Examples and Extensions for the NMOF package

Enrico Schumann
enricoschumann@yahoo.de

2012-02-10

Contents

This manuscript has now been superseded by the *NMOF Manual*, which is available from `http://enricoschumann.net/NMOF.htm`.

This manuscript – currently a draft – contains a number of examples for the NMOF package that did not make it into the book (Gilli, Maringer, and Schumann, 2011) or the vignettes (the vignettes were shortened to keep the build time of the package acceptable). I am grateful for comments and corrections.

I refer to the book as GMS, and NMOF stands for the package. The latest version of the package is available from `http://r-forge.r-project.org/` (Theußl and Zeileis, 2009). The package is also available from CRAN. To install the package from within R, type

```
> install.packages("NMOF") ### CRAN
> install.packages("NMOF", repos = "http://R-Forge.R-project.org")
```

to download and install it. For all examples, the package needs to be attached.

```
> require("NMOF")
> set.seed(1112233344)
```

This document is written with Sweave (Leisch, 2002). The code is part of the package; it can be found in the subdirectory `NMOFex`. To show the code in R, you can use the function `system.file`.

```
> whereToLook <- system.file("NMOFex/NMOFex.R", package = "NMOF")
> file.show(whereToLook, title = "NMOF examples")
```

The latest pdf version of this report can be obtained from
`http://enricoschumann.net/NMOF.htm`

1 Optimisation

1.1 New functions

This section describes functions that are not covered in the book.

1.1.1 Genetic Algorithms – `GAopt`

The function `GAopt` was added in version 0.13-0. It implements a typical Genetic Algorithm (GA). Its structure is very similar to that of `DEopt` and `PSopt`. The solutions are coded as binary strings (vectors of mode `logical`); the complete population is a matrix in which every column represents on solution. See `?GAopt` after attaching the package; an example is given in the next section.

1.1.2 gridSearch

The function `gridSearch` was added in version 0.14-0. `gridSearch` allows to distribute the evaluation of the objective function (through packages `multicore` or `snow`).

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

$$\mathrm{f}(x_1, x_2) = x_1 + x_2^2 . \tag{1}$$

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) ## just wasting time :-)
     x[1L] + x[2L]^2
 }
```

Now we can evaluate f for, say, $1 \leqslant x_1 \leqslant 5$ and $3 \leqslant x_2 \leqslant 5$, 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))
```

```
   user  system elapsed
   0.02    0.00    2.74
```

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,
                                method = "snow",  ### use 'snow' ...
                                cl = 2L))         ### ... with 2 cores
```

```
   user  system elapsed
   0.04    0.00    2.24
```

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

```
[1] TRUE
```

1.2 Examples

1.2.1 Asset selection with GA and TA

We first extend an example given in the book: select a 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; here, we will also use Threshold Accepting (TA) and GA. 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.

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).

```
> ## number of assets
> na <- 500L
> ## correlation matrix
> C <- array(0.6, dim = c(na,na)); diag(C) <- 1
> ## covariance matrix
> minVol <- 0.20; maxVol <- 0.40
> 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)
 }
```

...and 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))
```

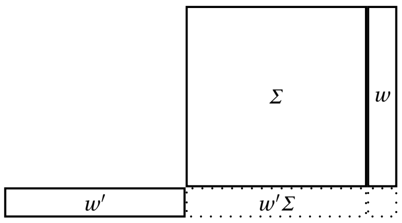
```
   user  system elapsed
   0.31    0.00    0.31
```

```
> ## 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))
```

```
   user  system elapsed
   0.47    0.00    0.47
```

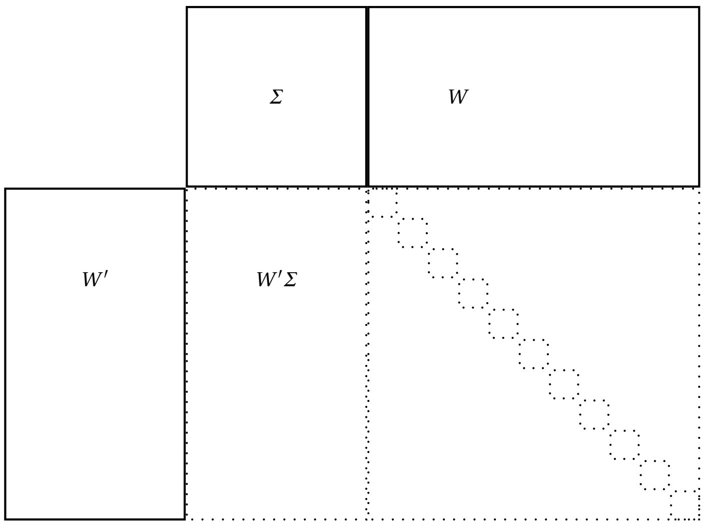
Now we use a GA, for which we need to write a new objective function. It is helpful in this case (and in many others) to cast the computation into matrix algebra notation. This makes the maths more concise and allows to use linear algebra routines. In fact, our objective function will evaluate the whole 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 $\mathsf{diag}(W'\Sigma W)$.



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 $\mathsf{diag}(W'\Sigma W)$ 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; see the following objective function. The function also handles the cardinality constraint through a simple penalty.

```
> ## objective function for Genetic Algorithm
> 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.

```
> ## Genetic Algorithm
> 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 0.0261 ;
standard deviation of OF in final population is 0 .

   user  system elapsed
   11.3     0.0    11.3
```

We should now compare the results of the three algorithms.

