Experimental Analysis of Optimization Heuristics Using R

Marco Chiarandini

October 5, 2012

Abstract

This document introduces the reader to the analysis of results of computational experiments on heuristic algorithms for optimization by means of R, the free software environment for statistical computing and graphics. In the first chapter relevant R commands are listed. In the second chapter, a basic analysis is developed on two example studies. In chapter three, the setup and use of the racing method for configuration and tuning of the algorithms is described. In chapter four, elements from survival analysis and extreme value theory are used to model the behaviour of the algorithms. In chapter five, advanced analysis from experimental design are reviewed.

This is a very preliminary version and suggestions and corrections are very welcome.

Contents

Chapter 1

Introduction

1.1 R Download and Installation

- R web page (http://www.r-project.org/), select Download - CRAN → Select a mirror → Select in the first frame, named 'Download and Install R', your operating system.
- Windows: → select 'base' → click on 'R-?.?.?-win32.exe'.
- Under Linux Ubuntu:

  ```
  sudo apt-get install r-base
  ```

- Alternatively, go to http://cran.r-project.org/doc/manuals/R-admin.html and follow description there.
- A package is a collection of functions and programs that can be used within R. To install new packages type from R command line:

  ```
  > install.packages("lattice")
  ```

  See http://mirrors.dotsrc.org/cran/web/packages/index.html for an alphabetic list of packages and http://mirrors.dotsrc.org/cran/web/views/ for a thematic organization of the packages.
- Package `Rcmdr` provides a graphical interface to R.
- R documentation: see http://www.sbtc.ltd.uk/freenotes.html: 'Getting Started in R' by Saghir Bashir, and the references under the R link http://cran.r-project.org/manuals.html.
- Emacs users can find a mode for R at http://ess.r-project.org/.

1.2 Basic commands

See help page for details on all commands listed here.

1.2.1 System, help and documentation commands

- `R` starts R from command line.
- `library(Rcmdr)` loads the `Rcmdr` package and starts graphical interface.
- `q()` quits your R session.
- `ls()` provides a list of objects in the current R workspace.
- `options(width=120)` determines the position of the line break in R output.
- `?plot` a question mark followed by the name of the function opens the help page relative to that function.
- `help.start()` opens a browser with documentation.
- `example(plot)` calls one or more examples implemented for the function.
- `demo(package.name)` calls a demonstration for the functionality of the defined package.
- `vignette(package="packagename",topic="name")` opens a pdf document on the package, if provided by the maintainer.

1.2.2 Data and operations

- `read.table()` and `write.table()` read and write from text file.
- `load()` and `save()` load and save R objects.
- `source("myscript.r")` loads and executes an R script.
- `^`, `%%`, `%/%` operators for power, modulus and integer part of the division, respectively.
- `c()` function used to collect objects together into a vector, example: `x <- c(1,2,3)`

```
> c(A1=1,list(A2=1))

$A1
[1] 1
$A2
[1] 1
```

- `vector()`, `matrix()`, `array()`, `data.frame()`, `list()` data structures. Tests or coercions can be done with `is.data.frame()`, `as.data.frame()`, respectively
- `integer()` `double()` data types. Can be queried or coerced with `is.integer()` and `as.integer()`
- `str()` `head()` `tail()` compactly displays the structure, head and tail of an arbitrary R object and a data.frame, respectively.

- `mean()`. `median()`, `sum()`, `var()`, `summary()`, `interquartile()` `range()` compute sample statistics.
- `factor()` offers an alternative way of storing character data. For example a *factor* can have four elements and two *levels*:

```
> algorithms <- c("greedy", "grasp", "greedy", "grasp")
> algorithms
[1] "greedy" "grasp"  "greedy" "grasp"

> algorithms <- factor(algorithms)
> algorithms

[1] greedy grasp  greedy grasp
Levels: grasp greedy
```

- Generate sequences of integers by:

```
> 1:12

 [1]  1  2  3  4  5  6  7  8  9 10 11 12

> seq(1, 21, by = 2)

 [1]  1  3  5  7  9 11 13 15 17 19 21

> rep(3, 12)

 [1] 3 3 3 3 3 3 3 3 3 3 3 3
```

- Generate factors by specifying the pattern of their levels

```
> gl(2, 8, labels = c("Control", "Treat"))

 [1] Control Control Control Control Control Control Control Control Treat
[10] Treat   Treat   Treat   Treat   Treat   Treat   Treat
Levels: Control Treat
```

- `expand.grid()` creates a data frame from all combinations of factors

```
> (table <- expand.grid(algorithm = algorithms, instance = c("A", "B")))

  algorithm instance
1    greedy        A
2     grasp        A
3    greedy        A
4     grasp        A
5    greedy        B
6     grasp        B
7    greedy        B
8     grasp        B
```

- `paste()`, `substr()` **`strsplit()`** work with strings. The first concatenates, the second returns substrings within two positions, the third splits strings. Example:

```
> colors <- c("red", "yellow", "green")
> paste(colors, "flowers")
[1] "red flowers"    "yellow flowers" "green flowers"

> paste("several ", colors, "s", sep = "")
[1] "several reds"    "several yellows" "several greens"

> paste("I like", colors, collapse = ", ")
[1] "I like red, I like yellow, I like green"

> substr(colors, 1, 2)
[1] "re" "ye" "gr"

> unlist(strsplit("a.b.c", "\\."))
[1] "a" "b" "c"


> sub("(.*)-(.*)-(.*)","\\1","a-b-c")
[1] "a"
```

1.2.3 Graphics

- `plot()`, `lines()`, `points()`, `curve()`, `hist()`, `barplot()`, `boxplot()`, graphics functions from the base installation
- `par()` lists and changes graphic setting
- `colors()` the colors available in R
- Graphics are device independent. Type `?device` to see on which device they can be printed and how.
- `dev.copy(dev=pdf,file='Rplot.pdf')` copies the graphic in a pdf file. Remember to close the pipeline with `dev.off()`
- `lattice` and `ggplot` two packages for multivariate conditional plots. Try `demo()` on them for a demonstration of their facilities. The package `lattice` is thoroughly explained in reference [6].

1.2.4 Data Manipulation and Reshaping Data

- `stack`, `unstack`, `reshape` and `merge` are useful functions for rearragging data.

```
> table$res <- runif(8, 0, 1)
> reshape(table, timevar = "algorithm", idvar = "instance", direction = "wide")

  instance res.greedy res.grasp
1        A     0.6576     0.958
5        B     0.0315     0.242
```

```
> tab <- data.frame(instance = c("A", "B"), opt = c(1, 2))
> merge(table, tab, by.x = "instance", by.y = "instance")

  instance algorithm    res opt
1        A    greedy 0.6576   1
2        A     grasp 0.9577   1
3        A    greedy 0.0712   1
4        A     grasp 0.0646   1
5        B    greedy 0.0315   2
6        B     grasp 0.2423   2
7        B    greedy 0.9633   2
8        B     grasp 0.7597   2
```

- Function `melt` from package `reshape`:

```
> L<-c(t1=list(table),t2=list(table))
> str(L,max.level=2)

List of 2
 $ t1:'data.frame':        8 obs. of  3 variables:
  ..$ algorithm: Factor w/ 2 levels "grasp","greedy": 2 1 2 1 2 1 2 1
  ..$ instance : Factor w/ 2 levels "A","B": 1 1 1 1 2 2 2 2
  ..$ res      : num [1:8] 0.6576 0.9577 0.0712 0.0646 0.0315 ...
  ..- attr(*, "out.attrs")=List of 2
 $ t2:'data.frame':        8 obs. of  3 variables:
  ..$ algorithm: Factor w/ 2 levels "grasp","greedy": 2 1 2 1 2 1 2 1
  ..$ instance : Factor w/ 2 levels "A","B": 1 1 1 1 2 2 2 2
  ..$ res      : num [1:8] 0.6576 0.9577 0.0712 0.0646 0.0315 ...
  ..- attr(*, "out.attrs")=List of 2

> require(reshape)
> melt(L)

   algorithm instance variable  value L1
1     greedy        A      res 0.6576 t1
2      grasp        A      res 0.9577 t1
3     greedy        A      res 0.0712 t1
4      grasp        A      res 0.0646 t1
5     greedy        B      res 0.0315 t1
6      grasp        B      res 0.2423 t1
7     greedy        B      res 0.9633 t1
8      grasp        B      res 0.7597 t1
```

```
9     greedy        A      res 0.6576 t2
10     grasp        A      res 0.9577 t2
11    greedy        A      res 0.0712 t2
12     grasp        A      res 0.0646 t2
13    greedy        B      res 0.0315 t2
14     grasp        B      res 0.2423 t2
15    greedy        B      res 0.9633 t2
16     grasp        B      res 0.7597 t2
```

```
> table$res.1 <- runif(8, 0, 1)
> table$res.2 <- runif(8, 0, 1)
> head(table)

  algorithm instance    res res.1  res.2
1    greedy        A 0.6576 0.563 0.5185
2     grasp        A 0.9577 0.941 0.7016
3    greedy        A 0.0712 0.521 0.0953
4     grasp        A 0.0646 0.570 0.4818
5    greedy        B 0.0315 0.296 0.1257
6     grasp        B 0.2423 0.541 0.4199

> reshape(table,direction="long",varying=list(3:5),times=c("0","1","2"))

    algorithm instance time    res id
1.0    greedy        A    0 0.6576  1
2.0     grasp        A    0 0.9577  2
3.0    greedy        A    0 0.0712  3
4.0     grasp        A    0 0.0646  4
5.0    greedy        B    0 0.0315  5
6.0     grasp        B    0 0.2423  6
7.0    greedy        B    0 0.9633  7
8.0     grasp        B    0 0.7597  8
1.1    greedy        A    1 0.5635  1
2.1     grasp        A    1 0.9410  2
3.1    greedy        A    1 0.5215  3
4.1     grasp        A    1 0.5703  4
5.1    greedy        B    1 0.2959  5
6.1     grasp        B    1 0.5411  6
7.1    greedy        B    1 0.2157  7
8.1     grasp        B    1 0.9964  8
1.2    greedy        A    2 0.5185  1
2.2     grasp        A    2 0.7016  2
3.2    greedy        A    2 0.0953  3
4.2     grasp        A    2 0.4818  4
5.2    greedy        B    2 0.1257  5
6.2     grasp        B    2 0.4199  6
7.2    greedy        B    2 0.7856  7
8.2     grasp        B    2 0.0616  8
```

- `which()` returns the index of an element in a vector.
  Example: `which(!(colnames(table) %in% c("algorithm","instace")))`
- `which.min()` returns the index of the minimal element

- drop unused levels from factors in a data frame after subsetting data frames.

```
CHEUR.01<-subset(CHEUR, variability!="no")
CHEUR.01 <- droplevels(CHEUR.01)
```

  See also `factor` for definition of factors `drop` for dropping array dimensions. `drop1` for dropping terms from a model.

