Financial Optimisation with R

(formerly a.k.a. the *NMOF Manual*)

Package version 2.7-0

Enrico Schumann

`es@enricoschumann.net`

19 August 2022

Contents

Preliminaries

The book and the package

This book explains how to use the NMOF package. There are three parts: (i) Tutorials, (ii) Function Reference and (iii) Case Studies and Examples. Some of the examples are purely pedagogical; others will, I hope, be useful as references or recipes for actual applications, giving the manual a 'cookbook' character.

> The book is still a draft: Comments, corrections and suggestions are very welcome.

I refer to both Gilli, Maringer, and Schumann (2011) and Gilli, Maringer, and Schumann (2019) as GMS. (In fact, all the R code examples of the first edition work with the current package version.) When I use the abbreviation NMOF, I mean the R package of that name. The latest version of the package is always available from

`http://enricoschumann.net/R/packages/NMOF.`

The stable version of the package is available from CRAN. That version is updated once or twice per year. The Appendix of this book describes how to obtain and install the package.

The book does not explain how particular optimisation methods work, and neither does it discuss the actual practice of financial optimisation; for this you may want to consult GMS. (This is also the reason why few examples in this book use real data.[†]) To put it more clearly: this book is about solving optimisation models, not about solving actual problems. Models should never be confused with problems; but models can be useful for solving problems. This distinction is not pedantic and, unfortunately, it appears not to be obvious. See Chapter 1 of GMS, or Gilli and Schumann (2010).

[†]Using artificial data has its advantages: we can scale datasets, e.g. use more assets or more 'observations'. And in any case, many useful and instructive results can already be illustrated with artificial data.

The book is written with Sweave (Leisch, 2002). It also makes use of the `weaver` package (Falcon, 2015). The tangled code is at

`http://enricoschumann.net/files/NMOFman.R`

The latest version of this book can be obtained from

`http://enricoschumann.net/NMOF.`

So, good luck, and, as the Perl folks might say, have the approriate amount of fun.

Tutorials

1. Choosing few from many – selecting a subset

All chapters will be self-contained and start with a clean workspace. We attach the package and we set a seed so to make the computations in this chapter reproducible.

```
> library("NMOF")
> set.seed(123321)
```

Code examples will always be typeset like this: in a typewriter font, with the leading > representing R's input prompt.

1.1. The problem

Suppose we were confronted with this conceptually simple problem:[†] we are given two ordered lists *a* and *b* that comprise random variates. Since this tutorial (and the whole book) is about using R, we can as well use correct terms, so we will speak of atomic vectors *a* and *b*. Each vector has a length of 100. If you like to think in financial terms, then suppose that *a* and *b* are return series.

[†]The problem is not as artificial as it might sound. A variant of it served to demonstrate that computing correlation to many digits is a waste. The original problem was: By removing up to 5 observations, out of about 250, how much can you increase the correlation between two series? How much can you decrease it?

We collect both vectors in a matrix *ab* of size 100×2. (It is more customary to use a single uppercase letter, such as *A*, to name a matrix; in mathematical notation, *ab* might otherwise be mistaken for the product of *a* and *b*. To avoid confusion, I shall from now on typeset `ab` in a typewriter font.)

The aim is to divide the rows of `ab` into two subsets in such a way that the correlation between the two columns is high in one subset and low in the other subset. Restrictions: (i) all data points must be used and (ii) a subset must comprise no fewer than 20 data points.

To make the idea clear, we create an example dataset and plot it. We define the number of rows in the dataset through a variable `nrows`; the minimum-row restriction is stored in `minrows`.

```
> nrows <- 100L
> minrows <- 20L
```

Note that I have added an `L` to the number. That is an indication for R that the number is an integer. (It is not necessary to do that; but it makes clearer that `nrows` is supposed to hold a whole number.)

For the example, the columns of ab should be correlated. First, we define a correlation rho and create a correlation matrix C. Actually, such a matrix construction is so common that we put it into a function const_cor.

```
> const_cor <- function(rho, n) {
      C <- array(rho, dim = c(n, n))
      diag(C) <- 1
      C
  }
```

```
> C <- const_cor(rho = 0.6, 2L)
```

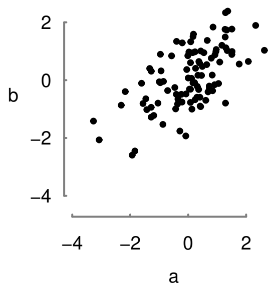
Next, we create ab. (See Chapter 7 of GMS for how to use the Cholesky factorisation to induce correlation into a random sample.)

```
> ab <- array(rnorm( nrows * 2L),
              dim = c(nrows, 2L)) %*% chol(C)
> colnames(ab) <- c("a", "b")
> head(ab, 5)
```

R's output will always be typeset in a typewriter font with a grey box around it.

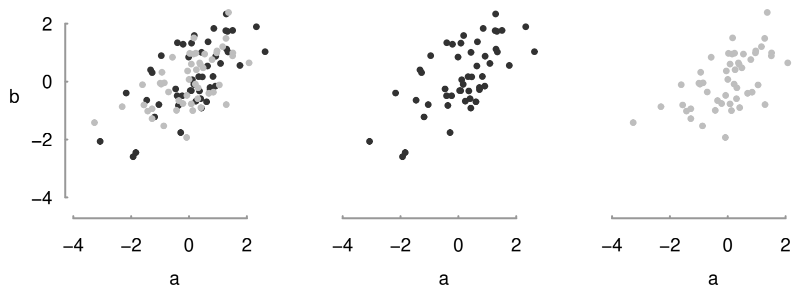
```
          a      b
[1,]  0.796  0.759
[2,]  0.967  1.059
[3,]  0.540  0.950
[4,]  1.288 -0.794
[5,] -1.277 -1.279
```

Let us look at the data.



One possible solution is to put the first 50 rows into subset 1 and the remaining rows into subset 2. This is probably a bad solution, but it does not violate the restrictions: we have used all rows and there are more than 20 data points in each subset.

We add the solution to the plot. The left panel shows all data points. Dark grey represents subset 1, which is also plotted in the middle figure. Light grey is used for subset 2, shown on the right. (Since we will later plot other solutions, we create a function `plot_subsets` for the figure. The function definition is omitted here, but is available in the source code.)



To have a more precise measure of the quality of this solution, we compute the correlations.

```
> cor(ab[ 1: 50, ])[1,2]
```

```
[1] 0.574
```

```
> cor(ab[51:100, ])[1,2]
```

```
[1] 0.654
```

They do not differ too much: the solution is – as we guessed – not very good. So let us find better solutions.

1.2. Solutions and their quality

Before we do can anything, we need to decide how to represent a solution;[†] we also need a mechanism for evaluating how good a solution is.

[†]Actually, not only the solution but also the data that are needed to evaluate a solution. Appropriate data structures can make quite a difference when it comes to performance, but also when it comes to clarity of the code.

1.2.1. Representing solutions

A solution needs to specify a set of rows in `ab`. Once we know the rows that belong to subset 1, we automatically know the rows of subset 2. Hence, we can store a solution as a logical vector with a length equal to the number of rows in `ab`. `TRUE` indicates subset 1; `FALSE` indicates subset 2.

The 50–50 solution, i.e. putting the first 50 rows into subset 1 and the remaining rows into subset 2, could have been created in this way:

```
> x0 <- rep(c(TRUE, FALSE), each = nrows/2)
```

We use the symbol x0 for an initial solution. (If nrows is odd, add length.out = nrows as an argument.)

An alternative starting point could be be a random solution:

```
> x0 <- runif(nrows) > 0.5
```

For such a random solution, we need to check the constraints:

```
> all(table(x0) >= minrows)
```

```
[1] TRUE
```

To extract subset 1 from ab, we just type:

```
> ab[ x0, ]
```

For subset 2, we take not subset 1.

```
> ab[!x0, ]
```

Now that we have a way to encode solutions, we can compute with them.

1.2.2. Solution quality: the objective function

We need a way to see how good or bad a solution is; that is, we need an objective function. The objective function takes as input a solution (and possibly some other data) and maps that solution into a real number. We use the convention that we always minimise, so a lower number is better.

The fact that we want to maximise the absolute difference between the correlations is not a problem: we just put a minus in front of the absolute difference. Clearly, the best possible solution corresponds to a numeric value of -2; the worst one has a value of 0. We put this computation into the function dcor.

```
> dcor <- function(x, ab)
      -abs(cor(ab[ x, ])[1L, 2L] - cor(ab[!x, ])[1L, 2L])
```

Now we would like to find a vector x that makes dcor small. We can test the function with our initial solution.

```
> x0 <- rep(c(TRUE, FALSE), each = nrows/2)
> dcor( x0, ab)
```

```
[1] -0.0803
```

```
> dcor(!x0, ab) ## should give the same result
```

```
[1] -0.0803
```

1.3. Strategy 1: Zero-intelligence solutions

1.3.1. Brute force

Ken Thompson is said to have suggested that 'When in doubt, use brute force'. The corresponding search strategy is to randomly choose subsets. Random sampling has several advantages. It is simple; it benefits from more computing power; and it can be distributed. But random sampling is also the least-efficient method we can think of. (The least-efficient method among the class of search strategies that really aim to find good solutions. Clearly, if we wanted to be bad on purpose, we could easily be less efficient.[†]

[†] The archetypical example of an perversely-awful algorithm is probably bogo-sort `http://www.catb.org/jargon/html/B` (as opposed to bubble sort, which is merely the generic bad algorithm).

Let us compute a number of random solutions. We store the objective function values in a vector `OFvalues` and the solutions in a list `solutions`.

First, we create a function that returns a random solution.

```
> random_x <- function(nrows, min) {
      c1 <- sample(min:(nrows - min), 1L)  ## cardinality of subset 1
      x0 <- logical(nrows)
      x0[sample.int(nrows, c1)] <- TRUE
      x0
  }
```

We wrap this function in a loop.

```
> trials <- 1e5
> OFvalues <- numeric(trials)
> solutions <- vector("list", trials)
> for (i in seq_len(trials)) {
      x0  <- random_x(nrows, min = minrows)
      OFvalues[i] <- dcor(x0, ab)
      solutions[[i]] <- x0
  }
```

We can summarise the results.

```
> summary(OFvalues)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 -0.751  -0.142  -0.083  -0.099  -0.039   0.000
```

Exercise 1.1 *Rewrite the sampling procedure so that no loop is required; start by vectorising `random_x`. Measure the performance difference.*

The best solution is the one with the lowest objective function value.

```
> xbest <- which.min(OFvalues)
> OFvalues[xbest]
```

```
[1] -0.751
```

Thus, the greatest absolute difference between the correlations in the subsets is -0.751. The corresponding solution is stored in `solutions[[xbest]]`. We store this solution as variable `xRandom` so that we can later compare it with other solutions.

```
> xRandom <- solutions[[xbest]]
```

(There is no need to store the corresponding objective-function value: we can always recover it with `dcor(xRandom, ab)`.)

1.3.2. Grid Search

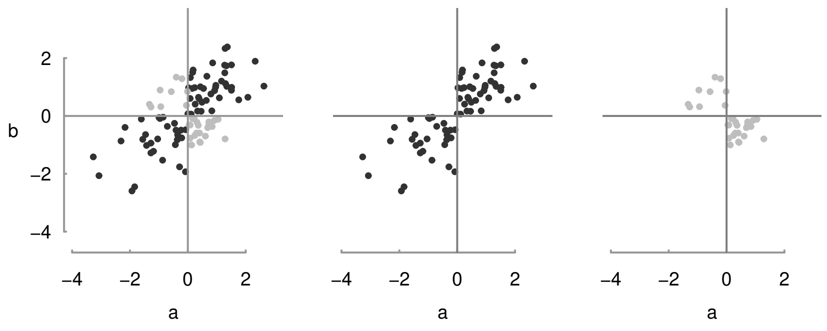
An alternative type of brute force is a grid search. For this particular problem, a grid search is not appropriate. See Chapter 2 for an example.

1.4. Strategy 2: A constructive solution

For the given problem, we know a and b are roughly zero-mean vectors (we know that because we created them so). We can divide them into two subsets like this: those data points i for which $a_i b_i$ is smaller than zero go into one subset, and those for which $a_i b_i$ is greater than zero go into the other.

```
> subset1 <- ab[ ,1L] * ab[ ,2L] >  0
> subset2 <- ab[ ,1L] * ab[ ,2L] <= 0
```

To make clear why such a solution strategy is reasonable, we can look at the corresponding plot.



Note that this is not a general strategy. It only works for this specific problem, and there is no guarantee that the constraints are satisfied. But nevertheless, for many problems we can use knowledge about the problem to come up with good solutions. Such a solution strategy is called constructive since we build – we construct – one single solution. Once we have this solution, we are done.

For a practical financial example of a constructive strategy, see Schumann (2013).

But back to our example.

```
> dcor(subset1, ab)
```

```
[1] -1.39
```

That's not bad – definitely better than random. But are the constraints violated?

```
> sum(subset1)
```

```
[1] 70
```

```
> sum(subset2)
```

```
[1] 30
```

For the chosen seed for the random-number generator, we were lucky. But suppose the solution had violated the restrictions. A variation of the construction mechanism is to sort the rows of ab by the size of $a_i b_i$.

```
> cr <- order(ab[ ,1L] * ab[ ,2L])
> OFvalues <- numeric(nrows)
> for (i in minrows:(nrows - minrows)) {
      x0 <- logical(nrows)
      x0[cr[seq_len(i)]] <- TRUE
      OFvalues[i] <- dcor(x0, ab)
  }
```

Now we can check the minimum of OFvalues.

```
> cutoff <- which.min(OFvalues)
> subset1 <- logical(nrows)
> subset1[cr[seq_len(nrows) <= cutoff]] <- TRUE
> subset2 <- !subset1
> dcor(subset1, ab)
```

```
[1] -1.42
```

For this particular dataset, that solution is even slightly better. We store the solution in the vector xConstr so that we can also compare this solutions with others.

```
> xConstr <- subset1
```

1.5. Strategy 3: A Greedy Search

Now we move to a fundamental strategy of numerical optimisation: we take a given solution and try to improve it iteratively.

Define a single change in a solution as a move in which one data point changes its subset, i.e., we pick one row and assign it to the other subset. There can be at most 100 such moves; there may be fewer, namely if a constraint is binding. We will call a solution created by such a move a neighbour to the original solution.

```
TRUE FALSE FALSE FALSE FALSE TRUE ... ## original solution

TRUE FALSE TRUE  FALSE FALSE TRUE ... ## neighbour solution
```

The strategy we will test in this section is called a Greedy Search and it works as follows. Start with a random solution and call it the current solution. Look at all its neighbours computed through single changes and compute their objective function values. Select the best neighbour; if that best neighbour is better than the current solution, make it the current solution.

If you have found a better solution in that way (ie, if we replaced the current solution), start again: look at all the neighbours of this new current solution, and so on. Repeat until there is no better solution.

We implement this approach in the function `greedy`. (Note that we could also write a more generic function for Greedy Search, but only at the price of making the function less understandable.)

```
> greedy <- function(fun, x0, ab, n, nmin, maxit = 1000L) {
      done <- FALSE
      xbest <- xc <- x0
      xbestF <- xcF <- fun(xbest, ab)
      ic <- 0

      while (!done) {
          if (ic > maxit)
              break
          else
              ic <- ic + 1L

          done <- TRUE
          xc <- xbest
          for (i in seq_len(n)) {

            ## create a new solution
            xn <- xc
            xn[i] <- !xn[i]

            ## check constraints
            sxn <- sum(xn)
            enough <- sxn >= nmin
            notTooMany <- sxn <= n - nmin

            if (enough && notTooMany) {
              xnF <- fun(xn, ab)
              if (xnF < xbestF) {
                xbest <- xn
                xbestF <- xnF
                done <- FALSE
              }
            }
          }
      }
      list(xbest = xbest, OFvalue = xbestF, ic = ic)
  }
```

greedy takes several arguments: an objective function fun, an initial solution x0 and the data. The search essentially is a while loop; thus, we also pass a number maxit that breaks the computation after a maximum number of steps.

```
> x0 <- random_x(nrows = nrows, min = minrows)
> result <- greedy(fun = dcor, x0 = x0, ab = ab,
                   n = nrows, nmin = minrows, maxit = 1000L)
> xGreedy <- result$xbest
> dcor(x0, ab)
```

```
[1] -0.142
```

```
> dcor(xGreedy, ab)
```

```
[1] -1.54
```

```
> result$OFvalue
```

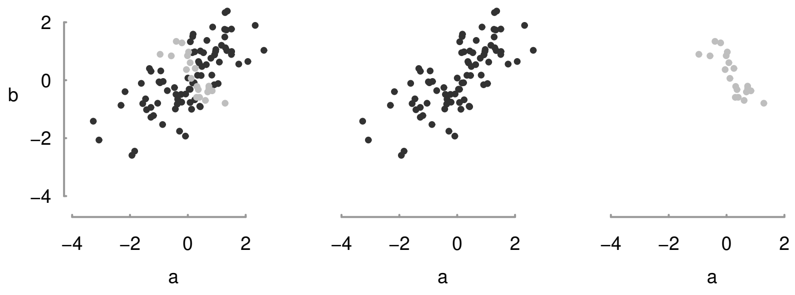
```
[1] -1.54
```

The function also returns number of moves it has made.

```
> result$ic
```

```
[1] 55
```

A plot of the resulting split.



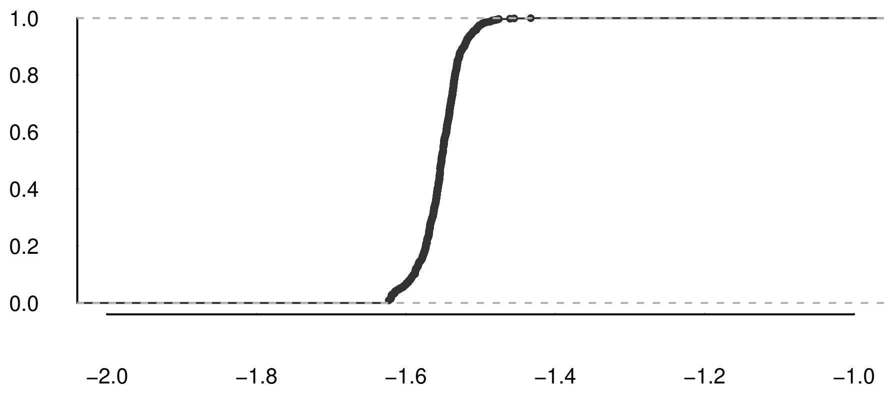
Seeing this result should immediately raise a question: if we choose a different starting value, do we get a different result? Let us try 1000 randomly-chosen starting values.

```
> trials <- 1000L
> OFvalues <- numeric(trials)
> solutions <- vector("list", trials)
> moves <- numeric(trials)
> for (i in seq_len(trials)) {

      x0 <- random_x(nrows = nrows, min = minrows)
      result <- greedy(fun = dcor, x0 = x0, ab = ab,
                       n = nrows, nmin = minrows, maxit = 1000L)

      OFvalues[i] <- result$OFvalue
      solutions[[i]] <- result$xbest
      moves[i] <- result$ic
  }
```

The results. Yes, the starting value matters.

```
> summary(OFvalues)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
  -1.62   -1.57   -1.55   -1.55   -1.54   -1.43 
```



It should not come as a surprise that these results are much better than the random solutions provided before. We have used much more knowledge about the problem, and the computation is actually quite expensive. We can check how many moves the algorithm made before it stopped.

```
> summary(moves)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   21.0    34.0    44.0    45.5    56.0    78.0
```

Since we test 100 neighbours in each iteration, one run of `greedy` creates and evaluates on average 4400 solutions.

```
> ## xbest <- which.min(OFvalues)
> ## OFvalues[xbest]
> ## xGreedy <- solutions[[xbest]]
```

1.6. Strategy 4: Local Search

Now we are going to move to a much simpler strategy, Local Search. See GMS, Chapter 11. As we did for the greedy search, we start a Local Search with a random solution. Then we create a neighbour, but this time we do not use any knowledge about the particular problem; rather, we change the solution randomly: pick one row at random and assign it to the other subset.

We will not implement Local Search, but use the function `LSopt` provided by the NMOF package.

We first collect all data in a list which we call `Data`. That way we keep all pieces of information in one place; later on, we will pass `Data` to functions. As an aside: this collecting-all-in-a-list makes it less likely to forget to explicitly pass some objects. If we did so, Local Search would still be able to use such objects: the functions that require these objects – for instance, the objective function – were defined at the top-level, as are the objects. But this is bad practice, which will not work, for instance, with functions defined in packages, and it will also get us into trouble when we distribute computations.

If you do not want to pass arguments to functions, see the appendix of this chapter for an alternative way, using closures.

```
> Data <- list(ab = ab, nrows = nrows, nmin = minrows)
```

Recall that `nrows` was the number of rows in `ab`; the variable `nmin` stores the minimum number of rows per subset. We start with a random initial solution.

```
> x0 <- random_x(nrows = nrows, min = minrows)
```

We define a neighbourhood function that chooses one data point randomly, then reassigns it to the other subset. Since we use a logical vector to represent a solution, we only need to switch the chosen elements, using the `!`-operator.

```
> neighbour <- function(xc, Data) {
      xn <- xc

      p <- sample.int(Data$nrows, size = 1L)
      xn[p] <- !xn[p]

      sxn <- sum(xn)
      enough <- sxn >= Data$nmin
      notTooMany <- sxn <= (Data$nrows - Data$nmin)

      if (enough && notTooMany)
          xn
      else
          xc
  }
```

The function also checks the constraints. If a constraint is violated, we reject the new solution and keep the old one. (See GMS, Section 12.5, for a general discussion of constraint handling.)

As a check: if we compare the neighbour solution with the original solution, they should only differ in one place, so an element-wise comparison should yield a single `FALSE` and $n - 1$ `TRUE` values:

```
> table(x0 == neighbour(x0, Data))
```

```
FALSE  TRUE
    1    99
```

We rewrite the objective function since now we pass `Data`.

```
> dcor <- function(x, Data)
      -abs(cor(Data$ab[x, ])[1L, 2L] - cor(Data$ab[!x, ])[1L, 2L])
```

We check the new function.

```
> dcor(x0, Data)
```

```
[1] -0.0356
```

```
> dcor(neighbour(x0, Data), Data)
```

```
[1] -0.0545
```

These two functions, dcor and neighbour, is all we need for LSopt. So we run a Local Search and check the solution.

```
> algo <- list(nS = 2500L,              ## number of steps to make
               neighbour = neighbour,  ## neighbourhood function
               x0 = x0,                ## initial solution
               printBar = FALSE)
> sol1 <- LSopt(dcor, algo = algo, Data = Data)
```

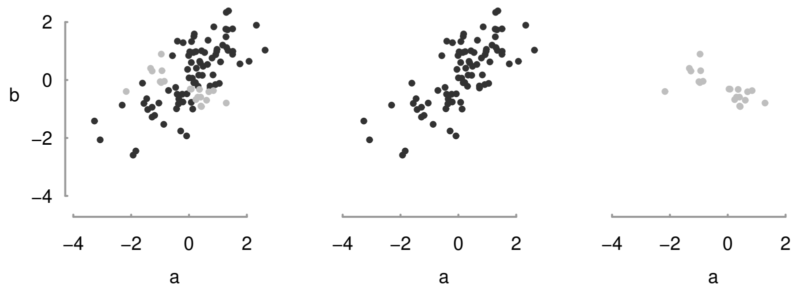
```
Local Search.
Initial solution:  -0.0356
Finished.
Best solution overall: -1.41
```

```
> sol1$OFvalue
```

```
[1] -1.41
```

```
> xLS <- sol1$xbest
```

Recall that the quality (i.e. objective function value) for the solution returned by the constructive method was -1.423. We plot the results.



The result of a single Local Search is random since we have chosen a random starting value and the moves that the Local Search takes are random as well. (Note that both sources of randomness could be eliminated by choosing a fixed starting solution and defining a neighbourhood function that does not involve chance. But the point is to show that chance is actually our friend.) How can we judge the results when it is random? By running a small experiment.

First we create a variation of our random-solution generator that takes no arguments.

```
> random_x_no_par <- function() {
      min <- 20
      nrows <- 100
      c1 <- sample(min:(nrows - min), 1L)
      x0 <- logical(nrows)
      x0[sample.int(nrows, c1)] <- TRUE
      x0
  }
```

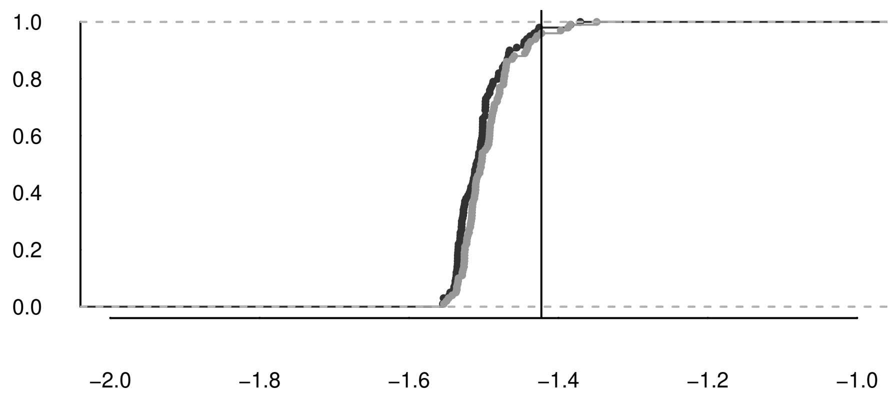
That is useful because the argument x0 that we give to LSopt may also be a function.

We run LSopt 100 times for two different settings. First, we use 5000 steps; then 10000. For convenience, NMOF provides a function restartOpt which runs these tests and collects the values.

```
> trials <- 100L
> algo <- list(nS = 5000L,              ## number of steps to make
               neighbour = neighbour,  ## neighbourhood function
               x0 = random_x_no_par,   ## initial solution
               printBar = FALSE,
               printDetail = FALSE)
> restarts1 <- restartOpt(LSopt, trials,
                          OF = dcor, algo = algo, Data,
                          cl = 4)
> restarts1OFvalues <- sapply(restarts1, `[[`, "OFvalue")
> algo$nS <- 10000L
> restarts2 <- restartOpt(LSopt, trials,
                          OF = dcor, algo = algo, Data,
                          cl = 4)
> restarts2OFvalues <- sapply(restarts2, `[[`, "OFvalue")
```

We can plot the results. The vertical lines indicates the constructive solution. Note that there is little improvement despite the doubling of the number of steps nS.

```
> par(bty = "n", las = 1, mar = c(3, 4, 0, 0), ps = 8, tck = 0.001)
> plot( ecdf(restarts1OFvalues), main = "", ylab = "", xlab = "",
       cex = 0.4, pch = 19, col = grey(.2), xlim = c(-2,-1))
> lines(ecdf(restarts2OFvalues),
       cex = 0.4, pch = 19, col = grey(.6))
> abline(v = dcor(xConstr, Data))
```



Exercise 1.2 *Take a random walk through your data like in a Local Search, but accept any new solution; store the best value along the way. Do this 100 times. What does the distribution of these best values look like?*

Now change the acceptance rule: if better, always accept. If worse, accept with probability of 50%. Again, run 100 such experiments.

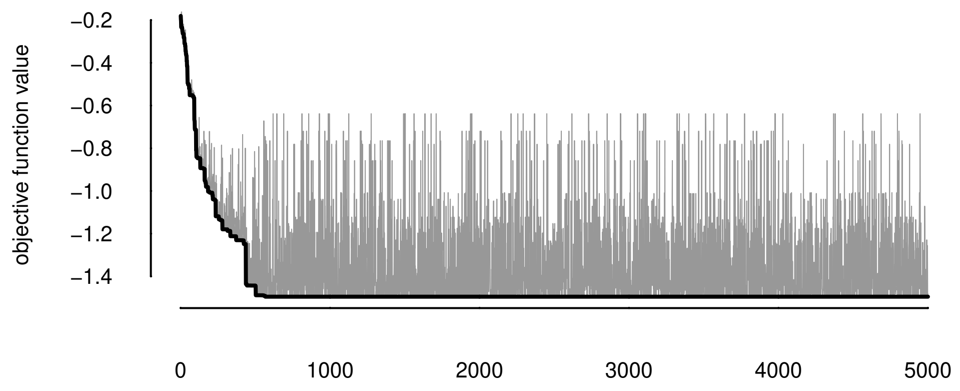
1.7. Strategy 5: A less-greedy Local Search

Now we use Threshold Accepting and Simulated Annealing instead of a Local Search. We use 10 thresholds.

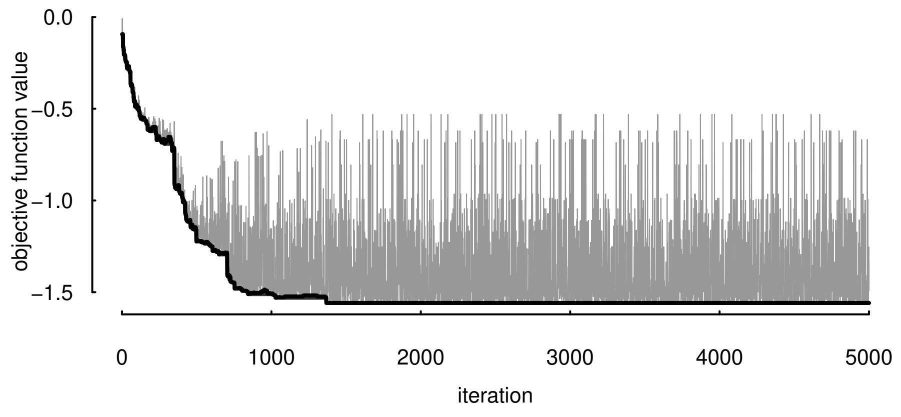
```
> x0 <- random_x(nrows = nrows, min = minrows)
> algo$nS <- 5000
> sol1 <- LSopt(dcor, algo = algo, Data = Data)
> sol1$OFvalue
> xLS <- sol1$xbest
> algo$nT <- 10
> algo$nS <- 500
> sol2 <- TAopt(dcor, algo = algo, Data = Data)
> sol2$OFvalue
> xTA <- sol2$xbest
```

We see that TA finds a better solution. Let us look at the solutions that the algorithms find over time. First LS.The dark black line shows the objective function value of the accepted solutions, which is equivalent, for LS, to the best solution found. The grey line shows the proposed solutions.

```
> par(mfrow = c(1, 1),
      bty = "n", las = 1, mar = c(3, 4, 0, 0), ps = 8, tck = 0.001)
> plot(sol1$Fmat[,1], type = "l", col = grey(.6),
       ylab = "objective function value", xlab = "iteration", lwd = 0.5)
> lines(sol1$Fmat[,2], type = "l", lwd = 2)
```



Here is TA. If you look carefully, you will see that the accepted-solutions' objective function value is not monotonically decreasing: TA may make uphill moves.

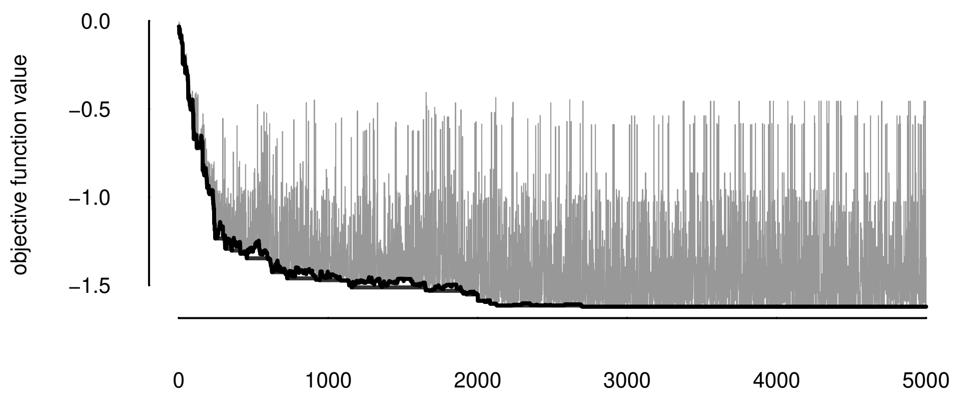


We can tell TA be even more forgiving by changing the parameter q, which is the size of the initial threshold. A value of 0.9 means that the first threshold would accept 90% of all changes.

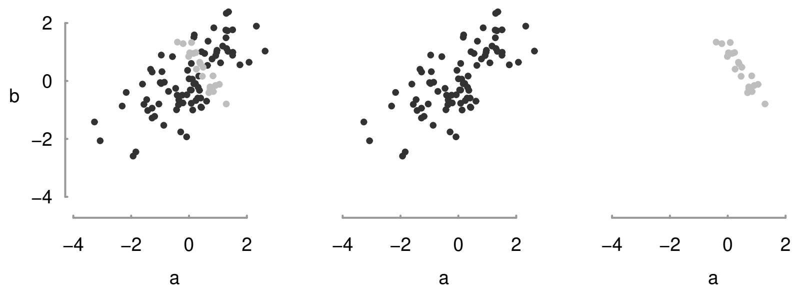
```
> algo$q <- 0.9
> sol2 <- TAopt(dcor, algo = algo, Data = Data)
> (sol2$OFvalue)
```

```
[1] -1.62
```

```
> par(mfrow = c(1, 1),
      bty = "n", las = 1, mar = c(3, 4, 0, 0), ps = 8, tck = 0.001)
> plot(sol2$Fmat[,1], type = "l", col = grey(.6), lwd = 0.5,
       ylab = "objective function value", xlab = "iteration")
> lines(cummin(sol2$Fmat[,2]), type = "l", lwd = 2, col = grey(.2))
> lines(sol2$Fmat[,2], type = "l", lwd = 2)
```



We plot the results.

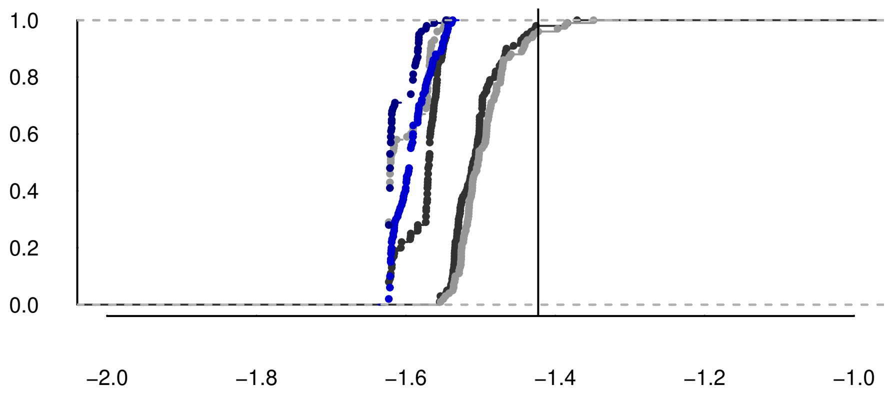


Just as with LSopt, we should start TAopt several times. Again, we test two different settings for the steps.

```
> algo$printBar <- FALSE
> algo$printDetail <- FALSE
> restarts3 <- restartOpt(TAopt, trials, OF = dcor, algo = algo, Data, cl = 4)
> restarts3OFvalues <- sapply(restarts3, `[[`, "OFvalue")
> algo$nS <- 1000
> restarts4 <- restartOpt(TAopt, trials, OF = dcor, algo = algo, Data, cl = 4)
> restarts4OFvalues <- sapply(restarts4, `[[`, "OFvalue")
```

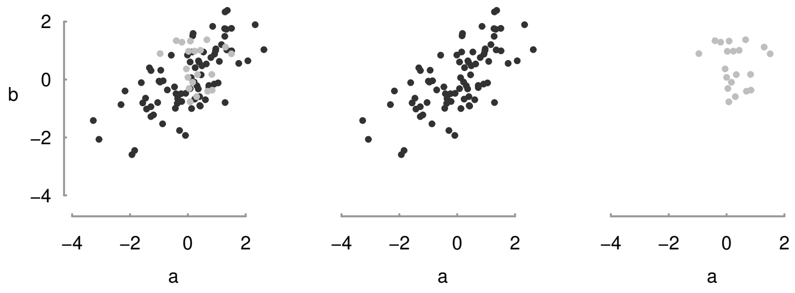
```
> algo$nS <- 500
> algo$printBar <- FALSE
> algo$printDetail <- FALSE
> restarts5 <- restartOpt(SAopt, trials, OF = dcor, algo = algo, Data, cl = 4)
> restarts5OFvalues <- sapply(restarts5, `[[`, "OFvalue")
> algo$nS <- 1000
> restarts6 <- restartOpt(SAopt, trials, OF = dcor, algo = algo, Data, cl = 4)
> restarts6OFvalues <- sapply(restarts6, `[[`, "OFvalue")
```

```
> par(bty = "n", las = 1, mar = c(3, 4, 0, 0), ps = 8, tck = 0.001)
> plot( ecdf(restarts1OFvalues), main = "", ylab = "", xlab = "",
       cex = 0.4, pch = 19, col = grey(.2), xlim = c(-2,-1))
> lines(ecdf(restarts20Fvalues),
       cex = 0.4, pch = 19, col = grey(.6))
> lines(ecdf(restarts30Fvalues),
       cex = 0.4, pch = 19, col = grey(.2), lty=2)
> lines(ecdf(restarts40Fvalues),
       cex = 0.4, pch = 19, col = grey(.6), lty=2)
> lines(ecdf(restarts50Fvalues),
       cex = 0.4, pch = 19, col = rgb(0,0,0.8), lty=2)
> lines(ecdf(restarts60Fvalues),
       cex = 0.4, pch = 19, col = rgb(0,0,0.5), lty=2)
> abline(v = dcor(xConstr, Data))
```

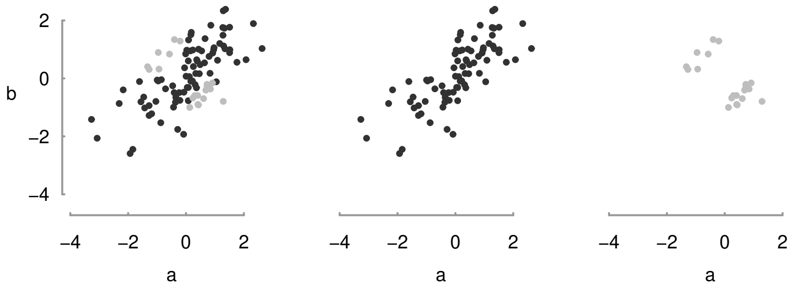


1.8. Comparing the results

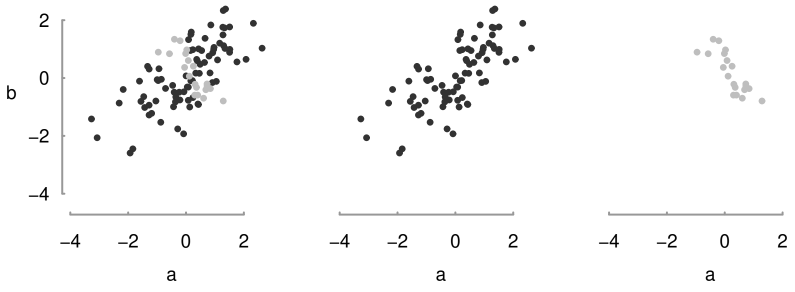
Random: -0.7508



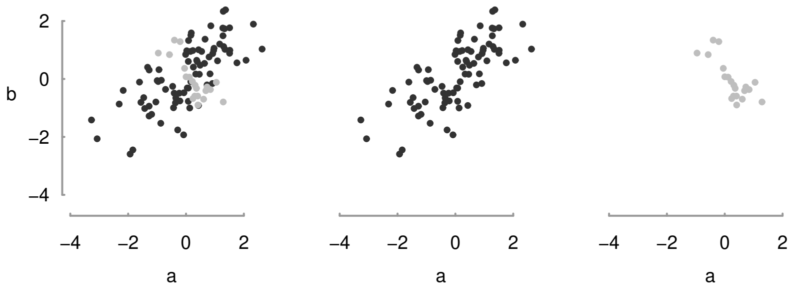
Constructive: -1.4228



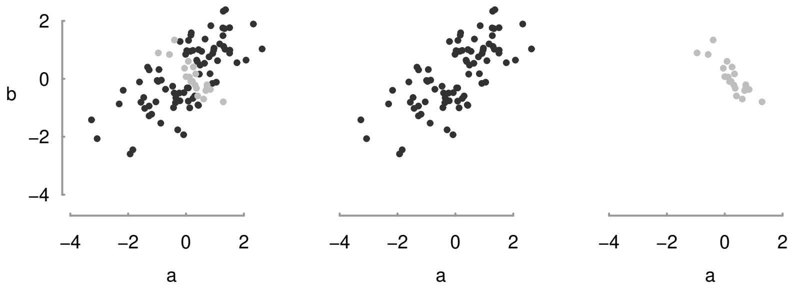
Greedy: -1.5379



Local Search: -1.4952



Threshold Accepting: -1.5588



1.A. Passing data via closures

Recall that we defined the neighbour function so that all information other than the solution was passed via a list Data. Suppose you do not like that: you want to write n instead of Data$n, and so on. You can do this with closures.