```
> cat("Local Search        ", format(sqrt(solLS$OFvalue), digits = 4), "\n",
     "Threshold Accepting ", format(sqrt(solTA$OFvalue), digits = 4), "\n",
     "Genetic Algorithm   ", format(sqrt(solGA$OFvalue), digits = 4), "\n",
     sep = "")
```

```
Local Search        0.1619
Threshold Accepting 0.1615
Genetic Algorithm   0.1615
```

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).

1.2.2 Minimising semivariance with DE, PS and TA

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:

$$\tfrac{1}{n_S} \sum_{r_i<\theta} (\theta - r_i)^2 \,. \tag{2}$$

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

$$\frac{1}{n_S}\underbrace{\frac{k}{k}}_{1}\sum_{r_i<\theta}(\theta - r_i)^2 = \underbrace{\frac{k}{n_S}}_{\mathrm{Prob}(r_i<\theta)}\ \underbrace{\frac{1}{k}\sum_{r_i<\theta}(\theta - r_i)^2}_{\text{conditional average}}\ . \tag{3}$$

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] -0.0365 -0.0306  0.0403  0.0158  0.0364
```

```
> sum(x0)
```

```
[1] 0.449
```

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] 6.89e-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] 6.89e-05
```

```
> OF(cbind(x0, x0, x0), data)
```

```
      x0       x0       x0
6.89e-05 6.89e-05 6.89e-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] 0.449
```

```
> 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.
-0.0609 -0.0201  0.0117  0.0055  0.0328  0.0540
```

```
> summary(repair2(x0, data)-x0)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
0.00551 0.00551 0.00551 0.00551 0.00551 0.00551
```

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 6.76e-05 ;
standard deviation of OF in final population is 2.13e-08 .

   user  system elapsed
   8.27    0.00    8.27
```

```
> 16 * 100 * sqrt(sol$OFvalue)   ## solution quality
```

```
[1] 13.2
```

```
> ## check constraints
> all(all.equal(sum(sol$xbest), 1),  ## budget constraint
      sol$xbest <= data$max,         ## holding size constraints
      sol$xbest >= data$min)
```

```
[1] TRUE
```

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 semivariance 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 6.59e-05 ;
standard deviation of OF in final population is 1.06e-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 7.09e-05 ;
standard deviation of OF in final population is 3.74e-08 .
```

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

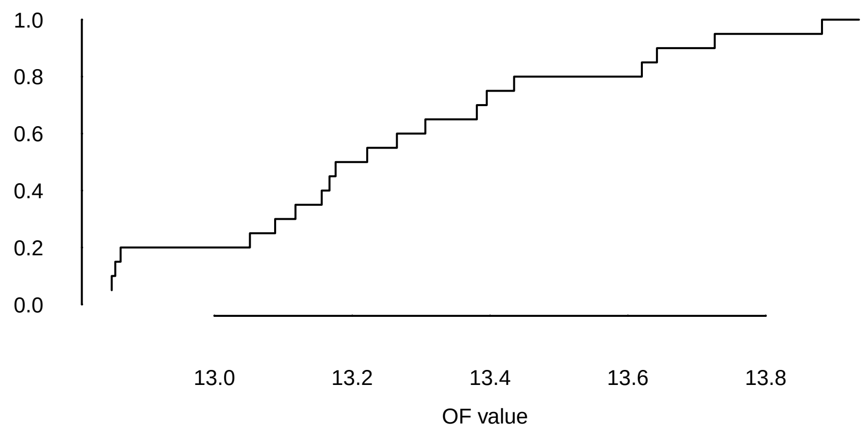
```
[1] 2.13
```

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,
                           method = "snow",  ### using package snow
                           cl = 2)           ### 2 cores
> ## extract best solution
> OFvaluesDE <- sapply(restartsDE, `[[`, "OFvalue")
> OFvaluesDE <- 16 * 100 * sqrt(OFvaluesDE)
> weightsDE  <- sapply(restartsDE, `[[`, "xbest")
```

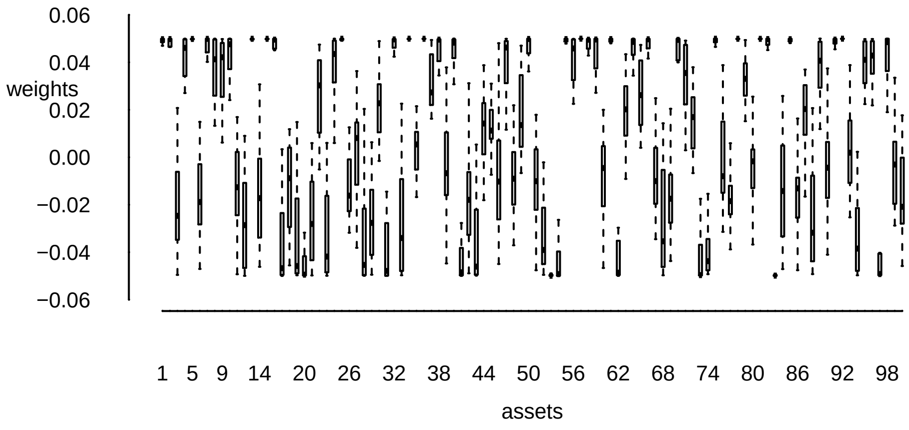
We check the objective function values associated with the restarts.

```
> 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, 1000L, 2500L)
> lstOFvaluesDE <- list()
> for (i in 1:3) {
     algo$nG <- nGs[i]
     restartsDE <- restartOpt(fun = DEopt,
                              n = restarts,
                              OF = OF,  algo = algo, data = data,
                              method = "snow", cl = 2)
     ## extract best solution
     OFvaluesDE <- sapply(restartsDE, `[[`, "OFvalue")
     OFvaluesDE <- 16 * 100 * sqrt(OFvaluesDE)
     lstOFvaluesDE[[i]] <- OFvaluesDE
 }
> res <- simplify2array(lstOFvaluesDE)
```

And now with repair2.