- For rank transformations within instances:

```
D<-CHEUR.01
D$rank <- D$res
split(D$rank, D$inst) <- lapply(split(D$res, D$inst), rank)
tapply(D$rank, D$alg, median)
```

- package **xtable** provides a function to convert an R object into a LaTeX or an HTML table.

```
> library(xtable)
> xtable(table)

% latex table generated in R 2.9.1 by xtable 1.5-5 package
% Mon Aug 31 19:41:09 2009
\begin{table}[ht]
\begin{center}
\begin{tabular}{rllr}
  \hline
 & algorithm & instance & res \\
  \hline
1 & greedy & A & 0.64 \\
  2 & grasp & A & 0.91 \\
  3 & greedy & A & 0.83 \\
  4 & grasp & A & 0.03 \\
  5 & greedy & B & 0.98 \\
  6 & grasp & B & 0.97 \\
  7 & greedy & B & 0.63 \\
  8 & grasp & B & 0.31 \\
   \hline
\end{tabular}
\end{center}
\end{table}
```

- `Sweave` is a tool that allows to embed the R code in latex documents[10, 11]. It implements the literate programming concept.

1.3 Development Commands

- `.libPaths()` to check which paths are considered for loading libraries
- `missing()` tests whether a value was specified as an argument to a function.
- `stopifnot()` stops if one of a list of conditions is not true reporting an error with the first violated condition.

- `debug()` debugs a function. `Q` to exit debug mode.
- `sessionInfo()` prints version information about R and attached or loaded packages (see also `Sys.getlocale()`)
- ```
  > Sys.getpid()

  [1] 31629

  > getAnywhere("friedman.test")

  A single object matching 'friedman.test' was found
  It was found in the following places
    package:stats
    namespace:stats
  with value
  function (y, ...)
  UseMethod("friedman.test")
  <bytecode: 0xa4aff78>
  <environment: namespace:stats>

  > methods("friedman.test")

  [1] friedman.test.default* friedman.test.formula*
     Non-visible functions are asterisked
  ```
- `prompt, promtData, package.skeleton` create documentation for functions, data sets and packages, respectively.
- to interrupt where a problem arise, `options(error=stop)` `options(error=recover)`, or to turn wanrings into erros `options(warn=2)`.
- `debug` to proceed step by step
- `browser()` to stop exectution at that point and open debug mode
- `if ( interactive() ) { ANSWER <- readline("Continue?") }` stops and asks whether to continue
- `R CMD build package`, `R CMD check package.tgz`, `R CMD INSTALL package.tgz`

1.4 Memory issues

- `.Machine` and `.Platform` variables holding information on the numerical characteristics of the machine and information on the platform on which R is running. Consult these for information such as the largest double or integer and the machine's precision. See also `?"Memory-limits"`
- `object.size()` provides an estimate of the memory that is being used to store an R object.
- `gc(verbose=TRUE)` causes a garbage collection to take place and prints memory usage statistics
  ```

Chapter 2

Comparing Distributions of Experimental Results

In this chapter, we discuss basic ways to visualize results for the analysis of computational experiments in optimization. We have chosen two running examples. The first considers the combinatorial optimization problem of finding a good approximation to the chromatic number of a graph and evaluates different heuristics for it. The second is a continuous optimization problem drawn from the field of statistics and considers both the implementation of the solvers and their assessment.

2.1 Example 1: Discrete Optimization

In this example we wish to assess different heuristics for the graph coloring problem, ie, finding a good approximation of the chromatic number of a graph.

2.1.1 Univariate Analysis

We have collected results for different heuristics run for the same amount of time on a class of instances randomly generated. By construction we know that each instance has a coloring that uses a certain number of colors. This number is encoded in the name of the instance. We load the data as an R object and produce a first exploratory plot to investigate the distribution of data. We may choose one of the three ways to represent distributions depicted in Figure 2.1. All summary statistics of the distributions, such as location measures (mean, median, max) or dispersion measures (variance, standard deviation, interquartile) necessarily omit information. Inspection of the full distribution of data is therefore a good practice to understand which of these summary statistics better represent the data at hand.

Boxplots are advantages because they allow to fit easily several distributions on the same plot.

The data are stored in a text file organized in columns. The import in a data frame is straightforward. By means of `str` we gain a view on the data. There are three algorithms and ten instances. For each instance and algorithm ten results are collected. We rename the columns according to the information they provide and produce the boxplots, shown in Figure 2.2, upper left plot.

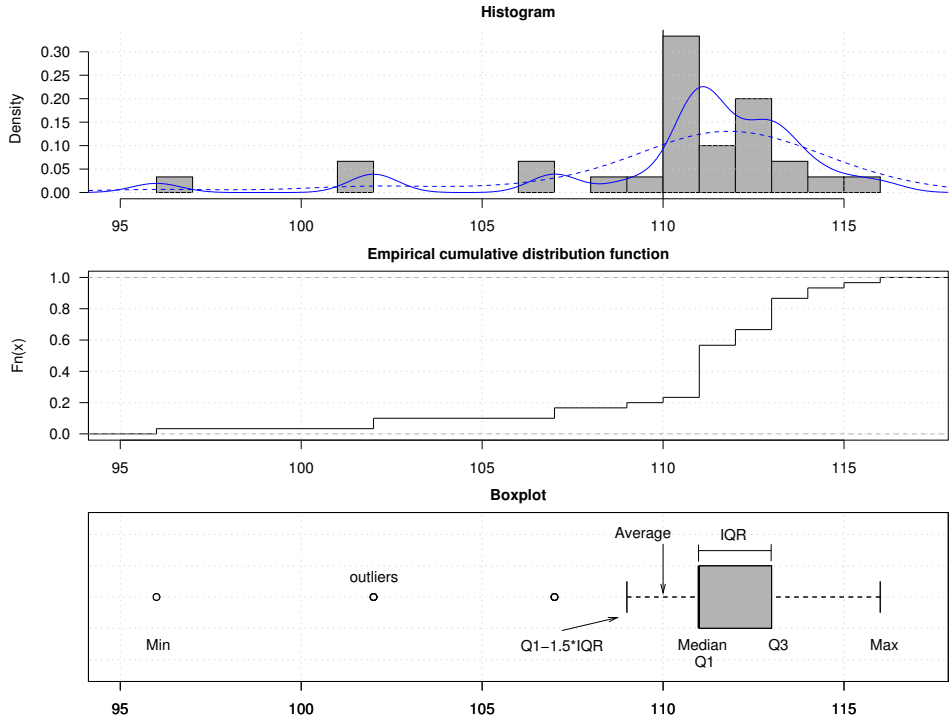


Figure 2.1: Histogram, density function, empirical distribution function and boxplots are different ways to look at the distribution of empirical data.

```
> G <- read.table("Data/TS-class-G.txt")
> names(G) <- c("alg", "inst", "trial", "sol", "time", "best")
> G[1:5, ]
```

```
  alg              inst trial sol time best
1 TS1 G-1000-0.5-30-1.1     1  59 9.90   30
2 TS1 G-1000-0.5-30-1.1     2  64 9.74   30
3 TS1 G-1000-0.5-30-1.1     3  64 9.91   30
4 TS1 G-1000-0.5-30-1.1     4  68 9.95   30
5 TS1 G-1000-0.5-30-1.1     5  63 9.91   30
```

```
> str(G)
```

```
'data.frame':        300 obs. of  6 variables:
 $ alg  : Factor w/ 3 levels "TS1","TS2","TS3": 1 1 1 1 1 1 1 1 1 1 ...
 $ inst : Factor w/ 10 levels "G-1000-0.5-30-1.1",..: 1 1 1 1 1 1 1 1 1 1 ...
 $ trial: int  1 2 3 4 5 6 7 8 9 10 ...
 $ sol  : int  59 64 64 68 63 63 65 65 71 62 ...
 $ time : num  9.9 9.74 9.91 9.95 9.91 ...
 $ best : int  30 30 30 30 30 30 30 30 30 30 ...
```

```
> par(mfrow = c(2, 2), las = 1, font.main = 1, mar = c(2, 3, 3,
    1))
> boxplot(sol ~ alg, data = G, horizontal = TRUE, main = "Original data")
```

This way of aggregating data tends to confound effects because instances may have different scales. For example an instance can have optimal solution 20 another 100 and if the algorithms are good they may find solutions close to these values, thus aggregating data would mix

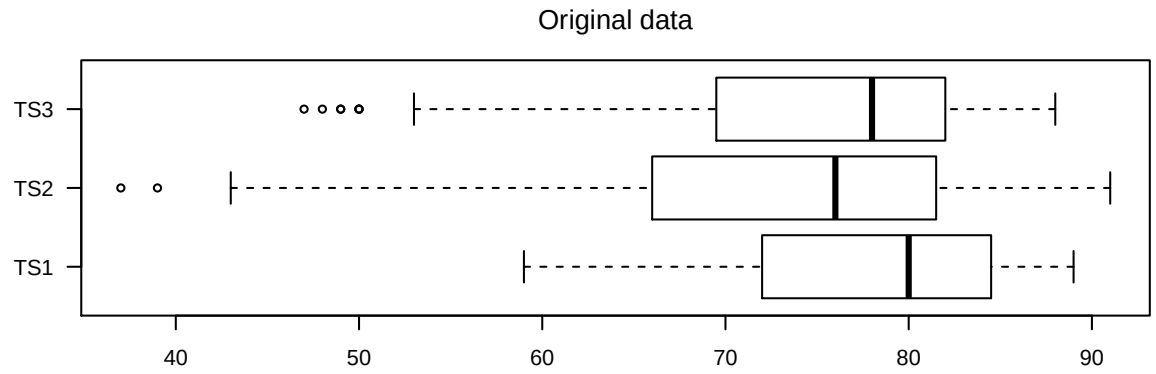


Figure 2.2: Aggregate analysis with original data

number that have little to do together. Let's have a closer look at these data by observing the distributions separately on each instance. For this we need the multi-panel functionality of the package `lattice`.