Let us make a simple example. First, as a reminder, the original function.

```
> neighbour <- function(xc, Data) {
      xn <- xc

      p <- sample.int(Data$nrows, size = Data$size)
      xn[p] <- !xn[p]

      sxn <- sum(xn)
      enough <- sxn >= Data$nmin
      notTooMany <- sxn <= (Data$nrows - Data$nmin)

      if (enough && notTooMany)
          xn
      else
          xc
  }
```

The new function neighbour_fun takes as arguments the variables that were in Data. The function returns a neighbourhood function.

```
> neighbour_fun <- function(n, nmin, size) {
      force(n)
      force(nmin)

      function(xc) {
          xn <- xc

          p <- sample.int(n, size = size)
          xn[p] <- !xn[p]

          sxn <- sum(xn)
          enough <- sxn >= nmin
          notTooMany <- sxn <= (n - nmin)

          if (enough && notTooMany)
              xn
          else
              xc
      }
  }
```

To create a neighbourhood, we call neighbour_fun.

```
> N <- neighbour_fun(n = 10, nmin = 2, size = 3)
> N
```

```
function(xc) {
        xn <- xc

        p <- sample.int(n, size = size)
        xn[p] <- !xn[p]

        sxn <- sum(xn)
        enough <- sxn >= nmin
        notTooMany <- sxn <= (n - nmin)

        if (enough && notTooMany)
            xn
        else
            xc
    }
<environment: 0x56517bbd7db8>
```

The function compareLogicals is described in Section 9.2.1.

```
> x0 <- rep(c(TRUE, FALSE), each = 5L)
> Data <- list(nrows = 10, nmin = 2, size = 1)
> compareLogicals(x0, neighbour(x0, Data))
```

```
1111100000
1111110000
     ^
```

```
> N <- neighbour_fun(n = 10, nmin = 2, size = 1)
> compareLogicals(x0, N(x0))
```

```
1111100000
0111100000
^
```

```
> compareLogicals(x0, N(x0))
```

```
1111100000
1011100000
 ^
```

```
> compareLogicals(x0, N(x0))
```

```
1111100000
1011100000
 ^
```

…and:

```
> N <- neighbour_fun(n = 10, nmin = 2, size = 3)
> compareLogicals(x0, N(x0))
```

```
1111100000
0110100010
^  ^    ^
```

```
> compareLogicals(x0, N(x0))
```

```
1111100000
1101100101
  ^    ^ ^
```

```
> compareLogicals(x0, N(x0))
```

```
1111100000
0011100001
^^       ^
```

2. Constrained regression

Again, we set a seed so that you can reproduce the results exactly. We will use the `rbenchmark` package (Kusnierczyk, 2010) for timing comparisons.

```
> library("NMOF")
> library("rbenchmark")
> set.seed(46457)
```

2.1. The problem

In this chapter we will see how to compute a constrained regression with Differential Evolution (DE). More specifically, we will run a regression that could be used in a style analysis.

Style analysis, described in Sharpe (1992), uses a linear factor model to describe the returns of a portfolio, with a few constraints added: the factors should be actual asset classes; the factor loadings, which are interpreted as weights, should sum to 100% and should take on reasonable values. For a typical mutual fund, for instance, the weights should be nonnegative.

Sharpe also stipulates that there is no constant in the model.

Sharpe uses quadratic programming (QP) to solve his model. But we suspect that some data points are not really reliable, so we want to run a Least-Trimmed-Squares (LTS) regression. That is, we want the algorithm to select *h* of the *n* observations, and only for those minimise the squared residuals.

2.2. Least Squares and Least Trimmed Squares

2.2.1. Data, solution representation and objective function

We create a dataset of *p* potential regressors and *n* observations. For this purpose we define the function `randomData`.

```
> randomData <- function(p, n, rscale = 0.5) {
      X <- array(rnorm(n * p), dim = c(n, p))
      k <- sample.int(p, 1L)    ## the number of regressors
      K <- sample.int(p, k)     ## the set of regressors
      betatrue <- numeric(p)
      betatrue[K] <- rnorm(k)   ## the true coefficients
```

```
    y <- X %*% betatrue + rnorm(n)*rscale
    list(X = X, y = y, betatrue = betatrue,
         K = K, n = n, p = p)
}
```

A dataset comprises a matrix X whose columns are the regressors and a single-column matrix y, the regressand.

We start with a modest dataset of $n = 60$ observations and $p = 5$ regressors.

```
> n <- 60L
> p <- 5L
> rD <- randomData(p, n)
```

As in Chapter 1, we need to decide how to represent a solution, and we need a way to evaluate its quality – the objective function. How solutions are changed is determined by the heuristic we use; Differential Evolution is essentially defined through these operations. We need, however, to discuss how to include constraints.

Let us start with how to represent the solution. Since we look for vector of coefficients, a numeric vector is natural. We create a random solution b0.

```
> b0 <- rnorm(p)
```

b0 will probably violate the constraints (non-negativity, likely; summing to one, definitely), but it is a solution that can be put into an objective function.

Since we will pass the different pieces of information (the data set, the number of observations n and so on) to our optimisation function, it will be convenient to collect them all in a list, which we call Data.

```
> Data <- list(X = rD$X,
               y = rD$y,
               p = rD$p,
               n = rD$n)
```

Now that we have some solution, we can evaluate its quality. We will start with the objective function for a simpler problem: Least Squares. It can be written like this.

```
> OFls <- function(b, Data) {
      tmp <- Data$y - Data$X %*% b
      sum(tmp * tmp)
  }
```

Instead of `tmp * tmp` we could have written `tmp^2`, which would not have been much different. But it makes a difference for higher exponents:

```
> tmp <- rnorm(1e4)
> benchmark(tmp * tmp * tmp,
            tmp^3,
            columns = c("test", "elapsed", "relative"),
            replications = 5000, order = "relative")
```

```
             test elapsed relative
1 tmp * tmp * tmp   0.226     1.00
2           tmp^3   1.501     6.64
```

The results are the same, but the computing time differs quite a bit.

```
> all.equal(tmp * tmp * tmp, tmp^3)
```

```
[1] TRUE
```

We can run Differential Evolution with a randomly-initialised population.

```
> algo <- list(nG = 200,  ## number of generations
               nP = 50,   ## population size
               min = rep(-20, p),
               max = rep( 20, p),
               printBar = FALSE)
> resDE <- DEopt(OFls, algo = algo, Data = Data)
```

```
Differential Evolution.
Best solution has objective function value 12.2 ;
standard deviation of OF in final population is 1.04e-12 .
```

The advantage of using Least Squares is that we can compare our results with those obtained through, for instance, the `lm` function, or obtained directly through qr.

```
> data.frame(QR = qr.solve(Data$X, Data$y),
             DE = resDE$xbest)
```

```
       QR      DE
1  1.7902  1.7902
2 -0.4769 -0.4769
3 -0.1943 -0.1943
4 -1.0499 -1.0499
5 -0.0764 -0.0764
```

In the example above I used 200 generations and a population size of 50. How did I know that these settings are appropriate? In fact, I didn't. And that posed no problem: *knowing* appropriate settings is not the goal – we only need to *decide* what settings to use. For this, we run experiments.

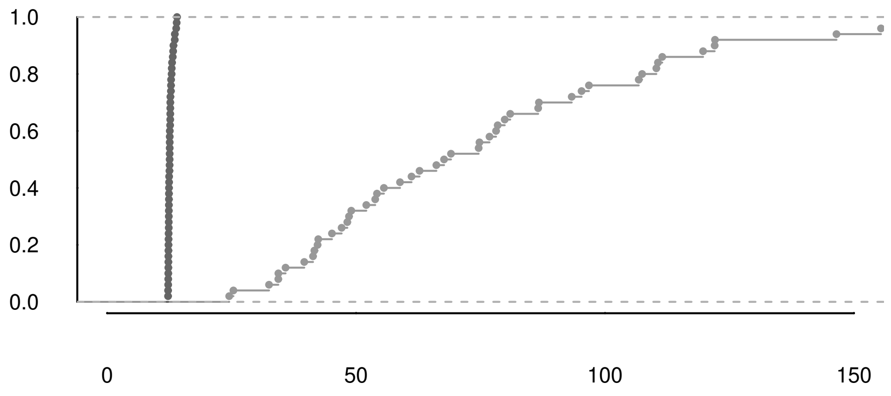
Suppose we leave the population size for now. (A rule of thumb is to use at least 2–5 times the number of decision variables.)

```
> algo <- list(nP = 50,
               min = rep(-20, p),
               max = rep( 20, p),
               printBar = FALSE, printDetail = FALSE)
```

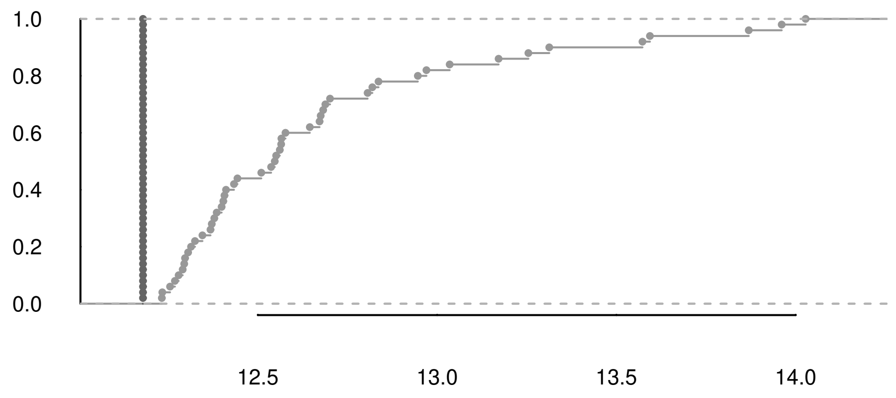
Now we can run the function `restartOpt`.

```
> algo$nG <- 25
> results1 <- restartOpt(DEopt, n = 50, OF = OFls, algo = algo, Data = Dat
> algo$nG <- 50
> results2 <- restartOpt(DEopt, n = 50, OF = OFls, algo = algo, Data = Dat
> algo$nG <- 100
> results3 <- restartOpt(DEopt, n = 50, OF = OFls, algo = algo, Data = Dat
> algo$nG <- 200
> results4 <- restartOpt(DEopt, n = 50, OF = OFls, algo = algo, Data = Dat
```

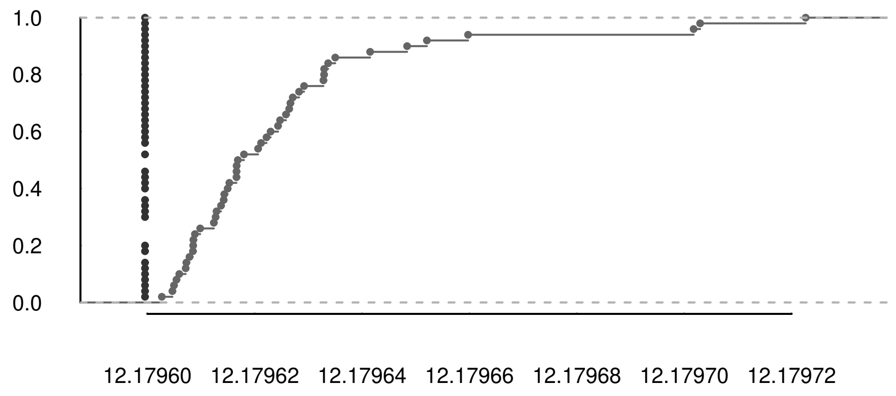
We compare 25 with 50 generations.



And 50 with 100 (note the changed y-scale).



And 100 with 200 (note the changed y-scale).



We will discuss more diagnostics for DEopt in Section 4.1.

2.2.2. Unconstrained Least Trimmed Squares

We define LTS such that the sum of the h smallest squared residuals is minimised. Thus, we need to add h to Data. The new objective function OFlts follows.

```
> OFlts <- function(b, Data) {
      tmp <- Data$y - Data$X %*% b
      tmp <- sort(tmp * tmp, partial = Data$h)
      sum(tmp[seq_len(Data$h)])
  }
```

For LTS, there exists a fast and typically very successful algorithm named FastLTS (Rousseeuw and Van Driessen, 2005), which is implemented in package robustbase. We can compare it with our method.

```
> library("robustbase")
> alpha <- 0.9
> Data <- list(X = rD$X, y = rD$y, p = rD$p, n = rD$n,
               h = h.alpha.n(alpha, n = n, p = p))
```

We use the function h.alpha.n to determine h (but we could also have set it to ceiling(alpha*n), say).

```
> resDE  <- DEopt(OFlts, algo = algo, Data = Data)
> resLTS <- ltsReg(rD$y ~ -1 + rD$X, alpha = alpha,
                   use.correction = FALSE)
> data.frame(fastLTS = resLTS$raw.coefficients,
             DE = resDE$xbest)
```

```
       fastLTS      DE
rD$X1   1.7689  1.7689
rD$X2  -0.3984 -0.3984
rD$X3  -0.1764 -0.1764
rD$X4  -1.1280 -1.1280
rD$X5  -0.0309 -0.0309
```

We can also compare the objective function values.

```
> cLTS <- resLTS$raw.coefficients
> cat("LTS")
```

```
LTS
```

```
> sum(sort((Data$X %*%cLTS - Data$y)^2)[1:Data$h])
```

```
[1] 5.89
```

```
> cDE <- resDE$xbest
> cat("DEopt")
```

```
DEopt
```

```
> sum(sort((Data$X %*%cDE -  Data$y)^2)[1:Data$h])
```

```
[1] 5.89
```

2.3. Constraints

2.3.1. Repairing or penalising

There are two constraints: have the weights sum to one, and no weights should be negative. b0 violates both.

```
> any(b0 < 0)
```

```
[1] TRUE
```

```
> sum(b0)
```

```
[1] 0.456
```

There are a number of ways to heal these violations; see also Section 3.1.2. In this tutorial we will discuss two possibilities: repairing a solutions, or using a penalty function.

A repair function is straightforward to write.

```
> repair <- function(b, Data) {
      b <- abs(b)
      b/sum(b)
  }
```

This function first reflects negative values to their positive counterparts; it then divides all coefficients by the sum of all coefficients. There are many variations to this, but for now, we stick with this simple solution.

```
> b1 <- repair(b0, Data)
> all(b1 >= 0)  ## should be TRUE
```

```
[1] TRUE
```

```
> sum(b1)       ## should be 1
```

```
[1] 1
```

An alternative is a penalty function. Whenever a solution violates a constraint, we add a positive number to its objective function value. Since we minimise, this will make the solution look bad. Note that in this way we change the model to solve into one of unconstrained optimisation.

Capturing negative coeffients is simple:

```
> b0
```

```
[1] -1.438 -0.680  0.437  0.269  1.868
```

```
> b0 - abs(b0)
```

```
[1] -2.88 -1.36  0.00  0.00  0.00
```

```
> sum(b0)
```

```
[1] 0.456
```

Why not write `b0[b0 < 0]`? Because the `b - abs(b)` is slightly more efficient. (The more elements `b` has, the bigger the advantage.)

```
> b <- rnorm(1000L)
> benchmark(sum(b - abs(b))/2,
            sum(b[b < 0]),
            columns = c("test", "elapsed", "relative"),
            replications = 1e4, order = "relative")
```

```
               test elapsed relative
1 sum(b - abs(b))/2   0.072     1.00
2     sum(b[b < 0])   0.105     1.46
```

```
> all.equal(sum(b - abs(b))/2, sum(b[b < 0]))
```

```
[1] TRUE
```

Similarly we can check whether the sum of the coefficients is one.

```
> abs(sum(b0) - 1)
```

```
[1] 0.544
```

We put these computations into a function. Each violation is weighted by a weight `pw`.

```
> Data$pw1 <- 500
> Data$pw2 <- 500
> penalty <- function(b, Data)
      Data$pw1 * -sum(b - abs(b)) + Data$pw2 * abs(sum(b) - 1)
> penalty(b0, Data)
```

```
[1] 2389
```

```
> penalty(b1, Data) ## recall that b1 was 'repaired'
```

```
[1] 5.55e-14
```

Let’s try. First, the unconstrained case.

```
> algo <- list(nG = 500, nP = 100,
               min = rep(-20, p), max = rep( 20, p),
               printBar = FALSE)
> resDE <- DEopt(OFls, algo = algo, Data = Data)
```

```
Differential Evolution.
Best solution has objective function value 12.2 ;
standard deviation of OF in final population is 1.07e-15 .
```

We should not be surprised that the constraints are violated.

```
> round(resDE$xbest, 5)
```

```
[1]  1.7902 -0.4769 -0.1943 -1.0499 -0.0764
```

```
> resDE$OFvalue
```

```
[1] 12.2
```

```
> sum(resDE$xbest)
```

```
[1] -0.00728
```

```
> all(resDE$xbest >= 0)
```

```
[1] FALSE
```

Now we use the `repair` function.

```
> algo$repair <- repair
> resDE <- DEopt(OFls, algo = algo, Data = Data)
```

```
Differential Evolution.
Best solution has objective function value 168 ;
standard deviation of OF in final population is 0 .
```

```
> round(resDE$xbest,5)
```

```
[1] 1 0 0 0 0
```

```
> resDE$OFvalue
```

```
[1] 168
```

```
> sum(resDE$xbest)
```

```
[1] 1
```

```
> all(resDE$xbest >= 0)
```

```
[1] TRUE
```

Now we use the penalty function.

```
> algo$repair <- NULL
> algo$pen <- penalty
> resDE <- DEopt(OF1s, algo = algo, Data = Data)
```

```
Differential Evolution.
Best solution has objective function value 168 ;
standard deviation of OF in final population is 3.97e-11 .
```

```
> round(resDE$xbest,5)
```

```
[1] 1 0 0 0 0
```

```
> resDE$OFvalue
```

```
[1] 168
```

```
> sum(resDE$xbest)
```

```
[1] 1
```

```
> all(resDE$xbest >= 0)
```

```
[1] FALSE
```

Exercise 2.1 *The non-negativity constraint is violated. Why? Try to fix it.*

2.4. Doing it the vectorised way

The population is repaired and evaluated through a loop. While this is natural, the R way to do it would be to evaluate all solutions in one step. For instance, if we take the absolute value of the twenty columns of a matrix, we can more naturally compute the value for the matrix at once. This section is going to discuss how to vectorise the objection function (OF).

Recall that the OF looked like this.

```
> OFlts
```

```
function(b, Data) {
    tmp <- Data$y - Data$X %*% b
    tmp <- sort(tmp * tmp, partial = Data$h)
    sum(tmp[seq_len(Data$h)])
}
<bytecode: 0x565184a7d068>
```

```
> b0 <- rnorm(p)
> b1 <- rnorm(p)
> P <- cbind(b0 = b0, b1 = b1)
```

What we first did in the OF is to compute residuals.

```
> head(Data$y - Data$X %*% b0)
```

```
       [,1]
[1,]  2.670
[2,]  0.530
[3,]  5.940
[4,] -0.727
[5,] -2.760
[6,]  2.978
```

```
> head(Data$y - Data$X %*% b1)
```

```
       [,1]
[1,]  1.805
[2,] -1.236
[3,]  1.915
[4,]  0.901
[5,] -0.434
[6,]  1.668
```

```
> head(drop(Data$y) - Data$X %*% P)
```

```
         b0     b1
[1,]  2.670  1.805
[2,]  0.530 -1.236
[3,]  5.940  1.915
[4,] -0.727  0.901
[5,] -2.760 -0.434
[6,]  2.978  1.668
```

Note that we had to drop the `dim` attribute from `Data$y`, which we did with the `drop` function.

Next, we had to square the residuals. Again, there is no need to loop.

```
> head(Data$y - Data$X %*% b0)^2
```

```
       [,1]
[1,]  7.129
[2,]  0.281
[3,] 35.278
[4,]  0.529
[5,]  7.617
[6,]  8.867
```

```
> head(Data$y - Data$X %*% b1)^2
```

```
      [,1]
[1,] 3.256
[2,] 1.528
[3,] 3.667
[4,] 0.812
[5,] 0.189
[6,] 2.783
```

```
> head((drop(Data$y) - Data$X %*% P)*(drop(Data$y) - Data$X %*% P))
```

```
         b0    b1
[1,]  7.129 3.256
[2,]  0.281 1.528
[3,] 35.278 3.667
[4,]  0.529 0.812
[5,]  7.617 0.189
[6,]  8.867 2.783
```

Here is the new OF.

```
> OFlts2 <- function(b, Data) {
      tmp <- drop(Data$y) - Data$X %*% b
      tmp <- tmp * tmp
      tmp <- apply(tmp, 2L, sort, partial = Data$h)
      .colSums(tmp[seq_len(Data$h), ,drop = FALSE], Data$h, ncol(b))
  }
```

We create a random population.

```
> nP <- 100
> P <- array(rnorm(p * nP), dim = c(p, nP))
> sol0 <- OFlts2(P, Data)
```

```
> sol1 <- numeric(nP)
> benchmark(for (i in seq_len(nP))
            sol1[i] <- OFlts(P[ , i, drop = FALSE], Data),
            sol2 <- OFlts2(P, Data),
            columns = c("test", "elapsed", "relative"),
            replications = 100, order = "relative")
```

```
                                                                test
2                                            sol2 <- OFlts2(P, Data)
1 for (i in seq_len(nP)) sol1[i] <- OFlts(P[, i, drop = FALSE], Data)
  elapsed relative
2   0.225      1.0
1   0.496      2.2
```

```
> all.equal(sol1, sol2)
```

```
[1] TRUE
```

We run DEopt.

```
> algo$repair <- repair
> algo$pen <- NULL
> algo$loopOF <- TRUE  ## loop
> resDE  <- DEopt(OFlts, algo = algo, Data = Data)
```

```
Differential Evolution.
Best solution has objective function value 101 ;
standard deviation of OF in final population is 0 .
```

```
> round(resDE$xbest,5)
```

```
[1] 1 0 0 0 0
```

```
> algo$loopOF <- FALSE  ## vectorised
> resDE <- DEopt(OFlts2, algo = algo, Data = Data)
```

```
Differential Evolution.
Best solution has objective function value 101 ;
standard deviation of OF in final population is 0 .
```

```
> round(resDE$xbest,5)
```

```
[1] 1 0 0 0 0
```

Function reference

3. Optimisation with single-solution methods

3.1. Heuristics

Numerical optimisation starts with a model, typically stated as

$$\underset{x}{\text{minimise}}\ \phi(x)\ , \tag{3.1}$$

in which ϕ is the function that we want to minimise, called the objective function, and x represents the decision variables. If we wanted to maximise, we would minimise $-\phi$ instead. In most models, there are restrictions on how we may choose x.

Heuristics,[†] the techniques described in this book, are a class of methods for solving such optimisation models.

[†] The term heuristics is actually is in different – though often related – meanings in different disciplies. For a brief overview, see Gilli and Schumann (2017), on which this section is based.

We find it helpful to not think in terms of a mathematical description, but rather to replace ϕ by something like

```
        solutionQuality = function(x, data).
```

That is, we need to be able to program a mapping from a solution to its quality, given the data. There is no need for a closed-form mathematical description of the function.[1] Indeed, in many applied disciplines there are no closed-form objective functions. The function ϕ could include an experimental setup, with x the chosen treatment and $\phi(x)$ the desirability of its outcome. Or evaluating ϕ might require a complicated stochastic simulation, such as an agent-based model.

A number of requirements describe an optimisation heuristic further (Zanakis and Evans, 1981, Barr et al., 1995, and Winker and Maringer, 2007, list similar criteria):

- The method should give a 'good' stochastic approximation of the true optimum, with 'goodness' measured in computing time or solution quality.
- The method should be robust when we change the model – for instance, when we modify the objective function or add a constraint – and also when we increase the problem size. Results should not vary too much for different parameter settings for the heuristic.

[1]Mathematically a function is nothing but a mapping, so there is no contradiction here. But when people see $\phi(x)$ they intuitively often think of something like $\phi(x) = \sqrt{x} + x^2$. We would prefer they thought of a programme, not a formula.

- The technique should be easy to implement.
- Implementation and application of the technique should not require subjective elements.

Such a definition is not unambiguous, but it is a start. Actually, we think that users can only gain intuition about heuristics through studying examples – which we will do in the next section. But for now, we shall go on dwelling on principles.

In a broad sense, we can differentiate between two classes of heuristics, constructive methods and iterative-search methods. In this chapter, we shall concentrate on the latter type, so let us give a quick example for constructive methods and then not mention them any further. For a constructive method, an algorithm starts with an empty solution and adds components step-by-step; the procedure terminates when it has completed one solution. An example: a reasonable low-variance equity portfolio of cardinality N can be constructed by (i) obtaining forecasts for the marginal variances of all eligible assets, (ii) sort the assets by forecast variance and (iii) keep the N assets with the lowest forecast variance in the portfolio (equally-weighted); see Schumann (2013).

For iterative search methods the algorithm moves from solution to solution, that is, a complete existing solution is modified to obtain a new solution. Such a new solution may be quite different from previous ones, as some methods, such as Genetic Algorithms, create new solutions in a rather discontinuous ways. But still, a new solution will share characteristics with its predecessor (if that was not the case, we would be doing random-sampling).

3.1.1. Principles

The following pseudocode should make the idea of an iterative method more precise.

```
1: generate initial solution $x^c$
2: **while** stopping condition not met **do**
3:     create new solution $x^n = N(x^c)$
4:     **if** $A(\phi, x^n, x^c, \ldots)$ **then** $x^c = x^n$
5: **end while**
6: return $x^c$
```

In words: we start with a solution x^c, typically randomly chosen. Then, in each iteration, the function N ('neighbour') makes a copy of x^c and modifies this copy; thus, we get a new candidate solution x^n. The function A ('accept') decides whether x^n replaces x^c, typically by comparing the objective function values of the solutions. The process is repeated until a stopping condition is satisfied; finally, x^c is returned.

To implement such a method, we need to specify

- how we represent a solution x,

- how we evaluate a solution (the function ϕ),
- how we change a solution (the function N),
- how to decide whether to accept a solution (the function A),
- when to stop.

These building blocks would still apply to a classical method. For example, for a gradient-based method x would be a numeric vector; N would evaluate the gradient at x^{c} and then move minus the gradient with a specified stepsize; A would evaluate x^{c} and x^{n}, and replace x^{c} only if x^{n} is better; if not, the search is stopped.

Heuristics use other, often simpler, mechanisms. In fact, two characteristics will show up in almost all methods. (i) Heuristics will not insist on the best possible moves. A heuristic may accept a new solution x^{n} even if it is worse than the current solution. (ii) Heuristics typically have random elements. For instance, a heuristic may change x^{c} randomly (instead of locally-optimally as in a gradient search). These characteristics make heuristics inefficient for well-behaved models. But for difficult models (for instance, such with many local optima as in Figure **??**), they enable heuristics to move away from local optima.[2]

Let us give a concrete example, namely the problem we already used earlier: we want to select N assets, equally-weighted, out of a large number of assets, such that the resulting portfolio has a small variance. We assume that we have a forecast for the variance–covariance matrix available. Then a simple method for getting a very good solution to this model is a local search. For a local search,

- the solution x is a list of the included assets;
- the objective function ϕ is a function that computes the variance forecast for a portfolio x;
- the function N picks one neighbour by randomly removing one asset from the portfolio and adding another one;
- the function A compares $\phi(x^{\mathrm{c}})$ and $\phi(x^{\mathrm{n}})$, and if x^{n} is not worse, accepts it;
- the stopping rule is to quit after a fixed number of iterations.

Note that local search is still greedy in a sense, since it will not accept a new solution that is worse than the previous one. Thus, if the search arrives at a solution that is better than all its neighbours, it can never move away from it –

[2] In principle, because of such mechanisms a heuristic could drift farther and farther off a good solution. But practically, that is very unlikely because every heuristic has a bias towards good solutions. In Threshold Accepting, the method that we describe in Section **??**, that bias comes into effect because a better solution is always accepted, a worse one only if it is not too bad. Since we repeat this creating of new candidate solutions thousands of times, we can be very certain that the scenario of drifting-off a good solution does practically not occur.

even if this solution is only a local optimum. Heuristic methods that build on local search thus employ additional strategies for escaping such local optima.

And indeed, with a small – but important – variation we arrive at Simulated Annealing (Kirkpatrick et al., 1983). We use a different acceptance rule A: If the new solution is better, accept it. If it is worse, do still accept it, but only with a specific probability. This probability in turn depends on the new solution's quality: the worse it is, the less likely it is the solution is accepted. Also, the probability of acceptence is typically lower in later iterations (that is, the algorithm becomes pickier). In many implementations, the probability at later stages is essentially zero; thus, Simulated Annealing turns into a local search.

3.1.2. Constraints

Nothing in the pseudocode that we showed above ensures that a constraint on a solution x is observed. But it is often constraints that make models realistic and difficult. Several strategies exist for including restrictions into heuristics.

Throw away

If our model has only few constraints that are not often hit, the simplest approach is to 'throw away' infeasible new solutions. That is, if a neighbour solution violates a constraint, we just select another neighbour. Note that this means that we include the contraints in the acceptance function A.

Include constraint in N

We can directly use the constraint to create new, feasible solutions. In portfolio selection models we usually have a budget constraint; that is, we require that all asset weights sum to one. This constraint can be enforced when we compute new solutions by increasing some weights and decreasing others such that the sum of all weight changes is zero.

Transform x

An older but still used idea is to transform variables. This approach sometimes works for constraints that require that the elements of x lie in certain ranges; see the discussion in Powell (1972). For instance, $\sin(x)$ will map any real x to the range $[-1, 1]$; $\alpha\,(\sin(x))^2$ will give a mapping to $[0, \alpha]$. But such transformations come with their own problems; see Gill et al. (1986, Section 7.4); in particular, it may become difficult to change a problem later on or to handle multiple constraints.

Repair x

We can introduce mechanisms to correct solutions that violate constraints. For example, if a solution x holds the portfolio weights, then dividing every element in x by the sum of the elements of x ensures that all weights sum to unity.

Penalise x

Finally, we can penalise infeasible solutions. Whenever a constraint is violated, we add a penalty term to the objective function and so downgrade the quality of the solution. In essence, we change the problem to an unconstrained one for which we can use the heuristic. The penalty is often made an increasing function of the magnitude of violation. Thus, the algorithm may move through infeasible areas of the search space, but will have guidance to return to feasible areas. The penalty approach is the most generic strategy to include constraints; it is convenient since the computational architecture needs hardly be changed. Penalties create soft constraints since the algorithm could in principle always override a penalty; practically, we can set the penalty so high that we have hard constraints.

3.1.3. Random solutions

The most common objection against using heuristics is the fact that, since heuristics explicitly rely on random mechanisms, their solutions are also random. This randomness, it is argued, makes it difficult to evaluate the quality of solutions computed by such algorithms. (The discussion in this section builds on Gilli et al., 2011.)

Randomness

A naïve approach to solving an optimisation model could be this: randomly generate a large number of candidate solutions, evaluate all solutions and pick the best one. This best solution is our overall solution.

If we repeated the whole procedure a second time, our overall solution would probably be a different one. Thus, the solution x we obtain through our sampling strategy is stochastic. The difference between our solution and the actual optimum would be a kind of truncation error, since if we sampled more and more, we should in theory come arbitrarily close to the optimum. Importantly, the variability of the solution stems from our numerical technique; it has nothing to do with the error terms that we may have in models to account for uncertainty. Stochastic solutions may even occur with non-stochastic methods: think of search spaces like those we showed in Figure **??**. Even if we used a deterministic method like a gradient search, the many local minima would make sure that repeated runs from different starting points result in different solutions.

We can treat the result of a stochastic algorithm as a random variable with some distribution D. What exactly the 'result' of a restart is depends on our setting. We will want to look at the objective function value (ie, the solution quality), but we may also look at the decision variables given by a solution, that is, the portfolio weights. In any case, we collect all the quantities of interest in a vector ϱ. The result ϱ_j of a restart j is a random draw from D.

The trouble is that we do not know what D looks like. But fortunately, there is a simple way to find out for a given model. We run a reasonably large number of restarts, each time store ϱ_j, and finally compute the empirical distribution function of the ϱ_j, $j = 1, \ldots,$ number-of-restarts as an estimate for D. For a given model or model class, the shape of the distribution D will depend on the chosen method. Some techniques will be more appropriate than others and give less variable and on average better results. And D will often depend on the particular settings of the method, in particular the number of iterations – the search time – that we allow for.

Unlike classical optimization techniques, heuristics can walk away from local minima; they will not necessarily get trapped. So if we let the algorithm search for longer, we can hope to find better solutions. For minimization problems, when we increase the number of iterations, the mass of D will move to the left and the distribution will become less variable. Ideally, when we let the computing time grow ever longer, D should degenerate into a single point, the global minimum. There exist proofs of this convergence to the global minimum for many heuristic methods (see Gelfand and Mitter, 1985, for Simulated Annealing; Rudolph, 1994, for Genetic Algorithms; Gutjahr, 2000, Stützle and Dorigo, 2002, for Ant Colony Optimisation; Bergh and Engelbrecht, 2006, for Particle Swarm Optimisation).

Unfortunately, these proofs are not much help for practical applications. They often rely on asymptotic arguments; and many such proofs are nonconstructive (eg, Althöfer and Koschnick, 1991, for Threshold Accepting): they demonstrate that parameter settings exist that lead (asymptotically) to the global optimum. Yet, practically, there is no way of telling whether the chosen parameter setting is correct in this sense; we are never guaranteed that D really degenerates to the global optimum as the number of iterations grows.

Fortunately, we do not need these proofs to make meaningful statements about the performance of specific methods. For a given model class, we can run experiments. Such experiments also help investigate the sensitivity of the solutions with respect to different parameter settings for the heuristic. Experimental results are of course no proof of the general appropriateness of a method, but they are evidence of how a method performs for a given class of models; often this is all that is needed for practical applications.

3.2. Local Search

See `?LSopt` after attaching the package.

3.2.1. Man page

`LSopt` *Stochastic Local Search*

3.2.2. Description

Performs a simple stochastic Local Search.

3.2.3. Usage

```
LSopt(OF, algo = list(), ...)
```

3.2.4. Arguments

`OF` The objective function, to be minimised. Its first argument needs to be a solution; `...` arguments are also passed.

`algo` List of settings. See Details.

`...` Other variables to be passed to the objective function and to the neighbourhood function. See Details.

3.2.5. Details

Local Search (LS) changes an initial solution for a number of times, accepting only such changes that lead to an improvement in solution quality (as measured by the objective function `OF`). More specifically, in each iteration, a current solution `xc` is changed through a function `algo$neighbour`. This function takes `xc` as an argument and returns a new solution `xn`. If `xn` is not worse than `xc`, ie, if `OF(xn,...)<=OF(xc,...)`, then `xn` replaces `xc`.

The list `algo` contains the following items:

`nS` The number of steps. The default is 1000; but this setting depends very much on the problem.

`nI` Total number of iterations, with default `NULL`. If specified, it will override `nS`. The option is provided to makes it easier to compare and switch between functions `LSopt`, `TAopt` and `SAopt`.

`x0` The initial solution. This can be a function; it will then be called once without arguments to compute an initial solution, ie, `x0 <- algo$x0()`. This can be useful when `LSopt` is called in a loop of restarts and each restart is to have its own starting value.

`neighbour` The neighbourhood function, called as `neighbour(x, ...)`. Its first argument must be a solution `x`; it must return a changed solution.

`printDetail` If `TRUE` (the default), information is printed. If an integer `i` greater then one, information is printed at very `i`th step.

`printBar` If `TRUE` (the default), a `txtProgressBar` (from package utils) is printed). The progress bar is not shown if `printDetail` is an integer greater than 1.

`storeF` if `TRUE` (the default), the objective function values for every solution in every generation are stored and returned as matrix `Fmat`.

`storeSolutions` default is `FALSE`. If `TRUE`, the solutions (ie, decision variables) in every generation are stored and returned in list `xlist` (see Value section below). To check, for instance, the current solution at the end of the `i`th generation, retrieve `xlist[[c(2L, i)]]`.

`OF.target` Numeric; when specified, the algorithm will stop when an objective-function value as low as `OF.target` (or lower) is achieved. This is useful when an optimal objective-function value is known: the algorithm will then stop and not waste time searching for a better solution.

At the minimum, `algo` needs to contain an initial solution `x0` and a `neighbour` function.

LS works on solutions through the functions `neighbour` and `OF`, which are specified by the user. Thus, a solution need not be a numeric vector, but can be any other data structure as well (eg, a list or a matrix).

To run silently (except for warnings and errors), `algo$printDetail` and `algo$printBar` must be `FALSE`.

3.2.6. Value

A list:

`xbest` best solution found.

`OFvalue` objective function value associated with best solution.

`Fmat` a matrix with two columns. `Fmat[ ,1L]` contains the proposed solution over all iterations; `Fmat[ ,2L]` contains the accepted solutions.

`xlist` if `algo$storeSolutions` is `TRUE`, a list; else `NA`. Contains the neighbour solutions at a given iteration (`xn`) and the current solutions (`xc`). Example: `Fmat[i, 2L]` is the objective function value associated with `xlist[[c(2L, i)]]`.

`initial.state` the value of `.Random.seed` when the function was called.

3.2.7. Diagnostics

- check the neighbourhood
- how often are solutions accepted?
- paths of different restarts

3.3. Threshold Accepting

See `?TAopt` after attaching the package.

3.3.1. Help page

`TAopt` *Optimisation with Threshold Accepting*

3.3.2. Description

The function implements the Threshold Accepting algorithm.

3.3.3. Usage

```
TAopt(OF, algo = list(), ...)
```

3.3.4. Arguments

`OF` The objective function, to be minimised. Its first argument needs to be a solution `x`; it will be called as `OF(x, ...)`.

`algo` A list of settings for the algorithm. See Details.

`...` other variables passed to `OF` and `algo$neighbour`. See Details.

3.3.5. Details

Threshold Accepting (TA) changes an initial solution iteratively; the algorithm stops after a fixed number of iterations. Conceptually, TA consists of a loop than runs for a number of iterations. In each iteration, a current solution `xc` is changed through a function `algo$neighbour`. If this new (or neighbour) solution `xn` is not worse than `xc`, ie, if `OF(xn,...) <= OF(xc,...)`, then `xn` replaces `xc`. If `xn` is worse, it still replaces `xc` as long as the difference in 'quality' between the two solutions is less than a threshold `tau`; more precisely, as long as `OF(xn,...) - tau <= OF(xc,...)`. Thus, we also accept a new solution that is worse than its predecessor; just not too much worse. The threshold is typically decreased over the course of the optimisation. For zero thresholds TA becomes a stochastic local search.

The thresholds can be passed through the list `algo` (see below). Otherwise, they are automatically computed through the procedure described in Gilli et al. (2006). When the thresholds are created automatically, the final threshold is always zero.

The list `algo` contains the following items.

`nS` The number of steps per threshold. The default is 1000; but this setting depends very much on the problem.

`nT` The number of thresholds. Default is 10; ignored if `algo$vT` is specified.

`nI` Total number of iterations, with default `NULL`. If specified, it will override `nS` with `ceiling(nI/nT)`. Using this option makes it easier to compare and switch between functions `LSopt`, `TAopt` and `SAopt`.

`nD` The number of random steps to compute the threshold sequence. Defaults to 2000. Only used if `algo$vT` is `NULL`.

`q` The highest quantile for the threshold sequence. Defaults to 0.5. Only used if `algo$vT` is `NULL`. If `q` is zero, `TAopt` will run with `algo$nT` zero-thresholds (ie, like a Local Search).

`x0` The initial solution. If this is a function, it will be called once without arguments to compute an initial solution, ie, `x0 <- algo$x0()`. This can be useful when the routine is called in a loop of restarts, and each restart is to have its own starting value.

`vT` The thresholds. A numeric vector. If `NULL` (the default), `TAopt` will compute `algo$nT` thresholds. Passing threshold can be useful when similar problems are handled. Then the time to sample the objective function to compute the thresholds can be saved (ie, we save `algo$nD` function evaluations). If the thresholds are computed and `algo$printDetail` is `TRUE`, the time required to evaluate the objective function will be measured and an estimate for the remaining computing time is issued. This estimate is often very crude.

`neighbour` The neighbourhood function, called as `neighbour(x, ...)`. Its first argument must be a solution `x`; it must return a changed solution.

`printDetail` If `TRUE` (the default), information is printed. If an integer `i` greater then one, information is printed at very `i`th iteration.

`printBar` If `TRUE` (default is `FALSE`), a `txtProgressBar` (from package utils) is printed. The progress bar is not shown if `printDetail` is an integer greater than 1.

`scale` The thresholds are multiplied by `scale`. Default is 1.

`drop0` When thresholds are computed, should zero values be dropped from the sample of objective-function values? Default is `FALSE`.

`stepUp` Defaults to `0`. If an integer greater than zero, then the thresholds are recycled, ie, `vT` is replaced by `rep(vT, algo$stepUp + 1)` (and the number of thresholds will be increased by `algo$nT` times `algo$stepUp`). This option works for supplied as well as computed thresholds. Practically, this will have the same effect as restarting from a returned solution. (In Simulated Annealing, this strategy goes by the name of 're-heating'.)

`thresholds.only` Defaults to `FALSE`. If `TRUE`, compute only threshold sequence, but do not actually run TA.

`storeF` if `TRUE` (the default), the objective function values for every solution in every generation are stored and returned as matrix `Fmat`.

`storeSolutions` Default is `FALSE`. If `TRUE`, the solutions (ie, decision variables) in every generation are stored and returned in list `xlist` (see Value section below). To check, for instance, the current solution at the end of the `i`th generation, retrieve `xlist[[c(2L, i)]]`.

`classify` Logical; default is `FALSE`. If `TRUE`, the result will have a class attribute `TAopt` attached. This feature is experimental: the supported methods (plot, summary) may change without warning.

`OF.target` Numeric; when specified, the algorithm will stop when an objective-function value as low as `OF.target` (or lower) is achieved. This is useful when an optimal objective-function value is known: the algorithm will then stop and not waste time searching for a better solution.

At the minimum, `algo` needs to contain an initial solution `x0` and a `neighbour` function.

The total number of iterations equals `algo$nT` times `(algo$stepUp + 1)` times `algo$nS` (plus possibly `algo$nD`).

3.3.6. Value

`TAopt` returns a list with four components:

`xbest` the solution

`OFvalue` objective function value of the solution, ie, `OF(xbest, ...)`

`Fmat` if `algo$storeF` is `TRUE`, a matrix with one row for each iteration (excluding the initial `algo$nD` steps) and two columns. The first column contains the objective function values of the neighbour solution at a given iteration; the second column contains the value of the current solution. Since TA can walk away from locally-optimal solutions, the best solution can be monitored through `cummin(Fmat[ ,2L])`.

