Asset Allocation using Particle Swarm Optimization in R

Axel Roth

2022-09-06

Contents

Preface

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

Chapter 1

Abstract

|||in progress|||
Things about this thesis. why and what question should be answered. and what are the answers. (zusammenfassung)

Chapter 2

Software information and usage

|||in progress||| wie ich das buch schreibe, R markodwn bookdown und so und welche versionen ich nutze

2.1 R-Version and Packages

2.2 Reproducibility

github und code im bookdown

2.3 R-functions

zb plotly_save

Chapter 3

Open Data Sources

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

3.1 R-Functions

The following functions were created to increase the ease of data collection with the quantmod R package, which can be found in the `R/` directory in the attached github repository.

3.1.1 `get_yf()`

This function is the main wrapper for collecting data with `getSymbols()` from yahoo-finance, and converts prices to returns with the `pri_to_ret()` function explained in 4.5.1. 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()`

3.1.2 `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 4

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 in PSO because the dimension of the objective function is based on the number of elements that can be selected, see chapter 6.

4.1 Basic Operators

A compendium comparing commonly used mathematical symbols with R code and their meaning is given in the following table:

Latex/Displayed	R-Code	Meaning
×	%*%	Matrix or Vector multiplication and Cross-Product
⊗	%*%	Outer Product
A^T	t(A)	Transpose of Matrix or Vector A
·	*	Scalar or elementwise Vector multiplication

4.2 Return Calculation

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

4.2.1 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 there is an asset with price P on working day t_i and the following working day t_{i+1}, it follows that the simple rate of return on t_{i+1} can be calculated as follows:

$$R_{i+1} = \frac{P_{t_{i+1}}}{P_{t_i}} - 1$$

4.3 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 true when assets are not perfectly correlated, as bad

and good performance can offset each other, reducing the likelihood of extreme events. For more information, see (Maringer, 2005).

4.3.1 Assumptions of Markowitz Portfolio Theory

The following list contains all Markowitz assumptions according to (Maringer, 2005):

- Perfect market without taxes or transaction costs.
- Short sales are disallowed.
- Assets are infinitely divisible.
- Expected Returns, Variances and Covariances contain all information.
- Investors are risk-adverse, they will only accept greater risk if they are compensated with a higher expected return.

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

4.4 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 R of random variables and a portfolio weight vector w, respectively:

$$R = \begin{bmatrix} R_1 & R_2 & \cdots & R_N \end{bmatrix}, \quad w = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_N \end{bmatrix}$$

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

4.4.1 Expected Returns

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

$$\begin{aligned} E[R] &= \begin{bmatrix} E[R_1] & E[R_2] & \cdots & E[R_N] \end{bmatrix} \\ &= \begin{bmatrix} \mu_1 & \mu_2 & \cdots & \mu_N \end{bmatrix} = \mu \end{aligned}$$

and μ_i can be estimated in R using historical data and the formula for the geometric mean of returns (also called compound returns). The function to calculate the geometric mean of the returns from an xts object can be found in 4.5.3.

4.4.2 Expected Portfolio Returns

The following equation can be used to obtain the linear combination of expected returns μ and a weighting vector w (e.g. portfolio weights):

$$\begin{aligned} \mu \times w &= \begin{bmatrix} E[\mu_1] & E[\mu_2] & \cdots & E[\mu_N] \end{bmatrix} \times \begin{bmatrix} w_1 \\ w_2 \\ \cdots \\ w_N \end{bmatrix} \\ &= E[\mu_1] \cdot w_1 + E[\mu_2] \cdot w_2 + \cdots + E[\mu_N] \cdot w_N = \mu_P \end{aligned}$$

4.4.3 Covariance

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

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

and can be estimated in R with the basis function `cov()` and historical data.

4.4.4 Portfolio Variance

Let R be a random vector with N normally distributed elements and w a weight vector. Assuming that the covariance matrix $\sum$ of R is known, the variance of

the linear combination of R can be calculated as follows:

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

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

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

The same is true for an estimate of $\sum$.