```
> library(lattice)
> print(bwplot(alg ~ sol | inst, data = G, layout = c(5, 2)))
```

It is clear from the resulting plot in Figure 2.3 that there are differences among the algorithms. Hence an aggregate analysis must make use of some data transformation in order to uniform instance scales.

We review four possible transformations, standard error, relative error, invariant error and ranks. The boxplots obtained by these transformations are reported in Figure 2.4

Standard error This is the classical transformation used in statistics indicating the distance from the mean value in terms of standard deviation. In R the function `scale` takes care of the transformation.

```
> T1 <- split(G$sol, list(G$inst))
> T2 <- lapply(T1, scale, center = TRUE, scale = TRUE)
> T3 <- unsplit(T2, list(G$inst))
> T4 <- split(T3, list(G$alg))
> T5 <- stack(T4)
> boxplot(values ~ ind, data = T5, horizontal = TRUE, main = expression(paste("Standard error: ",
    frac(x - bar(x), sqrt(sigma)))))
> library(latticeExtra)
> print(ecdfplot(~T5$values, groups = T5$ind, main = expression(paste("Standard error: ",
    frac(x - bar(x), sqrt(sigma))))))
```

More concisely, the addition of a column with the standard error could have been achieved as follows:

```
> G$scale <- 0
> split(G$scale, G$inst) <- lapply(split(G$sol, G$inst), scale,
    center = TRUE, scale = TRUE)
> bwplot(scale ~ reorder(alg, scale, mean), data = G, horizontal = TRUE,
    main = "scale")
```

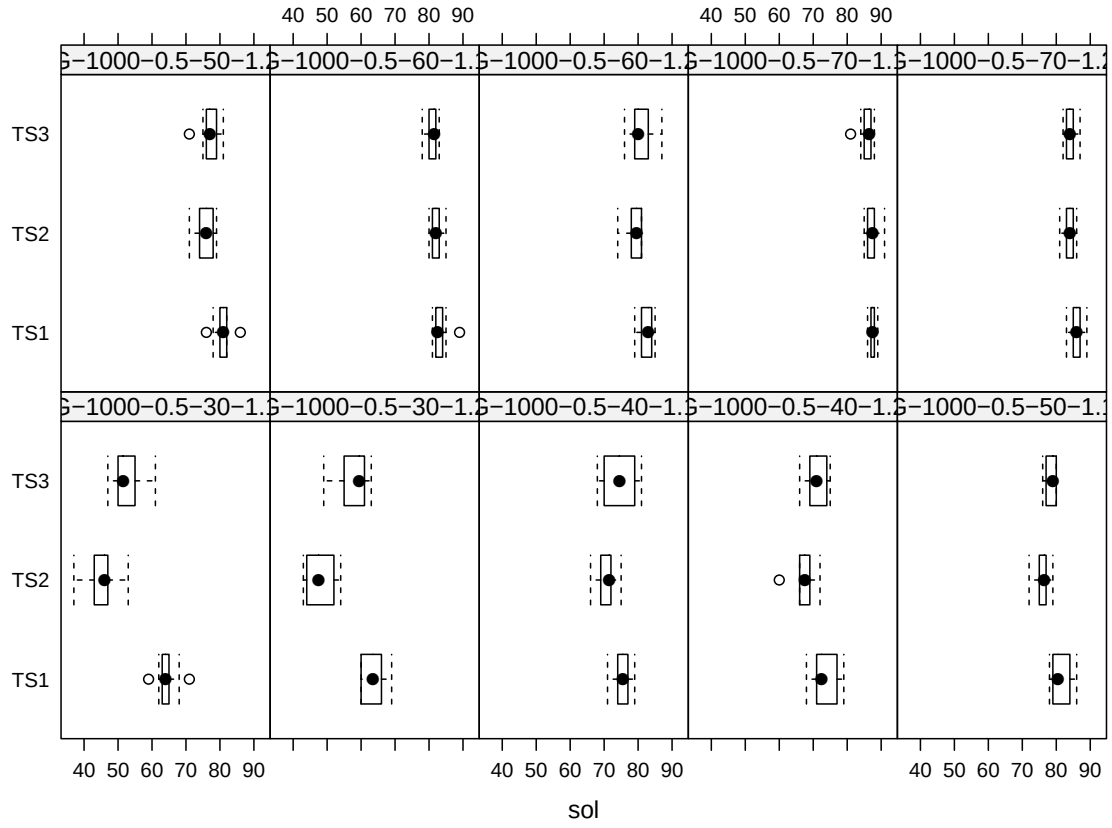


Figure 2.3: Boxplots of solution found by the algorithms on instance basis

Relative error Where lower bounds, best solutions or optimal solutions are known it is possible to use a measure more typically found in optimization, i.e., the relative error or optimality gap.

```
> G$err2 <- (G$sol - G$best)/G$best
> boxplot(err2 ~ alg, data = G, horizontal = TRUE, main = expression(paste("Relative error: ",
      frac(x - x^(best), x^(best)))))
> print(ecdfplot(~G$err2, groups = G$alg, main = expression(paste("Relative error: ",
      frac(x - x^(best), x^(best))))))
```

Invariant error The relative error is not invariant with repsct to linear transformation of the input data. It can be proved that substituting in the denominator x^{best} with $x^{worst} - x^{best}$, where x^{worst} is the worst possible result on the instance, the resulting mearue is invariant. Often x^{worst} is unknown and it might be hard to determine. Hence, we use as surrogate of x^{worst} the median solution returned by the simplest algorithm for the graph coloring, that is, the ROS heuristic.

```
> H <- read.table("Data/ROS-class-G.txt")
> names(H) <- c("alg", "inst", "trial", "sol", "time")
```

```
> X <- aggregate(H$sol, list(H$inst), median)
> G$ref <- sapply(G$inst, function(x) X[X$Group.1 == x, ]$x)
> G$err3 <- (G$sol - G$best)/(G$ref - G$best)
> boxplot(err3 ~ alg, data = G, horizontal = TRUE, main = expression(paste("Invariant error: ",
    frac(x - x^(best), x^(worst) - x^(best)))))
> print(ecdfplot(~G$err3, groups = G$alg, main = expression(paste("Invariant error: ",
    frac(x - x^(best), x^(worst) - x^(best))))))
```

Ranks The last transformation that we consider consists in ranking the data within each instance.

Let k be the number of candidate solvers and $X_{i1r}, X_{i2r}, \ldots, X_{ikr}$ the results they obtain on an instance i in the run r. Results over the r runs can be arranged as follows

	Solvers			
Instance	1	2	...	k
1	$X_{1,1,1}, \ldots, X_{1,1,r}$	$X_{1,2,1}, \ldots, X_{1,1,r}$	...	$X_{1,k,1}, \ldots, X_{1,1,r}$
2	$X_{2,1,1}, \ldots, X_{2,1,r}$	$X_{2,2,1}, \ldots, X_{2,1,r}$	...	$X_{2,k,1}, \ldots, X_{2,1,r}$
3	$X_{3,1,1}, \ldots, X_{3,1,r}$	$X_{3,2,1}, \ldots, X_{3,1,r}$	...	$X_{3,k,1}, \ldots, X_{3,1,r}$
...	...	...	...	...
10	$X_{10,1,1}, \ldots, X_{3,1,r}$	$X_{10,2,1}, \ldots, X_{10,1,r}$	...	$X_{10,k,1}, \ldots, X_{10,1,r}$

The results are transformed in ranks within each instance i assigning to each $X_{i,j,r}$ a value $R(X_{i,j,r})$ from 1 to rk. We can compute the sum of the ranks R_j and the average rank $\overline{R}_j$ for a solver j by

$$R_j = \sum_{i=1}^{10} \sum_{l=1}^{r} R(X_{i,j,r}) \qquad \overline{R}_j = \frac{R_j}{rk}$$

.

```
> G$rank <- 0
> split(G$rank, G$inst) <- lapply(split(G$sol, G$inst), rank)
> boxplot(rank ~ alg, data = G, horizontal = TRUE, main = "Ranks")
> print(ecdfplot(~rank, groups = alg, data = G, main = "Ranks"))
```

Numerical results and tables Once we observed the distributions we may choose the statistics worth to be reported in a table with numerical details. Since distributions are not perfectly symmetric we decide to use the median as summary measure of location and the interquartile, that is, the difference between the first and the third quartile[1]. Since we are also interested in the best results we also report the minimum of the results found. Then we use `xtable` to produce the table in LaTeX.