`xlist` if `algo$storeSolutions` is `TRUE`, a list; else `NA`. Contains the neighbour solutions at a given iteration (`xn`) and the current solutions (`xc`). Example: `Fmat[i, 2L]` is the objective function value associated with `xlist[[c(2L, i)]]`.

`initial.state` the value of `.Random.seed` when the function was called.

If `algo$classify` was set to `TRUE`, the resulting list will have a class attribute `TAopt`.

3.3.7. Note

If the `...` argument is used, then all the objects passed with `...` need to go into the objective function and the neighbourhood function. It is recommended to collect all information in a list `myList` and then write `OF` and `neighbour` so that they are called as `OF(x, myList)` and `neighbour(x, myList)`. Note that `x` need not be a vector but can be any data structure (eg, a `matrix` or a `list`).

Using thresholds of size `0` makes TA run as a Local Search. The function `LSopt` may be preferred then because of smaller overhead.

3.3.8. Diagnostics

- check the neighbourhood
- how often are solutions accepted?
- paths of different restarts

3.4. Simulated Annealing

See `?SAopt` after attaching the package.

3.4.1. Help page

`SAopt`	*Optimisation with Simulated Annealing*

3.4.2. Description

The function implements a Simulated-Annealing algorithm.

3.4.3. Usage

```
SAopt(OF, algo = list(), ...)
```

3.4.4. Arguments

`OF` The objective function, to be minimised. Its first argument needs to be a solution `x`; it will be called as `OF(x, ...)`.

`algo` A list of settings for the algorithm. See Details.

`...` other variables passed to `OF` and `algo$neighbour`. See Details.

3.4.5. Details

Simulated Annealing (SA) changes an initial solution iteratively; the algorithm stops after a fixed number of iterations. Conceptually, SA consists of a loop than runs for a number of iterations. In each iteration, a current solution `xc` is changed through a function `algo$neighbour`. If this new (or neighbour) solution `xn` is not worse than `xc`, ie, if `OF(xn,...) <= OF(xc,...)`, then `xn` replaces `xc`. If `xn` is worse, it still replaces `xc`, but only with a certain probability. This probability is a function of the degree of the deterioration (the greater, the less likely the new solution is accepted) and the current iteration (the longer the algorithm has already run, the less likely the new solution is accepted).

The list `algo` contains the following items.

`nS` The number of steps per temperature. The default is 1000; but this setting depends very much on the problem.

`nT` The number of temperatures. Default is 10.

`nI` Total number of iterations, with default `NULL`. If specified, it will override `nS` with `ceiling(nI/nT)`. Using this option makes it easier to compare and switch between functions `LSopt`, `TAopt` and `SAopt`.

`nD` The number of random steps to calibrate the temperature. Defaults to 2000.

`initT` Initial temperature. Defaults to `NULL`, in which case it is automatically chosen so that `initProb` is achieved.

`finalT` Final temperature. Defaults to 0.

`alpha` The cooling constant. The current temperature is multiplied by this value. Default is 0.9.

`mStep` Step multiplier. The default is 1, which implies constant number of steps per temperature. If greater than 1, the step number `nS` is increased to `m*nS` (and rounded).

x0 The initial solution. If this is a function, it will be called once without arguments to compute an initial solution, ie, `x0 <- algo$x0()`. This can be useful when the routine is called in a loop of restarts, and each restart is to have its own starting value.

neighbour The neighbourhood function, called as `neighbour(x, ...)`. Its first argument must be a solution x; it must return a changed solution.

printDetail If TRUE (the default), information is printed. If an integer i greater then one, information is printed at very ith iteration.

printBar If TRUE (default is FALSE), a txtProgressBar (from package utils) is printed. The progress bar is not shown if printDetail is an integer greater than 1.

storeF if TRUE (the default), the objective function values for every solution in every generation are stored and returned as matrix Fmat.

storeSolutions Default is FALSE. If TRUE, the solutions (ie, decision variables) in every generation are stored and returned in list xlist (see Value section below). To check, for instance, the current solution at the end of the ith generation, retrieve `xlist[[c(2L, i)]]`.

classify Logical; default is FALSE. If TRUE, the result will have a class attribute SAopt attached.

OF.target Numeric; when specified, the algorithm will stop when an objective-function value as low as OF.target (or lower) is achieved. This is useful when an optimal objective-function value is known: the algorithm will then stop and not waste time searching for a better solution.

At the minimum, algo needs to contain an initial solution x0 and a neighbour function.

The total number of iterations equals algo$nT times algo$nS (plus possibly algo$nD).

3.4.6. Value

SAopt returns a list with five components:

xbest the solution

OFvalue objective function value of the solution, ie, `OF(xbest, ...)`

Fmat if algo$storeF is TRUE, a matrix with one row for each iteration (excluding the initial algo$nD steps) and two columns. The first column contains the objective function values of the neighbour solution at a given iteration; the second column contains the value of the current solution. Since SA can walk away from locally-optimal solutions, the best solution can be monitored through `cummin(Fmat[ ,2L])`.

`xlist` if `algo$storeSolutions` is `TRUE`, a list; else `NA`. Contains the neighbour solutions at a given iteration (`xn`) and the current solutions (`xc`). Example: `Fmat[i, 2L]` is the objective function value associated with `xlist[[c(2L, i)]]`.

`initial.state` the value of `.Random.seed` when the function was called.

If `algo$classify` was set to `TRUE`, the resulting list will have a class attribute `TAopt`.

3.4.7. Note

If the `...` argument is used, then all the objects passed with `...` need to go into the objective function and the neighbourhood function. It is recommended to collect all information in a list `myList` and then write `OF` and `neighbour` so that they are called as `OF(x, myList)` and `neighbour(x, myList)`. Note that `x` need not be a vector but can be any data structure (eg, a `matrix` or a `list`).

Using an initial and final temperature of zero means that SA will be equivalent to a Local Search. The function `LSopt` may be preferred then because of smaller overhead.

3.4.8. Diagnostics

- check the neighbourhood
- how often are solutions accepted?
- paths of different restarts

4. Optimisation with multiple-solution methods (a.k.a. population-based methods)

Multiple-solution methods are – in principle – very similar to single-solution methods. In the previous chapter, we gave the following pseudocode to explain an iterative method.

```
1: generate initial solution $x^{\mathrm{c}}$
2: **while** stopping condition not met **do**
3:     create new solution $x^{\mathrm{n}} = N(x^{\mathrm{c}})$
4:     **if** $A(\phi, x^{\mathrm{n}}, x^{\mathrm{c}}, \ldots)$ **then** $x^{\mathrm{c}} = x^{\mathrm{n}}$
5: **end while**
6: return $x^{\mathrm{c}}$
```

4.1. Differential Evolution

See `?DEopt` after attaching the package.

4.1.1. Description

`DEopt` *Optimisation with Differential Evolution*

4.1.2. Description

The function implements the standard Differential Evolution algorithm.

4.1.3. Usage

```
DEopt(OF, algo = list(), ...)
```

4.1.4. Arguments

`OF` The objective function, to be minimised. See Details.

`algo` A list with the settings for algorithm. See Details and Examples.

`...` Other pieces of data required to evaluate the objective function. See Details and Examples.

4.1.5. Details

The function implements the standard Differential Evolution (no jittering or other features). Differential Evolution (DE) is a population-based optimisation heuristic proposed by Storn and Price (1997). DE evolves several solutions (collected in the 'population') over a number of iterations ('generations'). In a given generation, new solutions are created and evaluated; better solutions replace inferior ones in the population. Finally, the best solution of the population is returned. See the references for more details on the mechanisms.

To allow for constraints, the evaluation works as follows: after a new solution is created, it is (i) repaired, (ii) evaluated through the objective function, (iii) penalised. Step (ii) is done by a call to `OF`; steps (i) and (iii) by calls to `algo$repair` and `algo$pen`. Step (i) and (iii) are optional, so the respective functions default to `NULL`. A penalty is a positive number added to the 'clean' objective function value, so it can also be directly written in the `OF`. Writing a separate penalty function is often clearer; it can be more efficient if either only the objective function or only the penalty function can be vectorised. (Constraints can also be added without these mechanisms. Solutions that violate constraints can, for instance, be mapped to feasible solutions, but without actually changing them. See Maringer and Oyewumi, 2007, for an example.)

Conceptually, DE consists of two loops: one loop across the generations and, in any given generation, one loop across the solutions. `DEopt` indeed uses, as the default, two loops. But it does not matter in what order the solutions are evaluated (or repaired or penalised), so the second loop can be vectorised. This is controlled by the variables `algo$loopOF`, `algo$loopRepair` and `algo$loopPen`, which all default to `TRUE`. Examples are given in the vignettes and in the book. The respective `algo$loopFun` must then be set to `FALSE`.

All objects that are passed through `...` will be passed to the objective function, to the repair function and to the penalty function.

The list `algo` collects the the settings for the algorithm. Strictly necessary are only `min` and `max` (to initialise the population). Here are all possible arguments:

`CR` probability for crossover. Defaults to 0.9. Using default settings may not be a good idea.

`F` The step size. Typically a numeric vector of length one; default is 0.5. Using default settings may not be a good idea. (`F` can also be a vector with different values for each decision variable.)

`nP` population size. Defaults to 50. Using default settings may not be a good idea.

`nG` number of generations. Defaults to 300. Using default settings may not be a good idea.

`min`, `max` vectors of minimum and maximum parameter values. The vectors `min` and `max` are used to determine the dimension of the problem and to randomly initialise the population. Per default, they are no constraints: a solution may well be outside these limits. Only if `algo$minmaxConstr` is `TRUE` will the algorithm repair solutions outside the `min` and `max` range.

`minmaxConstr` if `TRUE`, `algo$min` and `algo$max` are considered constraints. Default is `FALSE`.

`pen` a penalty function. Default is `NULL` (no penalty).

`initP` optional: the initial population. A matrix of size `length(algo$min)` times `algo$nP`, or a function that creates such a matrix. If a function, it should take no arguments.

`repair` a repair function. Default is `NULL` (no repairing).

`loopOF` logical. Should the `OF` be evaluated through a loop? Defaults to `TRUE`.

`loopPen` logical. Should the penalty function (if specified) be evaluated through a loop? Defaults to `TRUE`.

`loopRepair` logical. Should the repair function (if specified) be evaluated through a loop? Defaults to `TRUE`.

`printDetail` If `TRUE` (the default), information is printed. If an integer `i` greater then one, information is printed at very `i`th generation.

`printBar` If `TRUE` (the default), a `txtProgressBar` is printed.

`storeF` if `TRUE` (the default), the objective function values for every solution in every generation are stored and returned as matrix `Fmat`.

`storeSolutions` default is `FALSE`. If `TRUE`, the solutions (ie, decision variables) in every generation are stored and returned as a list `P` in list `xlist` (see Value section below). To check, for instance, the solutions at the end of the `i`th generation, retrieve `xlist[[c(1L, i)]]`. This will be a matrix of size `length(algo$min)` times `algo$nP`. (To be consistent with other functions, `xlist` is itself a list. In the case of `DEopt`, it contains just one element.)

`classify` Logical; default is `FALSE`. If `TRUE`, the result will have a class attribute `TAopt` attached. This feature is experimental: the supported methods may change without warning.

`drop` If `FALSE` (the default), the dimension is not dropped from a single solution when it is passed to a function. (That is, the function will receive a single-column matrix.)

4.1.6. Value

A list:

`xbest` the solution (the best member of the population), which is a numeric vector

`OFvalue` objective function value of best solution

`popF` a vector. The objective function values in the final population.

`Fmat` if `algo$storeF` is `TRUE`, a matrix of size `algo$nG` times `algo$nP` containing the objective function values of all solutions over the generations; else `NA`.

`xlist` if `algo$storeSolutions` is `TRUE`, a list that contains a list `P` of matrices and a matrix `initP` (the initial solution); else `NA`.

`initial.state` the value of `.Random.seed` when the function was called.

4.1.7. Diagnostics

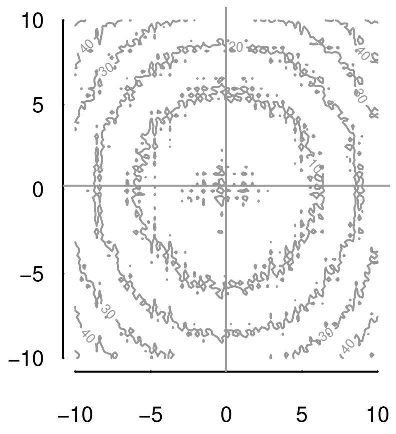
Example 1 – Trefethen's function

We use `tfTrefethen` as the objective function; see `?testFunctions`. To demonstrate the shape of the function, we evaluate it on a grid.

```
> OF <- tfTrefethen
> n <- 100L
> surf <- matrix(NA, n, n)
> x1 <- seq(from = -10, to = 10, length.out = n)
> for (i in seq_len(n))
      for (j in seq_len(n))
          surf[i, j] <- tfTrefethen(c(x1[i], x1[j]))
```

We can now plot these values, including the position of the true minimum. (Since we discretised the function, there may be a small discrepancy between the apparent position of the minimum as indicated by the contour plot and the position indicated by the lines.)

```
> par(bty = "n", las = 1, mar = c(3,4,0,0),
      ps = 8, tck = 0.001, mgp = c(3, 0.5, 0))
> contour(x1, x1, surf, nlevels=5, col = grey(0.6))
> ## the actual minimum
> abline(v = -0.02440308, h = 0.21061243, col = grey(0.6))
```



Now we solve it with DEopt. Note that storeSolutions is TRUE.

```
> algo <- list(nP = 50L,
               nG = 300L,
               F = 0.6,
               CR = 0.9,
               min = c(-10,-10),
               max = c(10,10),
               printDetail = FALSE,
               printBar = FALSE,
               storeF = TRUE,
               storeSolutions = TRUE)
> sol <- DEopt(OF = OF, algo = algo)
```

We can check the solution sol.

```
> names(sol)
```

```
[1] "xbest"         "OFvalue"       "popF"
[4] "Fmat"          "xlist"         "initial.state"
```

```
> sd(sol$popF)
```

```
[1] 5.92e-16
```

```
> ts.plot(sol$Fmat, xlab = "generations", ylab = "OF")
> length(sol$xlist)
```

```
[1] 2
```

```
> xlist <- sol$xlist[[1L]]
```

xlist actually holds a list of matrices (for symmetry: for other functions, xlist contains more than one item.)

Suppose we wanted to look at a particular solution (one column in the population matrix). We could do it like this.

```
> ## show solution 1 (column 1) in population over time
> xlist[[  1L]][ ,1L]  ## at the end of generation 1
```

```
[1]  4.23 -1.04
```

```
> ## ...
> xlist[[ 10L]][ ,1L]  ## at the end of generation 10
```

```
[1] -0.0474 -2.3155
```

```
> ## ...
> xlist[[300L]][ ,1L]  ## at the end of generation 300
```

```
[1] -0.0244  0.2106
```

```
> res  <- sapply(xlist, `[`, 1:2, 1)  ## get row 1 and 2 from column 1
> res2 <- sapply(xlist, `[`, TRUE, 1) ## simpler
> all.equal(res, res2)
```

```
[1] TRUE
```

```
> dim(res)
```

```
[1]   2 300
```

```
> res[ ,1L]
```

```
[1]  4.23 -1.04
```

```
> res[ ,2L]
```

```
[1]  4.23 -1.04
```

```
> res[ ,300L]
```

```
[1] -0.0244  0.2106
```

Alternatively, suppose we wanted to check how parameter 2 varies within the population over the course of the optimisation.

```
> ## show parameter 2 (row 2) in population over time
> xlist[[  1L]][2L, ]  ## at the end of generation 1
```

```
 [1]  -1.040   6.649  -1.638  -1.776   1.151 -11.627  -9.101
 [8]  -1.941  -8.918  -8.065  -2.907  -2.129   2.507  -4.725
[15]   1.354   5.575  -2.029   7.864  -4.609   5.100   2.979
[22]  -7.309   9.621   4.763   3.886  -2.094  -9.675  -3.164
[29]  -3.298  -1.487  -4.119   4.449  -3.038  -5.741  -7.248
[36]  -1.307  -1.100  -0.950   8.444   2.175  -0.828   7.609
[43]   2.555  -4.425  -5.022  -1.726  -9.117   1.955   8.837
[50]   4.780
```

```
> ## ...
> xlist[[ 10L]][2L, ]  ## at the end of generation 10
```

```
 [1] -2.3155 -1.4609  0.5155 -0.6754 -1.5081  0.5376  1.2569
 [8]  0.4732 -1.3417 -2.8404 -0.7210 -1.6990  0.6153 -0.3379
[15]  0.2788 -0.2409 -1.1948 -0.1910 -1.0127 -2.6950  1.2621
[22] -0.5671  0.4539 -0.6330 -0.0871  0.8432 -0.2112  0.2343
[29]  2.4353 -2.6804 -0.0963  0.2933 -0.2292 -0.2709 -0.8708
[36] -1.3065 -0.5952 -0.9499 -1.3652 -1.3470 -0.5043 -2.4387
[43]  1.2327  0.7278 -0.8681 -1.2910 -1.3034  1.2956 -0.9626
[50] -0.1911
```

```
> ## ...
> xlist[[300L]][2L, ]  ## at the end of generation 300
```

```
 [1] 0.211 0.211 0.211 0.211 0.211 0.211 0.211 0.211 0.211
[10] 0.211 0.211 0.211 0.211 0.211 0.211 0.211 0.211 0.211
[19] 0.211 0.211 0.211 0.211 0.211 0.211 0.211 0.211 0.211
[28] 0.211 0.211 0.211 0.211 0.211 0.211 0.211 0.211 0.211
[37] 0.211 0.211 0.211 0.211 0.211 0.211 0.211 0.211 0.211
[46] 0.211 0.211 0.211 0.211 0.211
```

```
> res <- sapply(xlist, `[`, 2, 1:50)
> res <- sapply(xlist, `[`, 2, TRUE)  ## simpler
> dim(res)
```

```
[1]  50 300
```

```
> res[ ,1L]
```

```
 [1]  -1.040   6.649  -1.638  -1.776   1.151 -11.627  -9.101
 [8]  -1.941  -8.918  -8.065  -2.907  -2.129   2.507  -4.725
[15]   1.354   5.575  -2.029   7.864  -4.609   5.100   2.979
[22]  -7.309   9.621   4.763   3.886  -2.094  -9.675  -3.164
[29]  -3.298  -1.487  -4.119   4.449  -3.038  -5.741  -7.248
```

```
[36]  -1.307  -1.100  -0.950   8.444   2.175  -0.828   7.609
[43]   2.555  -4.425  -5.022  -1.726  -9.117   1.955   8.837
[50]   4.780
```

```
> res[ ,2L]
```

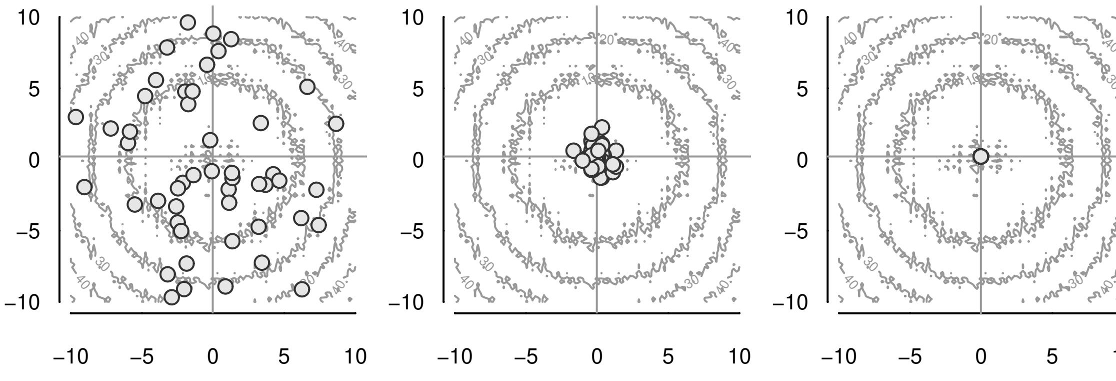
```
 [1] -1.040  6.649 -1.638 -1.776  1.151 -6.189 -8.636 -1.941
 [9] -8.918 -8.065 -2.907 -2.129  2.507 -4.725  1.354  5.575
[17] -2.029  0.131 -4.609 -0.320  2.979 -7.150  2.506  3.375
[25]  3.886 -2.094 -9.675  1.408  4.564 -2.680 -4.119  4.449
[33] -3.038  5.023 -7.248 -1.307 -1.100 -0.950  8.444  4.916
[41] -0.828 -5.670  2.555 -4.425 -5.022 -1.726 -6.850  1.955
[49]  8.837  4.780
```

```
> res[ ,300L]
```

```
 [1] 0.211 0.211 0.211 0.211 0.211 0.211 0.211 0.211 0.211
[10] 0.211 0.211 0.211 0.211 0.211 0.211 0.211 0.211 0.211
[19] 0.211 0.211 0.211 0.211 0.211 0.211 0.211 0.211 0.211
[28] 0.211 0.211 0.211 0.211 0.211 0.211 0.211 0.211 0.211
[37] 0.211 0.211 0.211 0.211 0.211 0.211 0.211 0.211 0.211
[46] 0.211 0.211 0.211 0.211 0.211
```

We can use this information to show how the solutions behaved over time.

```
> ## transposing xlist[[i]] gives a two-column matrix -- see ?points
> ## initial solutions
> points(t(xlist[[1L]]), pch = 21, bg=grey(0.9), col = grey(.2))
> ## solutions at the end of generation 100
> points(t(xlist[[100L]]), pch = 21, bg=grey(0.9), col = grey(.2))
> ## solutions at the end of generation 100
> points(t(xlist[[300L]]), pch = 21, bg=grey(0.9), col = grey(.2))
```



Example 2 – Nelson–Siegel with restrictions

As a second example, we look at the Nelson–Siegel model (see GMS, Chapter 14). We will try to answer two questions: (1) how relevant is the range over which we initialise the population?, and (2) how can we be sure that a constraint works?

We start with the objective function.

```
> OF <- function(par, Data) {
      ## compute model yields
      y <- Data$model(par, Data$tm)

      ## all rates finite?
      validRates <- !any(is.na(y))

      if (validRates) {
          ## any rates negative? if yes, add penalty
          pen1 <- sum(abs(y - abs(y))) * Data$ww

          F <- max(abs(y - Data$yM)) + pen1
      } else F <- 1e8
      F
  }
```

Now set up a true yield curve and try to recover its parameters with DEopt. The first true parameter is 5, but we initialise the population over the range from 0 to 1.

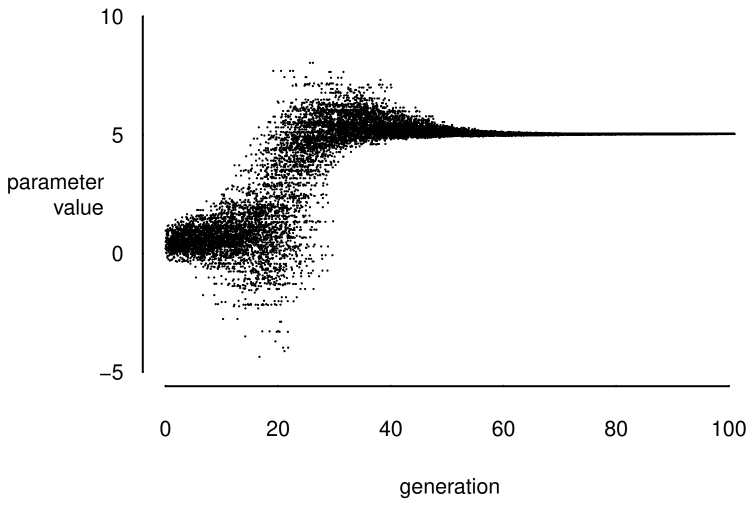
```
> algo <- list(nP = 200L, nG = 100L,
               F = 0.50, CR = 0.99,
               min = c( 0,-10,-10,  0),
               max = c( 1, 10, 10, 10),
               storeSolutions = TRUE, printBar = FALSE)
> ## set up yield curve and put information in Data
> tm <- 1:20                ## times to maturity
> parTRUE <- c(5, 3, 2, 1)  ## true parameters
> yM <- NS(parTRUE, tm)     ## true market yields
> Data <- list(yM = yM, tm = tm, model = NS, ww = 0.1, maxb1 = 4)
> ## solve with DEopt
> sol <- DEopt(OF = OF, algo = algo, Data = Data)
```

```
Differential Evolution.
Best solution has objective function value 0.00822 ;
standard deviation of OF in final population is 6.18e-05 .
```

```
> P <- sol$xlist[[1L]] ## all population matrices
> p1 <- sapply(P, `[`, 1L, TRUE)
```

We plot the values of the first parameter in the population over the course of the optimisation. We see that DE quickly 'escapes' from the initial range.

```
> par(bty = "n", las = 1, mar = c(4,4,0,0),
      ps = 8, tck = 0.001, mgp = c(3, 0.5, 0))
> plot(jitter(rep(seq_len(algo$nG), each = algo$nP), factor = 5),
       p1,
       pch = 21, cex = 0.01, ylim = c(-5,10),
       xlab = "", ylab = "")
> mtext("generation", 1, line = 2)
> mtext("parameter\nvalue", 2, line = 1)
```



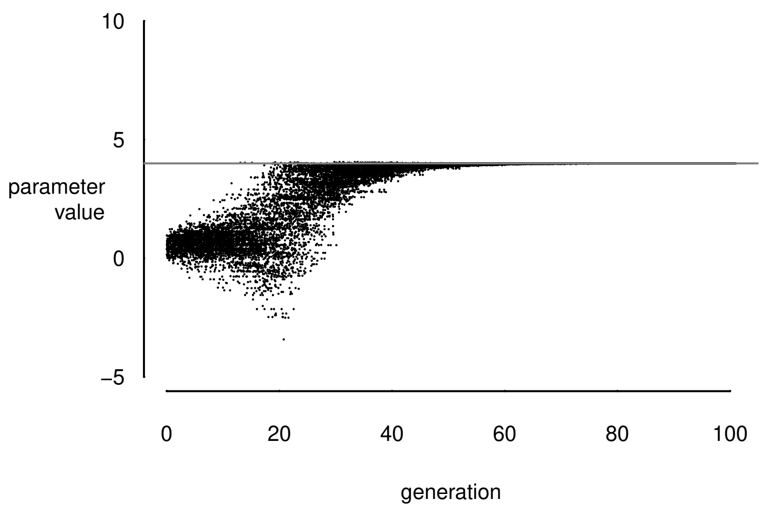
Now suppose we had included a constraint: the parameter should not be greater than 4. (Even though the true parameter is 5.) We adjust the objective function by adding a straightforward penalty. This could certainly be refined, but it is only an example here.

```
> OF2 <- function(par, Data) {
      ## compute model yields
      y <- Data$model(par, Data$tm)

      ## all rates finite?
      validRates <- !any(is.na(y))

      if (validRates) {
          ## any rates negative? if yes, add penalty
          pen1 <- sum(abs(y - abs(y))) * Data$ww

          ## is b1 greater than Data$maxb1? if yes, add penalty
          pen2 <- par[1L] - Data$maxb1
          pen2 <- pen2 + abs(pen2)
          pen2 <- pen2
```

```
        F <- max(abs(y - Data$yM)) + pen1 + pen2
    } else F <- 1e8
    F
  }
> ## solve with DEopt
> sol <- DEopt(OF = OF2, algo = algo, Data = Data)
```

```
Differential Evolution.
Best solution has objective function value 0.298 ;
standard deviation of OF in final population is 4.36e-05 .
```

```
> P <- sol$xlist[[1L]] ### all population matrices
> p1 <- sapply(P, `[`, 1, TRUE)
> par(bty = "n", las = 1, mar = c(4,4,0,0),
      ps = 8, tck = 0.001, mgp = c(3, 0.5, 0))
> plot(jitter(rep(seq_len(algo$nG), each = algo$nP), factor = 5),
       p1,
       pch = 21, cex = 0.01, ylim = c(-5,10),
       xlab = "", ylab = "")
> abline(h = 4, col=grey(0.5))
> mtext("generation", 1, line = 2)
> mtext("parameter\nvalue", 2, line = 1)
```



We see that now the population does not go beyond a value of 4.

4.2. Genetic Algorithm

See ?GAopt after attaching the package.

4.2.1. Description

`GAopt` *Optimisation with a Genetic Algorithm*

4.2.2. Description

A simple Genetic Algorithm for minimising a function.

4.2.3. Usage

```
GAopt (OF, algo = list(), ...)
```

4.2.4. Arguments

`OF` The objective function, to be minimised. See Details.

`algo` A list with the settings for algorithm. See Details and Examples.

`...` Other pieces of data required to evaluate the objective function. See Details and Examples.

4.2.5. Details

The function implements a simple Genetic Algorithm (GA). A GA evolves a collection of solutions (the so-called population), all of which are coded as vectors containing only zeros and ones. (In `GAopt`, solutions are of mode `logical`.) The algorithm starts with randomly-chosen or user-supplied population and aims to iteratively improve this population by mixing solutions and by switching single bits in solutions, both at random. In each iteration, such randomly-changed solutions are compared with the original population and better solutions replace inferior ones. In `GAopt`, the population size is kept constant.

GA language: iterations are called generations; new solutions are called offspring or children (and the existing solutions, from which the children are created, are parents); the objective function is called a fitness function; mixing solutions is a crossover; and randomly changing solutions is called mutation. The choice which solutions remain in the population and which ones are discarded is called selection. In `GAopt`, selection is pairwise: a given child is compared with a given parent; the better of the two is kept. In this way, the best solution is automatically retained in the population.

To allow for constraints, the evaluation works as follows: after new solutions are created, they are (i) repaired, (ii) evaluated through the objective function,

(iii) penalised. Step (ii) is done by a call to `OF`; steps (i) and (iii) by calls to `algo$repair` and `algo$pen`. Step (i) and (iii) are optional, so the respective functions default to `NULL`. A penalty can also be directly written in the `OF`, since it amounts to a positive number added to the 'clean' objective function value; but a separate function is often clearer. A separate penalty function is advantagous if either only the objective function or only the penalty function can be vectorised.

Conceptually a GA consists of two loops: one loop across the generations and, in any given generation, one loop across the solutions. This is the default, controlled by the variables `algo$loopOF`, `algo$loopRepair` and `algo$loopPen`, which all default to `TRUE`. But it does not matter in what order the solutions are evaluated (or repaired or penalised), so the second loop can be vectorised. The respective `algo$loopFun` must then be set to `FALSE`. (See also the examples for `DEopt` and `PSopt`.)

The evaluation of the objective function in a given generation can even be distributed. For this, an argument `algo$methodOF` needs to be set; see below for details (and Schumann, 2011, for examples).

All objects that are passed through `...` will be passed to the objective function, to the repair function and to the penalty function.

The list `algo` contains the following items:

`nB` number of bits per solution. Must be specified.

`nP` population size. Defaults to 50. Using default settings may not be a good idea.

`nG` number of iterations ('generations'). Defaults to 300. Using default settings may not be a good idea.

`crossover` The crossover method. Default is `"onePoint"`; also possible is "uniform".

`prob` The probability for switching a single bit. Defaults to 0.01; typically a small number.

`pen` a penalty function. Default is `NULL` (no penalty).

`repair` a repair function. Default is `NULL` (no repairing).

`initP` optional: the initial population. A logical matrix of size `length(algo$nB)` times `algo$nP`, or a function that creates such a matrix. If a function, it must take no arguments. If `mode(mP)` is not `logical`, then `storage.mode(mP)` will be tried (and a warning will be issued).

`loopOF` logical. Should the `OF` be evaluated through a loop? Defaults to `TRUE`.

`loopPen` logical. Should the penalty function (if specified) be evaluated through a loop? Defaults to `TRUE`.

`loopRepair` logical. Should the repair function (if specified) be evaluated through a loop? Defaults to `TRUE`.

`methodOF` `loop` (the default), `vectorised`, `snow` or `multicore`. Setting `vectorised` is equivalent to having `algo$loopOF` set to `FALSE` (and `methodOF` overrides `loopOF`). `snow` and `multicore` use functions `clusterApply` and `mclapply`, respectively. For `snow`, an object `algo$cl` needs to be specified (see below). For `multicore`, optional arguments can be passed through `algo$mc.control` (see below).

`cl` a cluster object or the number of cores. See documentation of package `parallel`.

`mc.control` a list of named elements; optional settings for `mclapply` (for instance,

```
list(mc.set.seed = FALSE))
```

`printDetail` If `TRUE` (the default), information is printed.

`printBar` If `TRUE` (the default), a `txtProgressBar` is printed.

`storeF` If `TRUE` (the default), the objective function values for every solution in every generation are stored and returned as matrix `Fmat`.

`storeSolutions` If `TRUE`, the solutions (ie, binary strings) in every generation are stored and returned as a list `P` in list `xlist` (see Value section below). To check, for instance, the solutions at the end of the `i`th generation, retrieve `xlist[[c(1L, i)]]`. This will be a matrix of size `algo$nB` times `algo$nP`.

`classify` Logical; default is `FALSE`. If `TRUE`, the result will have a class attribute `TAopt` attached. This feature is experimental: the supported methods may change without warning.

4.2.6. Value

A list:

`xbest` the solution (the best member of the population)

`OFvalue` objective function value of best solution

`popF` a vector. The objective function values in the final population.

`Fmat` if `algo$storeF` is `TRUE`, a matrix of size `algo$nG` times `algo$nP` containing the objective function values of all solutions over the generations; else `NA`

`xlist` if `algo$storeSolutions` is `TRUE`, a list that contains a list `P` of matrices and a matrix `initP` (the initial solution); else `NA`.

`initial.state` the value of `.Random.seed` when the function was called.

4.2.7. Diagnostics

4.3. Particle Swarm Optimisation

See `?PSopt` after attaching the package.

4.3.1. Description

`PSopt` *Particle Swarm Optimisation*

4.3.2. Description

The function implements Particle Swarm Optimisation.

4.3.3. Usage

```
PSopt(OF, algo = list(), ...)
```

4.3.4. Arguments

`OF` the objective function to be minimised. See Details.

`algo` a list with the settings for algorithm. See Details and Examples.

`...` pieces of data required to evaluate the objective function. See Details.

4.3.5. Details

The function implements Particle Swarm Optimisation (PS); see the references for details on the implementation. PS is a population-based optimisation heuristic. It develops several solutions (a 'population') over a number of iterations. PS is directly applicable to continuous problems since the population is stored in real-valued vectors. In each iteration, a solution is updated by adding another vector called velocity. Think of a solution as a position in the search space, and of velocity as the direction into which this solution moves. Velocity changes over the course of the optimization: it is biased towards the best solution found by the particular solution and the best overall solution. The algorithm stops after a fixed number of iterations.

To allow for constraints, the evaluation works as follows: after a new solution is created, it is (i) repaired, (ii) evaluated through the objective function,

(iii) penalised. Step (ii) is done by a call to `OF`; steps (i) and (iii) by calls to `algo$repair` and `algo$pen`. Step (i) and (iii) are optional, so the respective functions default to `NULL`. A penalty can also be directly written in the `OF`, since it amounts to a positive number added to the 'clean' objective function value. It can be advantageous to write a separate penalty function if either only the objective function or only the penalty function can be vectorised. (Constraints can also be added without these mechanisms. Solutions that violate constraints can, for instance, be mapped to feasible solutions, but without actually changing them. See Maringer and Oyewumi, 2007, for an example with Differential Evolution.)

Conceptually, PS consists of two loops: one loop across the iterations and, in any given generation, one loop across the solutions. This is the default, controlled by the variables `algo$loopOF`, `algo$loopRepair`, `algo$loopPen` and `loopChangeV` which all default to `TRUE`. But it does not matter in what order the solutions are evaluated (or repaired or penalised), so the second loop can be vectorised. Examples are given in the vignettes and in the book. The respective `algo$loopFun` must then be set to `FALSE`.

The objective function, the repair function and and the penalty function will be called as `fun(solution, ...)`.

The list `algo` contains the following items:

`nP` population size. Defaults to 100. Using default settings may not be a good idea.

`nG` number of iterations. Defaults to 500. Using default settings may not be a good idea.

`c1` the weight towards the individual's best solution. Typically between 0 and 2; defaults to 1. Using default settings may not be a good idea. In some cases, even negative values work well: the solution is then driven off its past best position. For 'simple' problems, setting `c1` to zero may work well: the population moves then towards the best overall solution.

`c2` the weight towards the populations's best solution. Typically between 0 and 2; defaults to 1. Using default settings may not be a good idea. In some cases, even negative values work well: the solution is then driven off the population's past best position.

`iner` the inertia weight (a scalar), which reduces velocity. Typically between 0 and 1. Default is 0.9.

`initV` the standard deviation of the initial velocities. Defaults to 1.

`maxV` the maximum (absolute) velocity. Setting limits to velocity is sometimes called velocity clamping. Velocity is the change in a given solution in a given iteration. A maximum velocity can be set so to prevent unreasonable velocities ('overshooting'): for instance, if a decision variable may lie between 0 and 1, then an absolute velocity much greater than 1 makes rarely sense.

`min`, `max` vectors of minimum and maximum parameter values. The vectors `min` and `max` are used to determine the dimension of the problem and to randomly initialise the population. Per default, they are no constraints: a solution may well be outside these limits. Only if `algo$minmaxConstr` is `TRUE` will the algorithm repair solutions outside the `min` and `max` range.

`minmaxConstr` if `TRUE`, `algo$min` and `algo$max` are considered constraints. Default is `FALSE`.

`pen` a penalty function. Default is `NULL` (no penalty).

`repair` a repair function. Default is `NULL` (no repairing).

`changeV` a function to change velocity. Default is `NULL` (no change). This function is called before the velocity is added to the current solutions; it can be used to impose restrictions like changing only a number of decision variables.

`initP` optional: the initial population. A matrix of size `length(algo$min)` times `algo$nP`, or a function that creates such a matrix. If a function, it should take no arguments.

`loopOF` logical. Should the `OF` be evaluated through a loop? Defaults to `TRUE`.

`loopPen` logical. Should the penalty function (if specified) be evaluated through a loop? Defaults to `TRUE`.

`loopRepair` logical. Should the repair function (if specified) be evaluated through a loop? Defaults to `TRUE`.

`loopChangeV` logical. Should the `changeV` function (if specified) be evaluated through a loop? Defaults to `TRUE`.

`printDetail` If `TRUE` (the default), information is printed. If an integer `i` greater then one, information is printed at very `i`th iteration.

`printBar` If `TRUE` (the default), a `txtProgressBar` (from package utils) is printed).

`storeF` If `TRUE` (the default), the objective function values for every solution in every generation are stored and returned as matrix `Fmat`.

`storeSolutions` default is `FALSE`. If `TRUE`, the solutions (ie, decision variables) in every generation are stored as lists `P` and `Pbest`, both stored in the list `xlist` which the function returns. To check, for instance, the solutions at the end of the `i`th iteration, retrieve `xlist[[c(1L, i)]]`; the best solutions at the end of this iteration are in `xlist[[c(2L, i)]]`. `P[[i]]` and `Pbest[[i]]` will be matrices of size `length(algo$min)` times `algo$nP`.

`classify` Logical; default is `FALSE`. If `TRUE`, the result will have a class attribute `TAopt` attached. This feature is experimental: the supported methods may change without warning.

`drop` Default is `TRUE`. If `FALSE`, the dimension is not dropped from a single solution when it is passed to a function. (That is, the function will receive a single-column matrix.)

4.3.6. Value

Returns a list:

`xbest` the solution

`OFvalue` objective function value of best solution

`popF` a vector: the objective function values in the final population

`Fmat` if `algo$storeF` is `TRUE`, a matrix of size `algo$nG` times `algo$nP`. Each column contains the best objective function value found by the particular solution.

`xlist` if `algo$storeSolutions` is `TRUE`, a list that contains two lists `P` and `Pbest` of matrices, and a matrix `initP` (the initial solution); else `NA`.

`initial.state` the value of `.Random.seed` when `PSopt` was called.

4.3.7. Diagnostics

4.4. Vectorisation and parallel evaluation of the population

When we look at heuristics in principle, we manipulate and evolve solutions through functions: new solutions are created as functions of existing solutions; solutions are evaluated through the objective function; whether new solutions are accepted is a function of (typically) the quality of the new solutions; and so on. This gives us much flexibility in how solutions are represented; in essence, any data structure (eg, a graph) could be directly handled, provided we define appropriate functions to work with it.

Yet a number of (quite successful) heuristics, such as Differential Evolution (DE) or Particle Swarm (PS), prescribe precisely how solutions are represented and manipulated. In fact, these specific prescriptions essentially define those heuristics. For DE and PS, for instance, a solution is a numeric vector; new solutions are created as (noisy) linear combinations of existing solutions. While this reduces the algorithms' flexibility, it allows for a simpler (and more efficient) generic implementation.

Let us be more concrete here. Since both DE and PS represent solutions as numerical vectors, a natural way to store the solutions is a matrix P. In this matrix, each column is one solution; each row represents a specific decision variable. When we compute the objective function values for these solutions, a straightforward strategy is to loop over the columns of P and call the objective function for each solution. In this case, the objective function should take as arguments a single numeric vector (and possibly other data passed through `...`); the function should return a single number.

In somes cases, however, it may be preferable to actually write the objective function such that it expects the whole population as an argument, and then returns a vector of objective function values. To accommodate this behaviour, the functions `DEopt`, `GAopt` and `PSopt` have settings `algo$loopFun`, in which '`Fun`' can be '`OF`' for objective function, but also, for instance, '`repair`'. These settings default to `TRUE`, so the functions will loop over the solutions. When such a loop-setting is `FALSE`, the respective function receives the whole population as an argument.

In the next section we give three examples when this 'evaluation in one step' can be advantegeous. The functions `DEopt`, `GAopt` and `PSopt` allow to implement the objective function (and also repair and penalty functions) like this. For more details and examples, see Gilli et al. (2011).

4.4.1. Examples for vectorised computations

We give three cases in which vectorised computations are preferred to loops. (Because the computations become faster.)