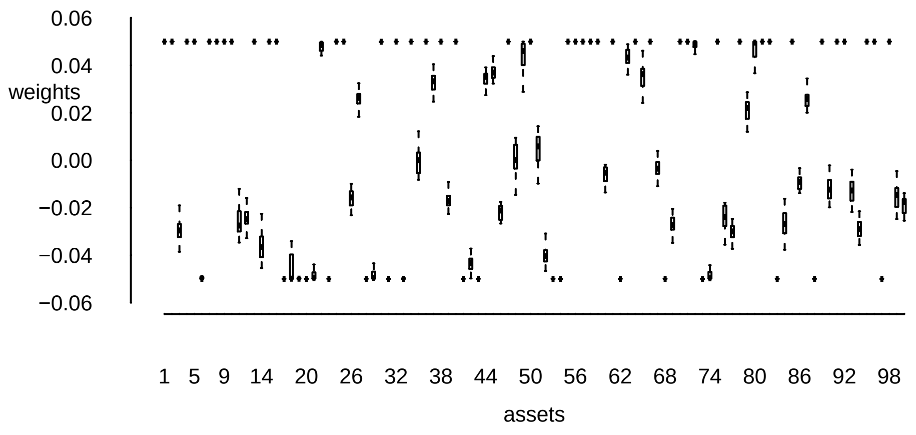
```
> algo$repair <- repair2
> lstOFvaluesDE <- list()
> for (i in 1:3) {
     algo$nG <- nGs[i]
     restartsDE <- restartOpt(fun = DEopt,
                              n = restarts,
                              OF = OF,  algo = algo, data = data,
                              method = "snow", cl = 2)
     ## 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)
```



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 9.7e-05 ;
standard deviation of OF in final population is 1.89e-06 .

   user  system elapsed
   8.66    0.02    8.68
```

```
> 16 * 100 * sqrt(sol$OFvalue)      ## solution quality
```

```
[1] 15.8
```

```
> ## check constraints
> all(all.equal(sum(sol$xbest),1),  ## budget constraint
      sol$xbest <= data$max,
      sol$xbest >= data$min)
```

```
[1] TRUE
```

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] -3.47e-17
```

```
> colSums(changeV(cbind(x0, x0, x0), data))
```

```
       x0        x0        x0
-3.47e-17 -3.47e-17 -3.47e-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 7.45e-05 ;
standard deviation of OF in final population is 0.0279 .


   user  system elapsed
   8.61    0.01    8.64
```

```
> 16 * 100 * sqrt(sol$OFvalue)   ## solution quality
```

```
[1] 13.8
```

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)
```

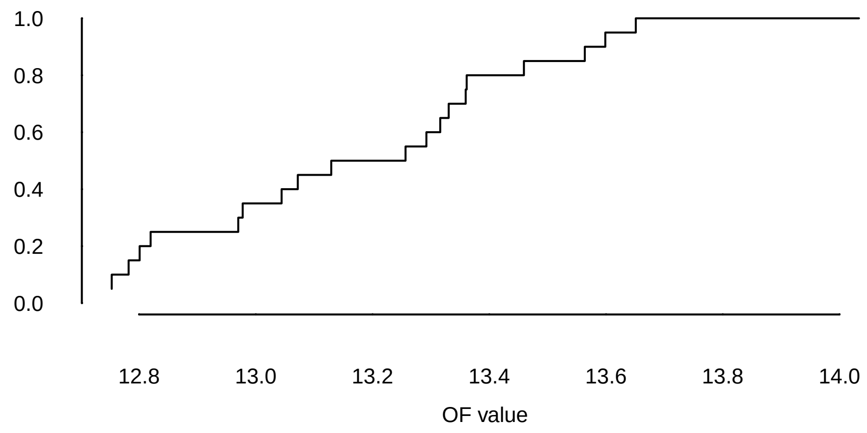
```
[1] TRUE
```

```
> 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 6.84e-05 ;
standard deviation of OF in final population is 0.0163 .

   user  system elapsed
   3.93    0.00    3.93
```

Finally, we can also run a small experiment here.

```
> algo$printDetail <- FALSE
> restartsPS <- restartOpt(fun = PSopt,
                           n = 20L,
                           OF = OF,
                           algo = algo, data = data,
                           method = "snow", 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)
```



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.

```
> ## a random initial weights
> 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))
```

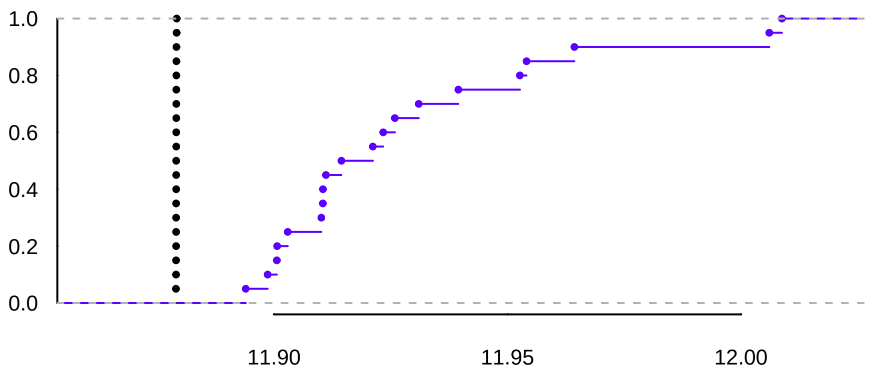
```
   user  system elapsed
   1.78    0.00    1.78
```

```
> 16 * 100 * sqrt(sol2$OFvalue)
```

```
[1] 11.9
```

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,
                          method = "snow", 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)
```



Implementing objective functions ...to be added.

1.2.3 Fitting yield curves with Differential Evolution

The material in this section was taken from to vignette ‘Fitting the Nelson–Siegel–Svensson model with Differential Evolution’ because the examples will run several minutes.