```
> df1 <- aggregate(G$sol, list(inst = G$inst, alg = G$alg), median)
> df2 <- aggregate(G$sol, list(inst = G$inst, alg = G$alg), IQR)
> df3 <- aggregate(G$sol, list(inst = G$inst, alg = G$alg), min)
> df <- cbind(df3, iqr = df1$x, min = df2$x)
```

[1]The p-quantile corresponds to the value x of a distribution such that $\Pr[X < x] = p$. The first quartile is the 25%-quantile, i.e., the value x such that $\Pr[X < x] = 0.25$.

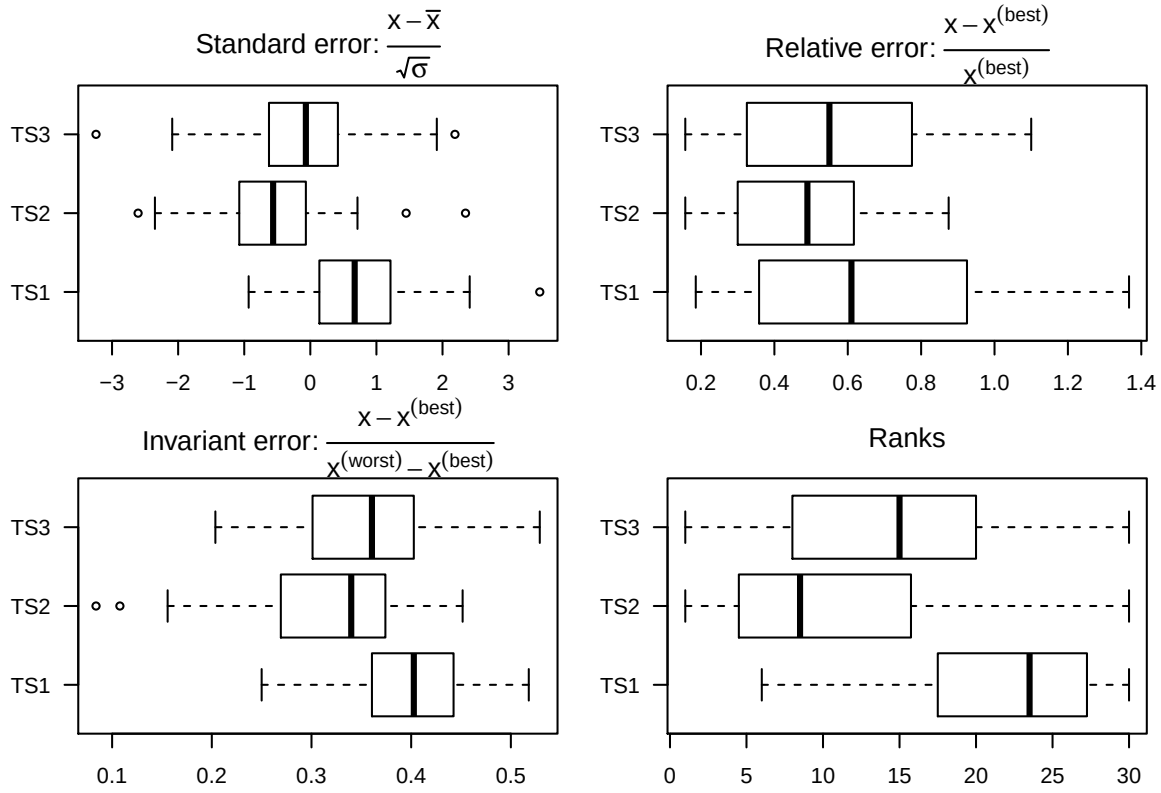


Figure 2.4: Aggregate analysis with different performance measures

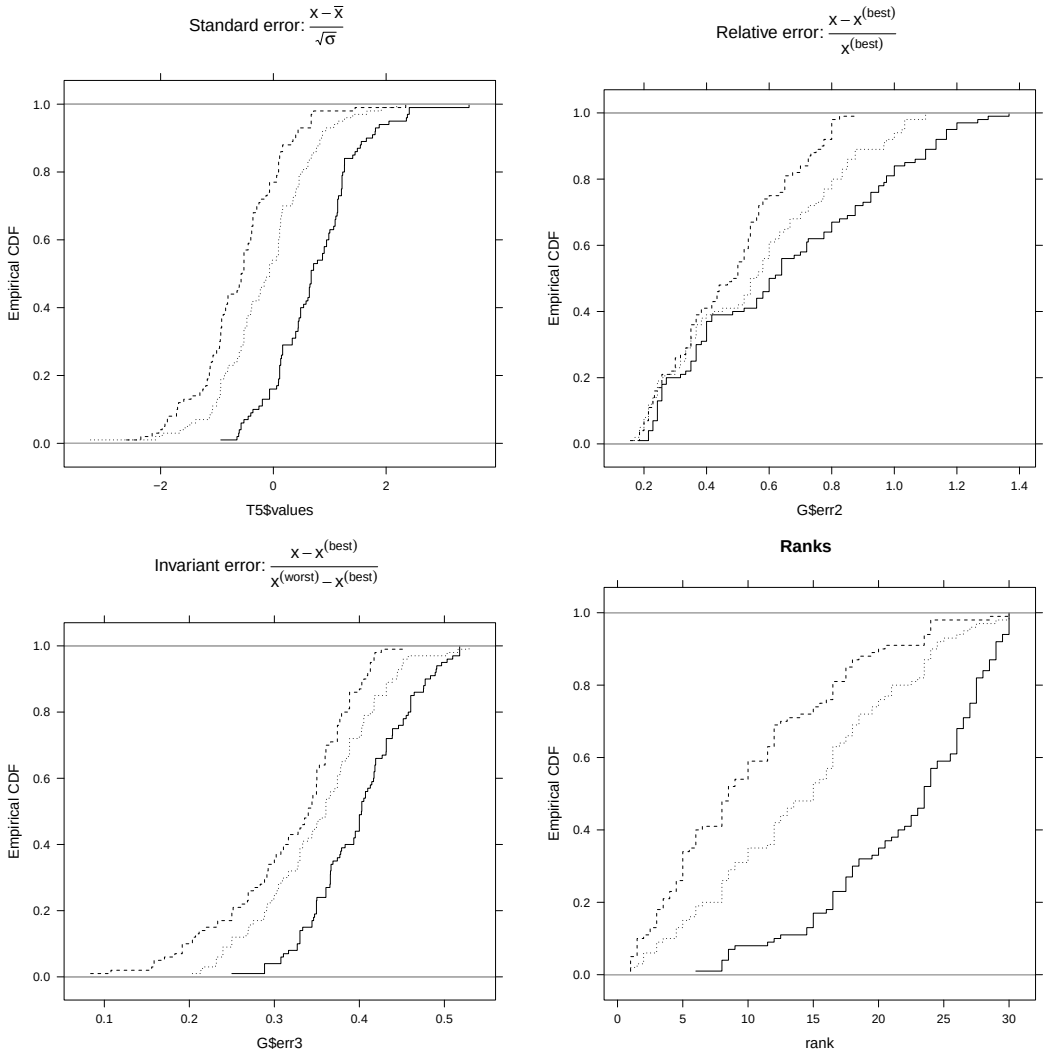


Figure 2.5: Analysis by means of empirical cumulative distribution functions for different performance measures

inst	x.TS1	iqr.TS1	min.TS1	x.TS2	iqr.TS2	min.TS2	x.TS3	iqr.TS3	min.TS3
G-1000-0.5-30-1.1	59	64.00	2.00	37	46.00	3.25	47	51.50	4.75
G-1000-0.5-30-1.2	60	63.50	5.50	43	47.50	7.00	49	59.50	6.00
G-1000-0.5-40-1.1	71	75.50	2.75	66	71.50	2.75	68	74.50	7.50
G-1000-0.5-40-1.2	68	72.50	5.25	60	67.50	3.00	66	71.00	4.00
G-1000-0.5-50-1.1	78	80.50	4.50	72	76.50	1.75	76	79.00	2.75
G-1000-0.5-50-1.2	76	81.00	2.00	71	76.00	3.50	71	77.00	2.75
G-1000-0.5-60-1.1	81	82.50	2.00	80	82.00	1.75	78	81.50	1.75
G-1000-0.5-60-1.2	79	83.00	2.75	74	79.50	3.00	76	80.00	3.50
G-1000-0.5-70-1.1	86	87.50	1.00	85	87.50	2.00	81	86.50	2.00
G-1000-0.5-70-1.2	83	86.00	2.00	81	84.00	1.75	82	84.00	2.00

```
> tab <- reshape(df, timevar = "alg", idvar = "inst", direction = "wide")
> library(xtable)
> print(xtable(tab), include.rownames = FALSE, floating = FALSE)

