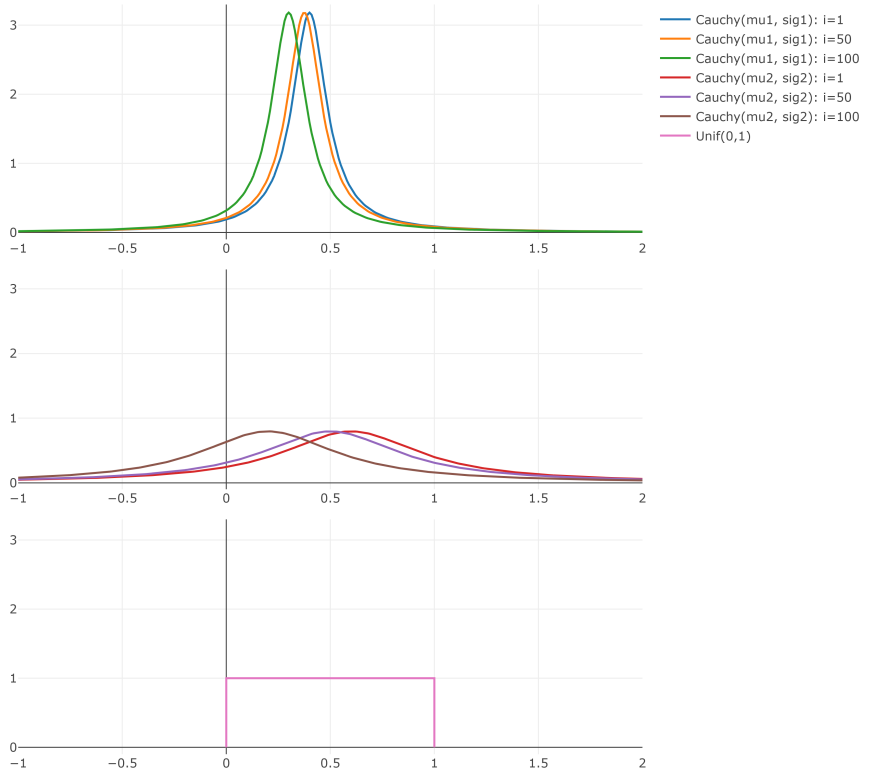
5) Update the best positions
Update the personal best $P$ and global best $p_g$ positions as in the standard PSO.

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

### 8.5.2 Analyse Implementation

The random use of the distributions for the velocity update increases the diversity of the swarm. The coefficients of iteration $i$ with 100 maximum

iterations are distributed as follows:

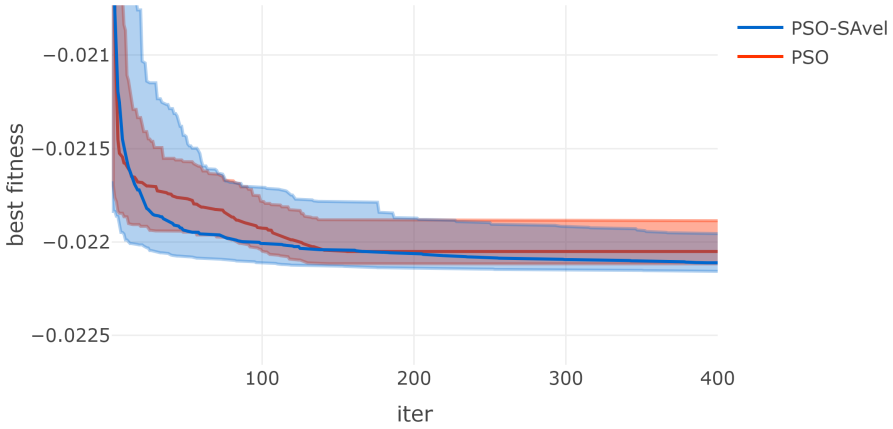



It can be seen that the randomness of the motion increases compared to the uniform distribution and the center of the Cauchy distributions slowly decreases towards the absolute term. In addition, the two Cauchy distributions differ in explorability and exploitability, indicated by probabilities outside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

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."