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. 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]
```

```
    [,1] [,2]  [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
0.5 5.75 0.00   3.5    0 3.25 5.75  0.0    5    0     3
1   0.00 4.25   0.0    3 0.00 0.00  3.5    0  105     0
1.5 5.75 0.00 103.5    0 3.25 5.75  0.0    5    0     3
2   0.00 4.25   0.0    3 0.00 0.00  3.5    0    0     0
2.5 5.75 0.00   0.0    0 3.25 5.75  0.0    5    0     3
3   0.00 4.25   0.0    3 0.00 0.00  3.5    0    0     0
3.5 5.75 0.00   0.0    0 3.25 5.75  0.0    5    0     3
4   0.00 4.25   0.0    3 0.00 0.00  3.5    0    0     0
4.5 5.75 0.00   0.0    0 3.25 5.75  0.0    5    0     3
5   0.00 4.25   0.0    3 0.00 0.00  3.5    0    0     0
```

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] 0.0274
```

```
> sol$OFvalue
```

```
[1] 0.0029
```

…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)))
```

```
   user  system elapsed
   0.08    0.00    0.08
```

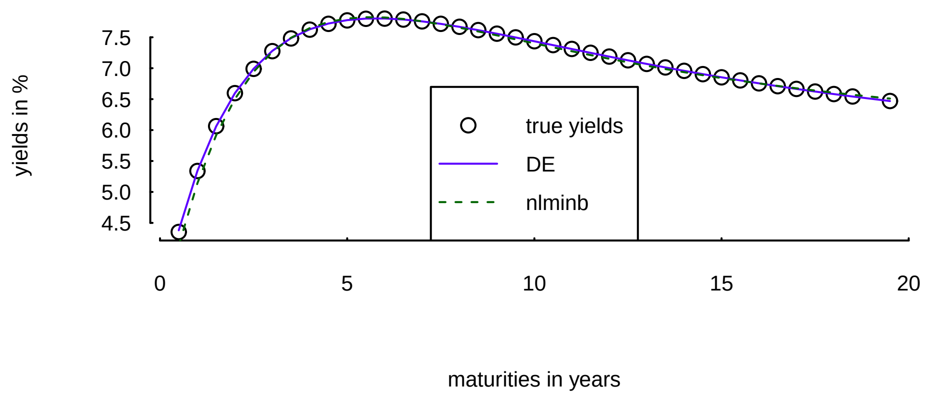
```
> max(abs(data$model(sol2$par,tm) - data$model(betaTRUE,tm)))
```

```
[1] 0.218
```

```
> sol2$objective
```

```
[1] 0.201
```

```
> 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))
```



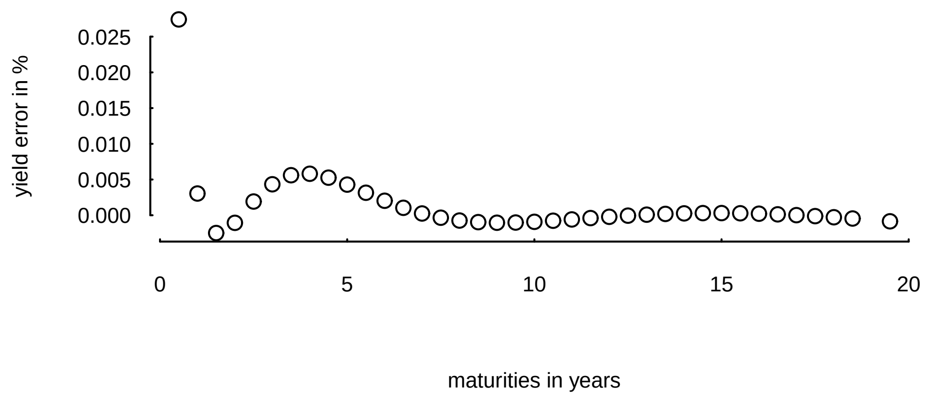
We can check the price errors.

```
> diFa <- 1 / ((1 + NSS(sol$xbest,tm)/100)^tm)
> b <- diFa %*% cfMatrix
> b - bM
```

```
        [,1]      [,2]    [,3]     [,4]     [,5]   [,6]     [,7]
[1,] 0.00107 -4.49e-05 0.00288 -0.00227 0.000525 -0.002 -0.00286
        [,8]     [,9]     [,10]  [,11]   [,12]    [,13]    [,14]
[1,] 0.00146 -0.00288 -0.000996 0.0019 0.00139 0.000802 4.57e-05
      [,15]   [,16]    [,17]   [,18]   [,19]    [,20]
[1,] 0.0029 -0.0029 -0.00117 0.00281 0.00202 -0.00268
```

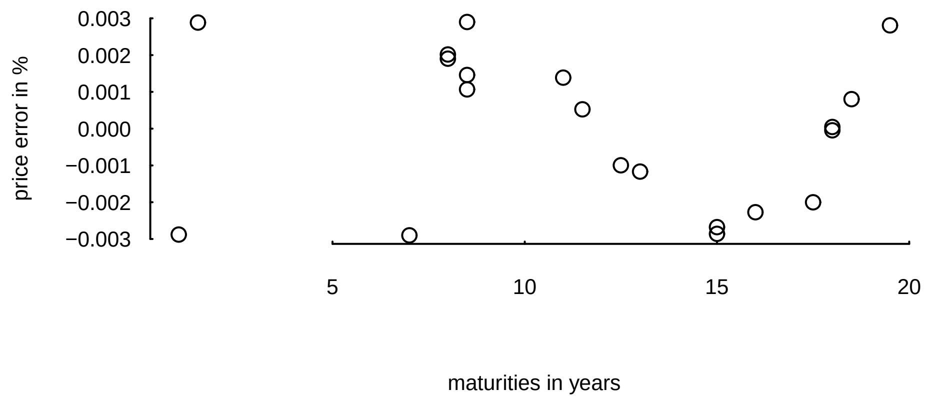
We can also plot the rate errors against time-to-payment.