% latex table generated in R 2.12.0 by xtable 1.5-6 package
% Sat Jan 22 14:54:53 2011
\begin{tabular}{lrrrrrrrrr}
  \hline
inst & x.TS1 & iqr.TS1 & min.TS1 & x.TS2 & iqr.TS2 & min.TS2 & x.TS3 & iqr.TS3 & min.TS3 \\
  \hline
G-1000-0.5-30-1.1 &  59 & 64.00 & 2.00 &  37 & 46.00 & 3.25 &  47 & 51.50 & 4.75 \\
  G-1000-0.5-30-1.2 &  60 & 63.50 & 5.50 &  43 & 47.50 & 7.00 &  49 & 59.50 & 6.00 \\
  G-1000-0.5-40-1.1 &  71 & 75.50 & 2.75 &  66 & 71.50 & 2.75 &  68 & 74.50 & 7.50 \\
  G-1000-0.5-40-1.2 &  68 & 72.50 & 5.25 &  60 & 67.50 & 3.00 &  66 & 71.00 & 4.00 \\
  G-1000-0.5-50-1.1 &  78 & 80.50 & 4.50 &  72 & 76.50 & 1.75 &  76 & 79.00 & 2.75 \\
  G-1000-0.5-50-1.2 &  76 & 81.00 & 2.00 &  71 & 76.00 & 3.50 &  71 & 77.00 & 2.75 \\
  G-1000-0.5-60-1.1 &  81 & 82.50 & 2.00 &  80 & 82.00 & 1.75 &  78 & 81.50 & 1.75 \\
  G-1000-0.5-60-1.2 &  79 & 83.00 & 2.75 &  74 & 79.50 & 3.00 &  76 & 80.00 & 3.50 \\
  G-1000-0.5-70-1.1 &  86 & 87.50 & 1.00 &  85 & 87.50 & 2.00 &  81 & 86.50 & 2.00 \\
  G-1000-0.5-70-1.2 &  83 & 86.00 & 2.00 &  81 & 84.00 & 1.75 &  82 & 84.00 & 2.00 \\
   \hline
\end{tabular}
```

2.1.2 Bivariate Analysis

We now turn our attention to heurstics with natural termination criterion, leading to performance measures that depend on both time and solution quality.

For three such heuristics we collected one single run on 30 different instances with same or similar characteristics. As we saw, an aggregate analysis calls for some transformation of data. We choose the standard error transofmation in this case, although better transformations could be found.

We use the lattice `xyplot` to produce a scatter plot that gives us a comprehensive visualization of the data. The analysis could be produced also using the standard `plot` and `text` functions.

Figure 2.6 left shows the clouds of points associated with the three algorithms. It is evident that the algorithm RLF has much higher variability than the other two. If we have many mor algorithms to compare, we may wish to repsent data more syntetically by plotting only the median values. Moreover, logarithmic transformations are useful to emphasize differences.

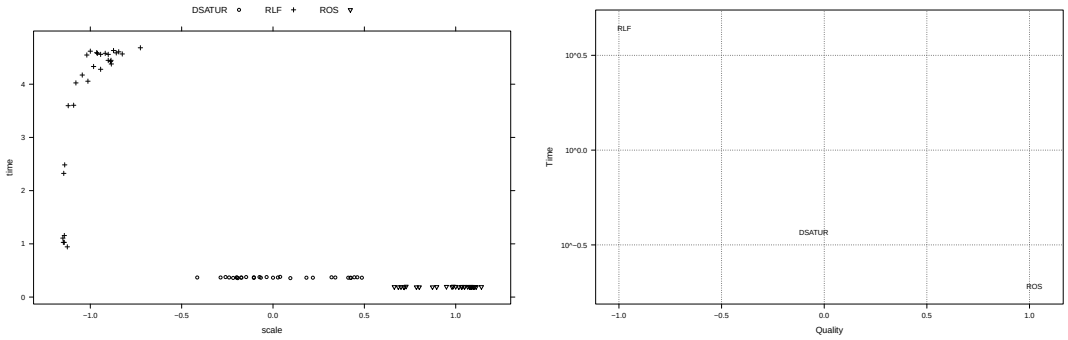


Figure 2.6: Bivariate analysis of time and quality. On the right the full data with scale transformation for the quality. On the left the median values and log transformation for the time

The resulting plot is given in Figure 2.6, right. The conclusion from this analysis is that the three algorithms are non domaniated in Pareto sense. Hence the decision on which to use will depend on the application.

```
> G <- read.table("/home/marco/Work08/RTutorial/Data/ch-class-G.txt")
> names(G) <- c("algo", "inst", "sol", "time")
> str(G)

'data.frame':        90 obs. of  4 variables:
 $ algo: Factor w/ 3 levels "DSATUR","RLF",..: 3 3 3 3 3 3 3 3 3 3 ...
 $ inst: Factor w/ 30 levels "G-1000-0.5-20-0.1",..: 1 2 3 4 5 6 7 8 9 10 ...
 $ sol : int  119 117 117 116 119 121 122 122 121 118 ...
 $ time: num  0.188 0.184 0.188 0.192 0.188 ...

> G$scale <- 0
> split(G$scale, G$inst) <- lapply(split(G$sol, G$inst), scale,
      center = TRUE, scale = TRUE)
> print(xyplot(time ~ scale, data = G, groups = algo, auto.key = list(columns = 3)))
> A <- aggregate(G$scale, list(algo = G$algo), median)
> B <- aggregate(G$time, list(algo = G$algo), median)
> G1 <- merge(A, B, by = "algo")
> names(G1) <- c("algo", "sol", "time")
> print(xyplot(time ~ sol, data = G1, groups = algo, scales = list(relation = "free",
      y = list(rot = 0, log = TRUE)), panel = function(x, y, subscripts,
      groups) {
      panel.grid(h = -1, v = -1, lty = 3, col = "grey30")
      ltext(x = x, y = y, label = groups[subscripts], cex = 0.8,
          fontfamily = "Helvetica")
 }, ylab = "Time", xlab = "Quality"))
```

2.2 Example 2: Continuous Optimization

Let's consider a basic linear regression model:

$$Y_i = \beta_0 + \beta_1 X_i + \epsilon \qquad i = 1, \ldots, n$$

where X_i is the value of the predictor variable in the ith trial, Y_i is the value of the response variable in the ith trial, β_0 and β_1 are parameters and ϵ_i are mutually independent random errors with $E[\epsilon_i] = 0$ and $\sigma[\epsilon_i] = 0$. We can rewrite in matrix notation:

$$\boldsymbol{Y} = \begin{bmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_n \end{bmatrix} \qquad \boldsymbol{X} = \begin{bmatrix} 1 & X_1 \\ 1 & X_2 \\ \vdots & \vdots \\ 1 & X_n \end{bmatrix} \qquad \boldsymbol{\beta} = \begin{bmatrix} \beta_0 \\ \beta_1 \end{bmatrix} \qquad \boldsymbol{\epsilon} = \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \vdots \\ \epsilon_n \end{bmatrix}$$

The $\mathbf{X}$ is often referred to as the *design matrix*. Hence,

$$\boldsymbol{Y} = \boldsymbol{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}$$

We simulate these data in R. The function `rexp(N,l)` generates `N` following an exponential distribution with mean `l`.

```
> set.seed(1)
> N <- 100
> X <- array(dim = c(N, 2))
> X[, 1] <- 1
> X[, 2] <- seq(0.01, 1, by = 0.1)
> Y <- X %*% matrix(c(0, 1)) + matrix(rexp(N, 1))
```

The usual way to estimate the values of the parameters $\boldsymbol{\beta}$ is by means of the least square method that minimizes:

$$\min \sum_{i=1}^{n} (Y_i - \beta_0 - \beta_1 X_i)^2$$

The estimators β_0 and β_1 can be found by analytical procedure and in R using the method `lm`.

```
> l <- lm.fit(X, Y)
> plot(X[, 2], Y)
> abline(l)
```

Here, we want instead to use the *least median of squares method*, that is,

$$z(\boldsymbol{\beta}) = \min \left\{ \text{median}[(Y_i - \beta_0 - \beta_1 X_i)^2] \right\}$$

The least median method should be more robust against outliers in the model, however the estimation of the parameters β_0 and β_1 requires minimizing a non-differentiable, non-linear, multi-modal function for which an analytical procedure is not known. We can inspect this visually using R methods for 3D plots.

First, we declare a function that implements the function we want to minimize, i.e., median$[(\boldsymbol{Y} - \boldsymbol{\beta}\boldsymbol{X})^2]$. R allows to declare functions in the following way:

```
lmedian <- function(beta, Response, Design) {
  counter <<- counter + 1
  median((Response - Design %*% beta)^2)
}
```

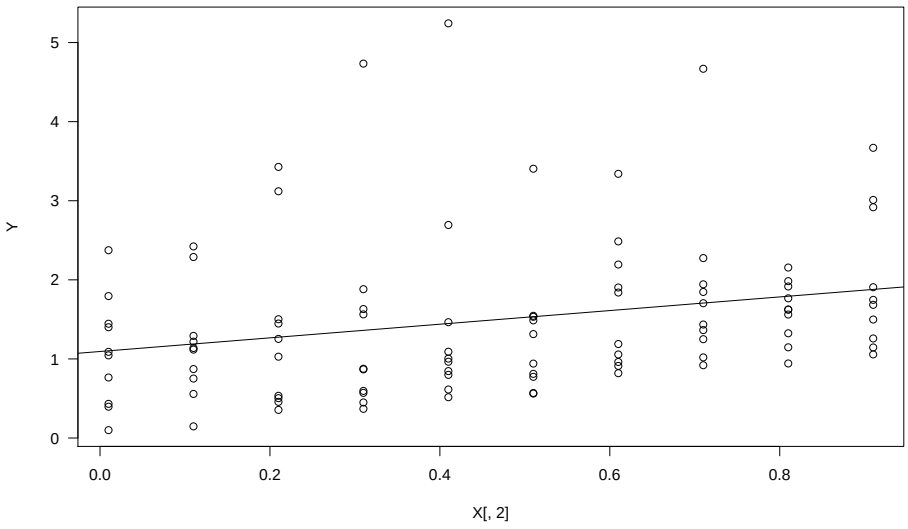


Figure 2.7: Scatter plot and linear regression line superimposed.

The operator `<-` is used for assignments. Assignments within functions are local. The operator `<<-` is a global assignment, the value of counter will remain modified also outside the function.

Then, we use the method `wireframe` from the package `lattice` to produce a 3D plot. In order to do this we must first determine plot points and then evaluate the function on these points

```
gr <- expand.grid(beta.0=seq(-1,1,0.08),
                  beta.1=seq(-1,5,0.1))
gr$z <- apply(gr,1,function(x) lmedian(x,Y,X))
trellis.par.set("axis.line",list(col=NA,lty=1,lwd=1))
print(
      wireframe(z ~ beta.0 * beta.1, data = gr,
                scales = list(arrows = FALSE),
                drape = TRUE, colorkey = FALSE,
                aspect=c(1,1),
                screen = list(z = 120, x = -60),zoom=1)
      )
```

The outcome is shown in Figure 2.8, left. At first sight this might seem a well-shaped convex function. However a closer look, shown in Figure 2.8, right, unveils its real nature:

```
gr <- expand.grid(beta.0=seq(-0.01,0.01,0.001),
                  beta.1=seq(1.2,3,0.01))

gr$z <- apply(gr[,1:2],1,function(x) lmedian(x,Y,X))
trellis.par.set("axis.line",list(col=NA,lty=1,lwd=1))
print(
      wireframe(z ~ beta.0 * beta.1, data = gr,
                scales = list(arrows = FALSE),
```

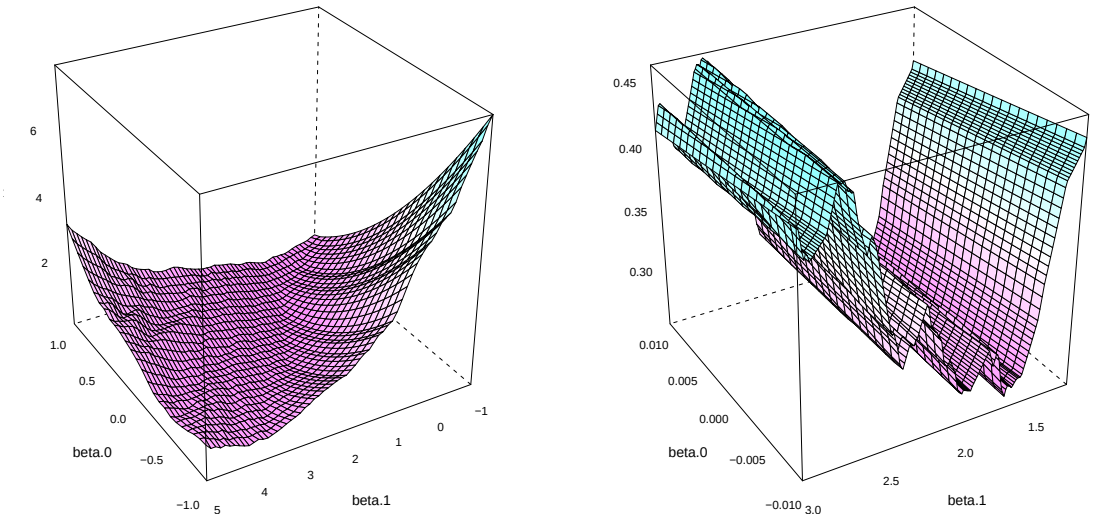


Figure 2.8: Surface of the function to minimize on the two parameters `beta.0` and `beta.1`.

```
                drape = TRUE, colorkey = FALSE,
                aspect=c(1,1),
                screen = list(z = 120, x = -60),zoom=1)
        )
```

The presence of local optima becomes evident and together with it the difficulty of solving this optimization problem.

We must therefore resort to numerical methods and heuristics. The R method `optim` offers an implementation of the Nelder-Mead [12] method and Simulated Annealing for continuous optimization. The Nelder-Mead is the default in the function `optim` provided by R. It requires a starting value for the parameters, the function to optimize and its parameters.

```
> beta.init <- matrix(c(0,0))
> O <- optim(c(beta=beta.init), lmedian, Response=Y, Design=X, control=list(trace=6))

Nelder-Mead direct search function minimizer
function value for initial parameters = 1.730979
  Scaled convergence tolerance is 2.57936e-08
Stepsize computed as 0.100000
BUILD              3 1.730979 1.477877
EXTENSION          5 1.599763 1.184003
EXTENSION          7 1.477877 0.856270
EXTENSION          9 1.184003 0.381746
REFLECTION        11 0.856270 0.306620
.....
REFLECTION       115 0.247260 0.247260
HI-REDUCTION     117 0.247260 0.247260
Exiting from Nelder Mead minimizer
    119 function evaluations used
> O
$par
  beta1   beta2
```

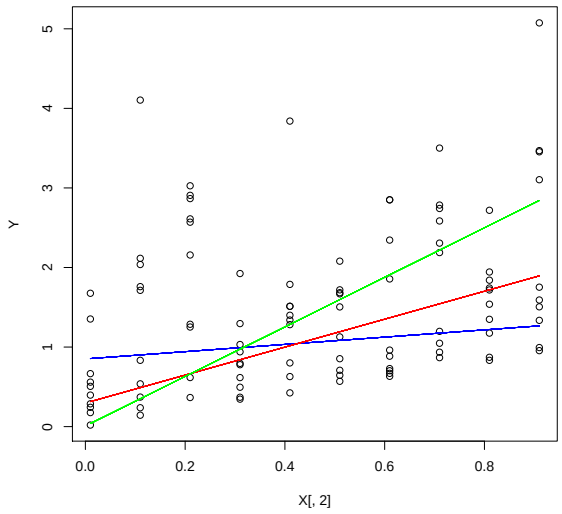


Figure 2.9: Least median of squares regression. Colors refer to different starting points: blue line refers to $\boldsymbol{\beta}_0 = [0, 0]^T$ with $z(\boldsymbol{\beta}) = 0.24726$, red to $\boldsymbol{\beta}_0 = [0, 2]^T$ with $z(\boldsymbol{\beta}) = 0.24581$ and green to $\boldsymbol{\beta}_0 = [0, 3]^T$ with $z(\boldsymbol{\beta}) = 0.39371$. Finally the black line is the supposed optimum found by one of the random restart strategies illustrated in the next section.

```
0.85100 0.45652

$value
[1] 0.24726

$counts
function gradient
     119       NA

$convergence
[1] 0

$message
NULL
```

```
> plot(X[,2],Y)
> lines( X[,2], O$par[1]+O$par[2]*X[,2], lty=2, col="blue")
```

Repeating the optimization from different starting points we obtain a value of 0.24581 after 107 function evaluations starting from $\boldsymbol{\beta}_0 = [0, 2]^T$ and a value of 0.39371 after 81 function evaluations starting from $\boldsymbol{\beta}_0 = [0, 3]^T$.

Optimization methods and algorithms In the previous section we saw that the initial solution has a strong impact on the final result of the Nelder-Mead algorithm. In this section we want to study the effect of random restart on this algorithm. Each descent starts from a solution randomly chosen in the square $[-2, 3] \times [0, 100]$. We will compare the algorithm without restart and two versions with 100 random restarts. In the first version start points

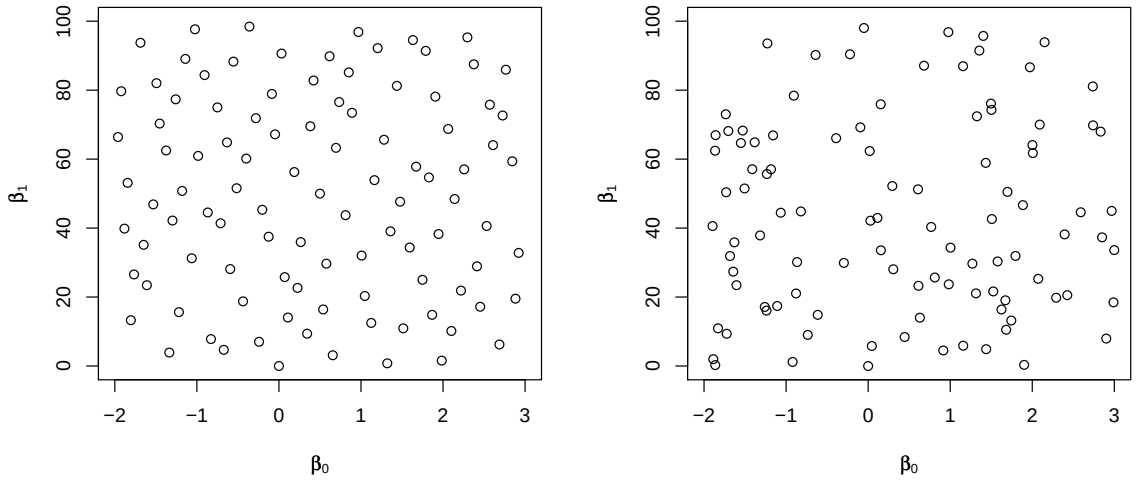


Figure 2.10: Comparison of a sample of 100 points in the square $[-2, 3] \times [0, 100]$ by quasi-Monte Carlo method (left) and uniform distribution (right). (For the use of mathematical symbols in R plots see `?plotmath`.)

are chosen uniformly at random. In the second version a quasi Monte Carlo method is used. Quasi-Monte Carlo methods are based on low-discrepancy sequences. We use the algorithm and the Fortran implementation by Niederreiter [5]. The difference between the two sampling methods is illustrated in Figure 2.10.

```
> no.restart <- function(seed = 1) {
      set.seed(seed)
      pc <- matrix(c(runif(1, -2, 3), runif(1, 0, 100)))
      O <- optim(c(beta = pc), lmedian, Response = Y, Design = X,
          control = list(trace = 0))
      return(O$value)
  }
> restart.uniform <- function(N, seed = 1) {
      set.seed(seed)
      values <- array(dim = c(N))
      for (i in 1:N) {
          pc <- matrix(c(runif(1, -2, 3), runif(1, 0, 100)))
          O <- optim(c(beta = pc), lmedian, Response = Y, Design = X,
              control = list(trace = 0))
          values[i] <- O$value
      }
      return(min(values))
  }
> restart.qmc <- function(N, seed = 1) {
      mylib <- file.path(".", "LowDiscrepancy", paste("niederreiter_prb",
          .Platform$dynlib.ext, sep = ""))
      dyn.load(mylib)
```

```
    values <- array(dim = c(N))
    s <- seed
    f <- vector(mode = "double", length = 2)
    for (i in 1:N) {
        qmc <- .Fortran("generate", as.integer(2), as.integer(2),
            s = as.integer(s), f = as.double(f))
        s <- qmc$s
        pc <- matrix(c(-2 + qmc$f[1] * (3 - (-2)), 0 + qmc$f[2] *
            (100 - 0)))
        O <- optim(c(beta = pc), lmedian, Response = Y, Design = X,
            control = list(trace = 0))
        values[i] <- O$value
    }
    return(min(values))
}
```

2.2.1 Comparison on one instance

Univariate analysis We compare these three algorithms on the basis of the solution cost. We first collect the data:

```
D <- data.frame(no.restart=sapply(1:30,function(x) no.restart(x)),
                uniform.restart=sapply(1:30,function(x) uniform.restart(10,x)),
                qmc.restart=sapply(1:30,function(x) qmc.restart(10,x)))
> D
   no.restart uniform.restart qmc.restart
1      1.7857          1.7857      1.7852
2      1.7857          1.7855      1.7852
3      1.8247          1.7852      1.7852
.....
29     1.7846          1.7846      1.7852
30     1.8069          1.7857      1.7852
> str(D)
'data.frame':        30 obs. of  3 variables:
 $ no.restart     : num  1.79 1.79 1.82 1.98 1.80 ...
 $ uniform.restart: num  1.79 1.79 1.79 1.78 1.79 ...
 $ qmc.restart    : num  1.79 1.79 1.79 1.79 1.79 ...
```

It is good practice to inspect the data and to check that there are no strange results. We can do this drawing some plots. The following lines show how to produce histograms, empirical cumulative distribution functions and boxplots (see Figure 2.1 and Figure 2.11).