### 8.5.3 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:



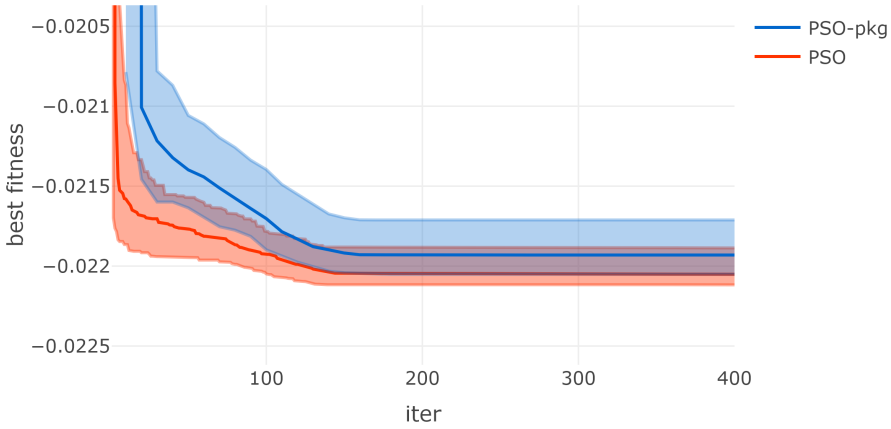

The aggregate statistics of the last iterations of all 100 runs can be found in the table below:

| iter | type | time_mean | const_break_mean | best_fit_q1 | best_fit_q3 | best_fit_mean | best_fit_median |
|---|---|---|---|---|---|---|---|
| 400 | PSO | 5,40 | 0,000000 | -0,022172 | -0,022108 | -0,022139 | -0,022140 |
| 400 | PSO-SAvel | 10,01 | 0,000000 | -0,022188 | -0,022143 | -0,022166 | -0,022166 |

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.

## 8.6 PSO R-Package

In this section, the existing package `pso` from R 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 diagram below compares the fitness:




The existing PSO package is slightly different from the standard PSO, which is most likely due to the different velocity initialization. It is important to note that the existing PSO package has many features and variants that are disabled for this comparison. 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 | 5,40 | 0,000000 | -0,022172 | -0,022108 | -0,022139 | -0,022140 |
| 400 | PSO-pkg | 5,92 | 0,000000 | -0,022052 | -0,021711 | -0,021909 | -0,021930 |

# Chapter 9

# Real Life ITP Example

In the previous chapters, the capabilities of a 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 time points, first by adding transaction costs to the problem. For the first rebalancing time point, 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 the historical data of each rebalancing date. Later, a full backtest of an ITP is evaluated and analyzed in a real-world environment.

## 9.1 Transaction Costs

The cost of buying or selling assets must be considered as it 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 an 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 cost by increasing the objective value, but it is difficult to make the intensity of the transaction cost value comparable to the objective value. The objective value $v^o$ of the ITP with MSTE approach, as in 5.2.2, 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 return $r_{tc}$:

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

with

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

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 cost, 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 cost 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.

## 9.2 Rebalancing Constraint

The rebalancing constraint restricts the changes in weights by considering the previous portfolio weight vector $w^{prev}$, which is recalculated using the previous shares vector and the rebalanced portfolio weight vector $w^{reba}$. The value constrained by the rebalancing constraint should take into account the weights moved between assets and additional weights added. Example: the previous portfolio had a weight vector $w^{prev} = [0.5, 0.4]$ as of the current rebalancing date and the rebalanced portfolio has a weight vector $w^{reba} = [0.8, 0.2]$. The rebalanced weight from the second to the first asset is 0.2 and the additional weight added is 0.1, resulting in a rebalance of $0.2 + 0.1 = 0.3$ weight. Below is the formula for calculating the rebalancing weight $w^{rb}$:

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

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

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

## 9.3 Objective

The goal is to simulate a tracking portfolio that tracks the SP500TR with a pool of 100 assets included in 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, continuously discarding assets as in section 6.4, 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 gives stable results. The tracking portfolio has the following constraints: discrete number of stocks, long only, maximum weight of 10%, $0.96 \leq \sum w_i \leq 0.995$, rebalancing under 30% weight, considering transaction cost 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 rebalancing portfolio is simulated with the portfolio return function from 3.5.5 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 following figure:

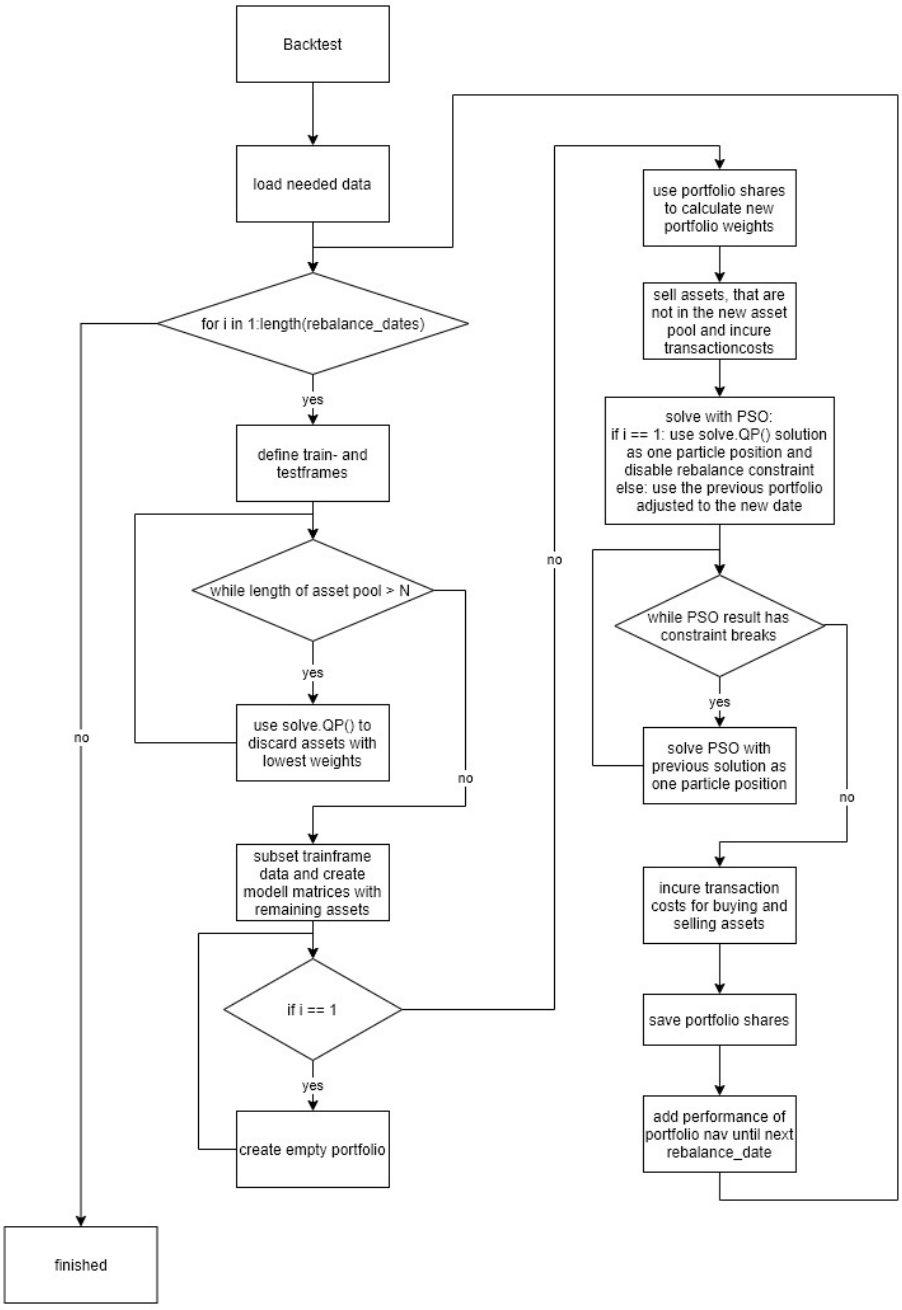
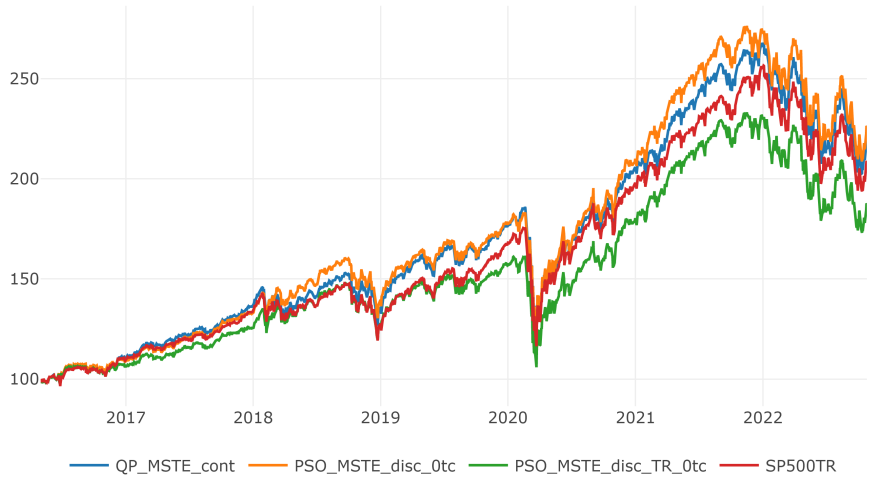



