

CONTENTS

DEDICATION	III
PREFACE	V
<i>About this Book</i>	v
<i>Computations</i>	v
<i>Audience Background</i>	vi
<i>Getting Help</i>	vi
<i>Getting Started</i>	vii
<i>Getting Support</i>	viii
<i>Acknowledgements</i>	viii
CONTENTS	XI
LIST OF FIGURES	XVII
LIST OF TABLES	XIX
I Computations	1
1 DATA STRUCTURES	3
1.1 <i>Vectors</i>	3
1.2 <i>Matrices</i>	6
1.3 <i>Arrays</i>	9
1.4 <i>Data Frames</i>	10
1.5 <i>Time Series</i>	12
1.6 <i>Lists</i>	14
1.7 <i>Printing the Object Structure</i>	18
2 DATA MANIPULATION	19
2.1 <i>Manipulating Vectors</i>	19
2.2 <i>Manipulating Matrices</i>	24
2.3 <i>Manipulating Data Frames</i>	28
2.4 <i>Working with Attributes</i>	34

2.5	<i>Manipulating Character Strings</i>	36
2.6	<i>Creating Factors from Continuous Data</i>	40
3	IMPORTING AND EXPORTING DATA	43
3.1	<i>Writing to Text Files</i>	43
3.2	<i>Reading from a Text File with scan()</i>	44
3.3	<i>Reading from a Text File with readLines()</i>	45
3.4	<i>Reading from a Text File with read.table()</i>	46
3.5	<i>Importing Example Data Files</i>	49
3.6	<i>Importing Historical Data Sets from the Internet</i>	50
4	OBJECT TYPES	51
4.1	<i>Characterization of Objects</i>	51
4.2	<i>Double</i>	52
4.3	<i>Integers</i>	57
4.4	<i>Complex</i>	60
4.5	<i>Logical</i>	61
4.6	<i>Missing Data</i>	62
4.7	<i>Character</i>	63
4.8	<i>NULL</i>	63
II	Programming	65
5	WRITING FUNCTIONS	67
5.1	<i>Writing your first function</i>	67
5.2	<i>Arguments and Variables</i>	69
5.3	<i>Scoping rules</i>	72
5.4	<i>Lazy evaluation</i>	73
5.5	<i>Flow control</i>	74
6	DEBUGGING YOUR R FUNCTIONS	79
6.1	<i>The traceback() function</i>	79
6.2	<i>The function warning() and stop()</i>	80
6.3	<i>Stepping Through a Function</i>	81
6.4	<i>The function browser()</i>	82
7	EFFICIENT CALCULATIONS	83
7.1	<i>Vectorized Computations</i>	83
7.2	<i>The Family of apply() Functions</i>	86
7.3	<i>The function by()</i>	89
7.4	<i>The Function outer()</i>	91
8	USING S3 CLASSES	93
8.1	<i>S3 Class Model Basics</i>	93

9	R PACKAGES	99
9.1	<i>Base R packages</i>	99
9.2	<i>Contributed R Packages from CRAN</i>	100
9.3	<i>R Packages under Development from R-forge</i>	100
9.4	<i>R Package Usage</i>	100
9.5	<i>Package Management Functions</i>	101
III Plotting		103
10	HIGH LEVEL PLOTS	105
10.1	<i>Scatter Plots</i>	105
10.2	<i>Line Plots</i>	106
10.3	<i>More about the plot() Function</i>	108
10.4	<i>Distribution Plots</i>	109
10.5	<i>Pie and Bar Plots</i>	112
10.6	<i>Stars- and Segments Plots</i>	114
10.7	<i>Bi- and Multivariate Plots</i>	116
11	CUSTOMIZING PLOTS	119
11.1	<i>More About Plot Function Arguments</i>	119
11.2	<i>Graphical Parameters</i>	123
11.3	<i>Margins, Plot and Figure Regions</i>	125
11.4	<i>More About Colours</i>	128
11.5	<i>Adding Graphical Elements to an Existing Plot</i>	130
11.6	<i>Controlling the Axes</i>	133
12	GRAPHICAL DEVICES	137
12.1	<i>Available Devices</i>	137
12.2	<i>Device Management under Windows</i>	137
12.3	<i>List of device functions</i>	139
IV Statistics and Inference		141
13	BASIC STATISTICAL FUNCTIONS	143
13.1	<i>Statistical Summaries</i>	143
13.2	<i>Distribution Functions</i>	145
13.3	<i>Random Numbers</i>	147
13.4	<i>Hypothesis Testing</i>	148
13.5	<i>Parameter Estimation</i>	150
13.6	<i>Distribution tails and quantiles</i>	150
14	LINEAR TIME SERIES ANALYSIS	153
14.1	<i>Overview of Functions for Time Series Analysis</i>	153