```
> hist(D$no.restart,
       breaks=seq(-0.01+min(D$no.restart),max(D$no.restart+0.01),0.01),
       probability=TRUE,panel.first=grid())
> lines(density(D$no.restart),col="blue")
> lines(density(D$no.restart,adjust=3),lty=2,col="blue")
> plot.ecdf(D$no.restart,panel.first=grid(),do.points=FALSE,verticals=TRUE)
> boxplot(D$no.restart,horizontal=TRUE,names=c("no.restart"),las=1)
```

Repeating the inspection for the other two algorithms we see that the `qmc.restart` procedure returns always the same value. Indeed the algorithm of Niederreiter [5] is deterministic if the same number of samples is required.

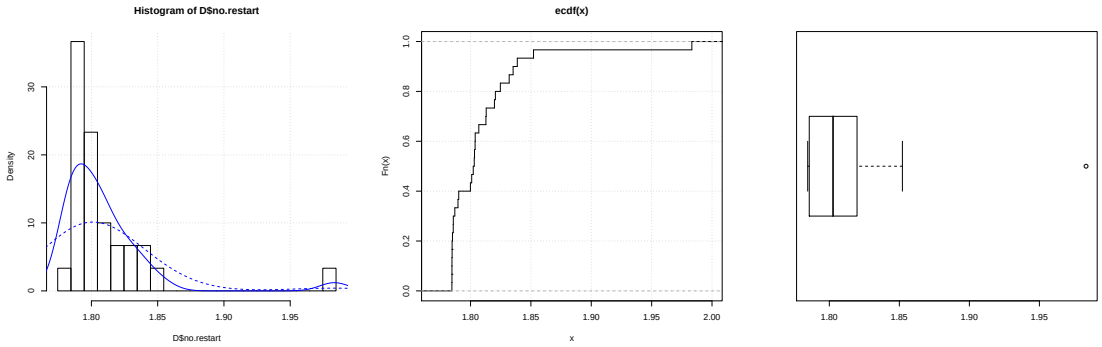


Figure 2.11: Three different ways to look at the distribution of sampled data.

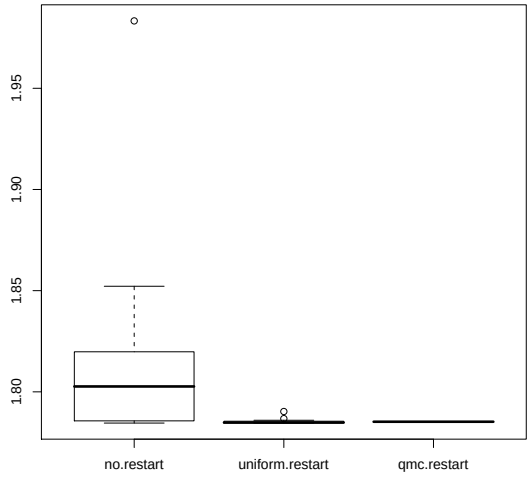


Figure 2.12: Boxplot comparing the three algorithms.

Boxplots can be used for the comparison of the three algorithms. The result of the following line is reported in Figure 2.12.

```
> boxplot(D)
```

It clearly arises that `qmc.restart` and `uniform.restart` outperform `no.restart` while in the comparison between `qmc.restart` and `uniform.restart` it is more difficult to distinguish a winner. It is also relevant to have a closer insight at some statistics of the numerical data:

```
> summary(D)
   no.restart   uniform.restart  qmc.restart
 Min.   :1.78   Min.   :1.78    Min.   :1.79
 1st Qu.:1.79   1st Qu.:1.78    1st Qu.:1.79
 Median :1.80   Median :1.78    Median :1.79
 Mean   :1.81   Mean   :1.79    Mean   :1.79
 3rd Qu.:1.82   3rd Qu.:1.79    3rd Qu.:1.79
 Max.   :1.98   Max.   :1.79    Max.   :1.79
```

The conclusion is that `qmc.restart` and `uniform.restart` perform about the same on this specific instance.

Bivariate analysis In the previous analysis we have not considered the computation time. In the following we take that also into account.

```
D2 <- data.frame()
for ( i in 1:30 )
  {
    T <- system.time(S <- no.restart(i))
    D2 <- rbind(D2,data.frame(algorithm="no.restart",trial=i,quality=S,time=T[[1]]))
    T <- system.time(S <- uniform.restart(10,i))
    D2 <- rbind(D2,data.frame(algorithm="uniform.restart",trial=i,quality=S,time=T[[1]]))
    T <- system.time(S <- qmc.restart(10,i))
    D2 <- rbind(D2,data.frame(algorithm="qmc.restart",trial=i,quality=S,time=T[[1]]))
  }
```

```
> D2
         algorithm trial quality  time
1       no.restart     1  1.7857 0.024
2  uniform.restart     1  1.7857 0.292
3      qmc.restart     1  1.7852 0.328
.....
88      no.restart    30  1.8069 0.028
89 uniform.restart    30  1.7857 0.352
90     qmc.restart    30  1.7852 0.312
> str(D2)
'data.frame':        90 obs. of  4 variables:
 $ algorithm: Factor w/ 3 levels "no.restart","uniform.restart",..: 1 2 3 1 2 3 1 2 3 1 ...
 $ trial    : int  1 1 1 2 2 2 3 3 3 4 ...
 $ quality  : num  1.79 1.79 1.79 1.79 1.79 ...
 $ time     : num  0.024 0.292 0.328 0.028 0.344 ...
```

This time we collected the data in the data frame structure in a long format, contrary to `D` that was in wide format. It is however easy to go from one form to the other by means of the functions `stack`, `unstack` and `reshape`, for example,

```
> unstack(D2[,c(3,1)])
```

```
   no.restart uniform.restart qmc.restart
1      1.7857          1.7857      1.7852
2      1.7857          1.7855      1.7852
.....
29     1.7846          1.7846      1.7852
30     1.8069          1.7857      1.7852
```

The second format is however more convenient for multivariate analysis, in which the responses for different variables are reported in different columns. In our case the two variables are time and quality. Moreover, this format allows us to use the methods from the package `lattice` which is specific for multivariate data visualization. For example,

```
> library(lattice)
> print(histogram(~quality | factor(algorithm), data = D2, layout = c(1,
     3)))
```

produces the output in Figure 2.13.

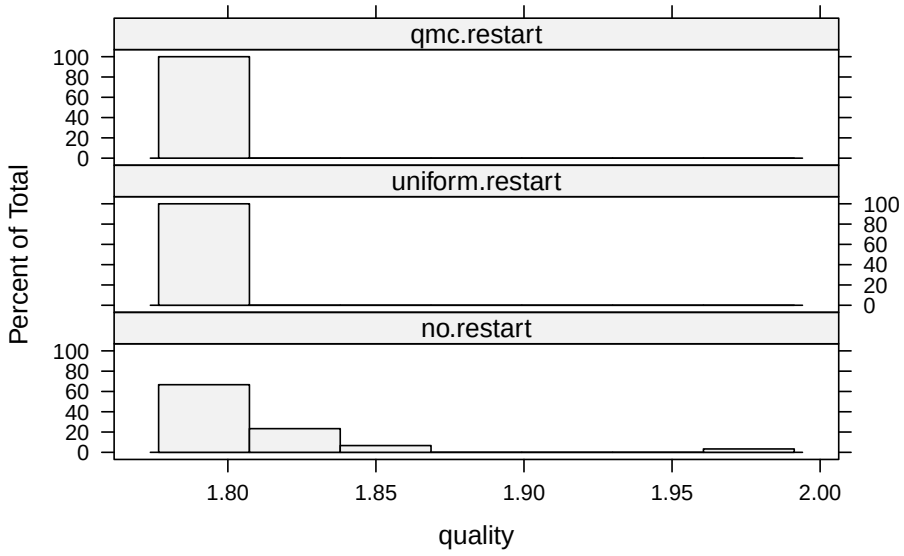


Figure 2.13: A conditional histogram. The different panels represent differnt algorithms.

Back to our comparison, we can plot the results on a time-quality plot. We may do this by plotting all the data or by summarizing them by means of the median. See Figure 2.14.

```
> print(xyplot(quality ~ time, groups = algorithm, data = D2))
```

```
> A <- aggregate(D2$quality, list(algorithm = D2$algorithm), median)
> B <- aggregate(D2$time, list(algorithm = D2$algorithm), median)
> D2s <- merge(A, B, by = "algorithm")
> names(D2s) <- c("algorithm", "quality", "time")
> D2s
```

```
       algorithm quality  time
1     no.restart    1.80 0.028
2    qmc.restart    1.79 0.304
3 uniform.restart    1.78 0.312
```