A test function

As an example, we use the Rosenbrock function, given by

$$\sum_{i=1}^{n-1}\Big(100(x_{i+1}-x_i^2)^2+(1-x_i)^2\Big)\,.$$

This test function is available in the package as the function `tfRosenbrock` (see `?testFunctions`). The Rosenbrock function has a minimum of zero when all elements of x are one. (In higher dimensions, this minimum may not be unique.)

```
> tfRosenbrock
```

```
function(x) {
    n <- length(x)
    xi <- x[seq_len(n-1L)]
    sum(100 * (x[2L:n] - xi * xi)^2 + (1 - xi)^2)
}
<bytecode: 0x5651826a9678>
<environment: namespace:NMOF>
```

So we define the objective function `OF` and test it with the known solution.

```
> OF <- tfRosenbrock     ## see ?testFunctions
> size <- 5L             ## set dimension
> x <- rep.int(1, size)  ## the known solution ...
> OF(x)                  ## ... should give zero
```

```
[1] 0
```

We set the parameters for DEopt. Note that in this example we are only concerned with the speed of the computation, so the actual settings do not matter so much.

```
> algo <- list(printBar = FALSE,
               nP = 50L,
               nG = 500L,
               F = 0.6,
               CR = 0.9,
               min = rep(-100, size),
               max = rep( 100, size))
```

Suppose we have several solutions, put into a matrix such that every column is one solution. Then we could rewrite the function like so:

```
> ## a vectorised OF: works only with *matrix* x
> OF2 <- function(x) {
      n <- dim(x)[1L]
      xi <- x[1L:(n - 1L), ]
      colSums(100 * (x[2L:n, ] - xi * xi)^2 + (1 - xi)^2)
  }
```

We can test it by creating a number of random solutions.

```
> x <- matrix(rnorm(size * algo$nP), size, algo$nP)
> c(OF(x[ ,1L]), OF(x[ ,2L]), OF(x[ ,3L]))
```

```
[1] 3769  157  326
```

```
> OF2(x)[1L:3L]  ## should give the same result
```

```
[1] 3769  157  326
```

```
> all.equal(OF2(x)[1L:3L], c(OF(x[ ,1L]), OF(x[ ,2L]), OF(x[ ,3L])))
```

```
[1] TRUE
```

As pointed out above, DEopt either can loop over the solutions, or it can evaluate the whole population in one step. The first behaviour is triggered when algo$loopOF is set to TRUE, which is the default setting.

When we want to use OF2, we need to set algo$loopOF to FALSE.

```
> set.seed(1223445)
> (t1 <- system.time(sol <- DEopt(OF = OF, algo = algo)))
```

```
Differential Evolution.
Best solution has objective function value 3.25e-16 ;
standard deviation of OF in final population is 7.07e-16 .

   user  system elapsed
  0.109   0.000   0.109
```

```
> algo$loopOF <- FALSE
> set.seed(1223445)
> (t2 <- system.time(sol2 <- DEopt(OF = OF2, algo = algo)))
```

```
Differential Evolution.
Best solution has objective function value 3.25e-16 ;
standard deviation of OF in final population is 7.07e-16 .

   user  system elapsed
  0.025   0.000   0.025
```

We can compare the solutions, and compute the speedup.

```
> sol$OFvalue    ## both should be zero (with luck)
```

```
[1] 3.25e-16
```

```
> sol2$OFvalue
```

```
[1] 3.25e-16
```

```
> t1[[3L]]/t2[[3L]]  ## speedup
```

```
[1] 4.36
```

Portfolio optimisation

A portfolio can be described by a weight vector w. Given a variance–covariance matrix Σ, we can calculate the variance of such a portfolio like so:

$$w'\Sigma w\,.$$

Suppose now that we have a number of solutions, and we collect them in a matrix W, such that every column is one solution w. One approach would be now to loop over the columns, and for every column compute the variance. But we can use a one-line computation as well: the variances of the solutions are given by

$$\mathsf{diag}(W'\Sigma W)\,.$$

This can be written consisely, but we are unnessarily computing the off-diagonal elements of the resulting matrix. One solution, then, is to recognise that $\mathsf{diag}(W'\Sigma W)$ is equivalent to

$$\iota' \underbrace{\overbrace{\Sigma W}^{\substack{\text{matrix}\\ \text{multiplication}}} \quad W}_{\substack{\text{elementwise}\\ \text{multiplication}}}$$

which is consise and more efficient. The following example illustrates this. We start by setting up a variance–covariance matrix `Sigma` and a population `W`. (We would not need to include the budget constraint here since we are only interested in computing time.)

```
> na <- 100L  ## number of assets
> np <- 100L  ## size of population
> trials <- seq_len(100L)  ## for speed test
> ## a covariance matrix
> Sigma <- array(0.7, dim = c(na, na)); diag(Sigma) <- 1
> ## set up population
> W <- array(runif(na * np), dim = c(na, np))
> ## budget constraint
> scaleFun <- function(x) x/sum(x); W <- apply(W, 2L, scaleFun)
```

Now we can test the three variants described above.

```
> ## variant 1
> t1 <- system.time({
      for (i in trials) {
          res1 <- numeric(np)
          for (j in seq_len(np)) {
              w <- W[ ,j]
              res1[j] <- w %*% Sigma %*% w
          }
```

```
        }
  })
> ## variant 2
> t2 <- system.time({
        for (i in trials) res2 <- diag(t(W) %*% Sigma %*% W)
  })
> ## variant 3
> t3 <- system.time({
        for (i in trials) res3 <- colSums(Sigma %*% W * W)
  })
```

All three computations should give the same result.

```
> all.equal(res1,res2)
```

```
[1] TRUE
```

```
> all.equal(res2,res3)
```

```
[1] TRUE
```

But the first variant requires more code than the others, and it is slower.

```
> t1  ##  speedup for variant 1
```

```
   user  system elapsed
  0.933   0.865   0.161
```

```
> t2  ##  speedup for variant 2
```

```
   user  system elapsed
  0.212   0.069   0.024
```

```
> t3  ##  speedup for variant 3
```

```
   user  system elapsed
  0.082   0.100   0.015
```

Residuals in a linear model

We wish to compute the residuals r of a linear model, $y = X\theta + r$. Suppose we have a population Θ of solution vectors; each column in Θ is one particular solution θ. Now, as before we could compute

$$\mathrm{r} = y - X\theta_i$$

for every $i \in \{1, \ldots, \text{population size}\}$. Alternatively, we may replace the loop over those solutions with the computation

$$\mathrm{R} = y\iota' - X\Theta\,,$$

in which R is the matrix of residuals.

Again, an example. As before, we set up random data and a random population of solutions.

```
> n <- 100L   ## number of observation
> p <- 5L     ## number of regressors
> np <- 100L  ## population size
> trials <- seq_len(1000L)
> ## random data
> X <- array(rnorm(n * p), dim = c(n, p))
> y <- rnorm(n)
> ## random population
> Theta <- array(rnorm(p * np), dim = c(p, np))
> ## empty residuals matrix
> R1 <- array(NA, dim = c(n, np))
```

Now we can compare both variants.

```
> system.time({
    for (i in trials)
    for (j in seq_len(np))
    R1[ ,j] <- y - X %*% Theta[ ,j]
  })
```

```
   user  system elapsed
  0.193   0.000   0.193
```

```
> system.time({
    for (i in trials)
    R2 <- y - X %*% Theta
  })
```

```
   user  system elapsed
  0.035   0.004   0.039
```

Note that we have not explicitly computed $y\iota'$ but have used R's recycling rule.

We check whether we actually obtain the same result.

```
> all.equal(R1, R2)  ## ... should be TRUE
```

```
[1] TRUE
```

See Chapter 14 in GMS.

4.4.2. Distributed computations

5. Other functions

In this chapter we briefly review a number of functions that were added to the NMOF package after the first edition of GMS had been published.

5.1. Bracketing

There are two approaches to numerical root finding. The first strategy is to approximate the function in question by a simpler function whose zero we can compute; doing this repeatedly should bring us closer to the desired root. An example for this approach is Newton's method.

The second approach uses the fact that if the sign of a function is different when evaluated at points *a* and *b*, then there must be at least one root between *a* and *b* (given the function is well-behaved in that interval). Making this interval smaller and smaller will bring us arbitrarily close a root.

The function `bracketing` uses the second approach, though it does not refine the interval. The function was added in NMOF version 0.16-0; it supports distributed evaluation of `fun` through package `parallel` (originally, `multicore` (Urbanek, 2011) and `snow` (Tierney et al., 2011)).

```
> testFun <- function(x) {
      Sys.sleep(0.1)  ## wasting time...
      cos(1/x^2)
  }
> system.time(sol1 <- bracketing(testFun,
                                 interval = c(0.3, 0.9),
                                 n = 100L))
> system.time(sol2 <- bracketing(testFun,
                                 interval = c(0.3, 0.9),
                                 n = 100L,
                                 cl = 2))
> all.equal(sol1, sol2)
```

`bracketing` *Zero-Bracketing*

5.1.1. Description

Bracket the zeros (roots) of a univariate function

5.1.2. Usage

```
bracketing(fun, interval, ...,
           lower = min(interval), upper = max(interval),
           n = 20L,
           method = c("loop", "vectorised", "multicore", "snow"),
           mc.control = list(), cl = NULL)
```

5.1.3. Arguments

`fun` a univariate function; it will be called as `fun(x, ...)` with `x` being a numeric vector

`interval` a numeric vector, containing the end-points of the interval to be searched

`...` further arguments passed to `fun`

`lower` lower end-point. Ignored if `interval` is specified.

`upper` upper end-point. Ignored if `interval` is specified.

`n` the number of function evaluations. Must be at least 2 (in which case `fun` is evaluated only at the end-points); defaults to 20.

`method` can be `loop` (the default), `vectorised`, `multicore` or `snow`. See Details.

`mc.control` a list containing settings that will be passed to `mclapply` if `method` is `multicore`. Must be a list of named elements. See the documentation of `mclapply` in package parallel.

`cl` default is `NULL`. If `method` is `snow`, this must be a cluster object or an integer (the number of cores to be used). See the documentation of packages parallel and snow.

5.1.4. Details

`bracketing` evaluates `fun` at equal-spaced values of `x` between (and including) `lower` and `upper`. If the sign of `fun` changes between two consecutive `x`-values, `bracketing` reports these two `x`-values as containing ('bracketing') a root. There is no guarantee that there is only one root within a reported interval. `bracketing` will not narrow the chosen intervals.

The argument `method` determines how `fun` is evaluated. Default is `loop`. If `method` is `"vectorised"`, `fun` must be written such that it can be evaluated for a vector x (see Examples). If method is multicore, function mclapply from package parallel is used. Further settings for mclapply can be passed through the list mc.control. If multicore is chosen but the functionality is not available (eg, currently on Windows), then method will be set to loop and a warning is issued. If method is snow, function clusterApply from package parallel is used. In this case, the argument cl must either be a cluster object (see the documentation of clusterApply) or an integer. If an integer, a cluster will be set up via makeCluster(c(rep("localhost", cl)), type = "SOCK"), and stopCluster is called when the function is exited. If snow is chosen but the package is not available or cl is not specified, then method will be set to loop and a warning is issued. In case that cl is a cluster object, stopCluster will not be called automatically.

5.1.5. Value

A numeric matrix with two columns, named *lower* and *upper*. Each row contains one interval that contains at least one root. If no roots were found, the matrix has zero rows.

5.2. Grid Search

The function gridSearch was added in version 0.14-0. gridSearch allows to distribute the evaluation of the objective function through package parallel (originally, multicore (Urbanek, 2011) and snow (Tierney et al., 2011)).

gridSearch *Grid Search*

5.2.1. Description

Evaluate a function for a given list of arguments.

5.2.2. Usage

```
gridSearch(fun, levels, ..., lower, upper, npar = 1L, n = 5L,
           printDetail = TRUE,
           method = NULL,
           mc.control = list(), cl = NULL,
           keepNames = FALSE, asList = FALSE)
```

5.2.3. Arguments

`fun` a function of the form `fun(x, ...)`, with `x` being a numeric vector or a list

`levels` a list of levels for the arguments (see Examples)

`...` objects passed to `fun`

`lower` a numeric vector. Ignored if levels are explicitly specified.

`upper` a numeric vector. Ignored if levels are explicitly specified.

`npar` the number of parameters. Must be supplied if `lower` and `upper` are to be expanded; see Details. Ignored when `levels` are explicitly specified, or when `lower`/`upper` are used and at least one has length greater than one. See Examples.

`n` the number of levels. Default is 5. Ignored if levels are explicitly specified.

`printDetail` print information on the number of objective function evaluations

`method` can be `loop` (the default), `multicore` or `snow`. See Details.

`mc.control` a list containing settings that will be passed to `mclapply` if `method` is `multicore`. Must be a list of named elements; see the documentation of `mclapply` in parallel.

`cl` default is `NULL`. If method `snow` is used, this must be a cluster object or an integer (the number of cores).

`keepNames logical`: should the names of `levels` be kept?

`asList` does `fun` expect a list? Default is `FALSE`.

5.2.4. Details

A grid search can be used to find 'good' parameter values for a function. In principle, a grid search has an obvious deficiency: as the length of `x` (the first argument to `fun`) increases, the number of necessary function evaluations grows exponentially. Note that `gridSearch` will not warn about an unreasonable number of function evaluations, but if `printDetail` is `TRUE` it will print the required number of function evaluations.

In practice, grid search is often better than its reputation. If a function takes only a few parameters, it is often a reasonable approach to find 'good' parameter values.

The function uses the mechanism of `expand.grid` to create the list of parameter combinations for which `fun` is evaluated; it calls `lapply` to evaluate `fun` if `method == "loop"` (the default).

If method is multicore, then function mclapply from package parallel is used. Further settings for mclapply can be passed through the list mc.control. If multicore is chosen but the functionality is not available, then method will be set to loop and a warning is issued. If method == "snow", the function clusterApply from package parallel is used. In this case, the argument cl must either be a cluster object (see the documentation of clusterApply) or an integer. If an integer, a cluster will be set up via makeCluster(c(rep("localhost", cl)), type = "SOCK") (and stopCluster is called when the function is exited). If snow is chosen but not available or cl is not specified, then method will be set to loop and a warning is issued.

5.2.5. Value

A list.

minfun the minimum of fun.

minlevels the levels that give this minimum.

values a list. All the function values of fun.

levels a list. All the levels for which fun was evaluated.

We start with a simple example. We have a function of two variables, x_1 and x_2:

$$f(x_1, x_2) = x_1 + x_2^2 \,. \tag{5.1}$$

This function can be computed very quickly for given x-values. To demonstrate the use of distributed evaluation, we slow it down.

```
> testFun  <- function(x) {
      Sys.sleep(0.1)  ## wasting time...
      x[1L] + x[2L]^2
  }
```

Now we can evaluate f for, say, $1 \leq x_1 \leq 5$ and $3 \leq x_2 \leq 5$, with five different levels.

```
> lower <- c(1, 3); upper <- 5; n <- 5L
> system.time(sol1 <- gridSearch(fun = testFun,
                                 lower = lower, upper = upper,
                                 n = n, printDetail = TRUE))
```

With those settings gridSearch has evaluated f for all combinations of these levels:

```
> seq(from = 1, to = 5, length.out= n)  ## x_1
```

```
[1] 1 2 3 4 5
```

```
> seq(from = 3, to = 5, length.out= n)  ## x_2
```

```
[1] 3.0 3.5 4.0 4.5 5.0
```

For the given function the minimum is at `c(1,3)`, which is exactly what `gridSearch` returns.

```
> sol1$minfun
```

```
[1] 10
```

```
> sol1$minlevels
```

```
[1] 1 3
```

To use a snow cluster, call `gridSearch` with arguments `method` and `cl`.

```
> system.time(sol2 <- gridSearch(fun = testFun,
                                 lower = lower,
                                 upper = upper,
                                 n = n, printDetail = FALSE,
                                 cl = 2L))         ## with 2 cores
> all.equal(sol1, sol2)
```

5.3. Integration of Gauss-type

The functions `xwGauss` and `changeInterval` were added in version 0.17-0.

`xwGauss` *Integration of Gauss-type*

5.3.1. Description

Compute nodes and weights for Gauss integration.

5.3.2. Usage

```
xwGauss(n, method = "legendre")
changeInterval(nodes, weights, oldmin, oldmax, newmin, newmax)
```

5.3.3. Arguments

`n` number of nodes

`method` character. default is `"legendre"`; also possible are `"laguerre"` and `"hermite"`

`nodes` the nodes (a numeric vector)

`weights` the weights (a numeric vector)

`oldmin` the minimum of the interval (typically as tabulated)

`oldmax` the maximum of the interval (typically as tabulated)

`newmin` the desired minimum of the interval

`newmax` the desired maximum of the interval

5.3.4. Details

`xwGauss` computes nodes and weights for integration for the interval -1 to 1. It uses the method of Golub and Welsch (1969).

`changeInterval` is a utility that transforms nodes and weights to an arbitrary interval.

5.3.5. Value

a list with two elements

`weights` a numeric vector

`nodes` a numeric vector

5.4. Option pricing

5.4.1. Vanilla options

The functions `vanillaOptionEuropean`, `vanillaOptionAmerican` and `vanillaOptionImpliedVol` were added in package version 0.25-9.

`vanillaOptionEuropean` *Pricing Plain-Vanilla (European and American) and Barrier Options (European)*

5.4.2. Description

Functions to calculate the theoretical prices and (some) Greeks for plain-vanilla and barrier options.

5.4.3. Usage

```
vanillaOptionEuropean(S, X, tau, r, q, v, tauD = 0, D = 0,
                      type = "call", greeks = TRUE,
                      model = NULL, ...)
vanillaOptionAmerican(S, X, tau, r, q, v, tauD = 0, D = 0,
                      type = "call", greeks = TRUE, M = 101)

vanillaOptionImpliedVol(exercise = "european", price, S, X, tau, r,
                        q = 0, tauD = 0, D = 0, type = "call",
                        M = 101,
                        uniroot.control = list(), uniroot.info = FALSE)

barrierOptionEuropean(S, X, H, tau, r, q = 0, v, tauD = 0, D = 0,
                      type = "call",
                      barrier.type = "downin",
                      rebate = 0,
                      greeks = FALSE,
                      model = NULL, ...)
```

5.4.4. Arguments

`S` spot

`X` strike

`H` barrier

`tau` time-to-maturity in years

`r` risk-free rate

`q` continuous dividend yield, see Details.

`v` variance (volatility squared)

`tauD` vector of times-to-dividends in years. Only dividends with `tauD` greater than zero and not greater than `tau` are kept.

`D` vector of dividends (in currency units); default is no dividends.

`type` call or put; default is call.

`barrier.type` string: combination of `up`/`down` and `in`/`out`, such as `downin`

`rebate` currently not implemented

`greeks` compute Greeks? Defaults to `TRUE`. But see Details for American options.

`model` what model to use to value the option. Default is `NULL` which is equivalent to `bsm`.

`...` parameters passed to pricing model

`M` number of time steps in the tree

`exercise` `european` (default) or `american`

`price` numeric; the observed price to be recovered through choice of volatility.

`uniroot.control` A list. If there are elements named `interval`, `tol` or `maxiter`, these are passed to `uniroot`. Any other elements of the list are ignored.

`uniroot.info` logical; default is `FALSE`. If `TRUE`, the function will return the information returned by `uniroot`. See paragraph Value below.

5.4.5. Details

For European options the formula of Messrs Black, Scholes and Merton is used. It can be used for equities (set `q` equal to the dividend yield), futures (Black, 1976; set `q` equal to `r`), currencies (Garman and Kohlhagen, 1983; set `q` equal to the foreign risk-free rate). For future-style options (e.g. options on the German Bund future), set `q` and `r` equal to zero.

The Greeks are provided in their raw ('textbook') form with only one exception: Theta is made negative. For practical use, the other Greeks are also typically adjusted: Theta is often divided by 365 (or some other yearly day count); Vega and Rho are divided by 100 to give the sensitivity for one percentage-point move in volatility/the interest rate. Raw Gamma is not much use if not adjusted for the actual move in the underlier.

For European options the Greeks are computed through the respective analytic expressions. For American options only Delta, Gamma and Theta are computed because they can be directly obtained from the binomial tree; other Greeks need to be computed through a finite difference (see Examples).

For the European-type options, the function understands vectors of inputs, except for dividends. American options are priced via a Cox-Ross-Rubinstein tree; no vectorisation is implemented here.

The implied volatility is computed with `uniroot` from the stats package (the default search interval is `c(0.00001, 2)`; it can be changed through `uniroot.control`).

Dividends (`D`) are modelled via the escrowed-dividend model.

5.4.6. Value

Returns the price (a numeric vector of length one) if `greeks` is `FALSE`, else returns a list.

5.4.7. Note

If `greeks` is `TRUE`, the function will return a list with named elements (`value`, `delta` and so on). Prior to version 0.26-3, the first element of this list was named `price`.

5.4.8. Merton's jump–diffusion model

The function `callMerton` was added in package version 0.31-0.

`callMerton` *Price of a European Call under Merton's Jump–Diffusion Model*

5.4.9. Description

Computes the price of a European Call under Merton's jump–diffusion model (and the equivalent Black–Scholes–Merton volatility)

5.4.10. Usage

```
callMerton(S, X, tau, r, q, v, lambda, muJ, vJ, N, implVol = FALSE)
```

5.4.11. Arguments

`S` current stock price

`X` strike price

`tau` time to maturity

`r` risk-free rate

`q` dividend rate

`v` variance

`lambda` jump intensity

`muJ` mean jump-size

`vJ` variance of log jump-size

`N` The number of jumps. See Details.

`implVol` compute equivalent Black–Scholes–Merton volatility? Default is `FALSE`.

5.4.12. Details

The function computes the value of a plain-vanilla European call under Merton's jump–diffusion model. Put values can be computed through put–call-parity (see `putCallParity`). If `implVol` is `TRUE`, the function also computes the implied volatility necessary to obtain the same price under Black–Scholes–Merton. The implied volatility is computed with `uniroot` from the stats package.

Note that the function takes variances as inputs (not volatilities).

The number of jumps `N` typically can be set 10 or 20. (Just try to increase `N` and see how the results change.)

5.4.13. Value

Returns the value of the call (numeric) or, if `implVol` is `TRUE`, a list of the value and the implied volatility.

5.4.14. Pricing with the characteristic function

The package has always contained the function `callHestoncf` (see GMS, pages 520–521). The function `callCF` was added in version 0.21-0; it allows to pass a user-defined characteristic function. As examples, characteristic functions for Black–Scholes–Merton, Merton's jump–diffusion model, the Bates model, the Heston model and Variance-Gamma were added.

As an example, we use Black–Scholes–Merton. The characteristic function can be coded as follows.

```
> cfBSM
```

```
function(om, S, tau, r, q, v)
    exp(1i * om * log(S) + 1i * tau * (r - q) * om -
            0.5 * tau * v * (1i * om + om ^ 2))
<bytecode: 0x5651839370f8>
<environment: namespace:NMOF>
```

So now we can compare the results of different pricing methods.

```
> S <- 100    ## spot
> X <- 100    ## strike
> tau <- 1    ## time-to-maturity
> r <- 0.02   ## interest rate
> q <- 0.08   ## dividend rate
> v <- 0.2    ## volatility
> ## the closed-form solution
> callBSM <- function(S,X,tau,r,q,v) {
      d1 <- (log(S/X) + (r - q + v^2 / 2)*tau) / (v*sqrt(tau))
      d2 <- d1 - v*sqrt(tau)
      S * exp(-q * tau) * pnorm(d1) -  X * exp(-r * tau) * pnorm(d2)
  }
> callBSM(S,X,tau,r,q,v)
```

```
[1] 5.06
```

```
> ## with the characteristic function
> callCF(cf = cfBSM, S = S, X = X, tau = tau, r = r, q = q,
         v = v^2,  ## variance, not vol
         implVol = TRUE)
```

```
$value
[1] 5.06

$impliedVol
[1] 0.2
```

`callCF` *Price a Plain-Vanilla Call with the Characteristic Function*

5.4.15. Description

Price a European plain-vanilla call with the characteric function.

5.4.16. Usage

```
callCF(cf, S, X, tau, r, q = 0, ...,
       implVol = FALSE, uniroot.control = list(), uniroot.info = FALSE)
cfBSM(om, S, tau, r, q, v)
cfMerton(om, S, tau, r, q, v, lambda, muJ, vJ)
cfBates(om, S, tau, r, q, v0, vT, rho, k, sigma, lambda, muJ, vJ)
cfHeston(om, S, tau, r, q, v0, vT, rho, k, sigma)
cfVG(om, S, tau, r, q, nu, theta, sigma)
```

5.4.17. Arguments

`cf` characteristic function

`S` spot

`X` strike

`tau` time to maturity

`r` the interest rate

`q` the dividend rate

`...` arguments passed to the characteristic function

`implVol` logical: compute implied vol?

`uniroot.control` A list. If there are elements named `interval`, `tol` or `maxiter`, these are passed to `uniroot`. Any other elements of the list are ignored.

`uniroot.info` logical; default is `FALSE`. If `TRUE`, the function will return the information returned by `uniroot`. See paragraph Value below.

`om` a (usually complex) argument

`v0` a numeric vector of length one

`vT` a numeric vector of length one

`v` a numeric vector of length one

`rho` a numeric vector of length one

`k` a numeric vector of length one

`sigma` a numeric vector of length one

`lambda` a numeric vector of length one

`muJ` a numeric vector of length one

`vJ` a numeric vector of length one

`nu` a numeric vector of length one

`theta` a numeric vector of length one

5.4.18. Details

The function computes the value of a plain vanilla European call under different models, using the representation of Bakshi/Madan. Put values can be computed through put–call parity (see `putCallParity`).

If `implVol` is `TRUE`, the function will compute the implied volatility necessary to obtain the same value under Black–Scholes–Merton. The implied volatility is computed with `uniroot` from the stats package. The default search interval is `c(0.00001, 2)`; it can be changed through `uniroot.control`.

The function uses variances as inputs (not volatilities).

The function is not vectorised (but see the NMOF Manual for examples of how to efficiently price more than one option at once).

5.4.19. Value

Returns the value of the call (numeric) under the respective model or, if `implVol` is `TRUE`, a list of the value and the implied volatility. (If, in addition, `uniroot.info` is `TRUE`, the information provided by `uniroot` is also returned.)

5.4.20. Note

If `implVol` is `TRUE`, the function will return a list with elements named `value` and `impliedVol`. Prior to version 0.26-3, the first element was named `callPrice`.

`callHestoncf` *Price of a European Call under the Heston Model*

5.4.21. Description

Computes the price of a European Call under the Heston model (and the equivalent Black–Scholes–Merton volatility)

5.4.22. Usage

```
callHestoncf(S, X, tau, r, q, v0, vT, rho, k, sigma, implVol = FALSE, ...)
```

5.4.23. Arguments

`S` current stock price

`X` strike price

`tau` time to maturity

`r` risk-free rate

`q` dividend rate

`v0` current variance

`vT` long-run variance

`rho` correlation between spot and variance

`k` speed of mean-reversion

`sigma` volatility of variance. A value smaller than 0.01 is replaced with 0.01.

`implVol` compute equivalent Black–Scholes–Merton volatility? Default is FALSE.

`...` named arguments, passed to `integrate`

5.4.24. Details

The function computes the value of a plain vanilla European call under the Heston model. Put values can be computed through put–call-parity. If `implVol` is `TRUE`, the function will compute the implied volatility necessary to obtain the same price under Black–Scholes–Merton. The implied volatility is computed with `uniroot` from the stats package.

Note that the function takes variances as inputs (not volatilities).

5.4.25. Value

Returns the value of the call (numeric) under the Heston model or, if `implVol` is `TRUE`, a list of the value and the implied volatility.

5.4.26. Note

If `implVol` is `TRUE`, the function will return a list with elements named `value` and `impliedVol`. Prior to version 0.26-3, the first element was named `callPrice`.

5.5. Bond pricing

See Chapter 14 of GMS. The functions `ytm` and `vanillaBond` have been included in the NMOF package since version 0.27-1.

`vanillaBond` *Pricing Plain-Vanilla Bonds*

5.5.1. Description

Calculate the theoretical price and yield-to-maturity of a list of cashflows.

5.5.2. Usage

```
vanillaBond(cf, times, df, yields)
ytm(cf, times, y0 = 0.05, tol = 1e-05, maxit = 1000L, offset = 0)

duration(cf, times, yield, modified = TRUE, raw = FALSE)
convexity(cf, times, yield, raw = FALSE)
```

5.5.3. Arguments

`cf` Cashflows; a numeric vector or a matrix. If a matrix, cashflows should be arranged in rows; times-to-payment correspond to columns.

`times` times-to-payment; a numeric vector

`df` discount factors; a numeric vector

`yields` optional (instead of discount factors); zero yields to compute discount factor; if of length one, a flat zero curve is assumed

`yield` numeric vector of length one (both duration and convexity assume a flat yield curve)

`y0` starting value

`tol` tolerance

`maxit` maximum number of iterations

`offset` numeric: a 'base' rate over which to compute the yield to maturity. See Details and Examples.

`modified` logical: return modified duration? (default `TRUE`)

`raw` logical: default `FALSE`. Compute duration/convexity as derivative of cashflows' present value? Use this if you want to approximate the change in the bond price by a Taylor series (see Examples).

5.5.4. Details

`vanillaBond` computes the present value of a vector of cashflows; it may thus be used to evaluate not just bonds but any instrument that can be reduced to a deterministic set of cashflows.

`ytm` uses Newton's method to compute the yield-to-maturity of a bond (a.k.a. internal interest rate). When used with a bond, the initial outlay (i.e. the bonds dirty price) needs be included in the vector of cashflows. For a coupon bond, a good starting value `y0` is the coupon divided by the dirty price of the bond.

An `offset` can be specified either as a single number or as a vector of zero rates. See Examples.

5.5.5. Value

numeric

A plain-vanilla bond can be represented as a list of cashflows, `cf`, with associated payment dates. The bond's theoretical price `b0` is the present value of these payments. As an example, we calculate b0 with a single yield y.

```
> cf <- c(5, 5, 5, 5, 5, 105) ## cashflows
> times <- 1:6                ## times to payment
> y <- 0.047                  ## the "true" yield
> b0 <- sum(cf/(1 + y)^times)
> b0
```

```
[1] 102
```

Since y is below the coupon rate, the theoretical price should be higher than par.

The function vanillaBond shows a simple implementation for computing the present value of cashflows.

```
> vanillaBond <- function(cf, times, df, yields) {
      if (missing(times))
          times <- seq_len(length(cf))
      if (missing(df))
          df <- 1/(1+yields)^times
      drop(cf %*% df)
  }
```

Some examples.

```
> cf <- c(rep(5, 9), 105)
> vanillaBond(cf, yields = 0.05)
```

```
[1] 100
```

```
> vanillaBond(cf, yields = 0.03)
```

```
[1] 117
```

If only a single yield is given, the function acts as if the term structure were flat. But we did not explicitly check for this case; R's recycling rule will handle this for us. Here is an example to show this more clearly:

```
> 2^(1:5)
```

```
[1]  2  4  8 16 32
```

(The ^ operator has precedence over : which is why we need the parentheses.)

Another example; this time we value the bond according to a Nelson–Siegel curve. With the given parameters, the curve should be flat.

```
> vanillaBond(cf, 1:10, yield = NS(c(0.03,0,0,2), 1:10))
```

```
[1] 117
```

Back to our problem: to recover y from b0, we append b0 to the cashflow vector, but switch its sign (since we need to buy the bond). The is now to find discount factors for which the sum over all cashflows (the net present value) is just zero.

```
>   cf <- c(5, 5, 5, 5, 5, 105) ## cashflows
>   times <- 1:6                ## times to payment
>   y <- 0.047                  ## the "true" yield
>   b0 <- sum(cf/(1 + y)^times)
>   cf <- c(-b0, cf); times <- c(0, times)
>   data.frame(times=times, cashflows=cf)
```

```
  times cashflows
1     0      -102
2     1         5
3     2         5
4     3         5
5     4         5
6     5         5
7     6       105
```

The function ytm evaluates the derivative of the discounted cashflows analytically; ytm2 uses a finite difference.

```
> ytm <- function(cf, times, y0 = 0.05,
                  tol = 1e-05, h = 1e-05, maxit = 1000L) {
      dr <- 1
      for (i in seq_len(maxit)) {
          y1 <- 1 + y0
          g <- cf / y1 ^ times
          g <- sum(g)
          t1 <- times - 1
          dg <- times * cf * 1/y1 ^ t1
          dg <- sum(dg)
          dr <- g/dg
          y0 <- y0 + dr
          if (abs(dr) < tol)
              break
      }
      y0
  }
> ytm2 <- function(cf, times, y0 = 0.05,
                   tol = 1e-04, h = 1e-08, maxit = 1000L) {
      dr <- 1
      for (i in seq_len(maxit)) {
          y1 <- 1 + y0
          g <- sum(cf/y1^times)
          y1 <- y1 + h
          dg <- (sum(cf/y1^times) - g)/h
          dr <- g/dg
          y0 <- y0 - dr
          if (abs(dr) < tol)
              break
      }
      y0
  }
> system.time(for (i in 1:2000) ytm(cf, times, y0=0.06))
```

```
   user  system elapsed
  0.028   0.000   0.027
```

```
> system.time(for (i in 1:2000) ytm2(cf, times, y0=0.06))
```

```
   user  system elapsed
  0.021   0.000   0.021
```

```
> ytm(cf, times, y0=0.062, maxit = 5000)
```

```
[1] 0.047
```

```
> ytm2(cf, times, y0=0.062, maxit = 5000)
```

```
[1] 0.047
```

The only reason for not using a finite difference is that with extreme rates or extremely far off starting values, the numerically-evaluated derivative is more stable. But note that /far-off/ really means far-off: something like the true yield is 5 percent and we use a starting value of 50 percent. (A reasonable starting value is the coupon divided by the price.)

```
> (initial.value <- 5/b0)
```

```
[1] 0.0492
```

```
> ytm(cf,  times, y0 = 0.7, maxit = 5000)
```

```
[1] 0.047
```

```
> ytm(cf,  times, y0 = initial.value)
```

```
[1] 0.047
```

```
> ytm2(cf, times, y0 = 0.7, maxit = 5000)
```

```
[1] Inf
```

```
> ytm2(cf, times, y0 = initial.value)
```

```
[1] 0.047
```

5.6. Resampling

The function `resampleC` was added in version 0.24-0. It samples with replacement from a number of samples (numeric vectors); the resulting vectors have a specified rank correlation.

`resampleC` *Resample with Specified Rank Correlation*

5.6.1. Description

Resample with replacement from a number of vectors; the sample will have a specified rank correlation.

5.6.2. Usage

```
resampleC(..., size, cormat)
```

5.6.3. Arguments

`...` numeric vectors; they need not have the same length.

`size` an integer: the number of samples to draw

`cormat` the rank correlation matrix

5.6.4. Details

See Gilli, Maringer and Schumann (2011), Section 7.1.2. The function samples with replacement from the vectors passed through `...`. The resulting samples will have an (approximate) rank correlation as specified in `cormat`.

The function uses the eigenvalue decomposition to generate the correlation; it will not break down in case of a semidefinite matrix. If an eigenvalue of `cormat` is smaller than zero, a warning is issued (but the function proceeds).

5.6.5. Value

a numeric matrix with `size` rows. The columns contain the samples; hence, there will be as many columns as vectors passed through `...`.

To give just one financial example when such a function could be handy: mutual-fund prices are often not determined at the same point in time during the day. Such asynchronous pricing introduces attenuation bias, i.e. measured correlations are too small in absolute terms. When creating return scenarios, for example, we may then want to add our own guess for a correlation.

Truncated normals

As a numerical example, we sample from a Gaussian, a uniform, a binomial and a truncated Gaussian. For the first three we have functions in R' base package (`rnorm`, `runif` and `rbinom`); but not for the truncated Gaussian.

A variable that is distributed as a *truncated* Gaussian with mean μ and variance σ^2 is just like normal Gaussian with the same mean and variance; but its support is limited to a range a to b.

The simplest way to create such variates is to sample from a Gaussian, and then throw away those variates smaller than a or greater b. But this is inefficient whenever the range is small. An alternative is to sample from the inverse:

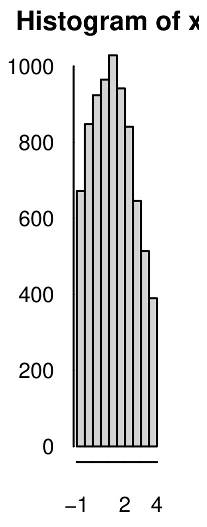
1. Transfrom a and b into $a_{\mathrm{t}} = (a - \mu)/\sigma$ and $b_{\mathrm{t}} = (/b/{-\mu})/\sigma$.
2. Draw a uniform u.
3. Compute $\Phi^{-1}\left(\Phi(a_{\mathrm{t}}) + u(\Phi(b_{\mathrm{t}}) - \Phi(a_{\mathrm{t}}))\right)$.

Here is an example: we create 10000 variates between -1 and 4.

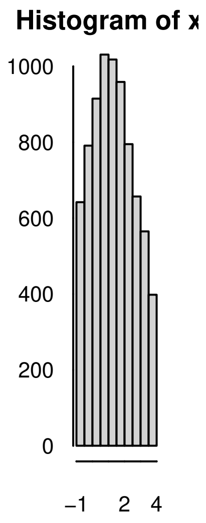
```
> mu <- 1
> sigma <- 2
> a <- -1
> b <-  4
> at <- (a - mu)/sigma
> bt <- (b - mu)/sigma
> ## "throw away" strategy
> x0 <- rnorm(10000L, mean = mu, sd = sigma)
> x0 <- x0[x0>=a & x0<=b]
> ## inverse strategy
> u <- runif(length(x0))
> z <- qnorm(pnorm(at) + u*(pnorm(bt) - pnorm(at)))
> x1 <- z * sigma + mu
```

We plot the results.

```
> par(mfrow = c(1, 2), mar = c(3, 3, 1, 1),
      bty = "n", las = 1, ps = 8, tck = 0.001, mgp = c(3, 0.5, 0))
> hist(x0, xlab = "")
```



```
> par(mfrow = c(1, 2), mar = c(3, 3, 1, 1),
      bty = "n", las = 1, ps = 8, tck = 0.001, mgp = c(3, 0.5, 0))
> hist(x1, xlab = "")
```



We create some other variates.

```
> x1 <- x1[1:750]
> x2 <- rnorm(200)
> x3 <- runif(500)
> x4 <- rbinom(100, size = 50, prob = 0.4)
```

Now suppose we want these to have a specified correlation.