14.2	<i>Simulation from an Autoregressive Process</i>	154
14.3	<i>AR - Fitting Autoregressive Models</i>	158
14.4	<i>Autoregressive Moving Average Modelling</i>	162
14.5	<i>Forecasting From Estimated Models</i>	165
15	REGRESSION MODELING	167
15.1	<i>Linear Regression Models</i>	167
15.2	<i>Parameter Estimation</i>	171
15.3	<i>Model Diagnostics</i>	174
15.4	<i>Updating a linear model</i>	176
16	DISSIMILARITIES OF DATA RECORDS	181
16.1	<i>Correlations and Pairwise Plots</i>	181
16.2	<i>Stars and Segments Plots</i>	188
16.3	<i>k-means Clustering</i>	189
16.4	<i>Hierarchical Clustering</i>	193
V	Case Studies: Utility Functions	197
17	COMPUTE SKEWNESS STATISTICS	199
17.1	<i>Assignment</i>	199
17.2	<i>R Implementation</i>	199
17.3	<i>Examples</i>	201
18	COMPUTE KURTOSIS STATISTICS	203
18.1	<i>Assignment</i>	203
18.2	<i>R Implementation</i>	203
18.3	<i>Examples</i>	204
19	EXTRACTING PACKAGE DESCRIPTION	205
19.1	<i>Assignment</i>	205
19.2	<i>R Implementation</i>	205
19.3	<i>Examples</i>	206
20	FUNCTION LISTING AND COUNTING	207
20.1	<i>Assignment</i>	207
20.2	<i>R Implementation</i>	207
20.3	<i>Examples</i>	208
VI	Case Studies: Asset Management	213
21	GENERALIZED ERROR DISTRIBUTION	215
21.1	<i>Assignment</i>	215

21.2	<i>R Implementation</i>	215
21.3	<i>Examples</i>	217
21.4	<i>Exercises</i>	221
22	SKEWED RETURN DISTRIBUTIONS	223
22.1	<i>Assignment</i>	223
22.2	<i>R Implementation</i>	223
22.3	<i>Examples</i>	225
22.4	<i>Exercise</i>	226
23	JARQUE-BERA HYPOTHESIS TEST	227
23.1	<i>Assignment</i>	227
23.2	<i>R Implementation</i>	228
23.3	<i>Examples</i>	229
24	PCA ORDERING OF ASSETS	231
24.1	<i>Assignment</i>	231
24.2	<i>R Implementation</i>	231
24.3	<i>Examples</i>	232
25	CLUSTERING OF ASSET RETURNS	235
25.1	<i>Assignment</i>	235
25.2	<i>R Implementation</i>	235
25.3	<i>Examples</i>	237
25.4	<i>Exercises</i>	237
VII	Case Studies: Option Valuation	239
26	BLACK-SCHOLES OPTION PRICE	241
26.1	<i>Assignment</i>	241
26.2	<i>R Implementation</i>	243
26.3	<i>Examples</i>	244
27	BLACK-SCHOLES OPTION GREEKS	247
27.1	<i>Assignment</i>	247
27.2	<i>R Implementation</i>	249
27.3	<i>Examples</i>	250
28	AMERICAN CALLS WITH DIVIDENDS	253
28.1	<i>Assignment</i>	253
28.2	<i>R Implementation</i>	254
28.3	<i>Examples</i>	256
29	MONTE CARLO OPTION PRICING	257