```
> 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 %")
```



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)))
```

```
[1] 0.0364
```

```
> sol$OFvalue
```

```
[1] 1.89e-05
```

With `nlminb`:

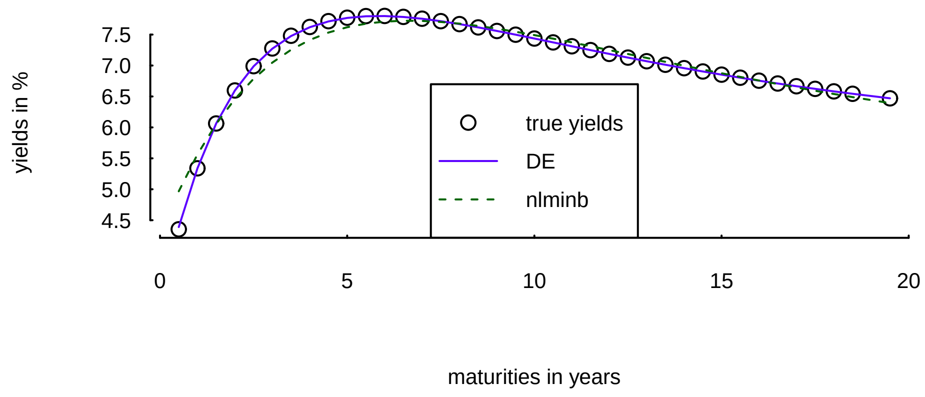
```
> s0 <- algo$min + (algo$max - algo$min) * runif(length(algo$min))
> sol2 <- nlminb(s0, OF3, data = data,
                                   lower = algo$min,
                                   upper = algo$max,
                                 control = list(eval.max = 50000L,
                                                iter.max = 50000L))
> max(abs(data$model(sol2$par,tm) - data$model(betaTRUE,tm)))
```

```
[1] 0.614
```

```
> sol2$objective
```

```
[1] 0.0067
```

```
> 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]  5.005 -1.680 -1.502 10.107  0.388  2.981
```

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`).

1.2.4 Model selection with Threshold Accepting

In this section we do a simple model selection for a linear regression:[1] 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. `k` of these are the 'true' regressors, and they define a response variable `y` like so:

$$y = \beta X_{\cdot,K} + s\epsilon \qquad (4)$$

[1] I would like to thank Victor Bystrov for comments on an earlier (MATLAB) version of this example.

The variable K is the set of true regressors (k == length(K)); s is the scale of 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 <- rnorm(n * p); dim(X) <- c(n, p)  # regressor matrix X
     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)
 }
> rD <- randomData(p = 100L, n = 200L, s = 1,
                   constant = TRUE, maxReg = 15L)
> data <- list(X = rD$X,
             y = rD$y,
             n = rD$n,
             p = rD$p,
             maxk  = 30L,  ### maximum number of regressors included in model
             lognn = log(rD$n)/rD$n)
```

We put all the data in a list called data. Next, we compute a random solution x0. Such a solution is a logical vector of length p. This vector will then be used to subset the columns of X.

```
> x0 <- logical(data$p)
> temp <- sample.int(data$maxk, 1L)
> temp <- sample.int(data$p, temp)
> x0[temp] <- TRUE
```

Any selection rule for a model will use the residuals of the fitted model as an ingredient. Thus, given such 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.

```
> 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)
```

```
                             test elapsed relative
1 lm(data$y ~ -1 + data$X[, x0])    1.67     13.9
2 qr.solve(data$X[, x0], data$y)    0.12      1.0
```

```
> ## ... should give the same coefficients
> ignore1 <- lm(data$y ~ -1 + data$X[ ,x0])
> ignore2 <- qr.solve(data$X[ ,x0], data$y)
> all.equal(as.numeric(coef(ignore1)), as.numeric(ignore2))
```

```
[1] TRUE
```

Now, for the actual selection criterion. We will use the Schwarz criterion, which is (for a linear model) given by

$$\log\left(\frac{\text{residuals}'\text{residuals}}{n}\right)+\frac{\log(n)\times\text{number of regressors}}{n};\tag{5}$$

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
 }
> OF(x0, data)
```

```
     [,1]
[1,] 1.87
```

The last ingredient 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
 }
> neighbour(x0, data)[1:10]
```

```
 [1] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE
```

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 thresholds
     neighbour = neighbour,
     x0 = x0,
     printBar = FALSE)
> system.time(sol1 <- TAopt(OF, algo = algo, data = data))
```

```
Threshold Accepting.

Computing thresholds ... OK.
Estimated remaining running time: 0.86 secs.


Running Threshold Accepting...
Initial solution:  1.87
Finished.
Best solution overall: 0.248
   user  system elapsed
   0.89    0.00    0.89
```

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,] 0.248
```

```
> which(sol1$xbest)  ### the selected regressors
```

```
 [1] 34 40 57 64 68 75 77 79 82 89
```

```
> rD$K               ### the true regressors
```

```
[1] 34 40 57 64 68 79 89
```

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,] 0.248
```

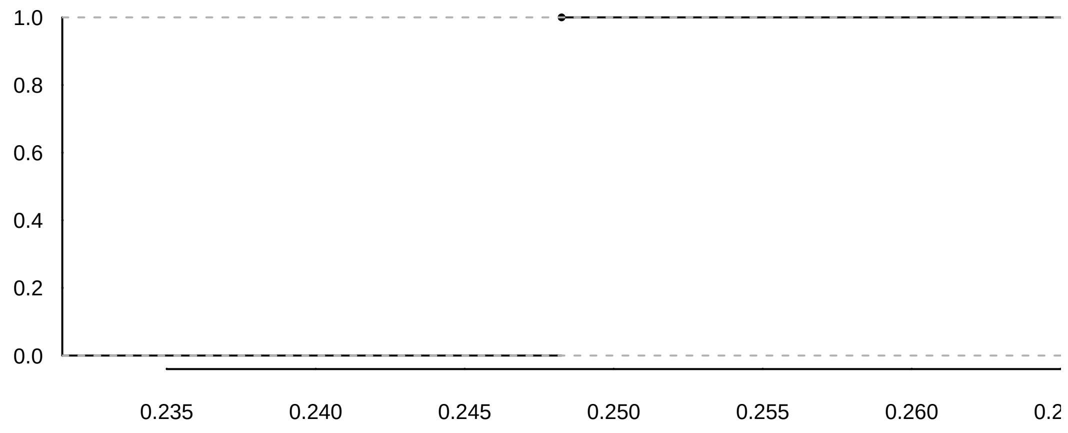
```
> OF(xtrue, data)
```

```
      [,1]
[1,] 0.285
```