4.4.5 Portfolio Returns

Suppose there are N assets forming a portfolio with weights w at time t_0, and the portfolio is to pass through several time steps until t_T without rebalancing. What are the portfolio returns at each time step t_i? Clearly, assets with positive performance in the current time step will have a higher weight in the next time step. This can be done by adjusting the weights after each time step depending on the returns. The formula for holding a portfolio with weights $\sum w = 1$ and return matrix $R \in \mathbb{R}^{T \times N}$, has return $Z_i - 1$ on t_i with $i = 0, 1, \cdots, T$ for:

$$Z_i = \begin{cases} (1 + R_i) \cdot w & \text{if } i = 0 \\ (1 + R_i) \cdot \frac{Z_{i-1}}{\sum Z_{i-1}} & \text{if } i > 0 \end{cases}$$

This calculation of portfolio returns is implemented in the function `calc_portfolio_returns()` below.

4.5 R-Functions

|||in progress|||

4.5.1 `pri_to_ret()`

|||in progress|||
((prices to returns))

4.5.2 `ret_to_cumret()`

|||in progress|||
((returns to cumulated returns normalized to 100))

4.5.3 `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^{id} can be calculated as:

$$r^{id} = ((1 + r_1) \cdot (1 + r_2) \cdot (1 + r_3))^{1/3} - 1 = 0.01996732$$

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

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

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

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

and as R code:

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

4.5.4 `calc_portfolio_returns()`

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

```
calc_portfolio_returns <-
  function(xts_returns, weights, name="portfolio"){
  if(sum(weights)!=1){
    xts_returns$temp___X1 <- 0
    weights <- c(weights, 1-sum(weights))
```

```
  }
  res <- cumprod((1+xts_returns)) * matrix(
    rep(weights, nrow(xts_returns)), ncol=length(weights), byrow=T)
  res <- xts(
    rowSums(res/c(1, rowSums(res[-nrow(xts_returns),])))-1,
    order.by=index(xts_returns)) %>%
    setNames(., name)
  return(res)
}
```

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

Chapter 5

Activ vs Passiv Investing

|||in progress|||
The fundation of Asset Management

passiv vs activ studie https://www.scirp.org/journal/paperinformation.aspx?paperid=92983

gut gut file:///C:/Users/Axel/Desktop/Master-Thesis-All/Ziel%20was%20beantwortet%20werden%20soll/Quellen%20nur%20wichtige/Rasmussen2003_Book_QuantitativePortfolioOptimisat.pdf

Chapter 6

Challenges of Passiv Investing

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

6.1 Mean-Variance Portfolio (MVP)

Markowitz showed that diversifying risk across multiple assets reduces overall portfolio risk. This result was the beginning of the widely used modern portfolio theory, which uses mathematical models to create portfolios with minimal variance for a given return target. All such optimal portfolios for a given return target are called efficient and constitute the efficient frontier.

6.1.1 MVP

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 covariance matrix of the returns is $\sum \in \mathbb{R}^{N \times N}$ and the expected returns are $\mu \in \mathbb{R}^N$. The MVP with the risk aversion parameter $\lambda \in [0, 1]$, as shown in (Maringer, 2005), can be formalized as follows:

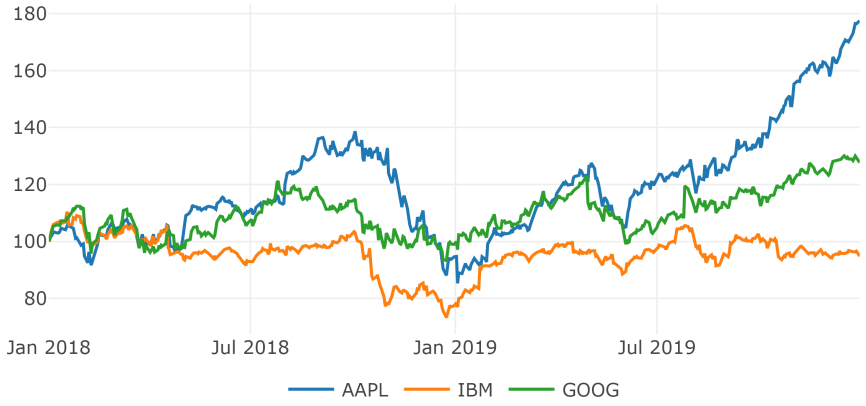
$$\underset{w}{minimize} \quad \lambda \, w^T \sum w - (1 - \lambda) \, \mu^T w \tag{6.1}$$

The risk aversion parameter λ defines the tradeoff between risk and return. With $\lambda = 1$, the minimization problem contains only the variance term, leading to a minimum variance portfolio, and $\lambda = 0$ transforms the problem into a minimization of negative expected returns, leading to a maximum return portfolio. All possible portfolios created by $\lambda \in [0, 1]$ define the efficient frontier.