29.1	<i>Assignment</i>	257
29.2	<i>R Implementation</i>	257
29.3	<i>Examples</i>	258
29.4	<i>Exercises</i>	260
VIII Case Studies: Portfolio Design		261
30	MEAN-VARIANCE MARKOWITZ PORTFOLIO	263
30.1	<i>Assignment</i>	263
30.2	<i>R Implementation</i>	264
30.3	<i>Examples</i>	265
31	MARKOWITZ TANGENCY PORTFOLIO	269
31.1	<i>Assignment</i>	269
31.2	<i>R Implementation</i>	270
31.3	<i>Examples</i>	271
32	LONG ONLY PORTFOLIO FRONTIER	273
32.1	<i>Assignment</i>	273
32.2	<i>R Implementation</i>	273
32.3	<i>Examples</i>	274
33	MINIMUM REGRET PORTFOLIO	277
33.1	<i>Assignment</i>	277
33.2	<i>R Implementation</i>	277
33.3	<i>Examples</i>	279
IX Appendix		281
A	PACKAGES REQUIRED FOR THIS EBOOK	283
A.1	<i>Rmetrics Package: fBasics</i>	283
A.2	<i>Contributed R Package: quadprog</i>	284
A.3	<i>Contributed R Package: Rglpk</i>	284
A.4	<i>Recommended Packages from base R</i>	285
B	R MANUALS ON CRAN	287
C	RMETRICS ASSOCIATION	289
D	RMETRICS TERMS OF LEGAL USE	293
	INDEX	295
	ABOUT THE AUTHORS	303

CHAPTER 5

WRITING FUNCTIONS

Most tasks are performed by calling a function in R. In fact, everything we have done so far is calling an existing function, which then performed a certain task resulting in some kind of output. A function can be regarded as a collection of statements and is an object in R of class `function`. One of the strengths of R is the ability to extend R by writing new functions.

5.1 WRITING YOUR FIRST FUNCTION

The general form of a function is given by:

```
functionname <- function(arg1, arg2,...) {  
  <<expressions>>  
}
```

In the above display `arg1` and `arg2` in the function header are input arguments of the function. Note that a function does not need to have any input arguments. The body of the function consists of valid R statements. For example, the commands, functions and expressions you type in the R console window. Normally, the last statement of the function body will be the return value of the function. This can be a vector, a matrix or any other data structure. Thus, it is not necessary to explicitly use `return()`. The following short function `tmean` calculates the mean of a vector `x` by removing the `k` percent smallest and the `k` percent largest elements of the vector. We call this mean a trimmed mean, therefore we named the function `tmean`

```
> tmean <- function(x, k) {  
  xt <- quantile(x, c(k, 1 - k))  
  mean(x[x > xt[1] & x < xt[2]])  
}
```

Once the function has been created, it can be run.

```
> test <- rnorm(100)
> tmean(test, 0.05)
[1] -0.012331
```

The function `tmean` calls two standard functions, `quantile` and `mean`. Once `tmean` is created it can be called from any other function.

If you write a short function, a one-liner or two-liner, you can type the function directly in the console window. If you write longer functions, it is more convenient to use a script file. Type the function definition in a script file and run the script file. Note that when you run a script file with a function definition, you will only define the function (you will create a new object). To actually run it, you will need to call the function with the necessary arguments.

Saving your function in a script file

You can use your favourite text editor to create or edit functions. Use the function source to evaluate expressions from a file. Suppose `tmean.R` is a text file, saved on your hard disk, containing the function definition of `tmean()`. In this example we use the function `dump()` to export the `tmean()` to a text file.

```
> tmean <- function(x, k) {
  xt <- quantile(x, c(k, 1 - k))
  mean(x[x > xt[1] & x < xt[2]])
}
> dump("tmean", "tmean.R")
```

You can load the function `tmean` in a new R session by using the `source()` function. It is important to specify the relative path to your file if R has not been started in the same directory where the source file is. You can use the function `setwd()` to change the working directory of your R session or use the GUI menu “Change working directory” if available.

```
> source("tmean.R")
```

Now we can run the function:

```
> tmean(test, 0.05)
[1] -0.012331
```

Using comments

If you want to put a comment inside a function, use the `#` symbol. Anything between the `#` symbol and the end of the line will be ignored.