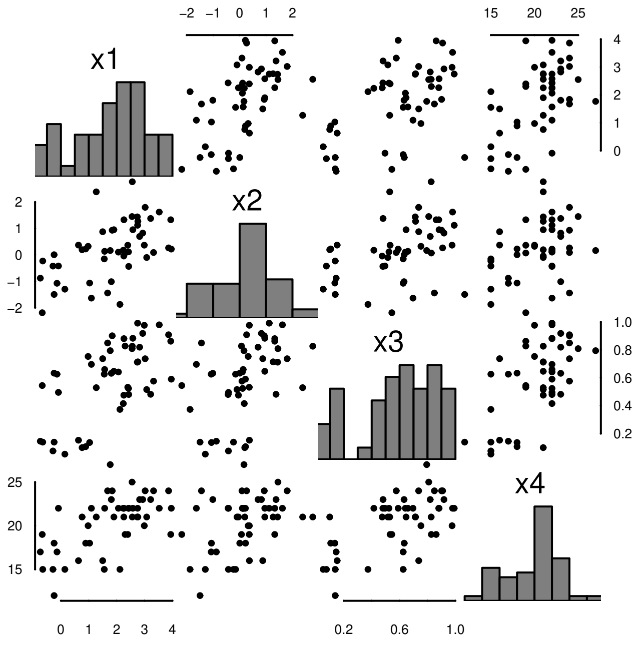
```
> cormat <- array(0.5, dim = c(4, 4))
> diag(cormat) <- 1
```

We resample 100 times from these vectors and plot the results.

```
> results <- resampleC(x1 = x1, x2 = x2, x3 = x3, x4 = x4,
                       size = 50, cormat = cormat)
> cor(results, method = "spearman")
```

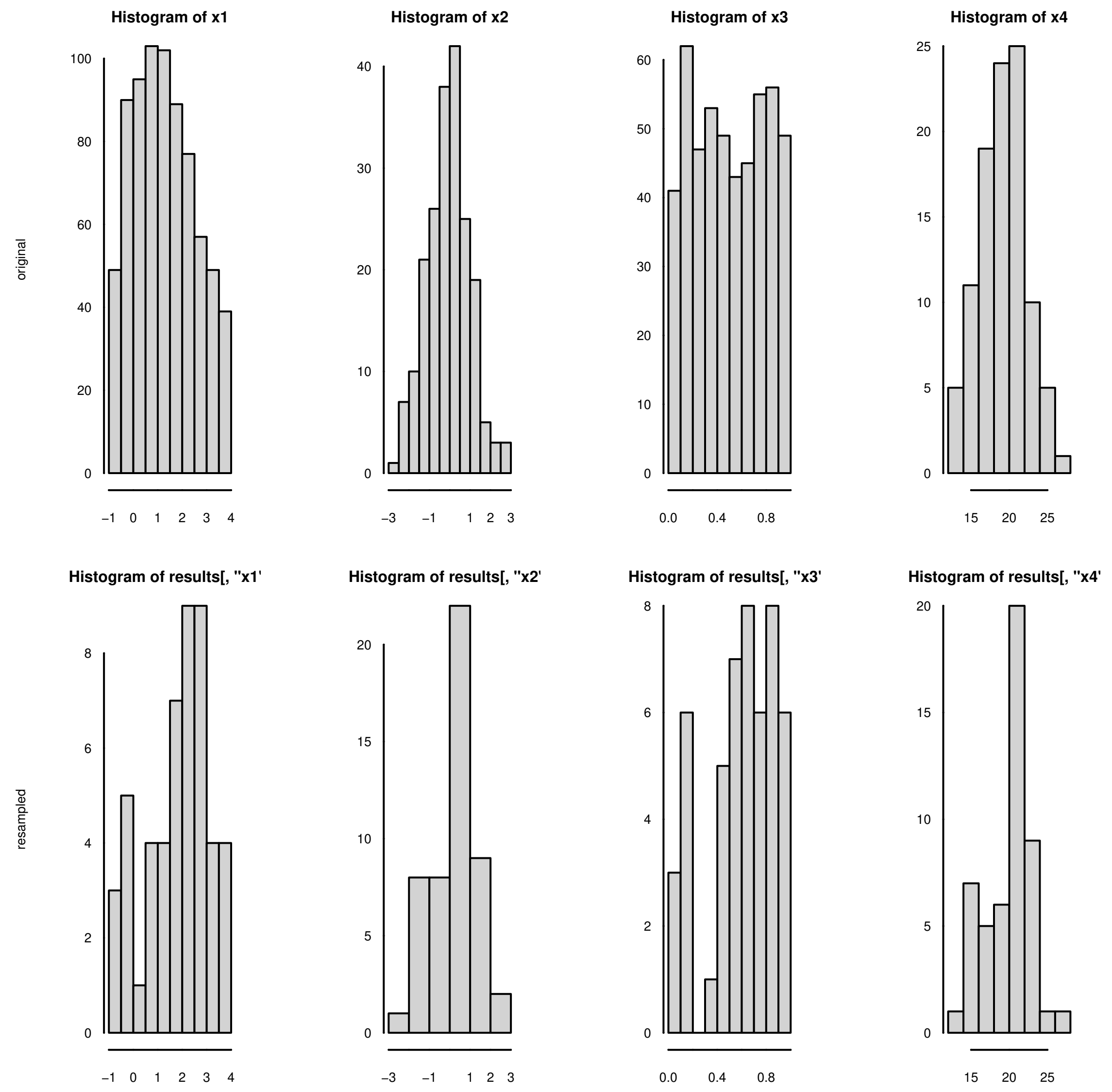
```
      x1    x2    x3    x4
x1 1.000 0.574 0.555 0.563
x2 0.574 1.000 0.535 0.354
x3 0.555 0.535 1.000 0.521
x4 0.563 0.354 0.521 1.000
```

```
> ## this function is taken from ?pairs
> panel.hist <- function(x, ...) {
      usr <- par("usr"); on.exit(par(usr))
      par(usr = c(usr[1:2], 0, 1.5) )
      h <- hist(x, plot = FALSE)
      breaks <- h$breaks; nB <- length(breaks)
      y <- h$counts; y <- y/max(y)
      rect(breaks[-nB], 0, breaks[-1L], y, col = grey(.5))
  }
> par(mar = c(3, 3, 1, 1),
      bty = "n", las = 1, ps = 8, tck = 0.001, mgp = c(3, 0.5, 0))
> pairs(results,
        diag.panel = panel.hist,
        gap = 0, pch = 19, cex = 0.5)
```



Checking the marginal distributions, before and after resampling.

```
> par(mfrow = c(2, 4), mar = c(3, 5, 1, 1),
      bty = "n", las = 1, ps = 8, tck = 0.001, mgp = c(3, 0.5, 0))
> hist(x1, xlab = "", ylab = "original")
> hist(x2, xlab = "", ylab = "")
> hist(x3, xlab = "", ylab = "")
> hist(x4, xlab = "", ylab = "")
> hist(results[ ,"x1"], xlab = "", ylab = "resampled")
> hist(results[ ,"x2"], xlab = "", ylab = "")
> hist(results[ ,"x3"], xlab = "", ylab = "")
> hist(results[ ,"x4"], xlab = "", ylab = "")
```



5.7. Constant-Proportion Portfolio Insurance (CCPI)

The function CPPI was added in version `0.99-0`. It simulates a simple constant-proportion portfolio-insurance strategy for a given price path.

`CPPI`	*Constant-Proportion Portfolio Insurance*

5.7.1. Description

Simulate constant-proportion portfolio insurance (CPPI) for a given price path.

5.7.2. Usage

```
CPPI(S, multiplier, floor, r, tau = 1, gap = 1)
```

5.7.3. Arguments

`S` numeric: price path of risky asset

`multiplier` numeric

`floor` numeric: a percentage, should be smaller than 1

`r` numeric: interest rate (per time period tau)

`tau` numeric: time periods

`gap` numeric: how often to rebalance. 1 means every timestep, 2 means every second timestep, and so on.

5.7.4. Details

Based on Dietmar Maringer's MATLAB code (function CPPIgap, Listing 9.1).

See Gilli, Maringer and Schumann, 2011, chapter 9.

5.7.5. Value

A list:

`V` normalised value (always starts at 1)

`C` cushion

`B` bond investment

`F` floor

`E` exposure

`N` units of risky asset

`S` price path

5.8. Moving averages

5.8.1. Simple moving average

Let y_t be a univariate time series, the subscript t indicates the point in time. Then a simple average M is defined thus:

$$M_t = \frac{1}{k}\sum_{i=0}^{k-1} y_i \tag{5.2}$$

If we compute a *moving* average, we do not need to recompute the whole sum at every t.

`MA` *Simple Moving Average*

5.8.2. Description

The function computes a moving average of a vector.

5.8.3. Usage

```
MA(y, order, pad = NULL)
```

5.8.4. Arguments

`y` a numeric vector

`order` An integer. The order of the moving average. The function is defined such that order one returns `y` (see Examples).

`pad` Defaults to `NULL`. If not `NULL`, all elements of the returned moving average with position smaller than `order` are replaced by the value of `pad`. Sensible values may be `NA` or `0`.

5.8.5. Value

Returns a vector of length `length(y)`.

5.8.6. Exponential moving average

$$E_t = \frac{y_t + \alpha y_{t-1} + \alpha^2 y_{t-2} + \cdots}{1 + \alpha + \alpha^2 + \cdots} \tag{5.3}$$

The sum $1 + \alpha + \alpha^2 + \cdots$ can be simplified.

$$\begin{aligned} s &= 1 + \alpha + \alpha^2 + \cdots \quad \text{und} \\ \alpha s &= \quad\;\; \alpha + \alpha^2 + \alpha^3 + \cdots \end{aligned}$$

Then:

$$s - \alpha s = 1 \quad \text{oder} \quad s = \frac{1}{1-\alpha}\,.$$

We rewrite Euqation (5.3).

$$E_t = (1-\alpha) y_t + \underbrace{(1-\alpha)[\alpha y_{t-1} + \alpha^2 y_{t-2} + \cdots]}_{\alpha E_{t-1}} \tag{5.4}$$

5.9. Monte Carlo

`mc` *Option Pricing via Monte-Carlo Simulation*

5.9.1. Description

Functions to calculate the theoretical prices of options through simulation.

5.9.2. Usage

```
gbm(npaths, timesteps, r, v, tau, S0,
    exp.result = TRUE, antithetic = FALSE)
gbb(npaths, timesteps, S0, ST, v, tau,
    log = FALSE, exp.result = TRUE)
```

5.9.3. Arguments

`npaths` the number of paths

`timesteps` timesteps per path

`r` the mean per unit of time

`v` the variance per unit of time

`tau` time

`S0` initial value

`ST` final value of Brownian bridge

`log` logical: construct bridge from log series?

`exp.result` logical: compute `exp` of the final path, or return log values?

`antithetic` logical: if `TRUE`, random numbers for only `npaths/2` are drawn, and the random numbers are mirrored

5.9.4. Details

`gbm` generates sample paths of geometric Brownian motion.

`gbb` generates sample paths of a Brownian bridge by first creating paths of Brownian motion `W` from time `0` to time `T`, with `W_0` equal to zero. Then, at each `t`, it subtracts `t/T * W_T` and adds `S0*(1-t/T)+ST*(t/T)`.

5.9.5. Value

A matrix of sample paths; each column contains the price path of an asset. Even with only a single time-step, the matrix will have two rows (the first row is `S0`).

5.10. Working with rank-deficient data matrices

`colSubset` *Full-rank Column Subset*

5.10.1. Description

Select a full-rank subset of columns of a matrix.

5.10.2. Usage

```
colSubset(x)
```

5.10.3. Arguments

`x` a numeric matrix

5.10.4. Details

Uses `qr`.

5.10.5. Value

A list:

`columns` indices of columns

`multiplier` a matrix

5.11. Drawdown

`drawdown` *Drawdown*

5.11.1. Description

Compute the drawdown of a time series.

5.11.2. Usage

```
drawdown(v, relative = TRUE, summary = TRUE)
```

5.11.3. Arguments

`v` a price series (a numeric vector)

`relative` if `TRUE`, maximum drawdown is chosen according to percentage losses; else in units of `v`

`summary` if `TRUE`, provide maximum drawdown and time when it occured; else return drawdown vector

5.11.4. Details

The drawdown at position t of a time series v is the difference between the highest peak that was reached before t and the current value. If the current value represents a new high, the drawdown is zero.

5.11.5. Value

If `summary` is `FALSE`, a vector of the same length as `v`. If `summary` is `TRUE`, a list

`maximum` maximum drawdown

`high` the max of `v`

`high.position` position of `high`

`low` the min of `v`

`low.position` position of `low`

5.12. Efficient frontiers

The function `mvFrontier` was added in version 1.5-0.

`mvFrontier` *Computing Mean–Variance Efficient Portfolios*

5.12.1. Description

Compute mean–variance efficient portfolios and efficient frontiers.

5.12.2. Usage

```
mvFrontier(m, var, wmin = 0, wmax = 1, n = 50, rf = NA,
           groups = NULL, groups.wmin = NULL, groups.wmax = NULL)
mvPortfolio(m, var, min.return, wmin = 0, wmax = 1, lambda = NULL,
            groups = NULL, groups.wmin = NULL, groups.wmax = NULL)
```

5.12.3. Arguments

`m` vector of expected returns

`var` expected variance–covariance matrix

`wmin` numeric: minimum weights

`wmax` numeric: maximum weights

`n` number of points on the efficient frontier

`min.return` minimal required return

`rf` risk-free rate

`lambda` risk–reward trade-off

`groups` a list of group definitions

`groups.wmin` a numeric vector

`groups.wmax` a numeric vector

5.12.4. Details

`mvPortfolio` computes a single mean–variance efficient portfolio, using package quadprog. It does so by minimising portfolio variance, subject to constraints on minimum return and budget (weights need to sum to one), and min/max constraints on the weights.

If λ is specified, the function ignores the `min.return` constraint and instead solves the model

$$\min_{w} \quad -\lambda \mathtt{m}'w + (1-\lambda)w'\mathtt{var}\,w$$

in which w are the weights. If λ is a vector of length 2, then the model becomes

$$\min_{w} \quad -\lambda_1 \mathtt{m}'w + \lambda_2 w'\mathtt{var}\,w$$

which may be more convenient (e.g. for setting λ_1 to 1).

`mvFrontier` computes returns, volatilities and compositions for portfolios along an efficient frontier. If `rf` is not `NA`, cash is included as an asset.

5.12.5. Value

For `mvPortfolio`, a numeric vector of weights.

For `mvFrontier`, a list of three components:

`return` returns of portfolios

`volatility` volatilities of portfolios

`weights` A matrix of portfolio weights. Each column holds the weights for one portfolio on the frontier. If `rf` is specified, an additional row is added, providing the cash weight.

The *i*-th portfolio on the frontier corresponds to the *i*-th elements of `return` and `volatility`, and the *i*-th column of `portfolio`.

We given an example for the case of four assets.

The variance-covariance matrix may be decomposed into the (matrix) product *S* times *C* times *S*, in which *S* is a diagonal matrix with the standard deviations on its main diagonal and zeros elsewhere, and in which *C* is the correlation matrix.

Assume you have the following forecasts for the assets:

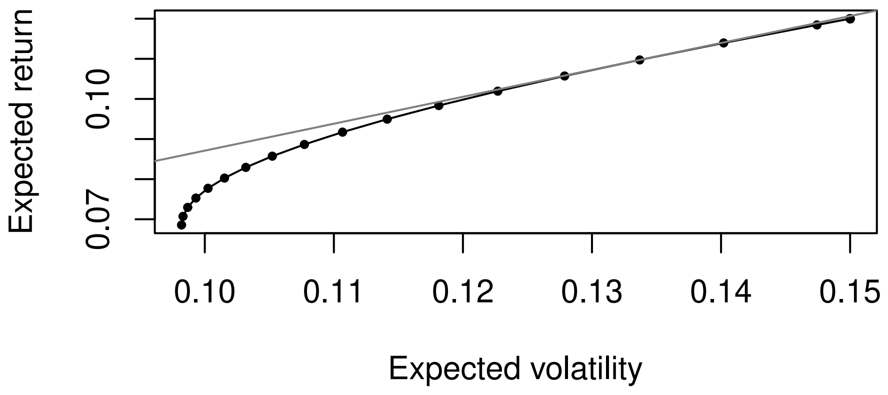
```
> na <- 4                               ## number of assets
> vols <- c(0.10, 0.15, 0.20, 0.22)  ## forecast vols
> m <- c(0.06, 0.12, 0.09, 0.07)     ## forecast returns
```

Then a covariance matrix for a constant correlation of 0.5 may be computed in this way:

```
> const_cor <- function(rho, na) {
      C <- array(rho, dim = c(na, na))
      diag(C) <- 1
      C
  }
> var <- diag(vols) %*% const_cor(0.5, na) %*% diag(vols)
```

We call `mvFrontier`.

```
> library("NMOF")
> wmax <- 1     ## maximum holding size
> wmin <- 0.0   ## minimum holding size
> rf <- 0.02
> ## without cash
> p1 <- mvFrontier(m, var, wmin = wmin, wmax = wmax, n = 50)
> ## with cash
> p2 <- mvFrontier(m, var, wmin = wmin, wmax = wmax, n = 50, rf = rf)
> plot(p1$volatility, p1$return, pch = 19, cex = 0.5, type = "o",
       xlab = "Expected volatility",
       ylab = "Expected return")
> lines(p2$volatility, p2$return, col = grey(0.5))
> abline(v = 0, h = rf)
```



5.13. Traditional portfolio-selection models

`minvar` *Minimum-Variance Portfolios*

5.13.1. Description

Compute minimum-variance portfolios, subject to lower and upper bounds on weights.

5.13.2. Usage

```
minvar(var, wmin = 0, wmax = 1, method = "qp",
       groups = NULL, groups.wmin = NULL, groups.wmax = NULL)
```

5.13.3. Arguments

`var` the covariance matrix: a numeric (real), symmetric matrix `wmin` numeric: a lower bound on weights. May also be a vector that holds specific bounds for each asset.

`wmax` numeric: an upper bound on weights. May also be a vector that holds specific bounds for each asset.

`method` character. Currently, only `"qp"` is supported.

`groups` a list of group definitions

`groups.wmin` a numeric vector

`groups.wmax` a numeric vector

5.13.4. Details

The function uses `solve.QP` from package quadprog. Because of the algorithm that `solve.QP` uses, `var` has to be positive definit (i.e. must be of full rank).

5.13.5. Value

a numeric vector (the portfolio weights) with an attribute `variance` (the portfolio's variance)

`minCVaR` *Minimum Conditional-Value-at-Risk (CVaR) Portfolios*

5.13.6. Description

Compute minimum-CVaR portfolios, subject to lower and upper bounds on weights.

5.13.7. Usage

```
minCVaR(R, q = 0.1, wmin = 0, wmax = 1,
        min.return = NULL, m = NULL,
        method = "Rglpk",
        groups = NULL, groups.wmin = NULL, groups.wmax = NULL,
        Rglpk.control = list())
```

5.13.8. Arguments

`R` the scenario matrix: a numeric (real) matrix

`q` the Value-at-Risk level: a number between 0 and 0.5

`wmin` numeric: a lower bound on weights. May also be a vector that holds specific bounds for each asset.

`wmax` numeric: an upper bound on weights. May also be a vector that holds specific bounds for each asset.

`m` vector of expected returns. Only used if `min.return` is specified.

`min.return` minimal required return. If `m` is not specified, the column means of `R` are used.

`method` character. Currently, only `"Rglpk"` is supported.

`groups` a list of group definitions

`groups.wmin` a numeric vector

`groups.wmax` a numeric vector

`Rglpk.control` a list: settings passed to `Rglpk_solve_LP`

5.13.9. Details

Compute the minimum CVaR portfolio for a given scenario set. The default method uses the formulation as a Linear Programme, as described in Rockafellar/Uryasev (2000).

The function uses `Rglpk_solve_LP` from package Rglpk.

5.13.10. Value

a numeric vector (the portfolio weights); attached is an attribute whose name matches the `method` name

---

`maxSharpe` *Maximum-Sharpe-Ratio/Tangency Portfolio*

5.13.11. Description

Compute maximum Sharpe-ratio portfolios, subject to lower and upper bounds on weights.

5.13.12. Usage

```
maxSharpe(m, var, min.return,
          wmin = -Inf, wmax = Inf, method = "qp",
          groups = NULL, groups.wmin = NULL, groups.wmax = NULL)
```

5.13.13. Arguments

`m` vector of expected (excess) returns.

`var` the covariance matrix: a numeric (real), symmetric matrix

`min.return` minimumm required return. This is a technical parameter, used only for QP.

`wmin` numeric: a lower bound on weights. May also be a vector that holds specific bounds for each asset.

`wmax` numeric: an upper bound on weights. May also be a vector that holds specific bounds for each asset.

`method` character. Currently, only `"qp"` is supported.

`groups` a list of group definitions

`groups.wmin` a numeric vector

`groups.wmax` a numeric vector

5.13.14. Details

The function uses `solve.QP` from package quadprog. Because of the algorithm that `solve.QP` uses, `var` has to be positive definit (i.e. must be of full rank).

5.13.15. Value

a numeric vector (the portfolio weights) with an attribute `variance` (the portfolio's variance)

5.14. Financial data

The function `French` was added in version 1.5-0.

`French` *Download Datasets from Kenneth French's Data Library*

5.14.1. Description

Download datasets from Kenneth French's Data Library.

5.14.2. Usage

```
French(dest.dir,
       dataset = "F-F_Research_Data_Factors_CSV.zip",
       weighting = "value", frequency = "monthly",
       price.series = FALSE, na.rm = FALSE,
       adjust.frequency = TRUE)
```

5.14.3. Arguments

`dest.dir` character: a path to a directory

`dataset` a character string: the CSV file name. Also supported are the keywords '`market`' and '`rf`'.

`weighting` a character string: `"equal"` or `"value"`

`frequency` a character string: `daily`, `monthly` or `annual`. Whether it is used or ignored depends on the particular dataset.

`price.series` logical: convert the returns series into prices series?

`na.rm` logical: remove missing values in the calculation of price series?

`adjust.frequency` logical: if `TRUE`, frequency is switched to '`"daily"`' when the word '`"daily"`' appears in the dataset's name

5.14.4. Details

The function downloads data provided by Kenneth French at `http://mba.tuck.dartmouth.edu/pages/faculty` The download file gets a date prefix (current date in format `YYYYMMDD`) and is stored in directory `dest.dir`. Before any download is attempted, the function checks whether a file with today's prefix exist in `dest.dir`; if yes, the file is used.

In the original data files, missing values are coded as `-99` or similar. These numeric values are replaced by `NA`.

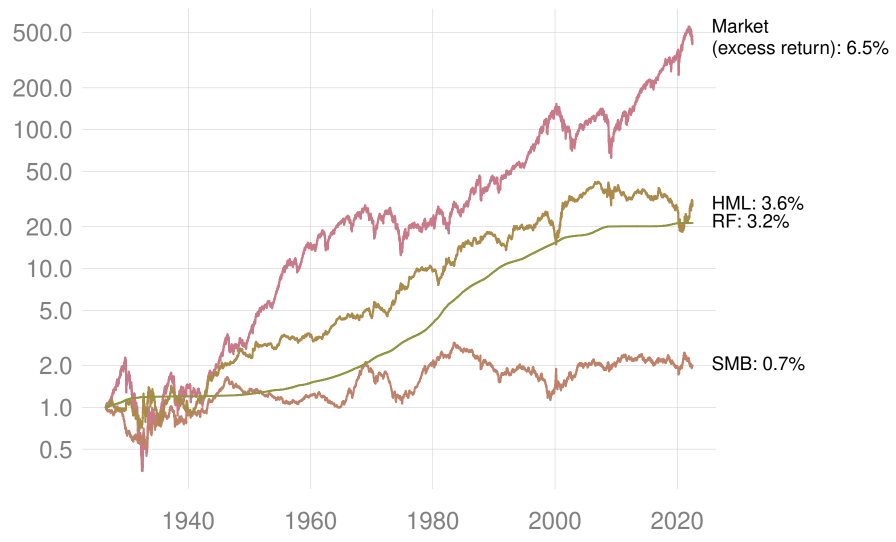
Calling the function without any arguments will print the names of the supported datasets (and return them insivibly).

5.14.5. Value

A `data.frame`, with contents depending on the particular dataset. If the download failes, the function evaluates to `NULL`.

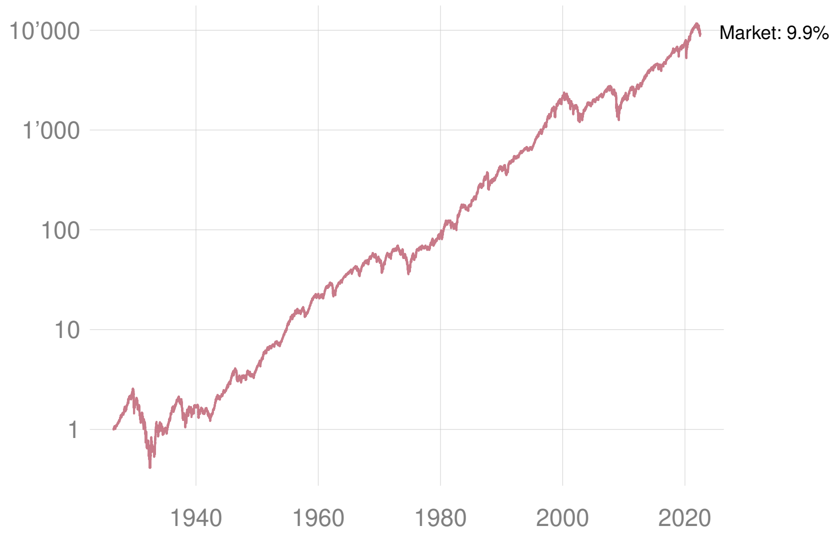
Some examples, which make use of function `plotseries`.

```
>   library("plotseries")
>   library("zoo")
>   series <- French("~/Downloads/French/",
                     "F-F_Research_Data_Factors_daily_CSV.zip",
                     frequency = "daily",
                     price.series = TRUE)
>   series <- zoo(series, as.Date(row.names(series)))
>   plotseries(series,
               col = hcl.colors(n = 12, palette = "Dark 2"),
               log.scale = TRUE,
               labels = c("Market\n(excess return)",
                          "SMB", "HML", "RF"),
               font.family = "")
```



The market (total return).

```
>   series <- French("~/Downloads/French/",
                     "market",
                     frequency = "daily",
                     price.series = TRUE)
>   series <- zoo(series, as.Date(row.names(series)))
>   plotseries(series,
               col = hcl.colors(n = 12, palette = "Dark 2"),
               log.scale = TRUE,
               labels = c("Market"),
               font.family = "")
```



The function `Shiller` was added in version 1.5-0.

`Shiller` *Download Robert Shiller's Data*

5.14.6. Description

Download the data provided by Robert Shiller and transform them into a data frame.

5.14.7. Usage

```
Shiller(dest.dir,
        url = "http://www.econ.yale.edu/~shiller/data/ie_data.xls")
```

5.14.8. Arguments

`dest.dir`	character: a path to a directory
`url`	the data URL

5.14.9. Details

The function downloads US stock-market data provided by Robert Shiller which he used in his book 'Irrational Exhuberance'. Since the data are provided in Excel format, package readxl is required.

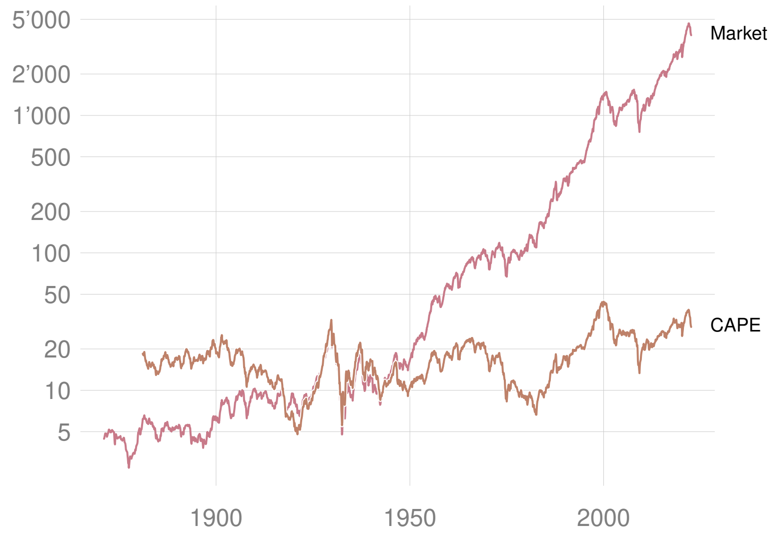
The downloaded Excel gets a date prefix (today in format `YYYYMMDD`) and is stored in directory `dest.dir`. Before any download is attempted, the function checks whether a file with today's prefix exist in `dest.dir`; if yes, the file is used.

5.14.10. Value

a `data.frame`:

`Date`	end of month
`Price`	numeric
`Dividend`	numeric
`Earnings`	numeric
`CPI`	numeric
`Long Rate`	numeric
`Real Price`	numeric
`Real Dividend`	numeric
`Real Earnings`	numeric
`CAPE`	numeric

```
> series <- Shiller("~/Downloads/Shiller")
> plotseries(series[, c("Price", "CAPE")],
             t = as.Date(series[["Date"]]),
             col = hcl.colors(n = 12, palette = "Dark 2"),
             log.scale = TRUE,
             returns.show = FALSE,
             labels = c("Market", "CAPE"),
             white.underlay = TRUE,
             font.family = "")
```



5.15. Random returns

The function `randomReturns` was added in version 2.0-1.

`randomReturns` *Create a Random Returns*

5.15.1. Description

Create a matrix of random returns.

5.15.2. Usage

```
randomReturns(na, ns, sd, mean = 0, rho = 0, exact = FALSE)
```

5.15.3. Arguments

`na` number of assets

`ns` number of return scenarios

`sd` the standard deviation: either a single number or a vector of length `na`

`mean` the mean return: either a single number or a vector of length `na`

`rho` correlation: either a scalar (i.e. a constant pairwise correlation) or a correlation matrix `exact` logical: if `TRUE`, return a random matrix whose column means, standard deviations and correlations match the specified values exactly (up to numerical precision)

5.15.4. Details

The function corresponds to the function `random_returns`, described in the second edition of NMOF (the book).

5.15.5. Value

a `numeric` `matrix` of size `na` times `ns`

5.15.6. Note

The function corresponds to the function `random_returns`, described in the second edition of NMOF (the book).

5.16. Greedy Search

The function `greedySearch` was added in version 2.0-1.

`greedySearch` *Greedy Search*

5.16.1. Description

Greedy Search

5.16.2. Usage

```
greedySearch(OF, algo, ...)
```

5.16.3. Arguments

`OF` The objective function, to be minimised. Its first argument needs to be a solution; `...` arguments are also passed.

`algo` List of settings. See Details.

`...` Other variables to be passed to the objective function and to the neighbourhood function. See Details.

5.16.4. Details

A greedy search works starts at a provided initial solution (called the current solution) and searches a defined neighbourhood for the best possible solution. If this best neighbour is not better than the current solution, the search stops. Otherwise, the best neighbour becomes the current solution, and the search is repeated.

5.16.5. Value

A list:

`xbest` best solution found.

`OFvalue` objective function value associated with best solution.

`Fmat` a matrix with two columns. `Fmat[ ,1L]` contains the proposed solution over all iterations; `Fmat[ ,2L]` contains the accepted solutions.

`xlist` a list

`initial.state` the value of `.Random.seed` when the function was called.

`x0` the initial solution

`iterations` the number of iterations after which the search stopped

Case studies and examples

In this part of the book, we we will look at a number of more specific examples.

6. Asset selection with GA and TA

6.1. Selection few from many assets

We first extend an example given in NMOF: selecting a small number of assets out of a large set of available assets such that the resulting portfolio has minimal variance. In the book, we solved this problem with a simple Local Search. In this chapter, we will also use Threshold Accepting (TA) and a Genetic Algorithm (GA). In fact, for this problem a Local Search is just fine. But the example serves to show how a GA could be used to solve such a model (TA and Local Search are quite similar).

6.2. Functions

We create random data: na assets with marginal volatilities between 20% and 40%, and a constant pairwise linear correlation of 0.6 (see GMS, Chapter 7).

```
> na <- 500L                          ## number of assets
> C <- array(0.6, dim = c(na,na)) ## correlation matrix
> diag(C) <- 1
> minVol <- 0.20; maxVol <- 0.40  ## covariance matrix
> Vols <- (maxVol - minVol) * runif(na) + minVol
> Sigma <- outer(Vols, Vols) * C
```

Next, we define the objective function and the neighbourhood function. They are the same for Local Search and TA. A solution will be coded as a logical vector. If an element of this vector is TRUE than the corresponding asset is in the portfolio; FALSE indicates that it is excluded. The budget constraint is handled in the objective function: we map a given logical vector to a numerical vectors that sums to unity. The cardinality restriction is enforced in the neighbourhood function, in which we simply reject new portfolios that violate the constraint.

```
> OF <- function(x, Data) {
      sx <- sum(x)
      w <- rep.int(1/sx, sx)
      res <- crossprod(w, Data$Sigma[x, x])
      tcrossprod(w, res)
  }
```

The neighbourhood function.

```
> neighbour <- function(xc, Data) {
      xn <- xc
      p <- sample.int(Data$na, Data$nn, replace = FALSE)
      xn[p] <- !xn[p]

      ## reject infeasible solution
      sumx <- sum(xn)
      if ( (sumx > Data$Ksup) || (sumx < Data$Kinf) )
          xc else xn
  }
```

To evaluate OF and neighbour, we typically need other pieces of information than just the solution itself. We collect them all in the list Data, and pass this list to both functions.

```
> Data <- list(Sigma = Sigma,   ## cov-matrix
               Kinf  = 30L,     ## min cardinality
               Ksup  = 60L,     ## max cardinality
               na    = na,      ## number of assets
               nn    = 1L)      ## how many assets to change per iteration
```

We create a random solution x0 with acceptable cardinality.

```
> card0 <- sample(Data$Kinf:Data$Ksup, 1L, replace = FALSE)
> assets <- sample.int(na, card0, replace = FALSE)
> x0 <- logical(na)
> x0[assets] <- TRUE
```

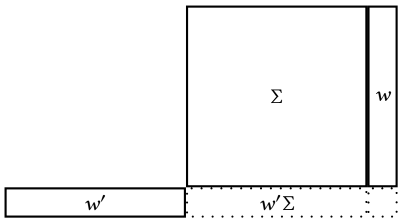
We define the settings for Local Search and TA and run both methods. Note that with these settings, both functions use the same starting value and the same number of objective function evaluations.

```
> ## Local Search
> algo <- list(x0 = x0, neighbour = neighbour, nS = 5000L,
               printDetail = FALSE, printBar = FALSE)
> system.time(solLS <- LSopt(OF, algo = algo, Data = Data))
> ## Threshold Accepting
> algo$nT <- 10L; algo$nS <- trunc(algo$nS/algo$nT); algo$q <- 0.2
> system.time(solTA <- TAopt(OF, algo = algo, Data = Data))
```

6.3. Using Genetic Algorithms

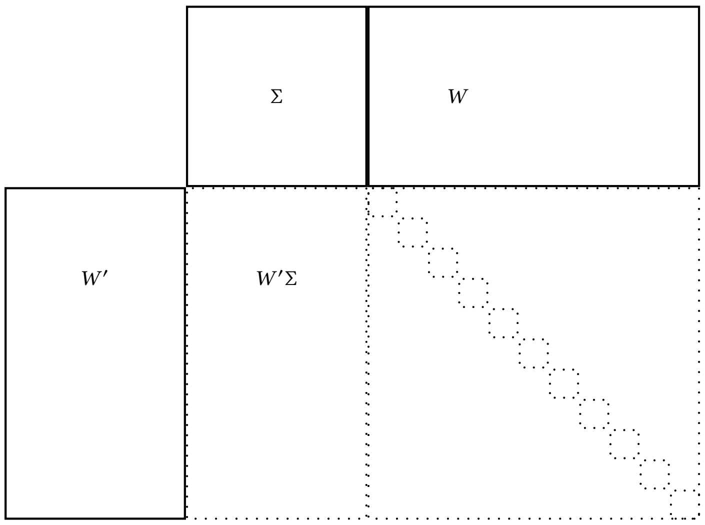
Now we use a GA, for which we need to write a new objective function. It is helpful in this case (and in many others) to cast the computation into matrix algebra notation. This makes the maths more concise and allows to use linear algebra routines. In fact, our objective function will evaluate the whole population in one step; thus, we have to set `algo$loopOF` to `FALSE`.

Suppose we have a portfolio vector w and a variance–covariance matrix Σ. For single portfolio, the computation would be as follows; the result is the scalar bottom right.



For an equal-weight portfolio, we can set w to a vector of ones and multiply the result by the squared weight (a scalar).

Now with a population $W = [w_1\ w_2\ \ldots]$, we could use matrix multiplication as well. The vector of variances is $\mathtt{diag}(W'\Sigma W)$.



But we are not interested in the off-diagonal elements. So while the code may be concise, the computation is inefficient. One solution, which we have chosen here, is to recognise that $\mathtt{diag}(W'\Sigma W)$ is equivalent to

$$\iota' \underbrace{\overbrace{\Sigma W}^{\text{matrix multiplication}} \; W}_{\text{elementwise multiplication}}$$

which is consise and more efficient; see the following objective function. The function also handles the cardinality constraint through a simple penalty.

```
> OF2 <- function(x, Data) {
      res <- colSums(Data$Sigma %*% x * x)
      n <- colSums(x); res <- res / n^2
      ## penalise
      p <- pmax(Data$Kinf - n, 0) + pmax(n - Data$Ksup, 0)
      res + p
  }
```

So we put all settings into the list `algo` and run `GAopt`. We wrap the call into `system.time` to get an idea how much time the algorithm requires.

```
> algo <- list(nB = na, nP = 100L, nG = 500L, prob = 0.002,
               printBar = FALSE, loopOF = FALSE)
> system.time(solGA <- GAopt(OF = OF2, algo = algo, Data = Data))
```

```
Genetic Algorithm.
Best solution has objective function value 0.0259 ;
standard deviation of OF in final population is 0 .
```

We should now compare the results of the three algorithms.

```
Local Search        0.1618
Threshold Accepting 0.1615
Genetic Algorithm   0.1608
```

All three algorithms give essentially the same answer. (Recall that the marginal volatilities were between 20% and 40%, so the result is reasonable.) Just looking at one outcome is not enough with stochastic algorithms; we should rerun the analysis several times (we can use the function `restartOpt` for that).

7. Minimising semi-variance with DE, PS and TA

7.1. The problem

We want to minimise the semivariance of a long–short portfolio, under the restrictions that (i) the asset weights sum to 100% (the budget constraint), and (ii) all asset weights are between -5% and 5% (holding size constraints). (Later, we will add further constraints.) We show how this can be done with Differential Evolution (DE), Particle Swarm (PS) and Threshold Accepting (TA).