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,
                    method = "snow", 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$K, inclReg))
> data.frame(
     regressor = inclReg,
     times_included = paste(rowSums(xbestAll)[inclReg], "/",
                            restarts, sep = ""),
     true_regressor = inclReg %in% rD$K)
```

```
   regressor times_included true_regressor
1         34          50/50           TRUE
2         40          50/50           TRUE
3         57          50/50           TRUE
4         64          50/50           TRUE
5         68          50/50           TRUE
6         75          50/50          FALSE
7         77          50/50          FALSE
8         79          50/50           TRUE
9         82          50/50          FALSE
10        89          50/50           TRUE
```

Across the restarts, we get a relatively clear answer which regressors should, according to the Schwarz criterion, be put into the model.

1.2.5 Neighbourhood functions for Threshold Accepting

The neighbourhood is the most important aspect of TA. Neighbourhoods have the tendency to become complicated; in particular, if we incorporate more knowledge or 'ideas' about the problem to be solved. Nevertheless, they are almost always built around simple building blocks (at least for data structures like vectors or matrices).

A typical problem is to change a vector of logicals.

```
> 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, ...) {
     argsL <- list(...)
     if (!("sep" %in% names(argsL))) argsL$sep <- ""
     do.call("cat",
             c(list("\n",as.integer(x), "\n", as.integer(y), "\n",
                    ifelse(x == y, " ", "^"), "\n"), argsL)
             )
         }
```

`compareLogicals` will print the vectors like `001110` and indicate differences by a `^`. Example:

```
> ## there should be no difference
> compareLogicals(x, x)
```

```
00010111100010010010
00010111100010010010
```

```
> ## change the second element
> z <- x; z[2L] <- !z[2L]
> compareLogicals(x, z)
```

```
00010111100010010010
01010111100010010010
 ^
```

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))
```

```
00010111100010010010
00010111100110101000
           ^  ^^^ ^
```

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))
```

```
00010111100010010010
01010111100010010000
 ^                 ^
```

1.2.6 Distributed evaluation of the objective function in `GAopt`

We use the example from the man-page of `GAopt`. Distributing the evaluation of the population will incur some overhead, so it should not be used for very cheap objective functions; see the next example.

```
> ## match a binary (logical) string y
> size <- 20L             ## the length of the string
> OF <- function(x, y)    ## the objective function
     sum(x != y)
> y <- runif(size) > 0.5  ## the true solution
> OF(y, y)                ## the optimum value is zero
```

```
[1] 0
```

```
> algo <- list(nB = size, nP = 50L, nG = 150L, prob = 0.002,
              printBar = FALSE, methodOF = "loop")
> system.time(sol <- GAopt(OF, algo = algo, y = y))
```

```
Genetic Algorithm.
Best solution has objective function value 0 ;
standard deviation of OF in final population is 0 .

   user  system elapsed
   0.17    0.00    0.18
```

```
> OF(sol$xbest, y)
```

```
[1] 0
```

```
> algo <- list(nB = size, nP = 50L, nG = 150L, prob = 0.002,
              printBar = FALSE, methodOF = "snow", cl = 2L)
> system.time(sol <- GAopt(OF, algo = algo, y = y))
```

```
Genetic Algorithm.
Best solution has objective function value 0 ;
standard deviation of OF in final population is 0 .

   user  system elapsed
   1.06    0.22    2.24
```

```
> OF(sol$xbest, y)
```

```
[1] 0
```

Now we make the objective function a bit slower (we also reduce the number of generations).

```
> OF <- function(x, y) {
     Sys.sleep(0.01)
     sum(x != y)
 }
> algo <- list(nB = size, nP = 50L, nG = 10L, prob = 0.002,
              printBar = FALSE, methodOF = "loop")
> system.time(sol <- GAopt(OF, algo = algo, y = y))
```

```
Genetic Algorithm.
Best solution has objective function value 1 ;
standard deviation of OF in final population is 0.948 .

   user  system elapsed
   0.00    0.00    8.87
```

```
> OF(sol$xbest, y)
```

```
[1] 1
```

```
> algo <- list(nB = size, nP = 50L, nG = 10L, prob = 0.002,
              printBar = FALSE, methodOF = "snow", cl = 2L)
> system.time(sol <- GAopt(OF, algo = algo, y = y))
```

```
Genetic Algorithm.
Best solution has objective function value 0 ;
standard deviation of OF in final population is 0.953 .

   user  system elapsed
   0.00    0.00    4.93
```

```
> OF(sol$xbest, y)
```

```
[1] 0
```

1.2.7 Combining different heuristics

You can also combine several heuristics.

Trajectory methods For `LSopt` or `TAopt`, 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.

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.

1.2.8 Diagnostics

All optimisation functions in the NMOF package can store the objective function value over the course of the optimisation, and also the actual solutions. Whether these data are stored depends on the parameters `algo$storeF` (defaults to `TRUE`) and `algo$storeSolutions` (defaults to `FALSE`).

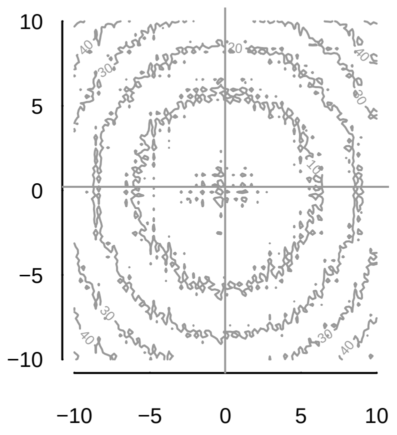
The functions return the objective function values as a matrix `Fmat` and the solutions as a list `xlist`. What exactly is stored depends on the specific function. If the results are not stored (ie, if `algo$storeF` and `algo$storeSolutions` are `FALSE`), then `Fmat` and `xlist` are `NA`.

Example 1 – Trefethen's function An example for `DEopt`. We use `tfTrefethen` as the objective function. To demonstrate the shape of the function, we evaluate it on a grid (see `?testFunctions`).