Viewing function code

Writing large functions in R can be difficult for novice users. You may wonder where and how to begin, how to check input parameters or how to use loop structures.

Fortunately, the code of many functions can be viewed directly. For example, just type the name of a function without brackets in the console window and you will get the code. Don't be intimidated by the (lengthy) code. Learn from it, by trying to read line by line and looking at the help of the functions that you don't know yet. Some functions call 'internal' functions or pre-compiled code, which can be recognized by calls such as: `.C`, `.Internal` or `.Call`.

5.2 ARGUMENTS AND VARIABLES

In this section we explain the difference between required and optional arguments, explain the meaning of the `...` argument, introduce local variables, and show the different options for returning an object from a function.

Required and optional arguments

When calling functions in R, the syntax of the function definition determines whether argument values are required or optional. With optional arguments, the specification of the arguments in the function header is:

```
argname = defaultvalue
```

In the following function, for example, the argument `x` is required and R will give an error if you don't provide it. The argument `k` is optional, having the default value 2:

```
> power <- function(x, k = 2) {
  x^k
}
```

Run it

```
> power(5)
[1] 25
```

Bear in mind that `x` is a required argument. You have to specify it, otherwise you will get an error.

```
> power()
Error in power() : argument "x" is missing, with no default
```

To compute the third power of x , we can specify a different value for k and set it to 3:

```
> power(5, k = 3)
[1] 125
```

The `'...'` argument

The three dots argument can be used to pass arguments from one function to another. For example, graphical parameters that are passed to plotting functions or numerical parameters that are passed to numerical routines. Suppose you write a small function to plot the `sin()` function from zero to `xup`.

```
> sinPlot <- function(xup = 2 * pi, ...) {
  x <- seq(0, xup, l = 100)
  plot(x, sin(x), type = "l", ...)
}

> sinPlot(col = "red")
```

The function `sinPlot` now accepts any argument that can be passed to the `plot()` function (such as `col()`, `xlab()`, etc.) without needing to specify those arguments in the header of `sinPlot`.

Local variables

Assignments of variables inside a function are local, unless you explicitly use a global assignment (the `"<-"` construction or the `assign` function). This means a normal assignment within a function will not overwrite objects outside the function. An object created within a function will be lost when the function has finished. Only if the last line of the function definition is an assignment, then the result of that assignment will be returned by the function. Note that it is not recommended to use global variables in any R code.

In the next example an object `x` will be defined with value zero. Inside the function `functionx`, `x` is defined with value 3. Executing the function `functionx` will not affect the value of the global variable `'x'`.

```
> x <- 0
> reassign <- function() {
  x <- 3
}

> reassign()
> x
[1] 0
```

If you want to change the global variable `x` with the return value of the function `reassign`, you must assign the function result to `x`. This overwrites the object `x` with the result of the `reassign` function

```
> x <- reassign()
> x
[1] 3
```

The arguments of a function can be objects of any type, even functions! Consider the next example:

```
> execFun <- function(x, fun) {
  fun(x)
}
```

Try it

```
> Sin <- execFun(pi/3, sin)
> Cos <- execFun(pi/3, cos)
> c(Sin, Cos, Sum = Sin * Sin + Cos * Cos)
              Sum
0.86603 0.50000 1.00000
```

The second argument of the function `execFun` needs to be a function which will be called inside the function.