We start by building an artificial dataset: we create random returns with random marginal volatilities between 20% and 40%, and induce correlation (see GMS, Chapter 7). We scale these returns so that their magnitude roughly resembles daily equity returns. We store the returns in a matrix R such that every column represents one asset.

```
> na <- 100L                                    ## number of assets
> ns <- 200L                                    ## number of scenarios
> vols <- runif(na, min = 0.2, max = 0.4)       ## marginal vols
> C <- matrix(0.6, na, na); diag(C) <- 1        ## correlation matrix
> R <- rnorm(ns * na)/16                        ## random returns
> dim(R) <- c(ns, na)
> R <- R %*% chol(C)
> R <- R %*% diag(vols)
```

The objective is to find a portfolio of minimal semivariance, given these return scenarios and constraints. Semivariance can be written like so:

$$\frac{1}{n_S}\sum_{r_i<\theta}(\theta - r_i)^2\,. \tag{7.1}$$

In words: we sum those returns below θ, and divide by n_S. A typical value for θ may be zero or a short-term deposit rate. Let there be k returns below θ, then

$$\underbrace{\frac{1}{n_S}\;\frac{k}{k}}_{1}\;\sum_{r_i<\theta}(\theta - r_i)^2 = \underbrace{\frac{k}{n_S}}_{\mathrm{Prob}(r_i<\theta)}\;\underbrace{\frac{1}{k}\sum_{r_i<\theta}(\theta - r_i)^2}_{\text{conditional average}}\;. \tag{7.2}$$

7.2. Differential Evolution

We first collect all information in a list Data. The specific meaning of the different variables will become clear shortly (as well as the reason for transposing R).

```
> Data <- list(R = t(R),                ## scenarios
               theta = 0.005,           ## return threshold
               na = na,                 ## number of assets
               ns = ns,                 ## number of scenarios
               max = rep( 0.05, na),    ## DE: vector of max. weight
               min = rep(-0.05, na),    ## DE: vector of min. weight
               wsup =  0.05,            ## TA: max weight
               winf = -0.05,            ## TA: min weight
               eps = 0.5/100,           ## TA: step size
               w = 1)                   ## penalty weight
```

To demonstrate how the ingredients of the optimisation algorithm work, we draw a random solution x0 (which very likely violates the budget constraints).

```
> x0 <- Data$min + runif(Data$na)*(Data$max - Data$min)
> x0[1:5]
```

```
[1]  0.00210 -0.04472 -0.04371  0.03970 -0.00118
```

```
> sum(x0)
```

```
[1] -0.0378
```

But nevertheless, we can compute semivariance for this solution step-by-step.

```
> temp <- R %*% x0             ## compute portfolio returns
> temp <- temp - Data$theta    ## subtract return threshold
> temp <- (temp[temp < 0])^2   ## select elements below threshold
> sum(temp)/ns                 ## compute semivariance
```

```
[1] 3.68e-05
```

We put this computation into the objective function, which could look as follows.

```
> OF <- function(x, Data) {
      Rx <- crossprod(Data$R, x)
      Rx <- Rx - Data$theta
      Rx <- Rx - abs(Rx)
      Rx <- Rx * Rx
      colSums(Rx) /(4*Data$ns)
  }
```

The function is written such that if we have several solutions, collected in the columns of a matrix, we can evaluate all solutions in one step. We use `crossprod` to compute the portfolio returns. `crossprod(a,b)` actually computes `t(a) %*% b`, which is why we have put `t(R)` into the list `Data`.

```
> OF(x0, Data)
```

```
[1] 3.68e-05
```

```
> OF(cbind(x0, x0, x0), Data)
```

```
      x0       x0       x0 
3.68e-05 3.68e-05 3.68e-05 
```

Now for the constraints. First, the budget constraint `all.equal(sum(x0),1)`. Here, we will repair the solutions. We can try two (quite similar) approaches: we can divide `x0` by `sum(x0)`; or we can add/subtract numbers such that `sum(x0)` is one.

```
> repair <- function(x, Data) {
      myFun <- function(x)
          x/sum(x)
      if (is.null(dim(x)[2L]))
          myFun(x) else apply(x, 2L, myFun)
  }
> repair2 <- function(x, Data) {
      myFun <- function(x)
          x + (1 - sum(x))/Data$na
      if (is.null(dim(x)[2L]))
          myFun(x) else apply(x, 2L, myFun)
  }
```

Like `OF`, the functions `repair` and `repair2` work with one solution, but also with a matrix of solutions.

```
> sum(x0)
```

```
[1] -0.0378
```

```
> sum(repair(x0, Data))
```

```
[1] 1
```

```
> sum(repair2(x0, Data))
```

```
[1] 1
```

```
> colSums(repair( cbind(x0, x0, x0), Data))
```

```
x0 x0 x0
 1  1  1
```

```
> colSums(repair2(cbind(x0, x0, x0), Data))
```

```
x0 x0 x0
 1  1  1
```

Note that repair2 will typically lead to smaller changes in a solution.

```
> summary(repair (x0, Data)-x0)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 -1.332  -0.661  -0.017   0.010   0.659   1.364
```

```
> summary(repair2(x0, Data)-x0)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 0.0104  0.0104  0.0104  0.0104  0.0104  0.0104
```

For the maximum holding sizes we use a penalty function.

```
> penalty <- function(x, Data) {
      up <- Data$max
      lo <- Data$min
      xadjU <- x - up
      xadjU <- xadjU + abs(xadjU)
      xadjL <- lo - x
      xadjL <- xadjL + abs(xadjL)
      if (is.null(dim(x)[2L]))
          Data$w * (sum(xadjU) + sum(xadjL)) else
      Data$w * (colSums(xadjU) + colSums(xadjL))
  }
```

The penalty function should evaluate to a positive number if a constraint is violated, and to zero if not. We can test it by increasing one weight. The weight Data$w allows us to control the impact of the penalty.

```
> x0[1L] <- 0.30
> penalty(x0, Data)
```

```
[1] 0.5
```

```
> penalty(cbind(x0, x0, x0), Data)
```

```
 x0  x0  x0
0.5 0.5 0.5
```

```
> x0[1L] <- 0
> penalty(x0, Data)
```

```
[1] 0
```

```
> penalty(cbind(x0, x0, x0), Data)
```

```
x0 x0 x0
 0  0  0
```

We collect the settings of DE in the list `algo`; see `?DEopt` for details.

```
> algo <- list(nP = 100,        ## population size
               nG = 1000,       ## number of generations
               F = 0.25,        ## step size
               CR = 0.9,
               min = Data$min,
               max = Data$max,
               repair = repair,
               pen = penalty,
               printBar = FALSE,
               printDetail = TRUE,
               loopOF = TRUE,      ## do not vectorise
               loopPen = TRUE,     ## do not vectorise
               loopRepair = TRUE)  ## do not vectorise
```

Now we can run DE. We scale the resulting objective function value into an ‘annualised’ figure in percentage points.

```
> system.time(sol <- DEopt(OF = OF,algo = algo,Data = Data))
```

```
Differential Evolution.
Best solution has objective function value 6.69e-05 ;
standard deviation of OF in final population is 3.01e-08 .
```

```
> 16 * 100 * sqrt(sol$OFvalue)   ## solution quality
> ## check constraints
> all(all.equal(sum(sol$xbest), 1),  ## budget constraint
      sol$xbest <= Data$max,         ## holding size constraints
      sol$xbest >= Data$min)
```

We can also see if there is a meaningful difference in computing time between looping over the solutions and evaluating them in on step – the answer, in this case, is yes. The difference is typically greater for smaller datasets. The semi-variance is cheap to compute for given returns; the main part of computing time is actually spent on calculating the portfolio returns `R %*% x`.

```
> ## looping over the population
> algo$loopOF <- TRUE; algo$loopPen <- TRUE; algo$loopRepair <- TRUE
> t1 <- system.time(sol <- DEopt(OF = OF,algo = algo, Data = Data))
```

```
Differential Evolution.
Best solution has objective function value 7.2e-05 ;
standard deviation of OF in final population is 3.46e-08 .
```

```
> ## evaluating the population in one step
> algo$loopOF <- FALSE; algo$loopPen <- FALSE; algo$loopRepair <- FALSE
> t2 <- system.time(sol <- DEopt(OF = OF,algo = algo, Data = Data))
```

```
Differential Evolution.
Best solution has objective function value 8.83e-05 ;
standard deviation of OF in final population is 2.72e-08 .
```

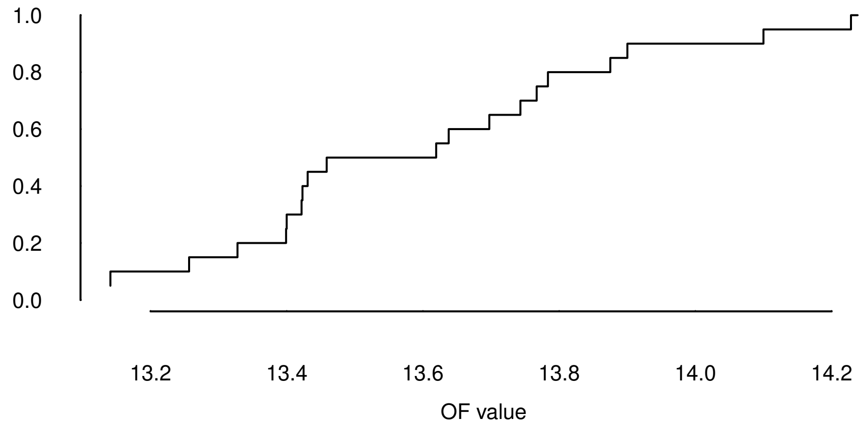
```
> ## speedup
> t1[[3L]]/t2[[3L]]
```

To see if the algorithm works properly, we run a number of restarts, and then check the solution quality of the results. For this, we can use the function `restartOpt`. The `method` and `cl` arguments specify that we use four cores to distribute the restarts, using package `snow` (Tierney et al., 2011). If the package is not available, `restartOpt` will fall back to its default (a loop) and issue a warning.

```
> algo$printDetail <- FALSE
> restartsDE <- restartOpt(fun = DEopt,      ## what function
                           n = 20L,          ## how many restarts
                           OF = OF,
                           algo = algo,
                           Data = Data,
                           cl = 2)           ## 2 cores
> ## extract best solution
> OFvaluesDE <- sapply(restartsDE, `[[`, "OFvalue")
> OFvaluesDE <- 16 * 100 * sqrt(OFvaluesDE)
> weightsDE  <- sapply(restartsDE, `[[`, "xbest")
```

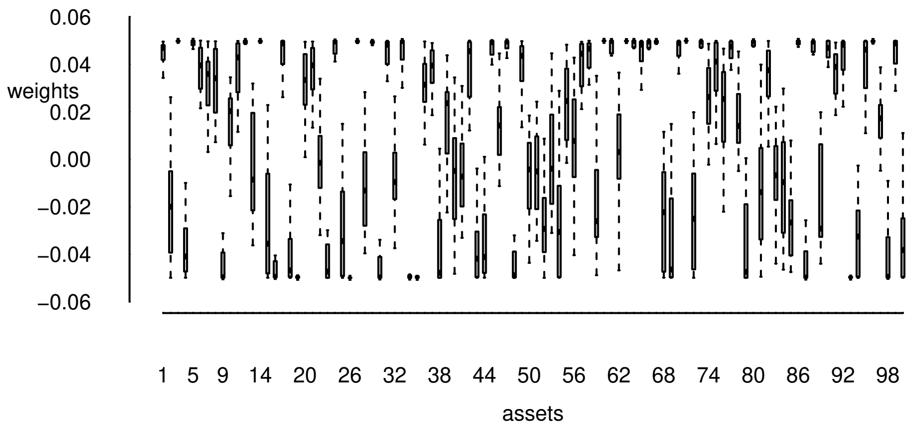
We check the objective function values associated with the restarts.

```
> par(bty = "n", las = 1, mar = c(3, 4, 0, 0),
      ps = 8, tck = 0.001)
> plot(sort(OFvaluesDE), (seq_len(length(OFvaluesDE))) / length(OFvaluesDE),
       type = "S", ylim = c(0, 1), xlab = "", ylab = "")
> mtext("OF value",  side = 1, line = 2)
```



Likewise, we may want to check the actual asset weights.

```
> par(bty = "n", las = 1, mar = c(3, 4, 0, 0),
      ps = 8, tck = 0.001)
> boxplot(t(weightsDE),
          outline = FALSE, boxwex = 0.4, ylim = c(-0.06,0.06))
> mtext("assets",  side = 1, line = 2)
> mtext("weights", side = 2, line = 1.3, las = 1, padj = -4)
```



We see that the results are quite variable, which is an indication that our settings for DE were not appropriate. In fact, in this case we simply did not grant the algorithm enough iterations. (See GMS, Chapter 10, and also Gilli and Schumann, 2011, for more discussion of the stochastics of the solutions.)

To see this, we run a small experiment in which we increase the number of iterations. We also test if there is a difference between the two different repair-approaches.

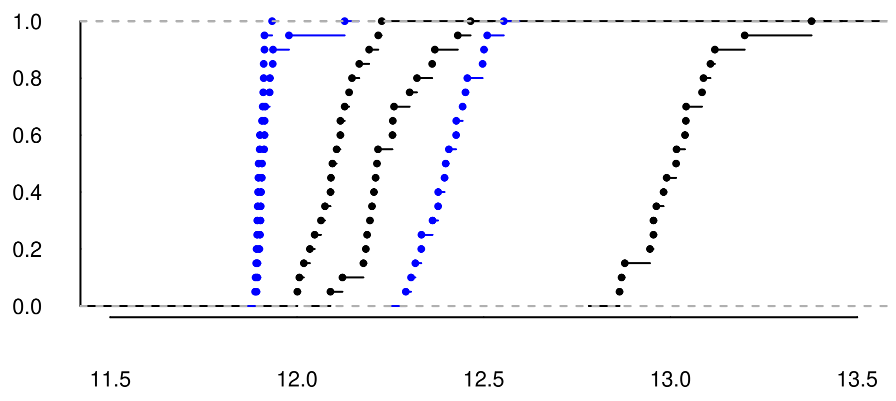
```
> algo$printDetail <- FALSE; algo$nP <- 200L; restarts <- 20L
> nGs <- c(500L, 1500L, 3000L)
> lstOFvaluesDE <- list()
> for (i in 1:3) {
      algo$nG <- nGs[i]
      restartsDE <- restartOpt(fun = DEopt,
                               n = restarts,
                               OF = OF,
                               algo = algo,
                               Data = Data,
                               cl = 8)
      ## extract best solution
      OFvaluesDE <- sapply(restartsDE, `[[`, "OFvalue")
      OFvaluesDE <- 16 * 100 * sqrt(OFvaluesDE)
      lstOFvaluesDE[[i]] <- OFvaluesDE
  }
> res <- simplify2array(lstOFvaluesDE)
```

And now with repair2.

```
> algo$repair <- repair2
> lstOFvaluesDE <- list()
> for (i in 1:3) {
      algo$nG <- nGs[i]
      restartsDE <- restartOpt(fun = DEopt,
                               n = restarts,
                               OF = OF,  algo = algo, Data = Data,
                               cl = 8)
      ## extract best solution
      OFvaluesDE <- sapply(restartsDE, `[[`, "OFvalue")
      OFvaluesDE <- 16 * 100 * sqrt(OFvaluesDE)
      lstOFvaluesDE[[i]] <- OFvaluesDE
  }
> res2 <- simplify2array(lstOFvaluesDE)
```

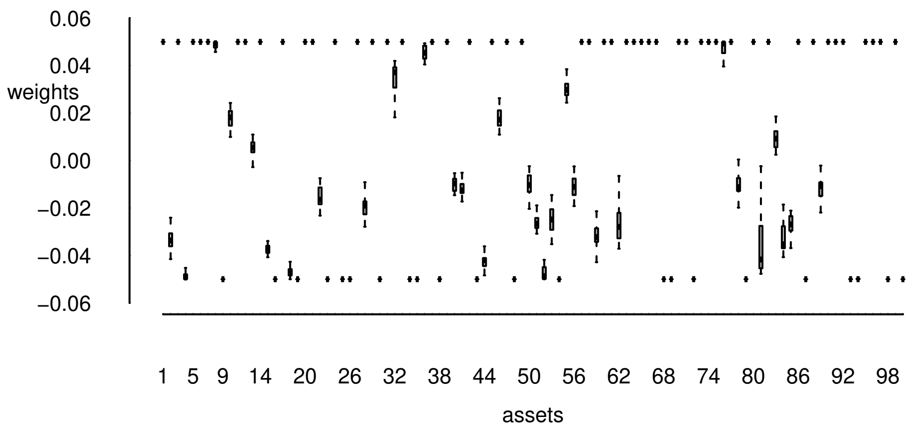
We plot the results.

```
> allres <- as.vector(rbind(res,res2))
> xlims <- pretty(allres); xlims <- c(min(xlims), max(xlims))
> par(bty = "n", las = 1, mar = c(3, 4, 0, 0),
        ps = 8, tck = 0.001)
> plot(ecdf(res[ ,3L]), xlim = xlims, cex = 0.4,
         main = "", ylab = "", xlab = "")
> for (i in 1:2)
        lines(ecdf(res[  ,i]), cex = 0.4)
> for (i in 1:3)
        lines(ecdf(res2[ ,i]), col = "blue", cex = 0.4)
```



The blue distributions are those obtained with `repair2`. We see that the distributions of the realised objective function values move to the left and become steeper, ie, they become less variable. We also check the weights, again. They also have become less variable. Many weights are at the boundaries with essentially no variation between the restarts.

```
> weightsDE <- sapply(restartsDE, `[[`, "xbest")
> par(bty = "n", las = 1, mar = c(3, 4, 0, 0),
  ps = 8, tck = 0.001)
> boxplot(t(weightsDE),
  outline = FALSE, boxwex = 0.4, ylim = c(-0.06, 0.06))
> mtext("assets",  side = 1, line = 2)
> mtext("weights", side = 2, line = 1.3, las = 1, padj = -4)
```



Exercise 7.1 *Compare two ways to implement the constraints with DE: the first is like described above, in which the population is actually repaired. In the alternative version, do not repair, but only 'map' infeasible solutions to feasible ones (Maringer and Oyewumi, 2007).*

7.3. Particle Swarm

The function PSopt is very similar to DEopt; thus, we can rerun the example almost without any changes with Particle Swarm.

```
> algo <- list(nP = 100L,      ## population size
               nG = 1000L,     ## number of generations
               c1 = 0.5,       ## weight for individually best solution
               c2 = 1.5,       ## weight for overall best solution
               min = Data$min,
               max = Data$max,
               repair = repair, pen = penalty,
               iner = 0.7, initV = 1, maxV = 0.2,
               printBar = FALSE, printDetail = TRUE)
> system.time(sol <- PSopt(OF = OF,algo = algo,Data = Data))
```

```
Particle Swarm Optimisation.
Best solution has objective function value 0.000106 ;
standard deviation of OF in final population is 5.88e-06 .
```

```
> 16 * 100 * sqrt(sol$OFvalue)      ## solution quality
> ## check constraints
> all(all.equal(sum(sol$xbest),1),  ## budget constraint
  sol$xbest <= Data$max,
  sol$xbest >= Data$min)
```

With PS we can easily impose a restriction on how a solution is changed by adjusting the velocity. We can, for instance, enforce the budget constraint by changing the weights such that the sum of the weight changes is zero.

```
> changeV <- function(x, Data) {
      myFun <- function(x) x - (sum(x))/Data$na
      if (is.null(dim(x)[2L]))
          myFun(x) else apply(x, 2L, myFun)
  }
> sum(changeV(x0, Data))
```

```
[1] -7.97e-17
```

```
> colSums(changeV(cbind(x0, x0, x0), Data))
```

```
       x0        x0        x0
-7.97e-17 -7.97e-17 -7.97e-17
```

We set up an initial population that meets the budget constraint.

```
> initP <- Data$min + diag(Data$max - Data$min) %*%
      array(runif(length(Data$min) * algo$nP),
            dim = c(length(Data$min),  algo$nP))
> colSums(initP <- repair(initP,Data))[1:10] ## check
```

```
 [1] 1 1 1 1 1 1 1 1 1 1
```

We add the function changeV and the initial population to algo.

```
> algo$changeV <- changeV        ## function to adjust velocity
> algo$initP <- initP            ## initial population
> algo$repair <- NULL            ## not needed anymore
> system.time(sol <- PSopt(OF = OF,algo = algo, Data = Data))
```

```
Particle Swarm Optimisation.
Best solution has objective function value 7.5e-05 ;
standard deviation of OF in final population is 0.0261 .

   user  system elapsed
   65.2    62.8    10.9
```

```
> 16 * 100 * sqrt(sol$OFvalue)   ## solution quality
```

```
[1] 13.9
```

We check whether the results violate the constraints.

```
> all(all.equal(sum(sol$xbest), 1), ## budget constraint
  sol$xbest <= Data$max,
  sol$xbest >= Data$min)
```

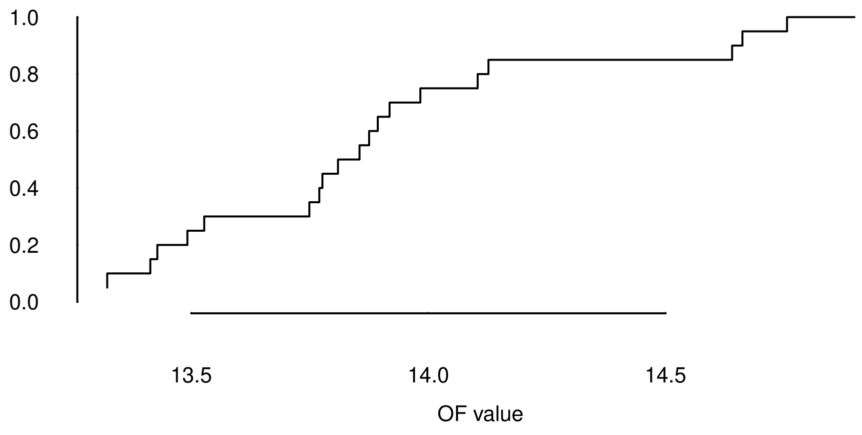
```
[1] TRUE
```

```
> algo$loopOF <- FALSE; algo$loopPen <- FALSE
> algo$loopRepair <- FALSE; algo$loopChangeV <- FALSE
> system.time(sol <- PSopt(OF = OF, algo = algo, Data = Data))
```

```
Particle Swarm Optimisation.
Best solution has objective function value 7.81e-05 ;
standard deviation of OF in final population is 0.0272 .

   user  system elapsed
   17.6    17.2     2.9
```

Finally, we can also run a small experiment here.

```
> algo$printDetail <- FALSE
> restartsPS <- restartOpt(fun = PSopt,
                           n = 20L,
                           OF = OF,
                           algo = algo, Data = Data,
                           cl = 2)
> ## extract best solution
> OFvaluesPS <- sapply(restartsPS, `[[`, "OFvalue")
> OFvaluesPS <- 16 * 100 * sqrt(OFvaluesPS)
> par(bty = "n", las = 1,mar = c(3,4,0,0),
      ps = 8, tck = 0.001)
> plot(sort(OFvaluesPS),
       (seq_len(length(OFvaluesPS))) / length(OFvaluesPS),
       type = "S", ylim = c(0, 1), xlab = "", ylab = "")
> mtext("OF value",  side = 1, line = 2)
```



7.4. Threshold Accepting

Now we solve the same problem with Threshold Accepting (TA). We first define a neighbourhood function and an objective function (in fact, we could have used the same objective function as for DE before; but this one is a bit simpler since it will never have to evaluate several solutions at once).

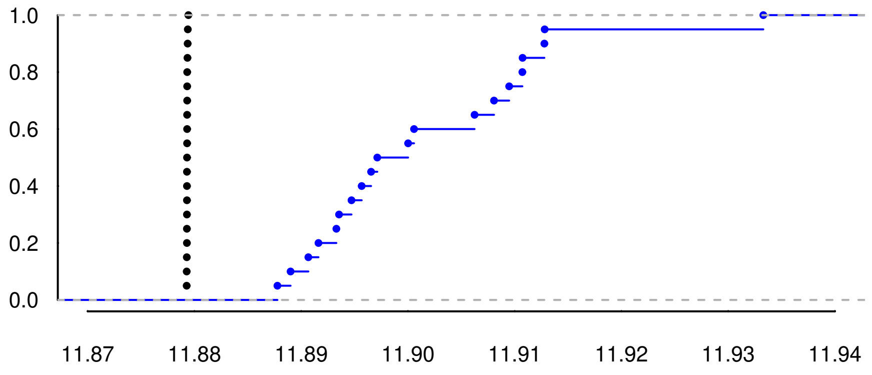
```
> Data$R <- R  ## not transposed any more
> neighbourU <- function(sol, Data){
      resample <- function(x, ...)
          x[sample.int(length(x), ...)]
      wn <- sol$w
      toSell <- wn > Data$winf
      toBuy  <- wn < Data$wsup
      i <- resample(which(toSell), size = 1L)
      j <- resample(which(toBuy), size = 1L)
      eps <- runif(1) * Data$eps
      eps <- min(wn[i] - Data$winf, Data$wsup - wn[j], eps)
      wn[i] <- wn[i] - eps
      wn[j] <- wn[j] + eps
      Rw <- sol$Rw + Data$R[,c(i,j)] %*% c(-eps,eps)
      list(w = wn, Rw = Rw)
  }
> OF <- function(x, Data) {
      Rw <- x$Rw - Data$theta
      Rw <- Rw - abs(Rw)
      sum(Rw*Rw) / (4*Data$ns)
  }
```

Next we choose a random initial solution, put all the settings in a list algo and run TA.

```
> w0 <- runif(Data$na); w0 <- w0/sum(w0)
> x0 <- list(w = w0, Rw = R %*% w0)
> algo <- list(x0 = x0,
               neighbour = neighbourU,
               nS = 2000L,
               nT = 10L,
               nD = 5000L,
               q = 0.20,
               printBar = FALSE,
               printDetail = FALSE)
> system.time(sol2 <- TAopt(OF,algo,Data))
> 16 * 100 * sqrt(sol2$OFvalue)
```

Finally, we also let the algorithm run several times. We can compare the solutions with those of DE (in blue).

```
> restartsTA <- restartOpt(fun = TAopt,
                           n = 20L,
                           OF = OF,
                           algo = algo,
                           Data = Data,
                           cl = 2)
> OFvaluesTA <- sapply(restartsTA, `[[`, "OFvalue") ## extract best solution
> OFvaluesTA <- 16 * 100 * sqrt(OFvaluesTA)
> weightsTA <- sapply(restartsTA, `[[`, "xbest")
> par(bty = "n", las = 1,mar = c(3,4,0,0), ps = 8,
      tck = 0.001, mgp = c(3, 0.5, 0))
> ## blue: DE solution with nP = 200 and nG = 2000
> xlims <- pretty(c(res2[,3], OFvaluesTA))
> plot(ecdf(res2[,3]), col = "blue", cex = 0.4,
       main = "", ylab = "", xlab = "",
       xlim = c(min(xlims), max(xlims)) )
> ## black: TA
> lines(ecdf(OFvaluesTA), cex = 0.4)
```



8. Equal-risk contribution

In this chapter we discuss how to compute equal-risk contribution portfolios – a.k.a. as risk-parity portfolios. As a benchmark, we use the functions provided in Bernhard Pfaff's FRAPO package.

```
> require("NMOF")
> require("FRAPO")
```

With Local Search.

```
> erc <- function(cov,  wmin = 0, wmax = 1, method = "ls") {

      fun <- function(x, Data) {
          tmp <- Data$S %*% x
          sd(x * tmp / c(sqrt(x %*% tmp)))
      }

      N <- function (w, Data) {
          toSell <- which(w > Data$wmin)
          toBuy <- which(w < Data$wmax)
          i <- toSell[sample.int(length(toSell), size = 1L)]
          j <- toBuy[sample.int(length(toBuy), size = 1L)]
          eps <- Data$epsmin + runif(1L) *
              (Data$epsmax-Data$epsmin) * (Data$nS-LS.info()$s)/Data$nS
          eps <- min(w[i] - Data$wmin, Data$wmax - w[j], eps)
          w[i] <- w[i] - eps
          w[j] <- w[j] + eps
          w
      }

      Data <- list(S = cov,
                   na = dim(cov)[1L],
                   wmin = 0,
                   wmax = 1,
                   epsmin = 0.0001,
                   epsmax = 0.1,
                   eps = 0.0005,
                   nS = 1000)

      sol <- LSopt(fun,
                   list(neighbour = N,
                        nS = Data$nS,
                        x0 = rep(1/Data$na, Data$na),
                        printDetail = FALSE, ## print info every 1000 step
                        printBar = FALSE),
                   Data)
      w <- sol$xbest
      w
  }
> set.seed(24244)
> S <- cov(fundData)
> vols <- c(0.05, 0.2)
> C <- c(1, 0,
         0, 1)
> dim(C) <- c(2,2)
> erc(diag(vols) %*% C %*% diag(vols))
```

```
[1] 0.8 0.2
```

```
> vols <- c(0.05, 0.2, 0.2)
> C <- c(1, 0, 0,
         0, 1, 1,
         0, 1, 1)
> dim(C) <- c(3,3)
> erc(diag(vols) %*% C %*% diag(vols))
```

```
[1] 0.739 0.131 0.131
```

```
> vols <- c(0.2, 0.05, 0.05)
> C <- c(1, 0, 0,
         0, 1, 1,
         0, 1, 1)
> dim(C) <- c(3,3)
> erc(diag(vols) %*% C %*% diag(vols))
```

```
[1] 0.150 0.425 0.425
```

```
> vols <- c(0.2, 0.05, 0.05)
> C <- c(1, 0, 0,
         0, 1, 0,
         0, 0, 1)
> dim(C) <- c(3,3)
> erc(diag(vols) %*% C %*% diag(vols))
```

```
[1] 0.111 0.444 0.445
```

```
> vols <- c(rates = 0.03, bonds = 0.06,
            equity = 0.12, commodities = 0.12)
```

```
> C <- c(1, 0, 0, 0,
         0, 1, 0, 0,
         0, 0, 1, 0,
         0, 0, 0, 1)
> dim(C) <- c(4,4)
> w <- erc(diag(vols) %*% C %*% diag(vols))
> round(100*w/sum(w))
```

```
[1] 50 25 13 13
```

```
> ## we use the dataset fundData from NMOF.
>
> require("NMOF")
> require("FRAPO")
> set.seed(24244)
> S <- cov(fundData[ ,1:50])
> ## -----------------[FRAPO]-----------------
>
> ## perc <- c(PERC(S,
> ##                ## par=sol$xbest,
> ##                control = list(abs.tol=1e-20, trace = 100))@weights)
> perc <- Weights(PERC(S)) ## FRAPO defines a generic Weights
```

```
Iteration: 0
pobj: 0
dobj: 2.91246
pinf: 1
dinf: 1
dgap: 51

Iteration: 1
pobj: 1.87693
dobj: 0.187281
pinf: 0.117354
dinf: 0.647111
dgap: 5.04169

Iteration: 2
pobj: 1.95823
dobj: 2.0103
pinf: 0.0288375
dinf: 0.118032
dgap: 0.271926

Iteration: 3
pobj: 1.69388
dobj: 1.88359
pinf: 0.0192786
dinf: 0.0641783
```

```
dgap: 0.0273029

Iteration: 4
pobj: 1.35214
dobj: 1.54212
pinf: 0.0167817
dinf: 0.0397297
dgap: 0.00346372

Iteration: 5
pobj: 1.04081
dobj: 1.21749
pinf: 0.0151601
dinf: 0.0232771
dgap: 0.000647534

Iteration: 6
pobj: 0.790766
dobj: 0.947684
pinf: 0.0133079
dinf: 0.0107125
dgap: 0.000125333

Iteration: 7
pobj: 0.724945
dobj: 0.819877
pinf: 0.00803009
dinf: 0.00422406
dgap: 3.51092e-05

Iteration: 8
pobj: 0.728335
dobj: 0.766192
pinf: 0.00319755
dinf: 0.00110173
dgap: 3.5436e-06

Iteration: 9
pobj: 0.739515
dobj: 0.756324
pinf: 0.00141968
dinf: 0.000472063
dgap: 1.27035e-06

Iteration: 10
pobj: 0.748074
dobj: 0.753132
pinf: 0.00042716
dinf: 0.000130865
```

```
dgap: 2.61754e-07

Iteration: 11
pobj: 0.751996
dobj: 0.752748
pinf: 6.35185e-05
dinf: 1.81009e-05
dgap: 2.47385e-08

Iteration: 12
pobj: 0.752685
dobj: 0.752738
pinf: 4.44726e-06
dinf: 1.25264e-06
dgap: 1.4676e-09

Iteration: 13
pobj: 0.752735
dobj: 0.752738
pinf: 2.262e-07
dinf: 6.36873e-08
dgap: 7.35877e-11

Optimal solution found.
```

```
> ## -----------------[NMOF]------------------
>
> ### --- objective function
> fun <- function(x, Data) {
      tmp <- Data$S %*% x
      sd(x * tmp / c(sqrt(x %*% tmp)))
  }
> ### --- neighbourhood function
> N <- function (w, Data) {
      toSell <- which(w > Data$wmin)
      toBuy <- which(w < Data$wmax)
      i <- toSell[sample.int(length(toSell), size = 1L)]
      j <- toBuy[sample.int(length(toBuy), size = 1L)]
      eps <- Data$epsmin + runif(1L) *
          (Data$epsmax-Data$epsmin) * (Data$nS-LS.info()$s)/Data$nS
      eps <- min(w[i] - Data$wmin, Data$wmax - w[j], eps)
      w[i] <- w[i] - eps
      w[j] <- w[j] + eps
      w
  }
> ### ---
> Data <- list(S = S,
              na = dim(S)[[1]],
              wmin = 0,
               wmax = 1,
               epsmin = 0.0001,
               epsmax = 0.01,
               eps = 0.0005,
               nS = 100000)
> ### --- run LSopt
> sol <- LSopt(fun,
               list(neighbour = N,
                    nS = Data$nS,
                    x0 = rep(1/Data$na, Data$na),
                    printDetail = 1000, ## print info every 1000 steps
                    printBar = FALSE),
               Data)
```

```
Local Search.
Initial solution:  0.000194
Best solution (step 1000/100000): 2.45e-05
Best solution (step 2000/100000): 1.44e-05
Best solution (step 3000/100000): 1.09e-05
Best solution (step 4000/100000): 7.5e-06
Best solution (step 5000/100000): 5.16e-06
Best solution (step 6000/100000): 4.63e-06
Best solution (step 7000/100000): 3.62e-06
Best solution (step 8000/100000): 3.39e-06
Best solution (step 9000/100000): 2.78e-06
Best solution (step 10000/100000): 2.64e-06
Best solution (step 11000/100000): 2.53e-06
Best solution (step 12000/100000): 2.5e-06
Best solution (step 13000/100000): 2.33e-06
Best solution (step 14000/100000): 2.26e-06
Best solution (step 15000/100000): 2.11e-06
Best solution (step 16000/100000): 2.11e-06
Best solution (step 17000/100000): 1.95e-06
Best solution (step 18000/100000): 1.92e-06
Best solution (step 19000/100000): 1.91e-06
Best solution (step 20000/100000): 1.64e-06
Best solution (step 21000/100000): 1.57e-06
Best solution (step 22000/100000): 1.55e-06
Best solution (step 23000/100000): 1.38e-06
Best solution (step 24000/100000): 1.38e-06
Best solution (step 25000/100000): 1.38e-06
Best solution (step 26000/100000): 1.38e-06
Best solution (step 27000/100000): 1.38e-06
Best solution (step 28000/100000): 1.27e-06
Best solution (step 29000/100000): 1.25e-06
Best solution (step 30000/100000): 1.25e-06
Best solution (step 31000/100000): 1.19e-06
Best solution (step 32000/100000): 1.19e-06
Best solution (step 33000/100000): 1.19e-06
Best solution (step 34000/100000): 1.18e-06
Best solution (step 35000/100000): 1.18e-06
Best solution (step 36000/100000): 1.12e-06
Best solution (step 37000/100000): 1.12e-06
Best solution (step 38000/100000): 1.04e-06
Best solution (step 39000/100000): 1.04e-06
Best solution (step 40000/100000): 1.04e-06
Best solution (step 41000/100000): 1.04e-06
Best solution (step 42000/100000): 1.02e-06
Best solution (step 43000/100000): 1.01e-06
Best solution (step 44000/100000): 9.79e-07
Best solution (step 45000/100000): 9.31e-07
Best solution (step 46000/100000): 9.31e-07
Best solution (step 47000/100000): 9.31e-07
Best solution (step 48000/100000): 9.31e-07
Best solution (step 49000/100000): 8.73e-07
Best solution (step 50000/100000): 8.73e-07
Best solution (step 51000/100000): 8.73e-07
Best solution (step 52000/100000): 8.61e-07
Best solution (step 53000/100000): 8.61e-07
Best solution (step 54000/100000): 8.61e-07
Best solution (step 55000/100000): 8.61e-07
Best solution (step 56000/100000): 8.61e-07
Best solution (step 57000/100000): 8.61e-07
Best solution (step 58000/100000): 8.61e-07
Best solution (step 59000/100000): 8.61e-07
Best solution (step 60000/100000): 8.59e-07
Best solution (step 61000/100000): 8.59e-07
Best solution (step 62000/100000): 8.59e-07
Best solution (step 63000/100000): 8.59e-07
Best solution (step 64000/100000): 8.59e-07
Best solution (step 65000/100000): 8.59e-07
Best solution (step 66000/100000): 8.59e-07
Best solution (step 67000/100000): 8.59e-07
Best solution (step 68000/100000): 8.59e-07
Best solution (step 69000/100000): 8.59e-07
Best solution (step 70000/100000): 8.59e-07
Best solution (step 71000/100000): 8.59e-07
Best solution (step 72000/100000): 8.59e-07
Best solution (step 73000/100000): 8.59e-07
Best solution (step 74000/100000): 8.59e-07
Best solution (step 75000/100000): 8.59e-07
Best solution (step 76000/100000): 8.59e-07
Best solution (step 77000/100000): 8.59e-07
Best solution (step 78000/100000): 8.59e-07
Best solution (step 79000/100000): 8.59e-07
Best solution (step 80000/100000): 8.56e-07
Best solution (step 81000/100000): 8.56e-07
Best solution (step 82000/100000): 8.56e-07
Best solution (step 83000/100000): 8.53e-07
Best solution (step 84000/100000): 8.53e-07
Best solution (step 85000/100000): 8.53e-07
Best solution (step 86000/100000): 8.37e-07
Best solution (step 87000/100000): 8.29e-07
Best solution (step 88000/100000): 8.29e-07
Best solution (step 89000/100000): 8.29e-07
Best solution (step 90000/100000): 8.29e-07
Best solution (step 91000/100000): 8.29e-07
Best solution (step 92000/100000): 8.29e-07
Best solution (step 93000/100000): 8.29e-07
Best solution (step 94000/100000): 8.22e-07
Best solution (step 95000/100000): 8.2e-07
Best solution (step 96000/100000): 8.2e-07
Best solution (step 97000/100000): 8.17e-07
Best solution (step 98000/100000): 8.17e-07
Best solution (step 99000/100000): 8.06e-07
Best solution (step 100000/100000): 8e-07
Finished.
Best solution overall: 8e-07
```

```
> w <- sol$xbest
> ### --- compare weights with FRAPO
> f <- function(x)
      format(round(x, 2), nsmall = 2)
> data.frame(TA = f(w*100), FR = f(perc))
```

```
          TA   FR
Asset1  7.01 7.01
Asset2  1.50 1.50
Asset3  1.12 1.12
Asset4  5.07 5.07
Asset5  1.56 1.57
Asset6  0.81 0.81
Asset7  0.79 0.79
Asset8  0.81 0.81
Asset9  0.91 0.91
Asset10 2.42 2.42
Asset11 1.23 1.23
Asset12 1.91 1.91
Asset13 2.34 2.34
Asset14 2.28 2.28
Asset15 1.70 1.70
Asset16 2.04 2.05
Asset17 4.33 4.33
Asset18 0.95 0.96
Asset19 1.86 1.86
Asset20 1.00 0.99
Asset21 1.45 1.45
Asset22 1.24 1.24
```

```
Asset23 1.10 1.10
Asset24 0.92 0.92
Asset25 2.59 2.59
Asset26 1.40 1.40
Asset27 1.36 1.37
Asset28 1.12 1.12
Asset29 1.23 1.23
Asset30 3.86 3.85
Asset31 3.25 3.24
Asset32 4.08 4.09
Asset33 2.09 2.09
Asset34 2.74 2.73
Asset35 1.73 1.73
Asset36 1.26 1.26
Asset37 2.54 2.55
Asset38 1.26 1.26
Asset39 1.04 1.04
Asset40 4.52 4.52
Asset41 2.00 2.00
Asset42 0.98 0.98
Asset43 1.63 1.63
Asset44 1.93 1.93
Asset45 0.96 0.95
Asset46 1.00 1.00
Asset47 1.22 1.21
Asset48 1.03 1.03
Asset49 1.11 1.11
Asset50 5.73 5.72
```

```
> cor(w, perc)
```

```
[1] 1
```

```
> ### --- compare OF values: lower is better
> fun(perc/100, Data)  ## FRAPO
```

```
[1] 5.18e-09
```

```
> fun(w, Data) ## NMOF
```

```
[1] 8e-07
```

```
> ### --- plot
> par(mfcol = c(2,2))
> plot(mrc(w, Data$S, TRUE), ylim = c(0,1.5), ylab = "in %",
       main = "Marginal risk contributions: NMOF")
> plot(mrc(perc, Data$S, TRUE), ylim = c(0,1.5), ylab = "in %",
       main = "Marginal risk contributions: FRAPO")
> plot(as.matrix(data.frame(NMOF = w*100, FRAPO = perc)),
       main = "Weights")
> ## allow short positions
>
>
>
>
>
>
>
> ## intuition
> na <- 3
> ns <- 10
> R <- array(rnorm(ns*na),
             dim = c(ns, na))
> vols <- seq(0.1,0.5, length.out = na)/16
> R <- R %*% diag(vols)
> apply(R,2,sd)*16
```

```
[1] 0.0927 0.2949 0.4700
```

```
> w <- rep(1/na, na)
> ctb <- R %*% diag(w)
> summary(ctb)
```

```
       V1                  V2                  V3
 Min.   :-0.00327   Min.   :-0.01290   Min.   :-0.00815
 1st Qu.:-0.00186   1st Qu.:-0.00376   1st Qu.:-0.00410
 Median :-0.00024   Median :-0.00145   Median : 0.00188
 Mean   :-0.00035   Mean   :-0.00217   Mean   : 0.00243
 3rd Qu.: 0.00133   3rd Qu.: 0.00218   3rd Qu.: 0.00544
 Max.   : 0.00195   Max.   : 0.00603   Max.   : 0.02636
```

```
> boxplot(ctb)
> rowSums(ctb)
```

```
 [1]  0.000698 -0.010016 -0.000371  0.003781 -0.022389
 [6]  0.002545 -0.014811  0.027655  0.004855  0.007193
```

```
>
```

9. Objective functions, neighbourhoods and constraints

9.1. Implementing objective functions for portfolio optimisation

In this section we discuss how specific objective functions for portfolio selection can be implemented; emphasis is on fast computation.

Note that even if some of these functions are faster than available implementations, that is not to be read as they are “better”. Speed may be valuable, but it is just one desirable property among many others, in particular, robustness (eg, handling missing values) and flexibility (eg, a function may work on different kinds of inputs).

9.1.1. Data

Data will always be a return-scenario matrix of `no` rows and `na` columns.

```
> na <- 50
> no <- 5000
> D <- array(rnorm(na*no)*0.01, dim = c(no,na))
> w <- runif(na)
> w <- w/sum(w)
> R <- D %*% w
```

We will also try with the `compiler` package.

```
> require("compiler")
```

9.1.2. Variance

Benchmark is var (see GMS, p. 397).

```
> var1 <- function(R) {
      n <- NROW(R)
      m <- sum(R)/n
      crossprod(R)/(n-1) - m^2
  }
> var(R) - var1(R)
```

```
          [,1]
[1,] -2.54e-13
```

The difference is purely numerical, but our implementation is not stable.

```
> var1 <- function(R) {
      n <- NROW(R)
      m <- sum(R)/n
      crossprod(R)/(n - 1) - m^2
  }
> var2 <- cmpfun(var1)
> runs <- 10000
> system.time(for (i in seq_len(runs))
              ignore <- var(R))
```

```
   user  system elapsed
  0.253   0.000   0.252
```

```
> system.time(for (i in seq_len(runs))
              ignore <- var1(R))
```

```
   user  system elapsed
  0.207   0.000   0.207
```

```
> system.time(for (i in seq_len(runs))
              ignore <- var2(R))
```

```
   user  system elapsed
  0.209   0.000   0.209
```

9.1.3. Partial moments

A straightforward implemenation could look like pm0.

```
> pm0 <- function(x, xp = 2, threshold = 0, lower = TRUE) {
      n <- NROW(x)
      x <- x - threshold
      if (lower)
          x <- x[x < 0] else x <- x[x > 0]
      sum(x^xp)/n
  }
> pm1 <- function(x, xp = 2, threshold = 0, lower = TRUE, keep.sign = FALS
      x <- x - threshold
```

```
        if (lower)
            x <- x - abs(x)
        else
            x <- x + abs(x)
        sx <- sign(x)
        x <- abs(x)
        if (xp == 1L)
            sum(x)/2/length(x)
        else if (xp == 2L)
            sum(x*x)/4/length(x)
        else if (xp == 3L)
            sum(x*x*x)/8/length(x)
        else if (xp == 4L)
            sum(x*x*x*x)/16/length(x)
        else
            sum(x^xp)/2^xp/length(x)
  }
> pm2 <- cmpfun(pm1)
```

For the default settings, there seems little difference.

```
> pm0(R)
```

```
[1] 1.23e-06
```

```
> pm1(R)
```

```
[1] 1.23e-06
```

```
> pm2(R)
```

```
[1] 1.23e-06
```

```
> runs <- 1000
> system.time(for (i in seq_len(runs))
              ignore <- pm0(R))
```

```
   user  system elapsed
  0.048   0.000   0.049
```

```
> system.time(for (i in seq_len(runs))
              ignore <- pm1(R))
```

```
   user  system elapsed
  0.080   0.011   0.091
```

```
> system.time(for (i in seq_len(runs))
              ignore <- pm2(R))
```

```
   user  system elapsed
  0.070   0.003   0.074
```

```
> pm0(R,2.2)
```

```
[1] NaN
```

```
> pm1(R,2.2)
```

```
[1] 3.64e-07
```

```
> pm2(R,2.2)
```

```
[1] 3.64e-07
```

```
> runs <- 1000
> system.time(for (i in seq_len(runs))
              ignore <- pm0(R, 2.5))
```

```
   user  system elapsed
  0.331   0.000   0.332
```

```
> system.time(for (i in seq_len(runs))
              ignore <- pm1(R, 2.5))
```

```
   user  system elapsed
  0.136   0.015   0.152
```

```
> system.time(for (i in seq_len(runs))
              ignore <- pm2(R, 2.5))
```

```
   user  system elapsed
  0.148   0.004   0.153
```

9.2. Neighbourhood functions for LSopt and TAopt

The neighbourhood is the most important aspect of TA. Neighbourhood functions have the tendency to become complicated; in particular, if we incorporate more knowledge about the problem to be solved. Nevertheless, they are almost always built from simple building blocks (at least for data structures like vectors or matrices). We will discuss a number of such building blocks for different cases:

- continuous decision variables
- continuous and integer (or categorical) parameters
- specific cases: portfolio optimisation
- minimum and maximum constraints
- ranges and 0 (eg $[-5,-1]\,[0]\,[1-5]$)

9.2.1. Logical vectors

Data, and comparing vectors

A typical problem is to flip one or a few elements of a vector of logicals (eg, for choosing a subset). As an example, we create a vector x and store its length, which we call size.

```
> size <- 20L
> x <- logical(size)
> x[runif(size) > 0.5] <- TRUE
> ## store information
> Data <- list()
> Data$size <- size
```

We first define a function to compare logical vectors.

```
> compareLogicals <- function(x,y, sep = "", ## true = "1", false = "0",
                              mark = "^", below = TRUE) {
      mark.line <- ifelse(x == y, " ", mark)
      if (!below)
          cat(mark.line, "\n", sep = sep)
      cat(as.integer(x), "\n",
          as.integer(y), "\n", sep = sep)
      if (below)
          cat(mark.line, "\n", sep = sep)

      sxy <- sum(x != y)
      if (!sxy)
          cat("The vectors do not differ.\n", sep = "")
      else if (sxy == 1L)
          cat("The vectors differ in 1 place.\n", sep = "")
      else
          cat("The vectors differ in ", sum(x != y), " place(s).\n", sep = "")
      invisible(x != y)
  }
>
```

`compareLogicals` will print the vectors like `001110` and indicate differences by a `^`. Example:

```
> compareLogicals(x, x)    ## there should be no difference
```

```
11100101010000000011
11100101010000000011

The vectors do not differ.
```

But now we compare two vectors that actually do differ.

```
> z <- x; z[2L] <- !z[2L]
> compareLogicals(x, z)
```

```
11100101010000000011
10100101010000000011
 ^
The vectors differ in 1 place.
```

Switch elements

We want to switch *n* elements of a logical vector (ie, make them `TRUE` of they are `FALSE`, or make them `FALSE` if they are `TRUE`).

```
> Data$n <- 5L  ## how many elements to change
> neighbour <- function(x, Data) {
      ii <- sample.int(Data$size, Data$n)
      x[ii] <- !x[ii]
      x
  }
> compareLogicals(x, neighbour(x, Data))
```

```
11100101010000000011
10100100110101000011
 ^     ^^  ^ ^
The vectors differ in 5 place(s).
```

Exchange two elements

Pick one `TRUE` and one `FALSE` element, and switch both. This way, the cardinality will not be changed. (The function requires that x has at least one `TRUE` and one `FALSE` element.)

```
> neighbour <- function(x, Data) {
      ## required: x must have at least one TRUE and one FALSE
      Ts <- which(x)
      Fs <- which(!x)
      lenTs <- length(Ts)
      O <- sample.int(lenTs,  1L)
      I <- sample.int(Data$size - lenTs, 1L)
      x[c(Fs[I], Ts[O])] <- c(TRUE, FALSE)
      x
  }
> compareLogicals(x, neighbour(x, Data))
```

```
11100101010000000011
11000101010010000011
  ^         ^
The vectors differ in 2 place(s).
```

9.2.2. Numeric vectors

We change all elements of the solution by adding a bit of noise.

```
> size <- 5L
> x0 <- runif(size)
> xTRUE <- runif(size)
> Data <- list(xTRUE = xTRUE,
               step = 0.02)
> OF <- function(x, Data)
      max(abs(x - Data$xTRUE))
> neighbour <- function(x, Data)
      x + runif(length(Data$xTRUE))*Data$step - Data$step/2
> algo <- list(q = 0.05, nS = 1000L, nT = 10L,
               neighbour = neighbour, x0 = x0,
               printBar = FALSE,
               printDetail = FALSE,
               storeSolutions = TRUE,
               storeF = TRUE)
> res <- TAopt(OF, algo = algo, Data = Data)
> res$OFvalue < 0.005
```

```
[1] TRUE
```

The obvious problem: what is a good step size? We can run experiments to find out; but even better is to use use knowledge about problem: use step sizes such that changes in the objective function are meaningful, or use meaningful changes in the decision variables.