Figure 9.1: Backtest Process

## 9.4 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 using the ITP MSTE approach 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_TE` considers all transaction costs in its performance. Everything is compared to the SP500TR which is the objective to track. Furthermore there are used different values for the transaction costs intensity $k \in \{0, 10, 20, 30\}$ of each backtest which can be seen in the legends by `0tc`, `10tc`, `20tc` and `30tc`.

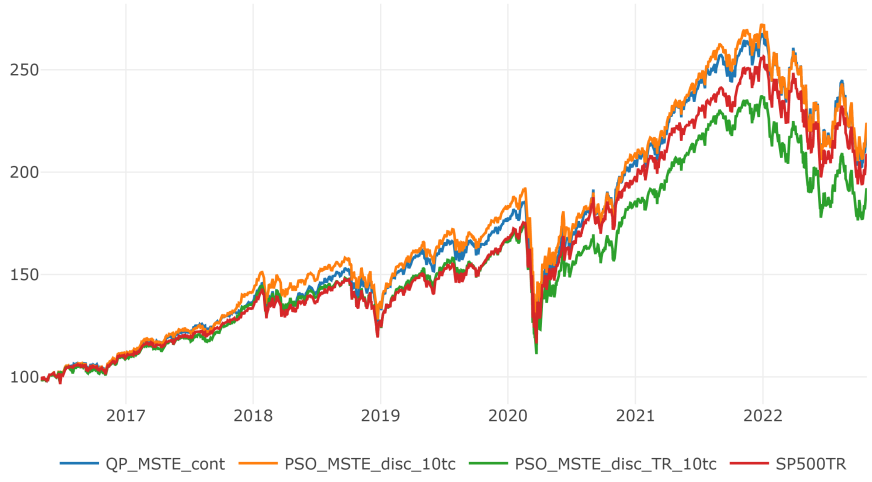
Backtest for $k = 0$:




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 |

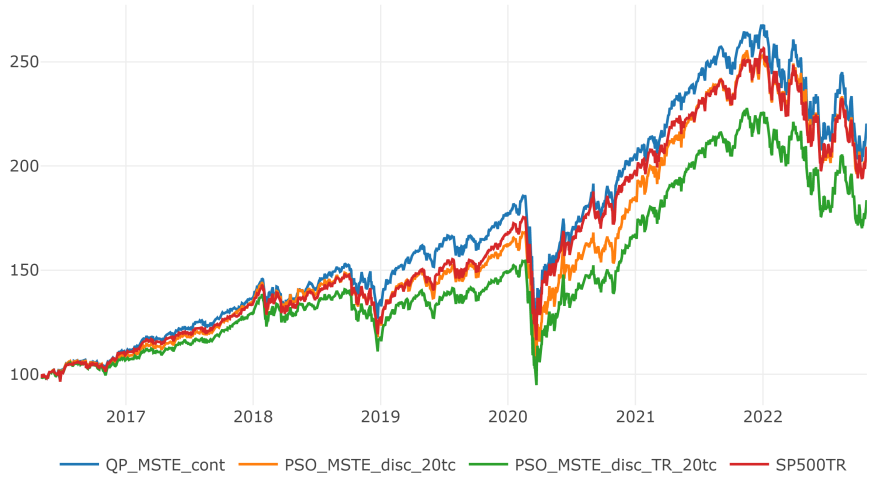
Backtest for $k = 10$:




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 |

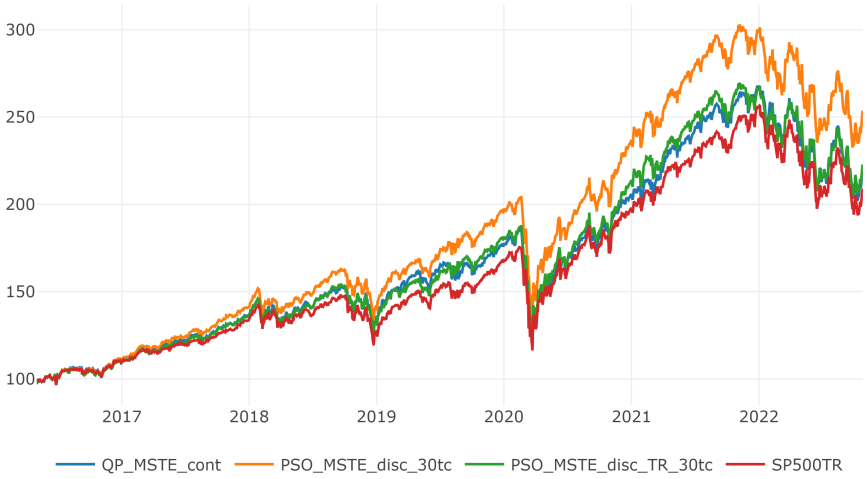
Backtest for $k = 20$:




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 |

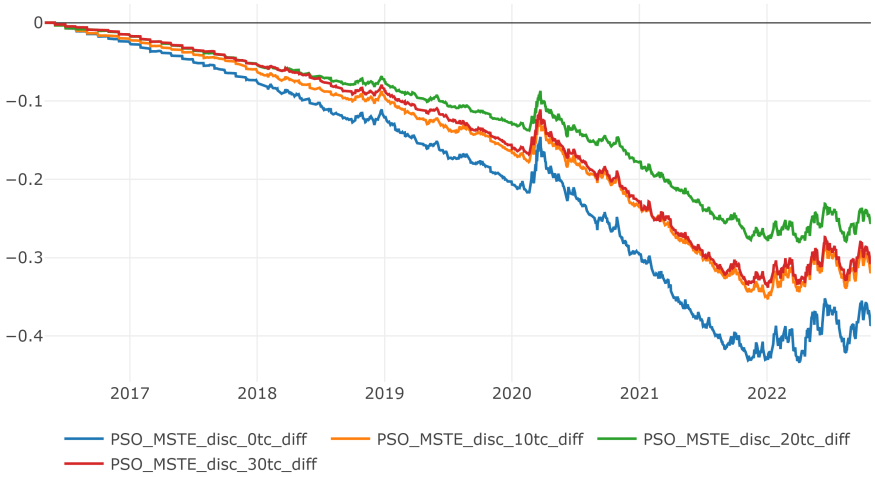
Backtest for $k = 20$:




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% | 0,001084 | 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 |

It can be seen that increasing $k$ tends to reduce the average transaction cost in USD. Since each backtest depends on different previous portfolios, further backtests are needed to confirm this result. The absolute loss of `nav` between the results of the PSO without and with transaction costs for each $k$ can be found in the chart below:




Another interesting result is that the `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 chart, comparing only the `solve.QP()` approach and the SP500TR:

# Chapter 10

# Future Research

Some PSO variants were analyzed and showed promising results in solving the index tracking problem. The best of them were the local and the self-adaptive velocity variants, both of which increase the ability to find good solutions by increasing the diversity in the swarm. This often prevents premature convergence in local minimum situations. The additional advantage of the self-adaptive velocity variant is the reduction in hyperparameters, making it capable of solving a wide range of problems without additional fine-tuning effort. The implementation is identical to the source and works well, but it would be interesting to analyze whether it can be improved by combining it with the local variant.

The last chapter analyzed backtests of a practical index tracking problem for retail investors, which yielded promising results but require further evaluation to verify their stability due to path-dependent portfolios.

# Chapter 11

# Conclusion

It has been shown that the PSO is capable of solving practical financial problems that are hardly solvable with analytical approaches without any limitations or losses in practicability. The advantage of the PSO is that problems can be formulated in any complexity and still be applied. This was shown especially in the last chapter, where a discrete index tracking problem was formulated, which is not analytically solvable. Exactly in such situations heuristics like the PSO are the last possibility to have the chance to formulate and solve the whole problem without restrictions. Nevertheless, an analytical approach should always be preferred if possible, since the PSO is not always able to find the best solution. If the discretization is the only obstacle for the analytical approach to be practical, the PSO chapter has shown that a continuous solution using the PSO can find a qualitative solution, taking into account the constraints. The chapter on PSO variants gave an insight into the many ways in which the standard PSO can be modified to improve behavior. It was shown that there are variants that increase diversification and reduce hyperparameters. In the chapter Active vs Passive Portfolio Management it has been analyzed how relevant it is to generate portfolios more and more automatically in order to save costs and to be more attractive in competition with other professional managed funds. This has an increasing relevance in the future, which is why it may be that PSO will be used more and more within portfolio management.

# Bibliography


Badary, A. (2017). How to build a basic particle swarm optimiser from scratch in r.

Desmond Pace, J. H. and Grima, S. (2016a). Active versus passive investing: An empirical study on the us and european mutual funds and etfs.

Desmond Pace, J. H. and Grima, S. (2016b). Active versus passive investing: An empirical study on the us and european mutual funds and etfs.

Engelbrecht, A. (2013). Particle swarm optimization: Global best or local best?

Fama, E. F. and French, K. R. (2010). Luck versus skill in the cross-section of mutual fund returns.

GANTI, A. (2022). What is a brokerage fee? how fees work, types, and expense.

Goldfarb, D. and Idnani, A. (1982). Dual and primal-dual methods for solving strictly convex quadratic programs.

Goldfarb, D. and Idnani, A. (1983). A numerically stable dual method for solving strictly convex quadratic programs.

Hu, X. and Eberhardt, R. (2002). Luck versus skill in the cross-section of mutual fund returns.

Iuliia Gavriushina, O. S. (2019). Widened learning of index tracking portfolios.

James Kennedy, R. E. (1995). Particle swarm optimization.

Konstantinos Parsopoulos, M. N. V. (2002). Recent approaches to global optimization problems through particle swarm optimization.

Maringer, D. (2005). *Portfolio Management with Heuristic Optimization.*

Markowitz, H. M. (1959). *Portfolio Selection EFFICIENT DIVERSIFICATION OF INVESTMENTS.*

Mauro S. Innocente, J. S. (2008). Constraint handling techniques for particle swarm optimization algorithms.

nyse.com (2022). New york stock exchange price list 2022.

Qinqin Fan, X. Y. (2014). Self-adaptive particle swarm optimization with multiple velocity strategies and its application for p-xylene oxidation reaction process optimization.

Roth, A. (2022). Asset allocation using particle swarm optimization in r.

R'tichoke (2021). How to build a basic particle swarm optimiser from scratch in r.

Zivot, E. (2021). *Introduction to Computational Finance and Financial Econometrics with R.*