Returning an object

Often the purpose of a function is to do some calculations on input arguments and return the result. As we have already seen in all previous examples, by default the last expression of the function will be returned.

```
> sumSinCos <- function(x, y) {
  Sin <- sin(x)
  Cos <- cos(y)
  Sin + Cos
}

> sumSinCos(0.2, 1/5)
[1] 1.1787
```

In the above example `Sin + Cos` is returned, whereas the individual objects `Sin` and `Cos` will be lost. You can only return one object. If you want to return more than one object, you can return them in a list where the components of the list are the objects to be returned. For example

```
> sumSinCos <- function(x, y) {
  Sin <- sin(x)
  Cos <- cos(y)
  list(Sin, Cos, Sum = Sin + Cos)
}
```

```

> sumSinCos(0.2, 1/5)
[[1]]
[1] 0.19867

[[2]]
[1] 0.98007

$Sum
[1] 1.1787

```

To exit a function before it reaches the last line, use the `return` function. Any code after the `return` statement inside a function will be ignored. For example:

```

> SinCos <- function(x, y) {
  Sin <- sin(x)
  Cos <- cos(y)
  if (Cos > 0) {
    return(Sin + Cos)
  }
  else {
    return(Sin - Cos)
  }
}

> SinCos(0.2, 1/5)
[1] 1.1787

> sin(0.2) + cos(1/5)
[1] 1.1787

> sin(0.2) - cos(1/5)
[1] -0.7814

```

5.3 SCOPING RULES

The scoping rules of a programming language are the rules that determine how the programming language finds a value for a variable. This is especially important for *free* variables inside a function and for functions defined inside a function. Let's look at the following example function.

```

> myScope <- function(x) {
  y <- 6
  z <- x + y + a1
  a2 <- 9
  insidef = function(p) {
    tmp <- p * a2
    sin(tmp)
  }
  2 * insidef(z)
}

```

In the above function

- `x`, `p` are formal arguments.
- `y`, `tmp` are local variables.
- `a2` is a local variable in the function `myScope`.
- `a2` is a free variable in the function `insidef`.

R uses a so-called *lexical scoping* rule to find the value of free variables. With lexical scoping, free variables are first resolved in the environment in which the function was created. The following calls to the function `myScope` shows this rule.

In the first example R tries to find `a1` in the environment where `myScope` was created but there is no object `a1`

```
> myScope(8)
Error in myf(8) : object "a1" not found
```

Now let us define the objects `a1` and `a2` but what value was assigned to `a2` in the function `insidef`?

```
> a1 <- 10
> a2 <- 1000
> myScope(8)
[1] 1.3921
```

It took `a2` in `myScope`, so `a2` has the value 9.

5.4 LAZY EVALUATION

When writing functions in R, a function argument can be defined as an expression like

```
> myf <- function(x, nc = length(x)) {
  }
```

When arguments are defined in such a way you must be aware of the *lazy evaluation* mechanism in R. This means that arguments of a function are not evaluated until needed. Consider the following examples.

```
> myf <- function(x, nc = length(x)) {
  x <- c(x, x)
  print(nc)
}

> xin <- 1:10
> myf(xin)
```

[1] 20

The argument `nc` is evaluated after `x` has doubled in length, it is not ten, the initial length of `x` when it entered the function.

```
> logplot <- function(y, ylab = deparse(substitute(y))) {
  y <- log(y)
  plot(y, ylab = ylab)
}
```

The plot will create a nasty label on the y axis. This is the result of lazy evaluation, `ylab` is evaluated after `y` has changed. One solution is to force an evaluation of `ylab` first

```
> logplot <- function(y, ylab = deparse(substitute(y))) {
  ylab
  y <- log(y)
  plot(y, ylab = ylab)
}
```

5.5 FLOW CONTROL

The following shows a list of constructions to perform testing and looping. These constructions can also be used outside a function to control the flow of execution.