A multiplicative constraint

We implement a neighbourhood for a constaint $a * b$ = constant

9.2.3. Testing a neighbourhood

GMS, Section 13.3.4, discuss a number of strategies to test neighbourhoods.

Random starting values and random walks

```
> ## N1: This neighbour enforces a budget constraint, a non-negativity
> ## constraint and a maximum holding size
>
> Data <- list(wmax = 0.22,  ## the maximal weight
               eps = 0.2/100,  ## the step size
               ## resample = function(x, ...)
               ##            x[sample.int(length(x), ...)],
               na = dim(fundData)[2L],
               R = fundData)
> cat("The portfolio will consist of at least ",
      ceiling(1/Data$wmax), " assets.\n", sep = "")
```

```
The portfolio will consist of at least 5 assets.
```

```
> neighbour1 <- function(w, Data){
      toSell <- which(w > 0)
      toBuy  <- which(w < Data$wmax)
      i <- toSell[sample.int(length(toSell), size = 1L)]
      j <- toBuy[ sample.int(length(toBuy),  size = 1L)]
      eps <- runif(1) * Data$eps
      eps <- min(w[i], Data$wmax - w[j], eps)
      w[i] <- w[i] - eps
      w[j] <- w[j] + eps
      w
  }
> neighbour1U <- function(x, Data){
      wn <- x$w
      toSell <- which(wn > 0)
      toBuy  <- which(wn < Data$wmax)
      i <- toSell[sample.int(length(toSell), size = 1L)]
      j <- toBuy[ sample.int(length(toBuy),  size = 1L)]
      eps <- runif(1) * Data$eps
      eps <- min(wn[i], Data$wmax - wn[j], eps)
      wn[i] <- wn[i] - eps
      wn[j] <- wn[j] + eps
      Rw <- x$Rw + Data$R[ ,c(i,j)] %*% c(-eps,eps)
      list(w = wn, Rw = Rw)
  }
> ## create a random solution
> makex <- function(Data) {
      resample <- function(x, ...)
          x[sample.int(length(x), ...)]
      w0 <- numeric(Data$na)
      nAssets <- resample(ceiling(1/Data$wmax):Data$na, 1L)
      w0[sample(seq_len(Data$na), nAssets)] <- runif(nAssets)
      w0/sum(w0)
  }
> isOK <- function(w, Data) {
      tooBig   <- any(w > Data$wmax)
      tooSmall <- any(w < 0)
      sumToOne <- abs(sum(w)-1) < 1e-12
      if (!tooBig && !tooSmall && sumToOne)
          TRUE
      else
          FALSE
  }
> ## TEST 1
> w0 <- makex(Data)
> x0 <- list(w = w0, Rw = fundData %*% w0)
> isOK(w0, Data)
```

```
[1] TRUE
```

```
> isOK(x0$w, Data)
```

```
[1] TRUE
```

```
> set.seed(545)
> w0 <- makex(Data)
> nTests <- 1e3
> for (i in seq(nTests)) {
      w1 <- neighbour1(w0,  Data)
      if (isOK(w1, Data))
          w0 <- w1
      else
          stop("error")
  }
> set.seed(545)
> w0 <- makex(Data)
> x0 <- list(w = w0, Rw = fundData %*% w0)
> nTests <- 1e3
> for (i in seq(nTests)) {
      x1 <- neighbour1U(x0,  Data)
      if (isOK(x1$w, Data))
```

```
          x0 <- x1
      else
          stop("error")
  }
> all.equal(fundData %*% w1, x1$Rw)
```

```
[1] TRUE
```

```
> ## TEST 2: reach a target solution
> makeOF <- function(wt)
      function(w0, Data)
          sum(abs(wt - w0))
> wt <- makex(Data)
> OF <- makeOF(wt)
> w0 <- makex(Data)
> OF(w0, Data)
```

```
[1] 0.989
```

```
> TAsettings <- list(neighbour = neighbour1,
                     x0 = w0, nS = 5000, q = 0.1,
                     printBar = FALSE)
> res <- TAopt(OF, algo = TAsettings, Data)
```

```
Threshold Accepting

  Computing thresholds ...   OK
  Estimated remaining running time: 1.32 secs

  Running Threshold Accepting ...
  Initial solution: 0.989
  Finished.
  Best solution overall: 0.00312
```

```
> round(head(sort(abs(res$xbest-wt), decreasing = TRUE),5),6)
```

```
[1] 0.000109 0.000077 0.000061 0.000060 0.000059
```

```
> ## N2: This long-only neighbour enforces a budget constraint, a
> ## non-negativity constraint, and a maximum holding
> ## size and a maximum cardinality.
>
> Data <- list(wmax = 0.3, ## the maximal weight
               Kmax = 10,  ## max cardinality
               eps = 1/100, ## the step size
               ## resample = function(x, ...)
               ##                 x[sample.int(length(x), ...)],
             na = dim(fundData)[2L],
             R = fundData)
> cat("The portfolio will consist of at least ",
      ceiling(1/Data$wmax), " assets.\n", sep = "")
```

```
The portfolio will consist of at least 4 assets.
```

```
> neighbour2 <- function(w, Data){
      tol <- 1e-12
      J <- sum(w > tol)
      if (J == Data$Kmax)
          toBuy <- which(w > tol & w < Data$wmax)
      else
          toBuy <- which(w < Data$wmax)
      toSell <- which(w > tol)
      i <- toSell[sample.int(length(toSell), size = 1L)]
      j <- toBuy[ sample.int(length(toBuy),  size = 1L)]
      eps <- runif(1) * Data$eps
      eps <- min(w[i], Data$wmax - w[j], eps)
      w[i] <- w[i] - eps
      w[j] <- w[j] + eps
      w
  }
> neighbour2U <- function(x, Data){
      tol <- 1e-12
      w <- x$w
      J <- sum(w > tol)
      if (J == Data$Kmax)
          toBuy <- which(w > tol & w < Data$wmax)
      else
          toBuy <- which(w < Data$wmax)
      toSell <- which(w > tol)
      i <- toSell[sample.int(length(toSell), size = 1L)]
      j <- toBuy[ sample.int(length(toBuy),  size = 1L)]
      eps <- runif(1) * Data$eps
      eps <- min(w[i], Data$wmax - w[j], eps)
      w[i] <- w[i] - eps
      w[j] <- w[j] + eps
      Rw <- x$Rw + Data$R[ ,c(i,j)] %*% c(-eps, eps)
      list(w = w, Rw = Rw)
  }
> makex <- function(Data) {
      w0 <- numeric(Data$na)
      nAssets <- sample(ceiling(1/Data$wmax):Data$Kmax, 1L)
      w0[sample(seq_len(Data$na), nAssets)] <- runif(nAssets)
      w0/sum(w0)
  }
> isOK <- function(w, Data) {
      tooBig   <- any(w > Data$wmax)
```

```
        tooMany <- sum(w > 1e-12) > Data$Kmax
        sumToOne <- abs(sum(w)-1) < 1e-12
        if (!tooBig && !tooMany && sumToOne)
            TRUE
        else
            FALSE
    }
> ## TEST 1
> w0 <- makex(Data)
> x0 <- list(w = w0, Rw = fundData %*% w0)
> isOK(w0, Data)
```

```
[1] TRUE
```

```
> isOK(x0$w, Data)
```

```
[1] TRUE
```

```
> set.seed(545)
> w0 <- makex(Data)
> nTests <- 1e3
> for (i in seq(nTests)) {
        w1 <- neighbour2(w0,  Data)
        if (isOK(w1, Data))
            w0 <- w1
        else
            stop("error")
    }
> set.seed(545)
> w0 <- makex(Data)
> x0 <- list(w = w0, Rw = fundData %*% w0)
> nTests <- 1e3
> for (i in seq(nTests)) {
        x1 <- neighbour2U(x0,  Data)
        if (isOK(x1$w, Data))
            x0 <- x1
        else
            stop("error")
    }
> all.equal(fundData %*% w1, x1$Rw)
```

```
[1] TRUE
```

```
> ## TEST 2: reach a target solution
> makeOF <- function(wt)
        function(w0, Data)
            sum(abs(wt - w0))
> wt <- makex(Data)
> OF <- makeOF(wt)
> w0 <- makex(Data)
> OF(w0, Data)
```

```
[1] 1.73
```

```
> OF(wt, Data)
```

```
[1] 0
```

```
> TAsettings <- list(neighbour = neighbour2,
                     x0 = w0, nS = 5000, q = 0.1,
                     printBar = FALSE)
> res <- TAopt(OF, algo = TAsettings, Data)
```

```
Threshold Accepting

  Computing thresholds ...   OK
  Estimated remaining running time: 1.45 secs

  Running Threshold Accepting ...
  Initial solution: 1.73
  Finished.
  Best solution overall: 0.000166
```

```
> isOK(res$xbest, Data)
```

```
[1] TRUE
```

```
> df <- data.frame(target=wt, w0 = w0, wTAopt = res$xbest)
> tmpfun <- function(x)
      !all(abs(x) < 1e-14)
> df[apply(df,1,tmpfun),]
```

```
    target     w0 wTAopt
6   0.0000 0.1273 0.0000
18  0.0497 0.0000 0.0496
42  0.1423 0.0000 0.1424
47  0.0219 0.0000 0.0219
56  0.0000 0.1951 0.0000
65  0.0385 0.0000 0.0386
93  0.0000 0.0419 0.0000
102 0.2044 0.0000 0.2044
109 0.1386 0.1374 0.1386
113 0.0164 0.0000 0.0164
125 0.0000 0.0134 0.0000
149 0.1759 0.0000 0.1760
```

```
159 0.2123 0.0000 0.2123
169 0.0000 0.0407 0.0000
187 0.0000 0.0310 0.0000
190 0.0000 0.2200 0.0000
197 0.0000 0.1933 0.0000
```

```
> apply(df, 2, sum)
```

```
target     w0 wTAopt
     1      1      1
```

```
> wt <- numeric(200)
> wt[1:4] <- c(0.3,0.3,0.3,0.1)
> OF <- makeOF(wt)
> TAsettings <- list(neighbour = neighbour2,
                     x0 = w0, nS = 5000, q = 0.1,
                     printBar = FALSE)
> res <- TAopt(OF, algo = TAsettings, Data)
```

```
Threshold Accepting

  Computing thresholds ...   OK
  Estimated remaining running time: 1.35 secs

  Running Threshold Accepting ...
  Initial solution: 2
  Finished.
  Best solution overall: 9.71e-16
```

```
> isOK(res$xbest, Data)
```

```
[1] TRUE
```

```
> df <- data.frame(target=wt, w0 = w0, wTAopt = res$xbest)
> tmpfun <- function(x)
      !all(abs(x) < 1e-14)
> df[apply(df,1,tmpfun),]
```

```
    target     w0 wTAopt
1      0.3 0.0000    0.3
2      0.3 0.0000    0.3
3      0.3 0.0000    0.3
4      0.1 0.0000    0.1
6      0.0 0.1273    0.0
56     0.0 0.1951    0.0
93     0.0 0.0419    0.0
109    0.0 0.1374    0.0
```

```
125    0.0 0.0134    0.0
169    0.0 0.0407    0.0
187    0.0 0.0310    0.0
190    0.0 0.2200    0.0
197    0.0 0.1933    0.0
```

```
> apply(df, 2, sum)
```

```
target     w0 wTAopt
     1      1      1
```

```
> w0 <- makex(Data)
> x0 <- list(w = w0, Rw = fundData %*% w0)
> ## the N is slower
> system.time(for (i in 1:10000) neighbour2(w0, Data))
```

```
   user  system elapsed
  0.238   0.000   0.239
```

```
> system.time(for (i in 1:10000) neighbour2U(x0, Data))
```

```
   user  system elapsed
  0.317   0.004   0.321
```

```
> TAsettings2 <- list(neighbour = neighbour2,
                      x0 = w0, nS = 500, q = 0.1,
                      printBar = FALSE, printDetail = FALSE)
> TAsettings2U <- list(neighbour = neighbour2U,
                       x0 = x0, nS = 500, q = 0.1,
                       printBar = FALSE, printDetail = FALSE)
> ofun <- function(w, Data) {
      Rw <- Data$R %*% w
      crossprod(Rw)
  }
> ofunU <- function(sol, Data)
      crossprod(sol$Rw)
> ign <- TAopt(ofun, TAsettings2, Data)
> ign <- TAopt(ofunU, TAsettings2U, Data)
>
> ##benchmark(ign <- TAopt(ofun, TAsettings2, Data),
> ##          ign <- TAopt(ofunU, TAsettings2U, Data),
> ##          replications = 1, order = "relative")
```

Finding a particular solution

We should be able to move from a given solution to a desired solution. (See also (Burns, 2010) on the problem of finding solutions.)

9.2.4. Finding a feasible starting solution

9.3. Repairing or penalising solutions?

We use Threshold Accepting and compare two ways to include constraints: through a penality and thorugh a repair function. The particular application will be portfolio optimisation

9.3.1. The problem

9.3.2. Enforcing the constraint via the neighbourhood

9.3.3. Enforcing the constraint via a penalty

9.3.4. Comparing both methods

9.4. Examples

9.4.1. Absolute position size

For a 130/30 portfolio, `lim` must not exceed 1.6

```
> maxabs <- function(x, lim)
      max(sum(abs(x)) - lim, 0)
```

9.4.2. Buy-in Thresholds

```
> require("compiler")
> wmin <- 0.01
> wmax <- 0.10
> w <- numeric(50)
> w[1:10] <- 0.1
> step <- 0.01
> N <- function(w) {

      wo <- w
      ## initial sell
      sel <- which(w >= wmin)
      i <- sel[sample.int(length(sel), size = 1)]

      if (w[i] == wmin) {
          eps <- wmin
          w[i] <- 0
```

```
        } else {
            eps <- min(runif(1)*step , w[i] - wmin)
            w[i] <- w[i] - eps
        }

        cash <- eps

        iter <- 0
        while (abs(cash) > 1e-14) {
            iter <- iter + 1
            if (iter > 10) {
                return(wo)
            }
            ##message("cash ", cash)
            if (cash > 0) {  ## buy something
                sel <- which(w < wmax)
                i <- sel[sample.int(length(sel), size = 1)]
                if (w[i] == 0) {
                    w[i] <- eps <- wmin
                } else {
                    eps <- min(runif(1)*step , wmax - w[i], cash)
                    w[i] <- w[i] + eps
                }
                cash <- cash - eps
            } else { ## sell something
                sel <- which(w >= wmin)
                i <- sel[sample.int(length(sel), size = 1)]
                if (w[i] == wmin) {
                    eps <- wmin
                    w[i] <- 0
                } else {
                    eps <- min(runif(1)*step , w[i] - wmin)
                    w[i] <- w[i] - eps
                }
                cash <- cash + eps
            }
        }
        ##message(iter)
        w
    }
> ##N <- cmpfun(N)
> ##w
> system.time(for (i in 1:10000) w <- N(w))
```

```
   user  system elapsed
  0.418   0.001   0.419
```

```
>
> ## goal <- numeric(50)
> ## goal[41:50] <- 0.1
>
> ## OF <- function(x) {
> ##     tmp <- x - goal
> ##     sum(tmp * tmp)
> ## }
> ## ans <- LSopt(OF, algo = list(nS = 1000000, neighbour = N, x0 = w))
> ## sum(ans$xbest > 0)
> ## ans$xbest
>
> ## ans <- numeric(10000)
> ## for (i in seq_along(ans))
> ##     ans[i] <- ceiling(runif(1)*10)
>
> ## system.time(for (i in 1:10000) ignore <- ceiling(runif(5)*99))
> ## system.time(for (i in 1:10000) ignore <- sample.int(99, 5))
>
```

10. Traditional portfolio selection models

A main topic of Gilli et al. (2019) is non-standard portfolio-selection models; see Chapters 12–14. Nevertheless, the `NMOF` package also offers several functions that help with standard portfolio models, i.e. models that can be solved with traditional optimisation techniques such as quadratic programming.

10.1. Minimum-variance portfolios

minvar

The function `minvar` computes the minimum-variance portfolio for a given variance–covariance matrix, subject to holding-size constraints. As example data, the variable `var` contains a small variance–covariance matrix, computed from daily returns of five German stocks. The data are taken from `http://enricoschumann.net/data/gilli_accuracy.html`; the code to build the matrix is in the source file of this vignette.

```
> var
```

```
           CBK.DE    VOW.DE   CON.DE   LIN.DE  MUV2.DE
CBK.DE   0.000988 -1.80e-05 3.69e-04 2.08e-04 2.63e-04
VOW.DE  -0.000018  1.72e-03 8.57e-05 2.15e-05 2.84e-05
CON.DE   0.000369  8.57e-05 7.59e-04 1.94e-04 1.89e-04
LIN.DE   0.000208  2.15e-05 1.94e-04 2.66e-04 1.33e-04
MUV2.DE  0.000263  2.84e-05 1.89e-04 1.33e-04 2.59e-04
```

An example call, with minimum and maximum holding sizes specified.

```
> minvar(var, wmin = 0, wmax = 0.5)
```

```
[1] 6.94e-18 9.25e-02 4.69e-05 4.45e-01 4.62e-01
attr(,"variance")
[1] 0.000182
```

The function returns the portfolio weights with an attribute `variance` that provides the variance of this portfolio. The holding size constraints can also be specified as vectors, with different values for different assets.

```
> minvar(var,
         wmin = c(0.1, 0, 0, 0, 0), ## enforce at least 10% weight in CBK.
         wmax = 0.5)
```

Use Inf to switch off weight constraints.

```
> minvar(var, wmin = -Inf, wmax = Inf)   ## no bounds
> minvar(var, wmin = -Inf, wmax = 0.45)  ## no lower bounds
> minvar(var, wmin =  0.1, wmax = Inf)   ## no upper bounds
```

The function also supports group constraints:

```
> ## group 1 consists of asset 1 only,   and must have weight [0.25,0.30]
> ## group 2 consists of assets 4 and 5, and must have weight [0.10,0.20]
> minvar(var, wmin = 0, wmax = 0.40,
         groups = list(1, 4:5),
         groups.wmin = c(0.25, 0.1),
         groups.wmax = c(0.30, 0.2))
```

```
[1] 0.250 0.217 0.333 0.149 0.051
attr(,"variance")
[1] 0.000357
```

Alternatively, group constraints can be specified through group names instead of positions.

```
> ## group A consists of asset 1 only,   and must have weight [0.25,0.30]
> ## group B consists of assets 4 and 5, and must have weight [0.10,0.20]
> minvar(var, wmin = 0, wmax = 0.40,
         groups = c("A", "none", "none", "B", "B"),
         groups.wmin = c(A = 0.25, B = 0.1),
         groups.wmax = c(A = 0.30, B = 0.2))
```

```
[1] 0.250 0.217 0.333 0.149 0.051
attr(,"variance")
[1] 0.000357
```

10.2. Mean–variance efficient portfolios and frontiers

mvPortfolio

The function mvPortfolio computes a mean–variance-efficient portfolio for a given variance–covariance matrix and mean-return assumption, subject to holding-size constraints. We make up some data for four assets, with a constant correlation of 0.5.

```
> vols <- c(0.10, 0.15, 0.20, 0.22)  ## expected vols
> m    <- c(0.06, 0.12, 0.09, 0.07)  ## expected mean returns
> const_cor <- function(rho, na) {
      C <- array(rho, dim = c(na, na))
      diag(C) <- 1
      C
  }
> var <- diag(vols) %*% const_cor(0.5, length(vols)) %*% diag(vols)
```

One way to compute a mean–variance-efficient portfolio is by requiring a minimum return.

```
> mvPortfolio(m, var, min.return = 0.08, wmax = 1)
```

```
[1] 0.667 0.333 0.000 0.000
```

```
> mvPortfolio(m, var, min.return = 0.10, wmax = 1)
```

```
[1] 3.33e-01 6.67e-01 0.00e+00 3.10e-18
```

```
> mvPortfolio(m, var, min.return = 0.12, wmax = 1)
```

```
[1] -1.11e-16  1.00e+00 -5.55e-17  2.64e-17
```

Alternatively, we may specify a trade-off between return and variance and minimise

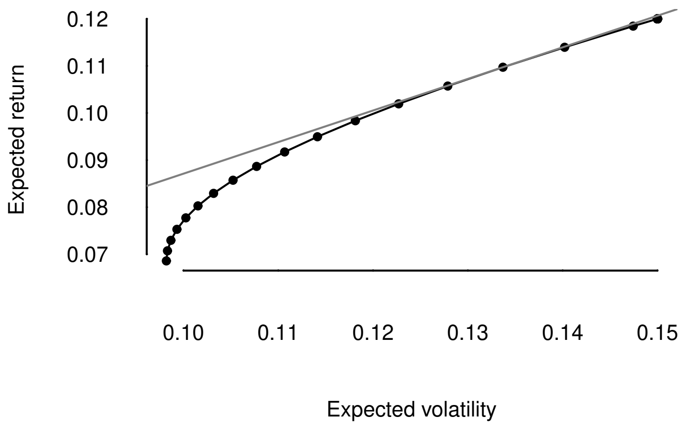
$$-\lambda \mathtt{m}'w + \frac{1}{2}(1-\lambda)w'\mathtt{var}\,w\,,$$

in which w are the weights. If λ is a vector of length 2, then the function minimises

$$-\lambda_1 \mathtt{m}'w + \frac{1}{2}\lambda_2 w'\mathtt{var}\,w\,.$$

The function `mvFrontier` traces out a whole frontier of mean–variance efficient portfolios. (But see the discussion on frontiers in Chapter 14 of Gilli et al., 2019.) mvFrontier

```
> wmin <- 0
> wmax <- 1
> p1 <- mvFrontier(m, var, wmin = wmin, wmax = wmax, n = 50)
> ## with a 'risk-free' asset rf
> rf <- 0.02
> p2 <- mvFrontier(m, var, wmin = wmin, wmax = wmax, n = 50, rf = rf)
> par(las = 1, bty = "n", tck = 0.001, ps = 8)
> plot(p1$volatility, p1$return, pch = 19, cex = 0.5, type = "o",
       xlab = "Expected volatility",
       ylab = "Expected return")
> lines(p2$volatility, p2$return, col = grey(0.5))
> abline(v = 0, h = rf)
```



10.3. Return-based tracking portfolios

trackingPortfolio

Function `trackingPortfolio` computes a portfolio that is close to another portfolio in the mean-square/variance sense. The function to be minimised is determined by argument `objective`: supported are `variance` (the default) or `sum.of.squares`.

```
> ns <- 120
> R <- randomReturns(na = 1 + 10,  ## first asset is the benchmark
                     ns = ns,
                     sd = 0.03,
                     mean = 0.005,
                     rho = 0.7)
> var <- cov(R)
> trackingPortfolio(var, wmax = 0.4)
```

```
 [1] 1.53e-01 1.72e-01 3.47e-18 9.64e-02 1.07e-01 1.61e-01
 [7] 9.50e-02 1.64e-01 2.48e-02 2.65e-02
```

10.4. Minimum-Absolute-Deviation (MAD) portfolios

Assume a matrix R of returns, with n_{A} columns (one for each asset) and n_{S} rows (one row for each scenario). For given portfolio weights w, we can compute the portfolio returns as Rw. A bare-bones model could then be the following:

$$\min_{w} \ \mathrm{mean}(|Rw|) \tag{10.1}$$

subject to

$$w'\iota = 1 \tag{10.2}$$

$$\left[\ \underbrace{1}_{\alpha}\quad \underbrace{0\ \cdots\ 0}_{x}\quad \underbrace{\tfrac{1}{(1-\beta)n_{\mathrm{S}}}\ \ \cdots\ \ \tfrac{1}{(1-\beta)n_{\mathrm{S}}}}_{u}\ \right]$$

10.5. Minimum-Expected-Shortfall portfolios

The function minCVaR computes a portfolio that minimises conditional Value-at-Risk; its default method is the LP approach described in Rockafellar and Uryasev (2000). See *Minimising Conditional Value-at-Risk (CVaR)* (http://enricoschumann.net/notes/minimis for more details

minCVaR

```
> ns <- 5000  ## number of scenarios
> na <- 20    ## nunber of assets
> R <- randomReturns(na, ns, sd = 0.01, rho = 0.5)
> sol <- minCVaR(R, q = 0.1)
```

11. Fitting yield curves with Differential Evolution

The material in this section was taken from the vignette 'Fitting the Nelson–Siegel–Svensson model with Differential Evolution' because the examples would run several minutes (at least they did in 2012 or so).

11.1. Fitting the Nelson–Siegel–Svensson model to given bond prices

A bond is a list of payment dates (given a valuation date, we can translate them into times-to-payment) and associated payments.

```
> makeCashFlows <- function(coupon, T) {
      t1 <- T - floor(T)                  ## time to first coupon
      tm <- seq(ifelse(t1 > 1e-5, t1, 1), ## 1e-5 is less than a calendar day
                T,
                by = 1)
      cf <- rep.int(coupon, length(tm))
      cf[length(cf)] <- cf[length(cf)] + 100
      list(cf = cf, tm = tm)
  }
> makeCashFlows(3, 10.2)
```

Suppose we are given the following set of bonds.

```
> cf1 <- c(rep(5.75,  8), 105.75); tm1 <- 0:8 + 0.5
> cf2 <- c(rep(4.25, 17), 104.25); tm2 <- 1:18
> cf3 <- c(3.5, 103.5); tm3 <- 0:1 + 0.5
> cf4 <- c(rep(3.00, 15), 103.00); tm4 <- 1:16
> cf5 <- c(rep(3.25, 11), 103.25); tm5 <- 0:11 + 0.5
> cf6 <- c(rep(5.75, 17), 105.75); tm6 <- 0:17 + 0.5
> cf7 <- c(rep(3.50, 14), 103.50); tm7 <- 1:15
> cf8 <- c(rep(5.00,  8), 105.00); tm8 <- 0:8 + 0.5
> cf9 <- 105; tm9 <- 1
> cf10 <- c(rep(3.00, 12), 103.00); tm10 <- 0:12 + 0.5
> cf11 <- c(rep(2.50,  7), 102.50); tm11 <- 1:8
> cf12 <- c(rep(4.00, 10), 104.00); tm12 <- 1:11
> cf13 <- c(rep(3.75, 18), 103.75); tm13 <- 0:18 + 0.5
> cf14 <- c(rep(4.00, 17), 104.00); tm14 <- 1:18
> cf15 <- c(rep(2.25,  8), 102.25); tm15 <- 0:8 + 0.5
> cf16 <- c(rep(4.00,  6), 104.00); tm16 <- 1:7
> cf17 <- c(rep(2.25, 12), 102.25); tm17 <- 1:13
> cf18 <- c(rep(4.50, 19), 104.50); tm18 <- 0:19 + 0.5
> cf19 <- c(rep(2.25,  7), 102.25); tm19 <- 1:8
> cf20 <- c(rep(3.00, 14), 103.00); tm20 <- 1:15
```

We put all cash flows into a matrix `cfMatrix`, such that one bond is one column, and one row corresponds to one payment date.

```
> cfList <- list( cf1, cf2, cf3, cf4, cf5, cf6, cf7, cf8, cf9,cf10,
                 cf11,cf12,cf13,cf14,cf15,cf16,cf17,cf18,cf19,cf20)
> tmList <- list( tm1, tm2, tm3, tm4, tm5, tm6, tm7, tm8, tm9,tm10,
                 tm11,tm12,tm13,tm14,tm15,tm16,tm17,tm18,tm19,tm20)
> tm <- unlist(tmList, use.names = FALSE)
> tm <- sort(unique(tm))
> nR <- length(tm)
> nC <- length(cfList)
> cfMatrix <- array(0, dim = c(nR, nC))
> for(j in seq(nC))
      cfMatrix[tm %in% tmList[[j]], j] <- cfList[[j]]
> rownames(cfMatrix) <- tm
> cfMatrix[1:10, 1:10]
```

Suppose we have zero rates for all maturities (ie, one for each row of `cfMatrix`), then we can transform this vector of rates into discount factors. Premultiplying `cfMatrix` by the row vector of discount factors then gives us a row vector of bond prices.

```
> betaTRUE <- c(5,-2,1,10,1,3)
> yM <- NSS(betaTRUE,tm)
> diFa <- 1 / ( (1 + yM/100)^tm )
> bM <- diFa %*% cfMatrix
```

So, with a vector of 'true' bond prices `bm`, we can set up DE.

```
> Data <- list(bM = bM, tm = tm, cfMatrix = cfMatrix, model = NSS,
               ww = 1,
               min = c( 0,-15,-30,-30,0  ,2.5),
               max = c(15, 30, 30, 30,2.5,5  ))
```

The objective function takes the path that we just saw: given parameters for the NSS model, it computes zero rates, and transforms these into discount factors. Given the matrix `cfMatrix`, it then computes theoretical bond prices, and compares these with the given prices `bm`. As the optimisation criterion, we use the maximum absolute difference.

```
> OF2 <- function(param, Data) {
    tm <- Data$tm
    bM <- Data$bM
    cfMatrix <- Data$cfMatrix
    diFa  <- 1 / ((1 + Data$model(param, tm)/100)^tm)
    b <- diFa %*% cfMatrix
    aux <- b - bM; aux <- max(abs(aux))
    if (is.na(aux)) aux <- 1e10
    aux
  }
```

We will enforce the constraints with a penalty.

```
> penalty <- function(mP, Data) {
      minV <- Data$min
      maxV <- Data$max
      ww <- Data$ww
      ## if larger than maxV, element in A is positiv
      A <- mP - as.vector(maxV)
      A <- A + abs(A)
      ## if smaller than minV, element in B is positiv
      B <- as.vector(minV) - mP
      B <- B + abs(B)
      ## beta 1 + beta2 > 0
      C <- ww*((mP[1L, ] + mP[2L, ]) - abs(mP[1L, ] + mP[2L, ]))
      A <- ww * colSums(A + B) - C
      A
  }
```

We set up the parameters and run DE.

```
> algo <- list(nP  = 200L,
               nG  = 1000L,
               F   = 0.50,
               CR  = 0.99,
               min = c( 0,-15,-30,-30,0  ,2.5),
               max = c(15, 30, 30, 30,2.5,5  ),
               pen = penalty,
               repair = NULL,
               loopOF = TRUE,
               loopPen = FALSE,
               loopRepair = FALSE,
               printBar = FALSE,
               printDetail = FALSE,
               storeF = FALSE)
> sol <- DEopt(OF = OF2, algo = algo, Data = Data)
```

Note that now the objective function value (the difference in bond prices) does not correspond to the yield difference anymore. It is instructive to compare them nevertheless.

```
> max( abs(Data$model(sol$xbest, tm) - Data$model(betaTRUE, tm)))
```

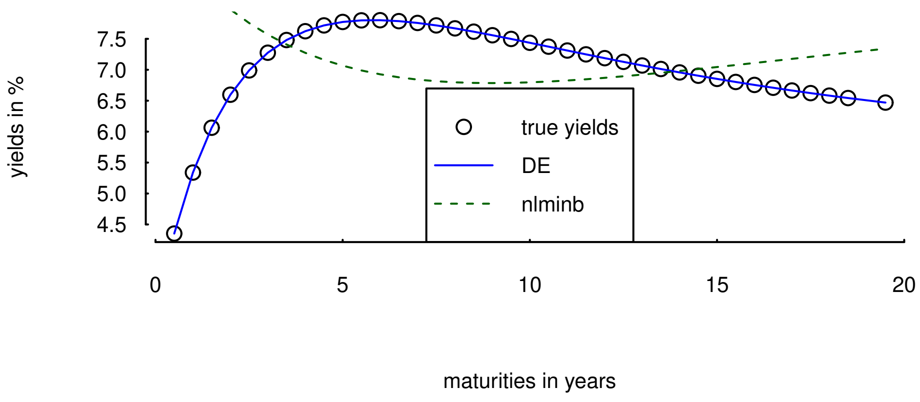
```
[1] 2.4e-14
```

```
> sol$OFvalue
```

```
[1] 0
```

…and we compare with `nlminb`.

```
> s0 <- algo$min + (algo$max - algo$min) * runif(length(algo$min))
> system.time(sol2 <- nlminb(s0,OF2,Data = Data,
                             lower = Data$min,
                             upper = Data$max,
                             control = list(eval.max = 50000,
                             iter.max = 50000)))
> max(abs(Data$model(sol2$par,tm) - Data$model(betaTRUE,tm)))
> sol2$objective
> par(ps = 8, bty = "n", las = 1, tck = 0.01,
      mgp = c(3, 0.5, 0), mar = c(4, 4, 1, 1))
> plot(tm, yM, xlab = "maturities in years", ylab = "yields in %")
> lines(tm,Data$model(sol$xbest,tm), col = "blue")
> lines(tm,Data$model(sol2$par,tm), col = "darkgreen", lty = 2)
> legend(x = "bottom", legend = c("true yields", "DE", "nlminb"),
         col = c("black", "blue", "darkgreen"),
         pch = c(1, NA, NA), lty = c(0, 1, 2))
```



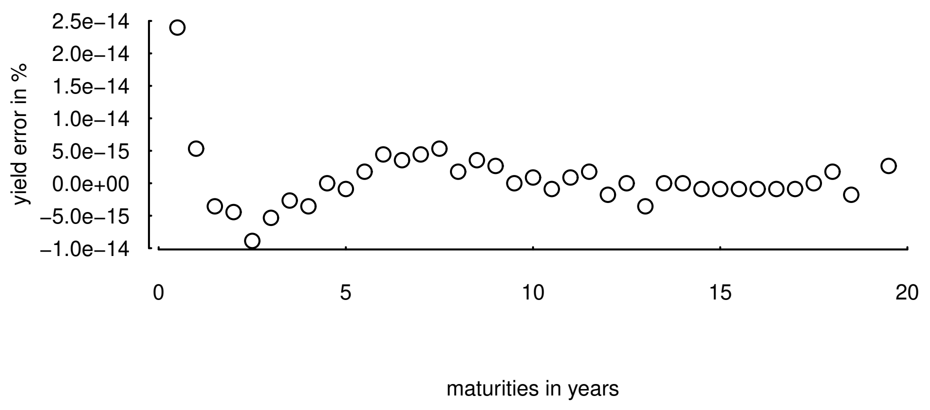
We can check the price errors.

```
> diFa <- 1 / ((1 + NSS(sol$xbest,tm)/100)^tm)
> b <- diFa %*% cfMatrix
> b - bM
```

```
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]    0    0    0    0    0    0    0    0    0     0
     [,11] [,12] [,13] [,14] [,15] [,16] [,17] [,18] [,19]
[1,]     0     0     0     0     0     0     0     0     0
     [,20]
[1,]     0
```

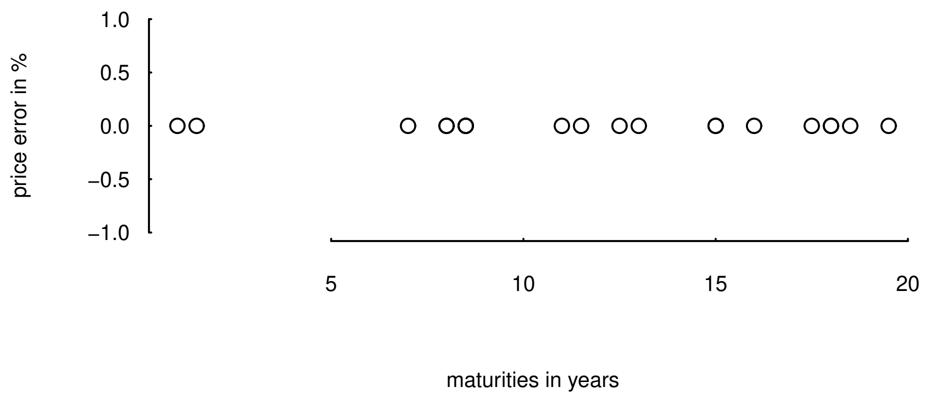
We can also plot the rate errors against time-to-payment.

```
> par(ps = 8, bty = "n", las = 1, tck = 0.01,
      mgp = c(3, 0.5, 0), mar = c(4, 4, 1, 1))
> plot(tm, NSS(sol$xbest,tm) - NSS(betaTRUE,tm),
  xlab = "maturities in years", ylab = "yield error in %")
```



These apparently systematic (albeit small) errors are less visible when we plot price errors against time-to-maturity (see the book for a discussion).

```
> par(ps = 8, bty = "n", las = 1, tck = 0.01,
  mgp = c(3, 0.5, 0), mar = c(4, 4, 1, 1))
> plot(as.numeric(unlist(lapply(tmList, max))), as.vector(b - bM),
  xlab = "maturities in years", ylab = "price error in %")
```



11.1.1. More vectorisation

We actually vectorise a bit more. We did like this to obtain the bond prices for one set of β-coefficients.

```
> beta <- c(5,-2,1,10,1,3)
> yM <- NSS(beta,tm)
> diFa <- 1 / ( (1 + yM/100)^tm )
> b <- diFa %*% cfMatrix
```

But we can actually save a number of steps.

```
> B <- cbind(c(5,-2,1,10,1,3), c(4,-2,1,10,1,3))
> Y <- array(0, dim = c(length(tm), ncol(B)))
> for (i in 1:ncol(Y))
      Y[ ,i] <- NSS(B[ ,i], tm)
> D <- 1/((1+Y/100)^tm)
> t(cfMatrix) %*% D - as.vector(b)
```

```
           [,1]   [,2]
 [1,]  0.00e+00  5.882
 [2,] -1.42e-14  9.112
 [3,]  0.00e+00  1.373
 [4,] -7.11e-15  7.648
 [5,]  0.00e+00  6.386
 [6,] -1.42e-14 10.060
 [7,]  0.00e+00  7.666
 [8,] -1.42e-14  5.692
 [9,]  0.00e+00  0.955
[10,]  0.00e+00  6.617
[11,]  0.00e+00  4.865
[12,]  0.00e+00  6.483
[13,]  0.00e+00  8.876
[14,]  0.00e+00  8.921
[15,]  0.00e+00  4.995
[16,]  0.00e+00  4.745
[17,]  0.00e+00  6.414
[18,]  0.00e+00  9.788
[19,]  0.00e+00  4.808
[20,]  0.00e+00  7.372
```

11.2. Fitting the NSS model to given yields-to-maturity

We will need the function `compYield`; it converts cash flows and times-to-payment into present values, and those present values into yields-to-maturities.