6.1.2 MVP example

All possible MVPs together define the efficiency 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 solution of the MVP for each λ. First, the daily returns of these three assets from 2018-01-01 to 2019-12-31 are loaded.

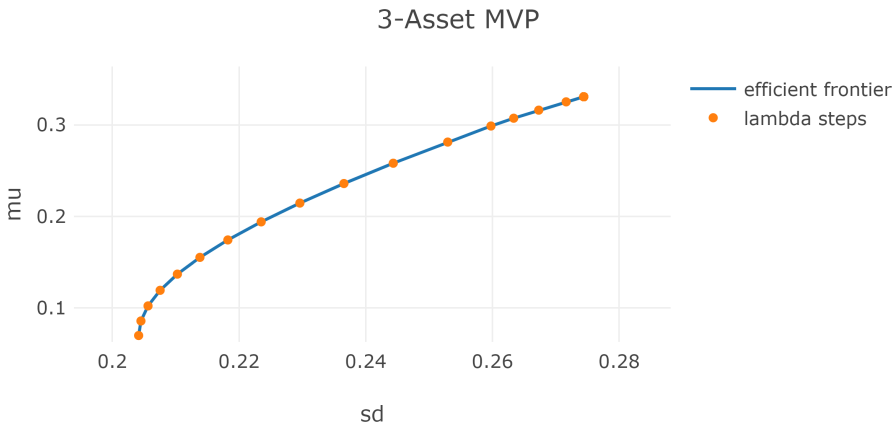
The cumulative daily returns are:



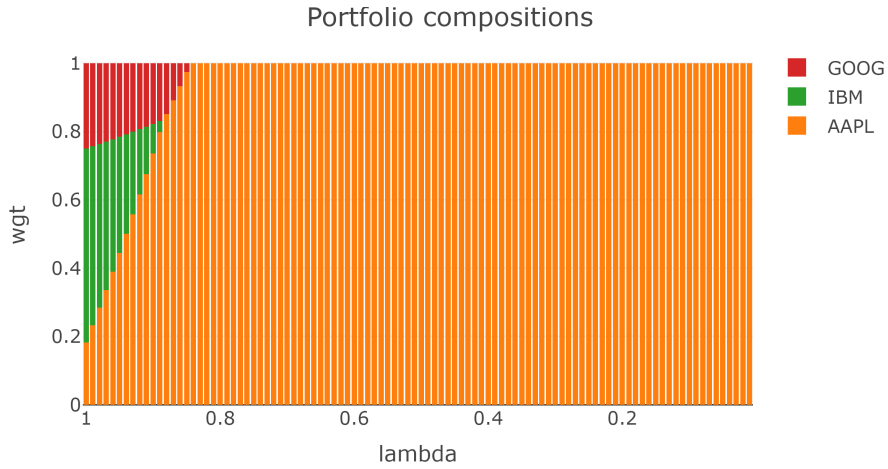
The expected daily returns and the covariance matrix for the three assets can be estimated using the formulas from chapter 4:

This is all the data necessary to solve the MVP 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" constraint, where the weights should sum to 1.

The resulting daily returns and standard deviations are converted to annual returns and standard deviations and plotted to create the efficiency frontier:



The portfolio compositions for each λ are:



6.2 Index-Tracking Portfolio (ITP)

Indices are baskets of assets that are used to track the performance of a particular asset group. For example, the well-known Standard and Poor's 500 Index (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, 2016). 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 is exactly equal to the performance of the benchmark, neglecting transaction costs. 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 reweight 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 this end, 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. Now is the time to optimize the portfolio to match the benchmark performance, taking into account investor constraints such as ratings, investment sectors, and others. Typically, this is accomplished by reducing the variance between portfolio and benchmark returns:

$$minimize \;\; Var(r_p - r_{bm})$$

To obtain the portfolio weights w, one needs to substitute r_p as follows:

$$r_p = R \times w$$

The variance is then solved until a quadratic problem is presented as a function of portfolio weights w:

$$\begin{aligned} Var(r_p - r_{bm}) &= Var(R \times w - r_{bm}) \\ &= Var(R \times w) + Var(r_{bm}) - 2 \cdot Cov(R \times w, r_{bm}) \end{aligned}$$