Tests with `if()`

The general form of the `if` construction has the form

```
if(test) {
  <<statements1>>
} else {
  <<statements2>>
}
```

where `test` is a logical expression such as `x < 0` or `x < 0 & x > -8`. R evaluates the logical expression; if it results in `TRUE`, it executes the `true` statements. If the logical expression results in `FALSE`, then it executes the `false` statements. Note that it is not necessary to have the `else` block. Adding two vectors in R of different length will cause R to recycle the shorter vector. The following function adds the two vectors by chopping of the longer vector so that it has the same length as the shorter.

```
> myplus <- function(x, y) {
  n1 <- length(x)
  n2 <- length(y)
  if (n1 > n2) {
```

```
        z <- x[1:n2] + y
      }
    else {
      z <- x + y[1:n1]
    }
  }
  z
}
```

```
> myplus(1:10, 1:3)
```

```
[1] 2 4 6
```

Tests with switch()

The switch function has the following general form.

```
switch(object,
  "value1" = {expr1},
  "value2" = {expr2},
  "value3" = {expr3},
  {other expressions}
)
```

If object has value value1 then expr1 is executed, if it has value2 then expr2 is executed and so on. If object has no match then other expressions is executed. Note that the block {other expressions} does not have to be present, the switch will return NULL if object does not match any value. An expression expr1 in the above construction can consist of multiple statements. Each statement should be separated with a ; or on a separate line and surrounded by curly brackets.

Example:

Choosing between two calculation methods:

```
> mycalc <- function(x, method = "ml") {
  switch(method, ml = {
    my.mlmethod(x)
  }, rml = {
    my.rmlmethod(x)
  })
}
```

Looping with for

The for, while and repeat constructions are designed to perform loops in R. They have the following forms.

```
for (i in for_object) {
```

```
<<some expressions>>
}
```

In the loop some expressions are evaluated for each element i in `for_object`.

Example: A recursive filter.

```
> arsim <- function(x, phi) {
  for (i in 2:length(x)) {
    x[i] <- x[i] + phi * x[i - 1]
  }
  x
}

> arsim(1:10, 0.75)
[1] 1.0000 2.7500 5.0625 7.7969 10.8477 14.1357 17.6018 21.2014 24.9010
[10] 28.6758
```

Note that the `for_object` could be a vector, a matrix, a data frame or a list.

Looping with while()

```
while (condition) {
  <<some expressions>>
}
```

In the while loop some expressions are repeatedly executed until the logical condition is FALSE. Make sure that the condition is FALSE at some stage, otherwise the loop will go on indefinitely.

Example:

```
> mycalc <- function() {
  tmp <- 0
  n <- 0
  while (tmp < 100) {
    tmp <- tmp + rbinom(1, 10, 0.5)
    n <- n + 1
  }
  cat("It took ")
  cat(n)
  cat(" iterations to finish \n")
}
```

Looping with repeat()

```
repeat
```

```
{  
  <<commands>>  
}
```

In the repeat loop «commands» are repeated *infinitely*, so repeat loops will have to contain a break statement to escape them.

CHAPTER 33

MINIMUM REGRET PORTFOLIO

33.1 ASSIGNMENT

The minimum regret portfolio maximizes the minimum return for a set of return scenarios. This can be accomplished by solving the following linear program.

$$\begin{aligned} \max_{R_{min}, w} \quad & R_{min} \\ \text{s.t.} \quad & \\ & w^\top \hat{\mu} = \bar{\mu} \\ & w^\top \mathbf{1} = 1 \\ & w_i \geq 0 \\ & w^\top r_s - R_{min} \geq 0 \end{aligned} \tag{33.1}$$

Let us write a function which solves this optimization problem.

References

Bernd Michael Scherer and R. Douglas Martin, 2005, Introduction to Modern Portfolio Optimization with NuOPT and S-PLUS, Springer Publishing, New York

GNU Linear Programming Kit
<http://www.gnu.org/software/glpk/glpk.html>

33.2 R IMPLEMENTATION

To solve a linear optimization program with linear constraints we use R's contributed `Rglpk`, which has implemented GNU's linear programming solver tool kit. Load the library and the arguments of the solver.

```

> library(Rglpk)
> args(Rglpk_solve_LP)
function (obj, mat, dir, rhs, bounds = NULL, types = NULL, max = FALSE,
         control = list(), ...)
NULL

```

The arguments have the following meaning

obj	a vector with the objective coefficients
mat	a vector or a matrix of the constraint coefficients
dir	a character vector with the directions of the constraints. Each element must be one of "<", "<=", ">", ">=", or "==".
rhs	the right hand side of the constraints
types	a vector indicating the types of the objective variables. types can be either "B" for binary, "C" for continuous or "I" for integer. By default all variables are of type "C".
max	a logical giving the direction of the optimization. TRUE means that the objective is to maximize the objective function, FALSE (default) means to minimize it.
bounds	NULL (default) or a list with elements upper and lower containing the indices and corresponding bounds of the objective variables. The default for each variable is a bound between 0 and Inf.
verbose	a logical for turning on/off additional solver output, Default: FALSE.