```
> 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"    "Fmat"    "xlist"
```

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

```
[1] 4.35e-16
```

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

```
[1] 1
```

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

xlist actually holds a list of matrices. (For symmetry: for other function, 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] -5.63 -9.36
```

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

```
[1]  0.0706 -0.5627
```

```
> ## ...
> 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] -5.63 -9.36
```

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

```
[1] -5.6340  0.0574
```

```
> 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] -9.357  6.021 -8.450  2.350 -5.207  1.670  4.826  3.467 -7.188
[10]  1.322 -1.536  5.658 -3.897  4.820  7.786  0.605  4.864  5.278
[19] -3.988  2.578 -4.201 -1.550 -0.757  1.362  1.122  3.077  2.292
[28] -9.314 -3.569 -2.605 -6.260  2.313  2.143  1.837 -9.428  0.377
[37] -4.315 -1.491 -8.472 -5.383 -1.174 -2.967  0.858 -6.659  2.792
[46] -1.304  6.425  5.646  4.164 -1.868
```

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

```
 [1] -0.56270 -1.60391  1.00276  0.31795 -0.99401  0.15201 -1.23598
 [8]  1.29887 -1.27915 -0.48666 -0.56002  0.61681  0.00225 -0.49622
[15]  0.77819 -1.12270 -0.72049 -1.74689  2.57744 -2.77564  0.90250
[22] -1.71915  0.22044 -0.99850  0.05218 -1.62263 -0.66818  0.71606
```

```
[29] -1.61120  0.92455 -2.53148  1.56449 -0.64056 -0.04591 -0.05559
[36]  0.37682  0.85663  2.34645 -0.88949 -0.71937  0.19768  0.98568
[43]  0.29600  1.56503 -1.62296 -1.29252  0.51845 -1.30113  2.34443
[50] -2.52988
```

```
> ## ...
> 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 0.211 0.211
[12] 0.211 0.211 0.211 0.211 0.211 0.211 0.211 0.211 0.211 0.211 0.211
[23] 0.211 0.211 0.211 0.211 0.211 0.211 0.211 0.211 0.211 0.211 0.211
[34] 0.211 0.211 0.211 0.211 0.211 0.211 0.211 0.211 0.211 0.211 0.211
[45] 0.211 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] -9.357  6.021 -8.450  2.350 -5.207  1.670  4.826  3.467 -7.188
[10]  1.322 -1.536  5.658 -3.897  4.820  7.786  0.605  4.864  5.278
[19] -3.988  2.578 -4.201 -1.550 -0.757  1.362  1.122  3.077  2.292
[28] -9.314 -3.569 -2.605 -6.260  2.313  2.143  1.837 -9.428  0.377
[37] -4.315 -1.491 -8.472 -5.383 -1.174 -2.967  0.858 -6.659  2.792
[46] -1.304  6.425  5.646  4.164 -1.868
```

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

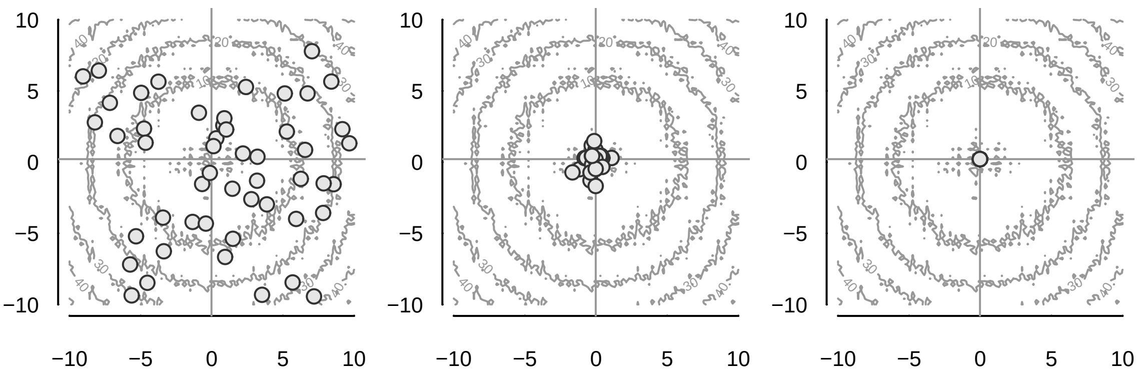
```
 [1]  0.0574  6.0207 -8.4496  2.3505 -5.2066  1.6697  4.8257  3.4671
 [9]  3.5688 -2.1743 -1.5358 -0.8434 -3.8966  3.5934  7.7861  0.6045
[17]  4.8636  5.2775 -3.9881 -2.7756  0.7547 -1.5500 -0.7569  1.3622
[25]  1.1223  3.0772  2.2917  0.8799 -3.6152 -2.3755 -6.6071  2.3127
[33] -0.6406  1.8372 -9.4284  0.3768  0.8566  5.2315 -8.4715  0.1514
[41] -1.1743 -2.9666  0.8579 -6.6588  2.7919 -1.3043  6.4251  5.6459
[49]  4.1640 -1.8682
```