The function fy computes the present value of vector of cash flows cf at times tm.

```
> fy <- function(ytm, cf, tm)
      sum( cf / ( (1 + ytm)^tm ) )
> compYield <- function(cf, tm, guess = NULL) {
      logik <- cf != 0
      cf <- cf[logik]
      tm <- tm[logik]
      if (is.null(guess)) {ytm <- 0.05} else {ytm <- guess}
      h <- 1e-8;        dF <- 1; ci <- 0
      while (abs(dF) > 1e-5) {
          ci <- ci + 1; if (ci > 5) break
          FF  <-  fy(ytm, cf, tm)
          dFF <- (fy(ytm + h, cf, tm) - FF) / h
          dF <- FF / dFF
          ytm <- ytm - dF
      }
      if (ytm < 0)
          ytm <- 0.99
      ytm
  }
```

The objective function, OF3, looks as follows.

```
> OF3 <- function(param, Data) {
      tm <- Data$tm
      rM <- Data$rM
      cfMatrix<- Data$cfMatrix
      nB <- dim(cfMatrix)[2L]
      zrates <- Data$model(param,tm); aux <- 1e10
      if ( all(zrates > 0,
               !is.na(zrates))
          ) {
          diFa <- 1 / ((1 + zrates/100)^tm)
          b <- diFa %*% cfMatrix
          r <- numeric(nB)
          if ( all(!is.na(b),
                   diFa < 1,
                   diFa > 0,
                   b > 1)
              ) {
              for (bb in 1:nB) {
                  r[bb] <- compYield(c(-b[bb], cfMatrix[ ,bb]), c(0,tm))
              }
              aux <- abs(r - rM)
              aux <- sum(aux)
          }
```

```
        }
        aux
    }
```

So the game plan is as follows: we compute prices `b` as in the last section, but then we convert them into yields-to-maturity `r` with the function `compYield`. The objective function evaluates the discrepancy between the market yields-to-maturity `rM` and our model yields `r`. We start by defining the 'true' `rM`.

```
> betaTRUE <- c(5,-2,1,10,1,3)
> yM <- NSS(betaTRUE, tm)
> diFa <- 1 / ( (1 + yM/100)^tm )
> bM <- diFa %*% cfMatrix
> rM <- apply(rbind(-bM, cfMatrix), 2, compYield, c(0, tm))
```

We set up `Data` and `algo`.

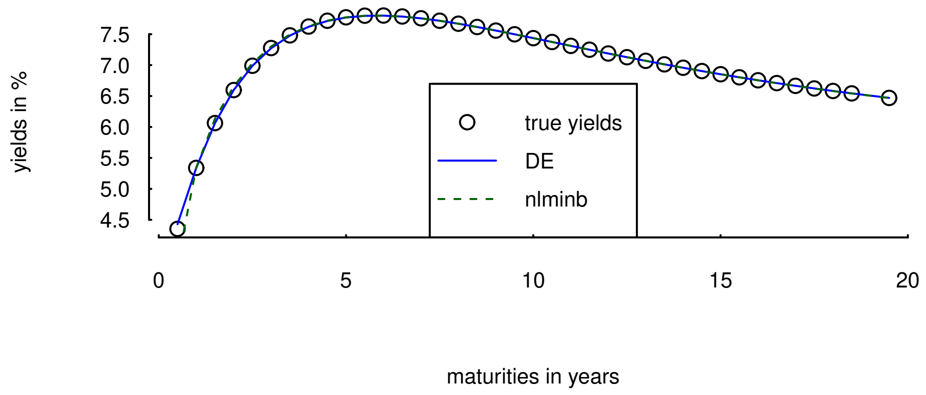
```
> Data <- list(rM = rM, tm = tm,
               cfMatrix = cfMatrix,
               model = NSS,
               min = c( 0,-15,-30,-30,0   ,2.5),
               max = c(15, 30, 30, 30,2.5,5   ),
               ww = 0.1,
               fy = fy)
> algo <- list(nP = 100L,
               nG = 1000L,
               F  = 0.50,
               CR = 0.99,
               min = c( 0,-15,-30,-30,0   ,2.5),
               max = c(15, 30, 30, 30,2.5,5   ),
               pen = penalty,
               repair = NULL,
               loopOF = TRUE,
               loopPen = FALSE,
               loopRepair = FALSE,
               printBar = FALSE,
               printDetail = FALSE)
```

```
> sol <- DEopt(OF = OF3, algo = algo, Data = Data)
> max(abs(Data$model(sol$xbest,tm) - Data$model(betaTRUE,tm)))
> sol$OFvalue
```

With `nlminb`:

```
> s0 <- algo$min + (algo$max - algo$min) * runif(length(algo$min))
> sol2 <- nlminb(s0, OF3, Data = Data,
                lower = algo$min,
                upper = algo$max,
                control = list(eval.max = 50000L,
                iter.max = 50000L))
> max(abs(Data$model(sol2$par,tm) - Data$model(betaTRUE,tm)))
> sol2$objective
```

```
> par(ps = 8, bty = "n", las = 1, tck = 0.01,
    mgp = c(3, 0.5, 0), mar = c(4, 4, 1, 1))
> plot(tm, yM, xlab = "maturities in years", ylab = "yields in %")
> lines(tm,Data$model(sol$xbest,tm), col = "blue")
> lines(tm,Data$model(sol2$par,tm), col = "darkgreen", lty = 2)
> legend(x = "bottom", legend = c("true yields","DE","nlminb"),
       col = c("black", "blue", "darkgreen"),
       pch = c(1, NA, NA), lty = c(0,1,2))
```



Compare the recovered parameters.

```
> betaTRUE
```

```
[1]  5 -2  1 10  1  3
```

```
> round(sol$xbest,3)
```

```
[1]  5.004 -1.329 -2.138 10.101  0.352  2.983
```

While the returned OF value will typically be acceptable, we need many more iterations to have the parameters converge. But compare the fitted yield curve: the fitted yields are generally fine. If you need more precision, just increase the number of generations (and possibly adjust the tolerance in the `while` condition in function `compYield`).

12. Model selection with Threshold Accepting

We load the package and set a seed.

```
> require("NMOF")
> set.seed(94679)
```

12.1. Linear models

In this section we do a simple model selection for a linear regression: out of p available regressors, select a subset such that a given selection criterion is minimised. We start with a function `randomData`; it creates a dataset `X` of `p` available regressors with `n` observations. A number `k` of these regressors are the 'true' regressors, and they define a response variable `y`:

I thank Victor Bystrov for comments on an earlier (MATLAB) version of this example.

$$y = X_{\mathrm{K}}\beta + s\epsilon \tag{12.1}$$

The variable `K` is the set of true regressors (ie, `k == length(K)`); thus, X_{K} are those columns of `X` that represent the true regressors. The number `ss` scales the residuals.

```
> randomData <- function(p = 200L,      ## number of available regressors
                         n = 200L,      ## number of observations
                         maxReg = 10L,  ## max. number of included regressors
                         s = 1,         ## standard deviation of residuals
                         constant = TRUE ) {

      X <- array(rnorm(n * p), dim = c(n, p))
      if (constant)
          X[ ,1L] <- 1

      k <- sample.int(maxReg, 1L)   ## the number of true regressors
      K <- sort(sample.int(p, k))   ## the set of true regressors
      betatrue <- rnorm(k)          ## the true coefficients

      ## the response variable y
      y <- X[ ,K] %*% as.matrix(betatrue) + rnorm(n, sd = s)

      list(X = X, y = y, betatrue = betatrue, K = K, n = n, p = p)
  }
```

We create a random dataset.

```
> rD <- randomData(p = 100L, n = 200L, s = 1,
                   constant = TRUE, maxReg = 10L)
```

We put all the data in a list called Data.

```
> Data <- list(X = rD$X,
               y = rD$y,
               n = rD$n,
               p = rD$p,
               maxk  = 30L,  ## maximum number of regressors included in m
               lognn = log(rD$n)/rD$n)
```

Next, we compute a random solution x0.

```
> x0 <- logical(Data$p)
> temp <- sample.int(Data$maxk, 1L)
> temp <- sample.int(Data$p, temp)
> x0[temp] <- TRUE
```

Such a solution is a logical vector of length p which can be used to subset the columns of X. Clearly, x0 is not going to be a particularly good solution. But it will help us to test the code and demonstrate how it works.

The true regressors…

```
> rD$K
```

```
[1]  2 31 41 50 62 81
```

…and the random solution.

```
> which(x0)
```

```
 [1]  7 14 19 20 23 43 58 66 74 90
```

12.2. Fast Least Squares

Any selection rule for a model will use the residuals of the fitted model as an ingredient. Thus, given a potential solution, we will have to compute a fit. Here we use Least Squares. Typically we would use `lm` for this. But `lm` computes a lot of things that we actually do not need: we only need the fitted coefficients to compute the residuals. Hence, we can use `qr` or `qr.solve` directly. As a test, we compute the coefficients for the random solution x0.

```
> result1 <- lm(Data$y ~ -1 + Data$X[ ,x0])
> result2 <- qr.solve(Data$X[ ,x0], Data$y)
> ## ... coefficients should be the same
> all.equal(as.numeric(coef(result1)), as.numeric(result2))
```

```
[1] TRUE
```

A timing test.

```
> require("rbenchmark")
> benchmark(lm(Data$y ~ -1 + Data$X[ ,x0]),
            qr.solve(Data$X[ ,x0], Data$y),
            columns = c("test", "elapsed", "relative"),
            order = "test",
            replications = 1000L)
```

12.3. Selection criterion

Now, for the actual selection criterion. We will use the Schwarz criterion, which is (for a linear model) given by

$$\log\left(\frac{\text{residuals}'\text{residuals}}{n}\right) + \frac{\log(n) \times \text{number of regressors}}{n}\,; \tag{12.2}$$

see for instance Johnston and DiNardo (1997). We put this computation in the objective function OF.

```
> OF <- function(x, Data) {
      q <- qr(Data$X[ ,x])
      e <- qr.resid(q, Data$y)
      log(crossprod(e)/Data$n) + sum(x) * Data$lognn
  }
```

With the random solution.

```
> OF(x0, Data)
```

```
     [,1]
[1,] 2.24
```

The final ingredient that we need is a neighbourhood function. It randomly chooses one element of a solution and switches its value. We reject solutions that include no or more than Data$maxk regressors.

```
> neighbour <- function(xc, Data) {
      xn <- xc
      ex <- sample.int(Data$p, 1L)
      xn[ex] <- !xn[ex]
      sumx <- sum(xn)
      if ( sumx < 1L || (sumx > Data$maxk) )
          xc else xn
  }
```

```
> OF(neighbour(x0, Data), Data)
```

```
     [,1]
[1,] 2.22
```

```
> OF(neighbour(x0, Data), Data)
```

```
     [,1]
[1,] 2.27
```

```
> OF(neighbour(x0, Data), Data)
```

```
     [,1]
[1,] 2.27
```

We collect all settings for the algorithm, including the neighbourhood function, in a list algo. Then we run TAopt.

```
> algo <- list(nT = 10L,    ## number of thresholds
               nS = 200L,   ## number of steps per threshold
               nD = 1000L,  ## number of random steps to compute threshold
               neighbour = neighbour,
               x0 = x0,
               printBar = FALSE)
> system.time(sol1 <- TAopt(OF, algo = algo, Data = Data))
```

```
Threshold Accepting

  Computing thresholds ...   OK
  Estimated remaining running time: 0.3 secs

  Running Threshold Accepting ...
  Initial solution: 2.24
  Finished.
  Best solution overall: 0.164
```

We check the resulting solution's objective function value `sol1$OFvalue`, and we compare the selected regressors with the true regressors.

```
> sol1$OFvalue
```

```
      [,1]
[1,] 0.164
```

```
> which(sol1$xbest)  ## the selected regressors
```

```
[1] 10 12 31 41 47 50 81 87 94
```

```
> rD$K               ## the true regressors
```

```
[1]  2 31 41 50 62 81
```

They are not the same. But in a relatively small sample we should actually not expect this to be the case. (You can increase `n` to see if the true model is eventually identified.) In fact, we can compare the value of the objective function for the true model and the selected model.

```
> xtrue <- logical(Data$p)
> xtrue[rD$K] <- TRUE
> OF(sol1$xbest, Data)
```

```
      [,1]
[1,] 0.164
```

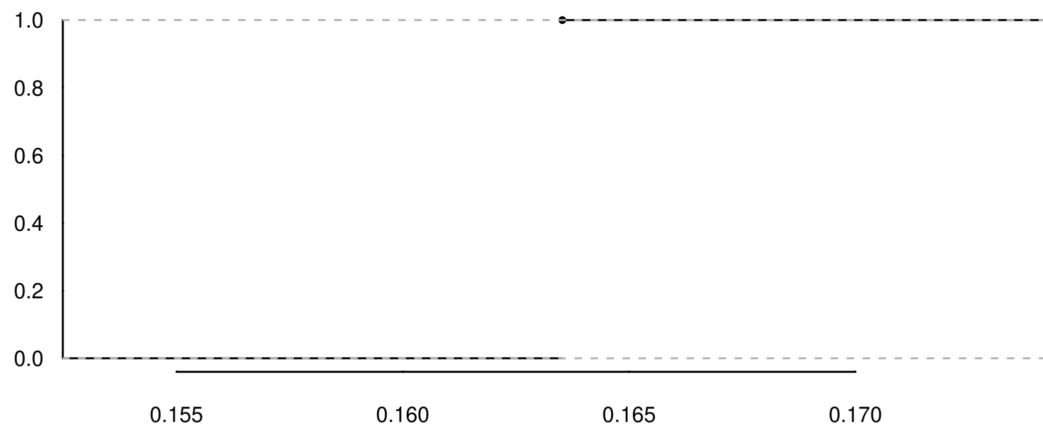
```
> OF(xtrue, Data)
```

```
      [,1]
[1,] 0.238
```

We see that the Schwarz criterion for our selected model is lower than for the true model.

Finally, we run a small experiment (note that all runs use the same starting value `x0`).

```
> restarts <- 50L
> algo$printDetail <- FALSE
> res <- restartOpt(TAopt, n = restarts,
                    OF = OF, algo = algo, Data = Data,
                    cl = 2)
> par(bty = "n", las = 1,mar = c(3,4,0,0),
      ps = 8, tck = 0.001, mgp = c(3, 0.5, 0))
> plot(ecdf(sapply(res, `[[`, "OFvalue")),  ## extract solution quality
       cex = 0.4, main = "", ylab = "", xlab = "")
```



For each solution, we compute the objective function value, and also the selected regressors.

```
> xbestAll <- sapply(res, `[[`, "xbest")    ## extract all solutions
> inclReg  <- which(rowSums(xbestAll) > 0L) ## get included regressors
> inclReg  <- sort(union(rD$K, inclReg))
> data.frame(regressor   = inclReg,
             `included`  = paste(rowSums(xbestAll)[inclReg], "/",
                                 restarts, sep = ""),
             `true regressor?` = inclReg %in% rD$K,
             check.names = FALSE)
```

```
   regressor included true regressor?
1          2     0/50            TRUE
2         10    50/50           FALSE
3         12    50/50           FALSE
4         31    50/50            TRUE
5         41    50/50            TRUE
6         47    50/50           FALSE
7         50    50/50            TRUE
8         62     0/50            TRUE
9         81    50/50            TRUE
10        87    50/50           FALSE
11        94    50/50           FALSE
```

Across the restarts, we get a relatively clear answer which regressors should, according to the Schwarz criterion, be put into the model.

```
> dim(rD$X)
```

```
[1] 200 100
```

```
> neighbour2 <- function(xc, Data) {
      if ((sumx <- sum(x0)) >= Data$maxk)
```

```
          ex <- sample(which(x0), 1L)
      else if (sumx == 1L)
          ex <- sample(which(!x0), 1L)
      else
          ex <- sample.int(Data$p, 1L)
      xc[ex] <- !xc[ex]
      xc
  }
> neighbour <- function(xc, Data) {
      xn <- xc
      ex <- sample.int(Data$p, 1L)
      xn[ex] <- !xn[ex]
      sumx <- sum(xn)
      if ( sumx < 1L || (sumx > Data$maxk) )
          xc else xn
  }
> algo <- list(nT = 20L,    ## number of thresholds
               nS = 200L,   ## number of steps per threshold
               nD = 1000L,  ## number of random steps to compute thresholds
               neighbour = neighbour,
               x0 = x0, q= 0.5,
               printBar = FALSE)
> system.time(sol1 <- TAopt(OF, algo = algo, Data = Data))
```

```
Threshold Accepting

  Computing thresholds ...   OK
  Estimated remaining running time: 0.728 secs

  Running Threshold Accepting ...
  Initial solution: 2.24
  Finished.
  Best solution overall: 0.164
   user  system elapsed
  0.623   0.001   0.625
```

```
> plot(cummin(sol1$Fmat[ ,2L]), type = "l", log = "y")
> ##rD
> OF <- function(x, Data) {
      q <- qr(Data$X[ ,x])
      e <- qr.resid(q, Data$y)
      crossprod(e)
  }
```

13. Calibrating option pricing models

In this chapter we look into fitting option pricing models to market prices.

13.1. Implied volatility

13.1.1. A single option

We first code a simple Black–Scholes–Merton formula, which we call `bsm`. We also calculate one of the Greeks, the vega. The function takes the usual parameters and the indicator `I`, which is `1` for a call and `-1` for a put.

```
> bsm <- function(S, X, tau, r, q, vol, I = 1) {
      d1 <- (log(S/X) + (r - q + vol^2/2) * tau)/
            (vol * sqrt(tau))
      d2 <- d1 - vol * sqrt(tau)
      list(value = I * (S * exp(-q * tau) * pnorm(I * d1) -
                        X * exp(-r * tau) * pnorm(I * d2)),
           vega = S * exp(-q*tau) * dnorm(d1 * I) * sqrt(tau))
  }
```

A numerical example.

```
> S <- 99   ## spot
> X <- 100  ## strike
> r <- 0.01
> q <- 0.0
> tau <- 0.25
> vol <- 0.2
> I <- 1    ## a call (-1 for a put)
> unlist(bsm(S, X, tau, r, q, vol, I))
```

```
value  vega
  3.6  19.7
```

As a check, we can use the function `vanillaOptionEuropean` that comes with the NMOF package Note that `vanillaOptionEuropean` takes variance as an input, i.e. volatility squared).

```
> tmp <- unlist(
      vanillaOptionEuropean(S = S, X = X, tau = tau,
                            r = r, q = q, v = vol^2,
                            type = ifelse(I == 1,
                                          "call", "put")))
> tmp[c("value", "vega")]
```

```
value  vega
  3.6  19.7
```

Now we code a simple Newton–Raphson root finder (see GMS, Chapter 11). We can use a `for` loop to automatically limit the number of iterations. We first compute a true price for given parameters, and then we will try to recover the volatility.

```
> S <- 99
> X <- 100
> r <- 0.01
> q <- 0.01
> tau <- 0.1
> I <- 1
> vol <- 0.247
> (price <- bsm(S, X, tau, r, q, vol, I)$value)
```

```
[1] 2.62
```

Here is the Newton–Raphson function. Note that `vol0` is the initial guess for the volatility (see also GMS, Chapter 15).

```
> impliedVol <- function(price, S, X, tau, r, q,
                         vol0 = 0.15, I = 1,
                         tol = 1e-4, maxit = 10) {

      for (i in seq_len(maxit)) {
          tmp <- bsm(S, X, tau, r, q, vol0, I)
          step <- (tmp$value - price)/tmp$vega
          vol0 <- vol0 - step
          if (all(abs(step) < tol))
              break
      }
      vol0
  }
```

You may wonder about the `all()` in the break condition; it will be explained shortly. Let us try the function; there is also a function `vanillaOptionImpliedVol` in the NMOF package, which we may use to check the result.

```
> impliedVol(price, S, X, tau, r, q, vol, I)
```

```
[1] 0.247
```

```
> vanillaOptionImpliedVol(exercise = "european",
                          price, S, X, tau, r,
                          q, type = "call")
```

```
[1] 0.247
```

impliedVol is faster then vanillaOptionImpliedVol (but the latter may become rewritten in the future).

```
> benchmark(iV = impliedVol(price, S, X, tau,
                            r, q, runif(1L) + 0.05, I),
            vanOptIV = vanillaOptionImpliedVol(
                exercise = "european",
                price, S, X, tau, r,
                q, tauD = 0, D = 0, type = "call",
                M = 101, uniroot.info = FALSE),
            columns = c("test", "elapsed", "relative"),
            replications = 1e3, order = "relative")
```

```
      test elapsed relative
1       iV   0.027     1.00
2 vanOptIV   0.144     5.33
```

13.1.2. Several options

For European options, many computations can be vectorised. First a pricing example; we reuse the function bsm, which we defined in the previous section. We only enter vectors of length greater than one as input.

```
> S <- rep(99, 21)  ## spot
> X <- 90:110       ## strike
> r <- 0.01; q <- 0.02
> tau <- 0.2; vol <- 0.24; I <- 1
> data.frame(S = S, X = X, bsm(S, X, tau, r, q, vol, I))
```

```
    S   X value vega
1  99  90 9.856 11.5
2  99  91 9.080 12.6
3  99  92 8.336 13.6
4  99  93 7.625 14.5
```

```
5  99  94 6.949 15.4
6  99  95 6.310 16.1
7  99  96 5.707 16.7
8  99  97 5.142 17.2
9  99  98 4.615 17.4
10 99  99 4.126 17.6
11 99 100 3.675 17.6
12 99 101 3.260 17.4
13 99 102 2.880 17.1
14 99 103 2.535 16.6
15 99 104 2.222 16.1
16 99 105 1.940 15.4
17 99 106 1.687 14.7
18 99 107 1.462 13.9
19 99 108 1.262 13.0
20 99 109 1.085 12.1
21 99 110 0.929 11.2
```

More usefully, let us price a surface with given maturities `tauvec` and strikes `Xvec` (all calls).

```
> Xvec <- 80:120
> tauvec <- c(c(3, 6, 9)/12,  ## 3, 6, 9 months
              1, 2, 3, 4, 5)  ## 1..5 years
```

One way the fill the surface with prices is to use two nested loops.

```
> loop <- function() {
      callprices <- array(NA, dim = c(length(Xvec), length(tauvec)))
      for (X in Xvec)
          for (tau in tauvec)
              callprices[X == Xvec, tau == tauvec] <- bsm(S,X,tau,r,q,vol)$
      callprices
  }
```

If we insist on vectorised computation, we can do it like this.

```
> vect <- function() {
      tmp <- expand.grid(Xvec,tauvec)
      callprices <- bsm(S, tmp[[1L]], tmp[[2L]], r, q, vol, I)$value
      dim(callprices) <- c(length(Xvec), length(tauvec))
      callprices
  }
```

An example follows; it checks whether `loop` and `vect` result in the same prices (they do).

```
> S <- 101
> Xvec <- 80:120
> tauvec <- c(c(3, 6, 9)/12,  ## 3, 6, 9 months
              1, 2, 3, 4, 5)  ## 1..5 years
>   r <- 0.01; q <- 0.01
>   tau <- 0.25; vol <- 0.2; I <- 1
>   callprices1 <- loop()
>   callprices2 <- vect()
>   all.equal(callprices1, callprices2)
```

```
[1] TRUE
```

The vectorised variant *may* be more obscure code, but we are compensated in terms of speed.

```
> benchmark(loop(), vect(),
            columns = c("test", "elapsed", "relative"),
            replications = 1e3, order = "relative")
```

```
    test elapsed relative
2 vect()   0.255     1.00
1 loop()   2.376     9.32
```

And now implied volatility. Again, we use the same function as before. Now the use of `all()` in the break criterion becomes apparent: only when all steps are smaller than `tol`, the loop is exited.

```
> S <- rep(99,21)  ## spot
> X <- 90:110
> r <- 0.01
> q <- 0.02
> tau <- runif(21)
> vol <- (runif(21)+0.2)/3
> ivol <- impliedVol(bsm(S, X, tau, r, q, vol, I)$value,
                     S, X, tau, r, q, vol = 0.2,
                     I, tol = 1e-5, maxit = 10)
> data.frame(S = S, X = X, vol = vol, ivol = ivol,
             diff = abs(vol-ivol))
```

```
    S   X   vol  ivol     diff
1  99  90 0.355 0.355 1.67e-16
2  99  91 0.152 0.152 5.55e-17
3  99  92 0.127 0.127 5.55e-16
4  99  93 0.186 0.186 4.44e-16
5  99  94 0.335 0.335 9.44e-16
6  99  95 0.219 0.219 1.33e-15
```

```
7  99  96 0.308 0.308 5.55e-17
8  99  97 0.155 0.155 1.11e-16
9  99  98 0.331 0.331 5.55e-17
10 99  99 0.167 0.167 3.89e-16
11 99 100 0.120 0.120 3.89e-16
12 99 101 0.323 0.323 3.33e-16
13 99 102 0.396 0.396 2.78e-16
14 99 103 0.383 0.383 2.22e-16
15 99 104 0.199 0.199 1.39e-16
16 99 105 0.380 0.380 5.55e-17
17 99 106 0.217 0.217 3.33e-16
18 99 107 0.307 0.307 7.77e-16
19 99 108 0.109 0.109 3.22e-11
20 99 109 0.340 0.340 6.11e-16
21 99 110 0.277 0.277 0.00e+00
```

And for a single option, it does not take long (and note that we compute the option price as well).

```
> system.time(
      for (i in 1:1e3)
          impliedVol(bsm(S, X, tau, r, q, vol, I)$value,
                     S, X, tau, r, q, tol = 1e-5, maxit = 5))
```

```
  user  system elapsed
 0.121   0.000   0.121
```

13.2. Alternative pricing models

```
> S <- 100    ## spot
> X <- 100    ## strike
> tau <- 1    ## time-to-maturity
> r <- 0.02   ## interest rate
> q <- 0.02   ## dividend rate
> v <- 0.2    ## volatility
```

The closed-form solution.

```
> callBSM <- function(S,X,tau,r,q,v) {
      d1 <- (log(S/X) + (r - q + v^2 / 2)*tau) / (v*sqrt(tau))
      d2 <- d1 - v*sqrt(tau)
      S * exp(-q * tau) * pnorm(d1) -  X * exp(-r * tau) * pnorm(d2)
  }
> callBSM(S,X,tau,r,q,v)
```

```
[1] 7.81
```

With the characteristic function.

```
> callCF(cf = cfBSM, S = S, X = X, tau = tau, r = r, q = q,
         v = v^2,  ## variance, not vol
         implVol = TRUE)
```

```
$value
[1] 7.81

$impliedVol
[1] 0.2
```

As a first exercise, we price a whole surface with the given parameters.

```
> Xvec <- 80:120
> tauvec <- c(c(3, 6, 9)/12,  ## 3, 6, 9 months
              1, 2, 3, 4, 5)  ## 1..5 years
```

As before, we may use loops or vectorise. The loop version first.

```
> loop2 <- function() {
      callprices <- array(NA,
                          dim = c(length(Xvec),
                                  length(tauvec)))
      for (X in Xvec)
          for (tau in tauvec)
              callprices[X == Xvec, tau == tauvec] <-
                  callBSM(S,X,tau,r,q,v)
      callprices
  }
```

And the vectorised version.

```
> vect2 <- function() {
      tmp <- expand.grid(Xvec,tauvec)
      callprices <- callBSM(S, tmp[[1]], tmp[[2]], r, q, v)
      dim(callprices) <- c(length(Xvec), length(tauvec))
      callprices
  }
```

```
> callprices1 <- loop2()
> callprices2 <- vect2()
> all.equal(callprices1, callprices2)
```

```
[1] TRUE
```

Speed difference.

```
> benchmark(loop2(), vect2(),
            columns = c("test", "elapsed", "relative"),
            replications = 1e3, order = "relative")
```

```
     test elapsed relative
2 vect2()   0.184      1.0
1 loop2()   1.999     10.9
```

Not let us move to pricing with the cf.

```
> priceMatrix <- function(cf, S, Xvec, tauvec,
                          r, q = 0, ...,
                          nodes = NULL, weights = NULL,
                          n = 200) {

      if (is.null(nodes)) {
          tmp <- xwGauss(n)
          tmp <- changeInterval(tmp$nodes, tmp$weights,
                                oldmin = -1, oldmax = 1,
                                newmin =  0, newmax = 200)
          nodes <- tmp$nodes
          weights <- tmp$weights
      }

      callprices <- array(NA, dim = c(length(Xvec), length(tauvec)))
      tmpmat <- array(NA, dim = c(length(Xvec), length(weights)))

      inodes <- 1i * nodes
      itau <- 0L
      for (tau in tauvec) {
          itau <- itau + 1L
          cfi <- S * exp((r - q) * tau)
          cf1 <- cf(nodes - 1i, S, tau, r, q, ...)/inodes/cfi
          cf2 <- cf(nodes,      S, tau, r, q, ...)/inodes
          iX <- 0L
          for (X in Xvec) {
              iX <- iX + 1L
              if (itau == 1L)
                  tmpmat[iX, ] <- exp(-inodes * log(X))
              P1 <- 0.5 + weights %*% Re(tmpmat[iX, ] * cf1)/pi
              P2 <- 0.5 + weights %*% Re(tmpmat[iX, ] * cf2)/pi
              callprices[iX, itau] <-
```

```
                exp(-q*tau) * S * P1 - exp(-r*tau) * X * P2
        }
    }
     callprices
  }
> system.time(
     for (i in 1:100)
         ignore <- priceMatrix(cf = cfBSM, S, Xvec, tauvec, r, q = q,
                    v = 0.2^2, n = 50) )
```

```
   user  system elapsed
  1.307   1.174   0.307
```

```
> require("compiler")
> priceMatrix2 <- cmpfun(priceMatrix)
> system.time(
     for (i in 1:100)
         ignore <- priceMatrix2(cf = cfBSM, S, Xvec, tauvec, r, q = q,
                    v = 0.2^2, n = 50) )
```

```
   user  system elapsed
  1.419   1.258   0.324
```

```
>
```

It should not come as a surprise that the classical Black–Scholes–Merton formula is faster.

```
> cfp <- priceMatrix(cf = cfBSM, S, Xvec, tauvec, r, q = q,
                    v = 0.2^2, n = 100)
> callprices1[1:5, 1:5]
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,] 19.9 20.1 20.4 20.8 22.2
[2,] 19.0 19.2 19.5 19.9 21.5
[3,] 18.0 18.3 18.7 19.1 20.8
[4,] 17.0 17.4 17.9 18.4 20.1
[5,] 16.1 16.5 17.0 17.6 19.4
```

```
> callprices2[1:5, 1:5]
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,] 19.9 20.1 20.4 20.8 22.2
[2,] 19.0 19.2 19.5 19.9 21.5
[3,] 18.0 18.3 18.7 19.1 20.8
[4,] 17.0 17.4 17.9 18.4 20.1
[5,] 16.1 16.5 17.0 17.6 19.4
```

```
> cfp[1:5, 1:5]
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,] 19.9 20.1 20.4 20.8 22.2
[2,] 19.0 19.2 19.5 19.9 21.5
[3,] 18.0 18.3 18.7 19.1 20.8
[4,] 17.0 17.4 17.9 18.4 20.1
[5,] 16.1 16.5 17.0 17.6 19.4
```

```
> all.equal(callprices1, cfp)
```

```
[1] TRUE
```

```
> system.time(
      for (i in 1:100)
          ignore <- loop2() )
```

```
   user  system elapsed
  0.226   0.000   0.226
```

```
> system.time(
      for (i in 1:100)
          ignore <- vect2() )
```

```
   user  system elapsed
  0.020   0.000   0.021
```

```
> system.time(
      for (i in 1:100)
          ignore <- priceMatrix(cf = cfBSM, S, Xvec, tauvec,
                                r, q = q,
                     v = 0.2^2, n = 50) )
```

```
   user  system elapsed
  1.364   1.081   0.275
```

But it turns out we can save quite some time by precomputing the nodes and weights for the numerical integration.

```
> tmp <- xwGauss(50)
> tmp <- changeInterval(tmp$nodes, tmp$weights,
                        oldmin = -1, oldmax = 1,
                        newmin =  0, newmax = 200)
> nodes <- tmp$nodes
> weights <- tmp$weights
> system.time(
      for (i in 1:100)
          ignore <- priceMatrix(cf = cfBSM, S, Xvec, tauvec,
                                r, q = q,
                                v = 0.2^2,
                                nodes = nodes, weights = weights))
```

```
   user  system elapsed 
   0.27    0.00    0.27 
```

14. Combining different heuristics

You can also combine several heuristics; see the discussion in Section 12.4 of GMS.

14.1. Single-solution methods

For `LSopt`, `TAopt` and `SAopt`, the simplest way to incorporate another method is through the neighbourhood function. TA could, for instance, every k iterations not draw a neighbour from some specific neighbourhood, but instead call some other method, pass the current solution as the starting value, and then return this method's solution as the new solution.

14.2. Population-based methods

The way to call new methods would be through the `repair` function. We could, for instance, write a repair mechanism (or rather an 'improve' mechanism) that every k iterations picks the best member of the population and performs some type of trajectory method (eg, a direct search). The solution returned by this second method then (possibly) replaces the member in the population.

A. Resources

A.1. Solutions to exercises

A.2. Installing the package

The latest version of the package is available from `http://enricoschumann.net`. The package is also available from CRAN. To install the package from within R, type

```
> install.packages("NMOF") ## CRAN
> install.packages("NMOF",
                   repos = c('http://enricoschumann.net/R',
                             getOption('repos')))
```

to download and install it.

The source of the package is also pushed to GitHub and GitLab:

`https://gitlab.com/NMOF/NMOF`

`https://github.com/enricoschumann/NMOF`

You can directly access all the R scripts that are displayed in the book with the function `showExample`. For instance:

```
> require("NMOF")
> showExample("exampleOF.R")
```

A.3. Other resources

You can download all the code examples from GMS from the book's home page,

`http://nmof.net`

Much additional material (working papers, presentations, errata) are available from http://enricoschumann.net/NMOF

New versions of the package and other news are announced through the NMOF-news mailing list. To browse the archives or to subscribe, go to

https://lists.r-forge.r-project.org/cgi-bin/mailman/listinfo/nmof-news

Applications, as long as they are finance-related, should be discussed on the R-SIG-Finance mailing list: https://stat.ethz.ch/mailman/listinfo/r-sig-finance

Please send bug reports or suggestions directly to the package maintainer, for instance by using bug.report.

```
> require("utils")
> bug.report("[NMOF] Unexpected behaviour in function XXX",
             maintainer("NMOF"), package = "NMOF")
```

A.4. Package Version and Session Information

```
> toLatex(sessionInfo())
```

- R version 4.2.1 (2022-06-23), `x86_64-pc-linux-gnu`
- Locale: `LC_CTYPE=en_US.UTF-8`, `LC_NUMERIC=C`, `LC_TIME=en_GB.UTF-8`, `LC_COLLATE=en_US.UTF-8`, `LC_MONETARY=en_GB.UTF-8`, `LC_MESSAGES=en_US.UTF-8`, `LC_PAPER=en_GB.UTF-8`, `LC_NAME=C`, `LC_ADDRESS=C`, `LC_TELEPHONE=C`, `LC_MEASUREMENT=en_GB.UTF-8`, `LC_IDENTIFICATION=C`
- Running under: `Ubuntu 22.04.1 LTS`
- Matrix products: default
- BLAS: `/usr/lib/x86_64-linux-gnu/openblas-pthread/libblas.so.3`
- LAPACK: `/usr/lib/x86_64-linux-gnu/openblas-pthread/libopenblasp-r0.3.20.so`
- Base packages: base, compiler, datasets, graphics, grDevices, methods, stats, tools, utils
- Other packages: cccp 0.2-7, codetools 0.2-18, digest 0.6.29, FRAPO 0.4-1, NMOF 2.7-0, plotseries 0.2-0, rbenchmark 1.0.0, Rglpk 0.6-4, robustbase 0.93-9, slam 0.1-49, timeDate 3043.102, timeSeries 3062.100, weaver 1.60.0, zoo 1.8-9
- Loaded via a namespace (and not attached): cellranger 1.1.0, crayon 1.4.2, datetimeutils 0.6-0, DEoptimR 1.0-9, ellipsis 0.3.2, fansi 0.5.0, fastmatch 1.1-3, grid 4.2.1, lattice 0.20-45, lifecycle 1.0.1, magrittr 2.0.1, orgutils 0.5-0, parallel 4.2.1, pillar 1.6.4, pkgconfig 2.0.3, PMwR 0.18-0, quadprog 1.5-8, Rcpp 1.0.7, readxl 1.3.1, rlang 0.4.12, snow 0.4-4, textutils 0.2-2, tibble 3.1.6, utf8 1.2.2, vctrs 0.3.8

Index

Bibliography

Ingo Althöfer and Klaus-Uwe Koschnick. On the convergence of "Threshold Accepting". *Applied Mathematics and Optimization*, 24(1):183–195, 1991.

Richard S. Barr, Bruce L. Golden, James P. Kelly, Mauricio G. C. Resende, and William R. Stewart. Designing and reporting on computational experiments with heuristic methods. *Journal of Heuristics*, 1(1):9–32, 1995.

Frans van den Bergh and Andries P. Engelbrecht. A study of Particle Swarm Optimization particle trajectories. *Information Sciences*, 176(8):937–971, 2006.

Patrick Burns. Random portfolios for performance measurement. In Erricos John Kontoghiorghes and Cristian Gatu, editors, *Optimisation, Econometric and Financial Analysis*. Springer, 2010.

Seth Falcon. *weaver: Tools and extensions for processing Sweave documents*, 2015. R package version 1.34.0.

Saul B. Gelfand and Sanjoy K. Mitter. Analysis of Simulated Annealing for Optimization. Technical Report LIDS-P-1494, MIT, 1985.

Philip E. Gill, Walter Murray, and Margaret H. Wright. *Practical Optimization*. Elsevier, 1986.

Manfred Gilli and Enrico Schumann. Optimization in financial engineering – an essay on 'good' solutions and misplaced exactitude. *Journal of Financial Transformation*, 28:117–122, 2010.

Manfred Gilli and Enrico Schumann. Optimal enough? *Journal of Heuristics*, 17(4):373–387, 2011.

Manfred Gilli and Enrico Schumann. Heuristics for portfolio selection. In Giorgio Consigli, Daniel Kuhn, and Paolo Brandimarte, editors, *Optimal Financial Decision Making under Uncertainty*, pages 225–253. Springer, 2017. ISBN 978-3-319-41613-7. doi: 10.1007/978-3-319-41613-7_10. URL `http://dx.doi.org/10.1007/978-3-319-41613-7_10`.

Manfred Gilli, Dietmar Maringer, and Enrico Schumann. *Numerical Methods and Optimization in Finance*. Elsevier/Academic Press, 2011. URL `http://nmof.net`.

Manfred Gilli, Dietmar Maringer, and Enrico Schumann. *Numerical Methods and Optimization in Finance*. Elsevier/Academic Press, 2 edition, 2019. doi: 10.1016/C2017-0-01621-X. URL `http://nmof.net`.

Gene H. Golub and John H. Welsch. Calculation of Gauss quadrature rules. *Mathematics of Computation*, 23(106):221–230+s1–s10, 1969.

Walter J. Gutjahr. A graph-based Ant System and its convergence. *Future Generation Computer Systems*, 16(9):873–888, 2000.

Jack Johnston and John DiNardo. *Econometric Methods*. McGraw-Hill, 4 edition, 1997.

Scott Kirkpatrick, C. Daniel Gelatt, and Mario P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, 1983.

Wacek Kusnierczyk. *rbenchmark: Benchmarking routine for R*, 2010. URL `http://CRAN.R-project.org/package=rbenchmark`. R package version 0.3.

Friedrich Leisch. Sweave: Dynamic generation of statistical reports using literate data analysis. In Wolfgang Härdle and Bernd Rönz, editors, *Compstat 2002 – Proceedings in Computational Statistics*, pages 575–580. Physica Verlag, 2002.

Dietmar Maringer and Olufemi Oyewumi. Index tracking with constrained portfolios. *Intelligent Systems in Accounting, Finance & Management*, 15 (1):57–71, 2007.

Michael J.D. Powell. Problems related to unconstrained optimization. In W. Murray, editor, *Numerical Methods for Unconstrained Optimization*. Academic Press, 1972.

R. Tyrrell Rockafellar and Stanislav Uryasev. Optimization of Conditional Value-at-Risk. *Journal of Risk*, 2(3):21–41, Spring 2000.

Peter J. Rousseeuw and Katrien Van Driessen. Computing LTS regression for large data sets. *Data Mining and Knowledge Discovery*, 12(1):29–45, 2005.

Günter Rudolph. Convergence analysis of canonical genetic algorithms. *IEEE Transactions on Neural Networks*, 5(1):96–101, 1994.

Enrico Schumann. Take-the-best in portfolio selection. available from `http://enricoschumann.net`, 2013.

William F. Sharpe. Asset allocation: Management style and performance measurement. *Journal of Portfolio Management*, 18(2):7–19, 1992. URL `http://www.stanford.edu/~wfsharpe/art/sa/sa.htm`.

Thomas Stützle and Marco Dorigo. A short convergence proof for a class of Ant Colony Optimization algorithms. *IEEE Transactions On Evolutionary Computation*, 6(4):358–365, 2002.

Luke Tierney, A. J. Rossini, Na Li, and H. Sevcikova. *snow: Simple Network of Workstations*, 2011. URL `http://CRAN.R-project.org/package=snow`. R package version 0.3-7.

Simon Urbanek. *multicore: Parallel processing of R code on machines with multiple cores or CPUs*, 2011. URL `http://CRAN.R-project.org/package=multicore`. R package version 0.1-7.

Peter Winker and Dietmar Maringer. The Threshold Accepting optimisation algorithm in economics and statistics. In Erricos John Kontoghiorghes and Cristian Gatu, editors, *Optimisation, Econometric and Financial Analysis*, volume 9 of *Advances in Computational Management Science*, pages 107–125. Springer, 2007.

Stelios H. Zanakis and James R. Evans. Heuristic "optimization": Why, when, and how to use it. *Interfaces*, 11(5):84–91, 1981.