The function returns a list with the following components:

solution	the vector of optimal coefficients
objval	the value of the objective function at the optimum
status	an integer with status information about the solution returned: 0 if the optimal solution was found, a non-zero value otherwise.

Alternatively we can use the solver function `Rsymphony_solve_LP()` from the contributed package `Rsymphony`.

Now we are ready to write a function to optimize the minimum regret portfolio for a set of asset returns and a given target return. The function body consists of several parts: 1 defining the vector for the objective function, 2 setting up the matrix of linear constraints excluding the simple bounds, 3 creating the vectors of directions and the value of the right hand side, and 4 setting the values for the lower and upper bounds. And the final step is the optimization itself.

```

> portfolioWeights <- function(assetReturns, targetReturn) {
  assetNames = colnames(assetReturns)
  assetReturns = as.matrix(assetReturns)
  nAssets = ncol(assetReturns)
  nScenarios = nrow(assetReturns)
  mu = colMeans(assetReturns)
  obj <- c(R_min = 1, Weights = rep(0, nAssets))
  mat = rbind(cbind(matrix(0, ncol = 1), t(mu)), cbind(matrix(0,
    ncol = 1), t(rep(1, nAssets))), cbind(matrix(rep(-1,

```

```

nScenarios), nScenarios), -assetReturns))
dir = c(Return = "==", Budget = "==", Scenarios = rep(">=",
nScenarios))
rhs = c>Returns = targetReturn, Budget = 1, Scenarios = rep(0,
nScenarios))
bounds = list()
bounds$lower$ind = 1:length(obj)
bounds$upper$ind = 1:length(obj)
bounds$lower$val = c(Rmin = -Inf, Weights = rep(0, nAssets))
bounds$upper$val = c(Rmin = Inf, Weights = rep(1, nAssets))
ans = Rglpk_solve_LP(obj = obj, mat = mat, dir = dir, rhs = rhs,
bounds = bounds, max = TRUE)
weights = ans$solution[-1]
names(weights) = assetNames
weights
}

```

33.3 EXAMPLES

As an example we consider again as in the previous case study the Swiss pension fund benchmark. The data can be loaded from the Rmetrics package `fBasics`. We use daily percentage returns

```

> library(fBasics)
> assetReturns <- 100 * LPP2005REC[, 1:6]
> head(assetReturns)
GMT
          SBI      SPI      SII      LMI      MPI      ALT
2005-11-01 -0.061275  0.841460 -0.31909 -0.110888  0.154806 -0.257297
2005-11-02 -0.276201  0.251934 -0.41176 -0.117594  0.034288 -0.114160
2005-11-03 -0.115309  1.270729 -0.52094 -0.099246  1.050296  0.500744
2005-11-04 -0.323575 -0.070276 -0.11272 -0.119853  1.167956  0.948268
2005-11-07  0.131097  0.620523 -0.17958  0.036037  0.270962  0.472395
2005-11-08  0.053931  0.032926  0.21034  0.232704  0.034684  0.085362
> end(assetReturns)
GMT
[1] [2007-04-11]

```

In this example we choose the grand mean of all assets as the values for the target return.

```

> targetReturn = mean(assetReturns)
> targetReturn
[1] 0.043077

```

The next step will be the optimization of the portfolio

```

> weights = portfolioWeights(assetReturns, targetReturn)
> weights
          SBI      SPI      SII      LMI      MPI      ALT
0.000000  0.033482  0.118310  0.440168  0.000000  0.408041

```

Now compare the weights with those from the mean-variance Markowitz portfolio.