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

```
 [1] 0.211 0.211 0.211 0.211 0.211 0.211 0.211 0.211 0.211 0.211 0.211
[12] 0.211 0.211 0.211 0.211 0.211 0.211 0.211 0.211 0.211 0.211 0.211
[23] 0.211 0.211 0.211 0.211 0.211 0.211 0.211 0.211 0.211 0.211 0.211
[34] 0.211 0.211 0.211 0.211 0.211 0.211 0.211 0.211 0.211 0.211 0.211
[45] 0.211 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? (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.

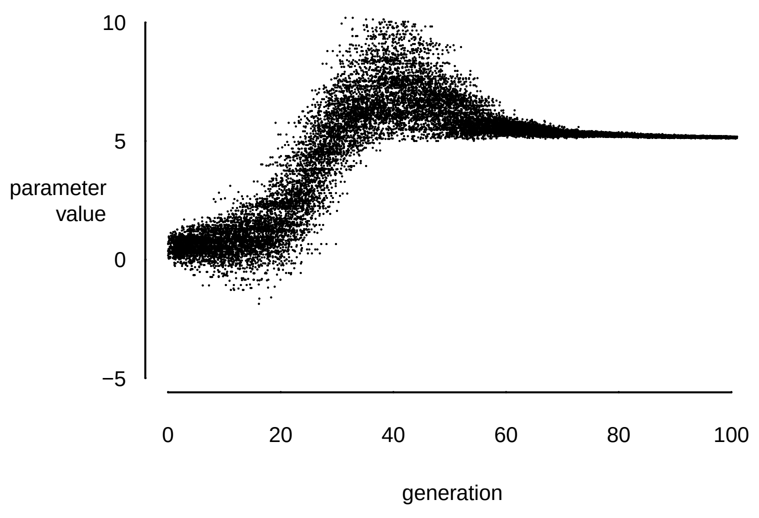
```
> algo <- list(nP = 200L, nG = 100L,
               F = 0.50, CR = 0.99,
             min = c( 0,-10,-10,  0),
             max = c( 1, 10, 10, 10),
             storeSolutions = TRUE, printBar = FALSE)
> ## set up yield curve and put information in Data
> tm <- 1:20                ### times to maturity
> 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 0.0238 ;
standard deviation of OF in final population is 0.00131 .
```

```
> 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)
```

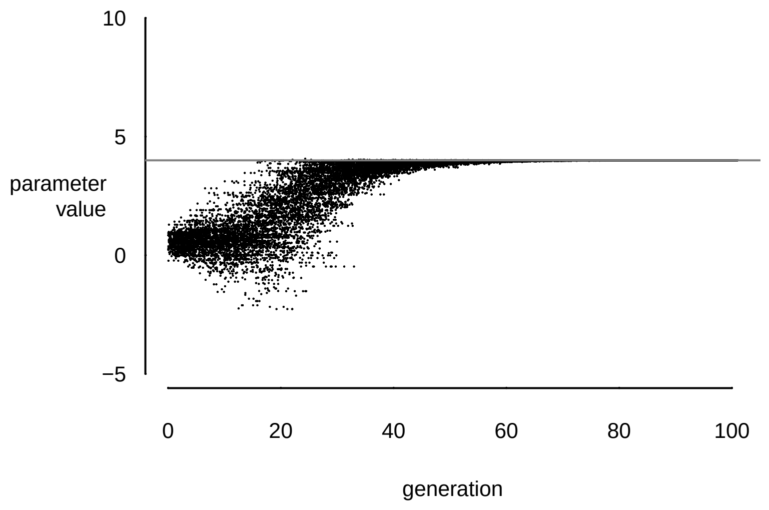


Now suppose we had included a constraint: the parameter should not be greater than 4. (Even though the true parameter is 5.) We adjust the objective function by adding a straightforward penalty. This could certainly be refined, but it is only an example here.

```
> 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 5e-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 4.

2 Numerical Methods

2.1 New functions

2.1.1 bracketing

The function `bracketing` was added in version 0.16-0. The function supports distributed evaluation of `fun` through `multicore` (Urbanek, 2011) or `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))
```

```
   user  system elapsed
    0.0     0.0    10.9
```

```
> system.time(sol2 <- bracketing(testFun, interval = c(0.3, 0.9),
                                 n = 100L, method = "snow", cl = 2))
```

```
   user  system elapsed
   0.00    0.00    6.09
```

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

```
[1] TRUE
```

2.1.2 Integration of Gauss-type

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

2.1.3 Option pricing with the characteristic function

The package always contained the function `callHestoncf`. 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.

2.2 Examples

2.2.1 Option pricing with the characteristic function

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((0+1i) * om * log(S) + (0+1i) * tau * (r - q) * om -
        0.5 * tau * v * ((0+1i) * om + om^2))
}
<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)
```

```
$callPrice
[1] 5.06
```

```
$impliedVol
[1] 0.2
```

A Resources

You can download all the code examples from GMS from the book's home page,

`http://nmof.net`

The latest version of the NMOF package is hosted on R-Forge; please visit

`http://nmof.r-forge.r-project.org/`
for more information.

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`

B Package version

```
> toLatex(sessionInfo())
```

- R version 2.14.1 (2011-12-22), `x86_64-pc-mingw32`
- Locale: `LC_COLLATE=German_Germany.1252`, `LC_CTYPE=German_Germany.1252`, `LC_MONETARY=German_Germany.1252`, `LC_NUMERIC=C`, `LC_TIME=German_Germany.1252`
- Base packages: base, datasets, graphics, grDevices, methods, stats, utils
- Other packages: NMOF 0.23-0, rbenchmark 0.3, snow 0.3-8
- Loaded via a namespace (and not attached): tools 2.14.1

References

Manfred Gilli and Enrico Schumann. Optimal enough? *Journal of Heuristics*, 17(4):373–387, 2011. available from `http://dx.doi.org/10.1007/s10732-010-9138-y`.

Manfred Gilli, Dietmar Maringer, and Enrico Schumann. *Numerical Methods and Optimization in Finance*. Academic Press, 2011.

Jack Johnston and John DiNardo. *Econometric Methods*. McGraw-Hill, 4th edition, 1997.

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. URL `http://www.stat.uni-muenchen.de/~leisch/Sweave`.

Stefan Theußl and Achim Zeileis. Collaborative software development using R-Forge. *R Journal*, 1(1):9–14, 2009. URL `http://journal.r-project.org/2009-1/RJournal_2009-1_Theussl+Zeileis.pdf`.

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.