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

$$Var(r_{bm}) = \sigma_{bm}^2 = constant$$

The variance of the portfolio can be solved with 4.4.4:

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

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

$$\begin{aligned} Cov(A \times a, b) &= Cov(b, A \times a) \\ &= E[(b - \mu_b)(A \times a - \mu_A \times a)] \\ &= E[(b - \mu_b)(A - \mu_A) \times a] \\ &= E[(b - \mu_b)(A - \mu_A)] \times a \\ &= Cov(A, b) \times a \end{aligned}$$

This results in the final formula of the ITP:

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

The minimization problem of the ITP in the general structure required by many optimizers is:

$$min(\frac{1}{2} \cdot b^T \times D \times b - d^T \times b)$$

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 as follows:

$$D = Cov(R)$$

and

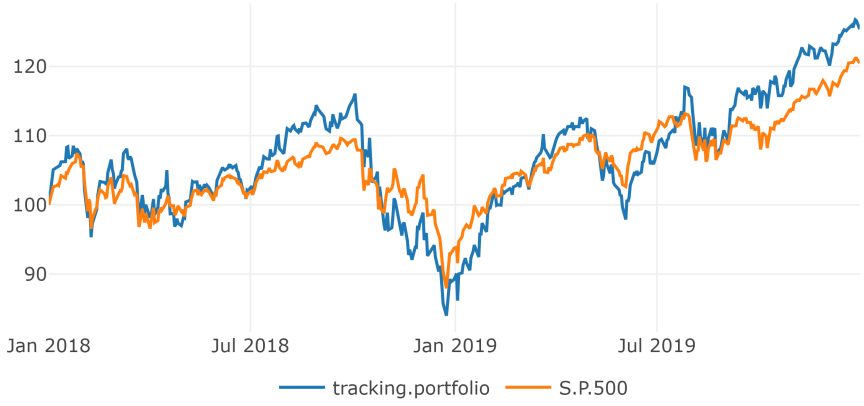
$$d = Cov(r_{bm}, R)$$

It is possible to add some basic constraints, as in the MVP to sum the weights to 1 and be long only.

6.2.1 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. The time frame is 2018-01-01/2019-12-31 and the goal is to minimize the variance of the difference in returns between the portfolio and the benchmark.

```
##       AAPL       IBM      GOOG
## 0.2594763 0.4164918 0.3240319
```


120
110
100
90
Jan 2018
Jul 2018
Jan 2019
Jul 2019
tracking.portfolio
S.P.500

Chapter 7

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 quadratic programming solution methods is to use them as a benchmark for PSO.

7.1 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:

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

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.

7.2 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 $x^T Dx > 0 \; \forall \; x \in \mathbb{R}^N$, 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 row of A is treated as an equality constraint

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

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

7.3 Example: Solving MVP with `solve.QP()`

This section provides insights into the effects of diversification and the use of `solve.QP` by creating ten different efficiency frontiers from a pool of ten assets.

Each efficiency 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 with constraints $\sum w = 1$ and $w \geq 0$, which yields the key features `mu`, `var` and `composition` of the portfolio.

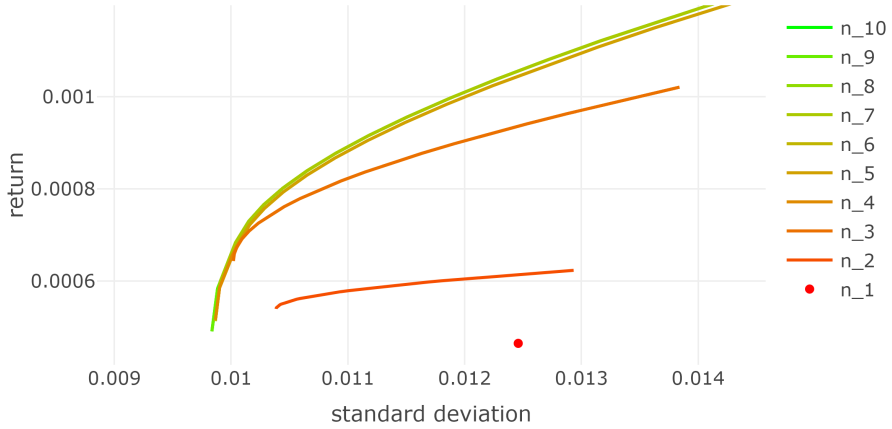
```
mvp <- function(returns, lambda){
  tc <- tryCatch({
    mu <- ret_to_geomeanret(returns)

    cov <- as.matrix(nearPD(cov(returns))$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.

7.4 Example: Solving ITP with `solve.QP()`

This example analyzes how many assets are needed to minimize the variance between the replication and historical returns of the S&P 500 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 serve as the new asset pool for the next replication until only five assets are left. 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 S&P 500. 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 S&P 500:

```
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 = "%5EGSPC", from = from, to = to)$returns,
  "AS_sp500"
) %>% setNames(., "S&P 500")
```

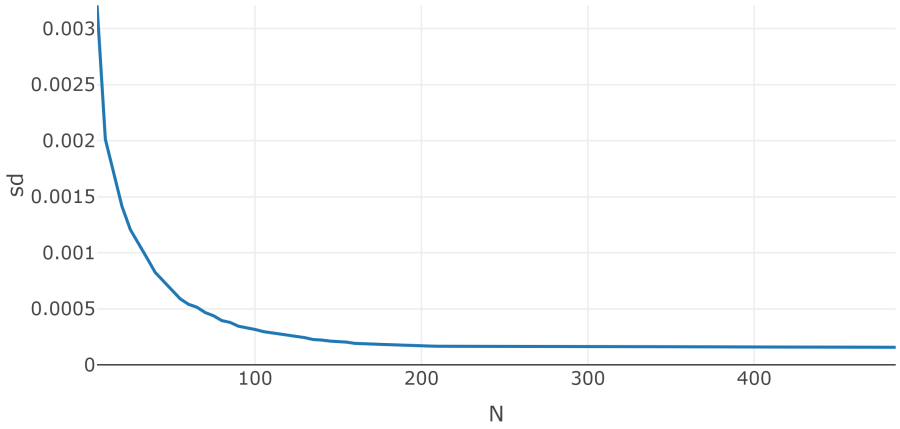
The required data is now available and the function for the ITP can be created. It requires `pool_returns` with variable number of columns and the single-column matrix `bm_returns`.

```
itp <- function(pool_returns, bm_returns){
  mat <- list(
    Dmat = cov(pool_returns),
    dvec = cov(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 duplication and successive discarding of assets can begin. The results are stored in `res` and used to display the results.

```
res <- 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{
    itp(pool_returns_raw[, names(sort(temp$solution, decreasing = T)[1:i])], bm_returns)
  }
  res <- rbind(res, data.frame("N"=i, "var"=temp$var, "sd"=sqrt(temp$var), row.names = NULL)
}
```



It can be seen that the standard deviation stagnates at about $N = 100$. This leads to the conclusion that a sparse replication with one hundred assets is sufficient in this particular case to track the historical performance of the S&P 500 over this period.

Chapter 8

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). Its advantages over other global optimization methods are that it does not require gradient information. 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.

8.1 The Algorithm

Each particle d with position x_d moves in the search space 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):

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

Where r_1 and r_2 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:

$$V^{i+1} = w \cdot V^i + c_1 \cdot r_1^i \cdot (P^i - X^i) + c_2 \cdot r_2^i \cdot (p_g^i - X^i)$$
$$X^{i+1} = X^i + V^{i+1}$$

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, c_g, r_1 and r_2 are stile scalars.

8.2 `pso()` Function

In this section, a general PSO function is created that follows the structure of other optimization heuristics in R, in particular the existing PSO implementation from the R package `pso`. At the center of it all is an objective function `fn()` that returns a scalar to be minimized. The function itself mainly needs a vector `pos` that describes the position of a particle (e.g. weights). The other main parameters for the PSO function are `par`, which is a position of a particle used to derive the dimension of the problem and as the first position of a particle. The argument can only contain `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 input 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)
)/4
```

The velocities are compressed by a factor of 4 to start with a maximum movement of one quarter 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 <- which.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, they are pushed back to the edge and the velocities are 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 * runif(1) * (P-X) +
    control$c.g * runif(1) * (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)
  }

  if(control$save_traces==TRUE){
    trace_data <- rbind(trace_data, data.frame("iter"=i, t(X)))
  }
}
```

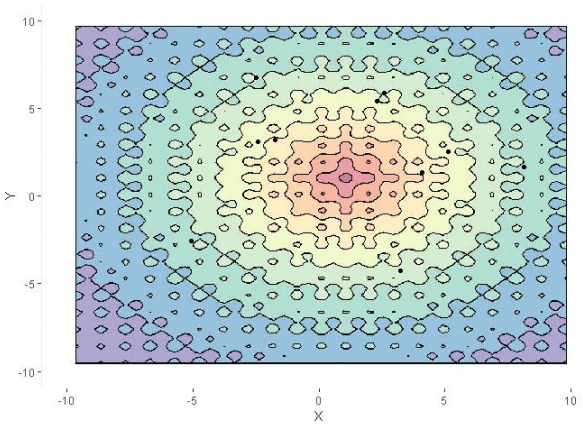
The best fitness after 100 iterations is 0.0000048 and the best possible solution is 0.

8.3 Animation 2-Dimensional

This section provides insights into the behavior of the PSO by visualizing multiple iterations in a GIF. The GIF only works in Adobe Acrobat DC or in the Markdown/HTML version of this paper. The amazing animation template is inspired by R'tichoke. The PSO core from the above chapter was used to complete the `pso()` function and is tested here with seed 0. The function `fn` to be evaluated can be found in R'tichoke.

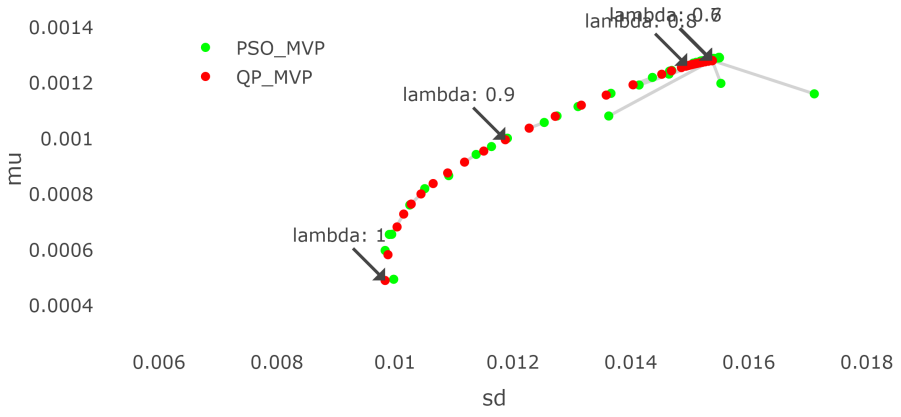
```
set.seed(0)

fn <- 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 = fn,
  lower = -10,
  upper = 10,
  control = list(
    s = 10,
    maxiter = 30,
    w0 = 0.8,
    save_traces = T
  )
)
```

The function `fn` 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:



8.4 Example MVP

This example uses the `solve.QP` approach from 7.3 with ten assets as the benchmark. Briefly, the goal is to create an MVP from ten of the largest U.S. stocks between 2018-01-01 and 2019-12-31 for each possible λ. The PSO has 300 particles and 200 iterations for each lambda. The main characteristics of all portfolios created with the `solve.QP` compared to the PSO are shown below:



The dots for each λ are connected with a grey line to visualize the error of the PSO. It can be seen that its possible to solve MVP's with a PSO approach.

8.5 Example ITP

The same ITP solved with `solve.QP()` in 7.4 is used as the benchmark for the PSO. In summary, the goal is to create a portfolio that minimizes the variance of the returns of itself and the S&P 500 between 2016 and 2021. The pool of assets includes all assets that are present in 2021 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 0.9, the upper bound is 0.1, and a starting position is the zero vector. 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_QP	0.0002	-0.0000444	0	0.3
ITP_PSO_best	0.0018	-0.0000429	0	24.4
ITP_PSO_mean	0.0019	-0.0000427	0	25.2

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

8.6 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 x is to be a integer vector. These 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.

8.7 Functions

Bibliography

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

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.

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.*

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