```
> print(xyplot(quality ~ time, data = D2s, groups = algorithm,
     scales = list(relation = "free", y = list(rot = 0, log = FALSE)),
     panel = function(x, y, subscripts, groups) {
         panel.grid(h = -1, v = -1, lty = 3)
         ltext(x = x, y = y, label = groups[subscripts], cex = 0.8,
             fontfamily = "Helvetica")
     }, ylab = "Mean time", xlab = "Mean quality"))
```

From Figure 2.14, right, we may conclude that since the quality of the solutions returned is slightly better for the `uniform.restart` and the computation time is about the same, `uniform.restart` seems a better algorithm in this experiment.

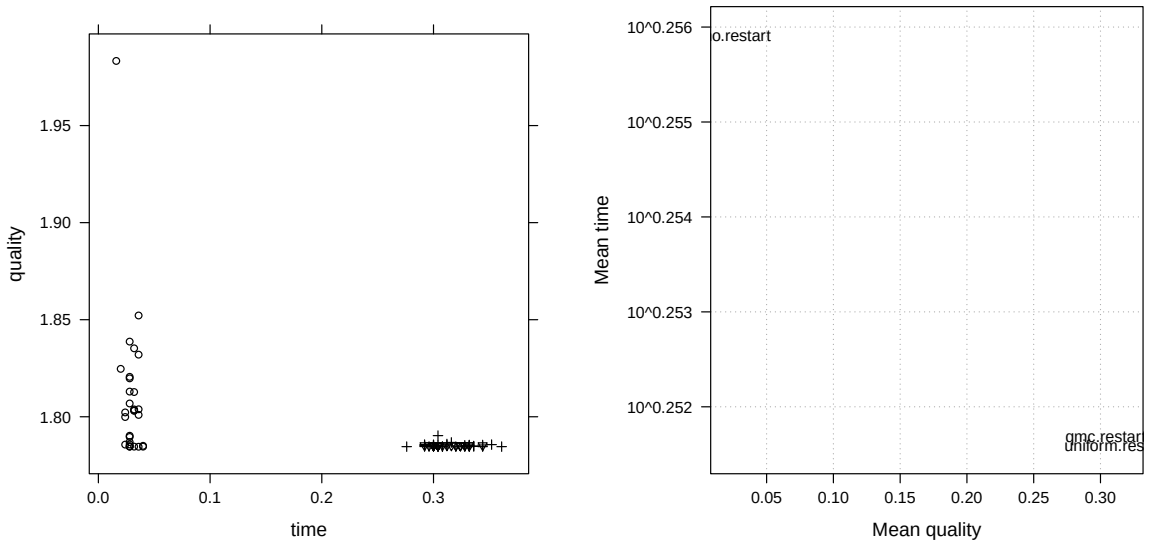


Figure 2.14: Time-quality plot of the results of 30 runs of the three algorithms. On the left, all data are plotted and algorithms differ by the sign of the points. On the right, the median results are used as coordinate for the algorithm's label.

2.2.2 Comparison on a set of instances

So far we only evaluated our three algorithms on a single instance. This might be misleading if we want to take a conclusion on which algorithm is the best for instances of a certain type because the instance used for comparison might not be a good representative of the whole population. In this section we will focus on the comparison over a set of instances sampled from the distribution of instances of a certain type.

Let's first generate the instances and store them in a list

```
generate.instance <- function(seed)
  {
    set.seed(seed)
    N <- 1000
    X <- array(dim=c(N,2))
    X[,1] <- 1
    X[,2] <- 10*seq(0.01,1,by=0.1)#rnorm(N,5,10)#seq(0.01,1,by=0.1)
    Y <- X %*% matrix(c(0,1)) + matrix(rnorm(N,0,2))
    return(list(X=X,Y=Y))
  }

instances <- list()
for (i in 1:30)
  {
    instances[[i]] <- generate.instance(i)
  }
```

and then let's collect the results

```
D3 <- data.frame()
```

```
for ( i in 1:30 )
  {
    X <<- instances[[i]]$X
    Y <<- instances[[i]]$Y
    T <- system.time(S <- no.restart(i))
    D3 <- rbind(D3,data.frame(algorithm="no.restart",instance=i,quality=S,time=T[[1]]))
    T <- system.time(S <- uniform.restart(10,i))
    D3 <- rbind(D3,data.frame(algorithm="uniform.restart",instance=i,quality=S,time=T[[1]]))
    T <- system.time(S <- qmc.restart(10,i))
    D3 <- rbind(D3,data.frame(algorithm="qmc.restart",instance=i,quality=S,time=T[[1]]))
  }
```

This time we present the analysis of the two variates time and quality using the conditional plots of `lattice`. We first need to reshape the data as follows

```
> str(D3)
'data.frame':        90 obs. of  4 variables:
 $ algorithm: Factor w/ 3 levels "no.restart","uniform.restart",..: 1 2 3 1 2 3 1 2 3 1 ...
 $ instance : int  1 1 1 2 2 2 3 3 3 4 ...
 $ quality  : num  1.79 1.79 1.79 1.9 1.83 ...
 $ time     : num  0.032 0.312 0.332 0.044 0.364 ...
> D3r <- reshape(D3, idvar = "id", timevar = "response", varying = list(c("time",
      "quality")), direction = "long", times = c("time", "quality"),
      v.names = "values")
> str(D3r)
'data.frame':        180 obs. of  5 variables:
 $ algorithm: Factor w/ 3 levels "no.restart","uniform.restart",..: 1 2 3 1 2 3 1 2 3 1 ...
 $ instance : int  1 1 1 2 2 2 3 3 3 4 ...
 $ response : chr  "time" "time" "time" "time" ...
 $ values   : num  0.032 0.312 0.332 0.044 0.364 ...
 $ id       : int  1 2 3 4 5 6 7 8 9 10 ...
 - attr(*, "reshapeLong")=List of 4
  ..$ varying:List of 1
  .. ..$ : chr  "time" "quality"
  ..$ v.names: chr "values"
  ..$ idvar  : chr "id"
  ..$ timevar: chr "response"
```

and then produce the plot of Figure 2.15 with the command

```
> print(bwplot(algorithm ~ values | response, data = D3r, layout = c(1,
      2), scales = "free"))
```

Looking at the quality response we see that apparently there are no significant differences. This contrasts with the conclusion of Figure 2.12 and should make us suspicious. Indeed, if no transformation of data is applied the different scale of the instances is likely to hide differences among the performance of the algorithms. We redo the analysis by using rank transformation of the results within each instance (hence results are mapped in the interval $[1, 3]$ because we have three algorithms and one run per instance). The result is illustrated in Figure 2.16.

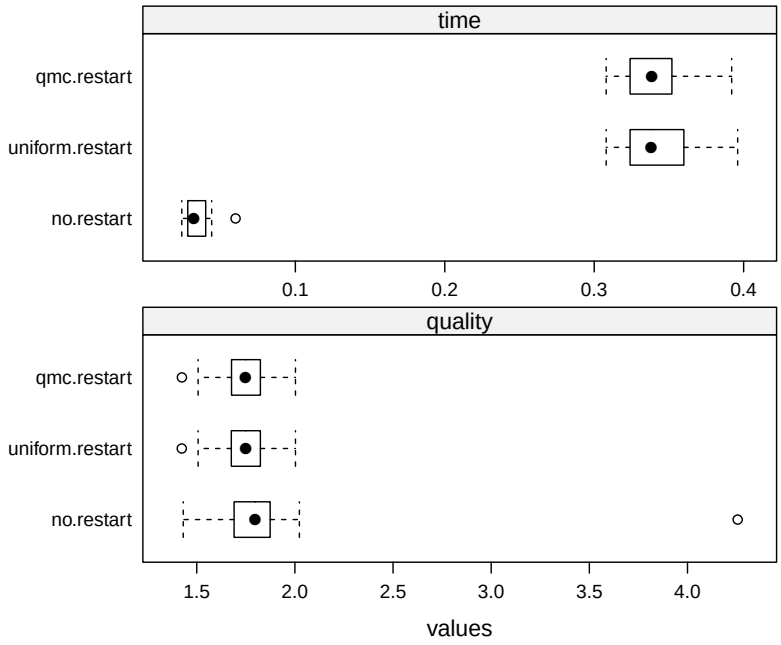


Figure 2.15: Boxplots of time and quality responses of the three algorithms over 30 instances of the least median of squares regression problem. Times are expressed in seconds and refer to a Intel(R) Core(TM)2 CPU at 1.86GHz.

```
> D3$rank <- unsplit(tapply(D3$quality, D3$instance, rank, ties.method = "average"),
     D3$instance)
> F <- aggregate(D3$rank, list(D3$algorithm), mean)
> l <- F[order(F$x), 1]
> print(bwplot(factor(algorithm, levels = l) ~ rank, data = D3))
```

The conclusion we should draw from Figure 2.16 is opposite to what we had before. When we consider a set of instances the algorithm `qmc.restart` shows best performance.

It might be also worth reporting a table with numerical results. For LaTeX this can be easily achieved with the package `xtable`:

```
> D3t <- unstack(D3[, c(3, 1)])
> library(xtable, lib.loc = "/home/marco/.R/library")
> xtable(D3t[1:5, ])
```

```
% latex table generated in R 2.12.0 by xtable 1.5-6 package
% Sat Jan 22 14:54:57 2011
\begin{table}[ht]
\begin{center}
\begin{tabular}{rrrr}
  \hline
 & no.restart & uniform.restart & qmc.restart \\
  \hline
1 & 1.79 & 1.79 & 1.79 \\
  2 & 1.90 & 1.83 & 1.83 \\
```

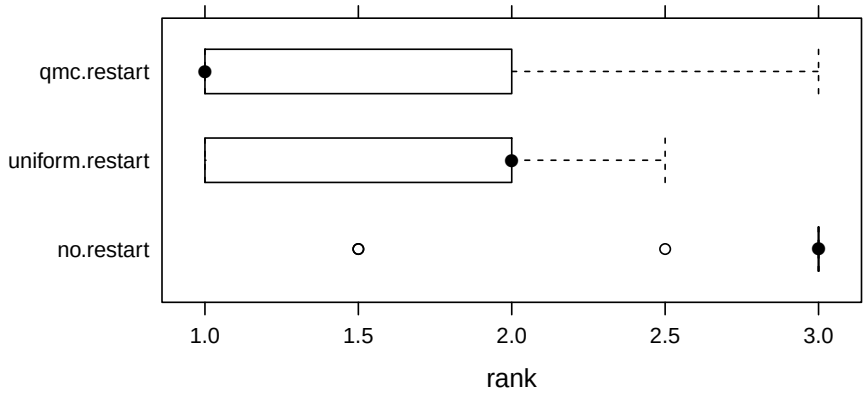


Figure 2.16: The comparison on a set of instances after rank transformation.

```
  3 & 1.81 & 1.78 & 1.78 \\
  4 & 1.67 & 1.67 & 1.67 \\
  5 & 4.26 & 1.72 & 1.72 \\
   \hline
\end{tabular}
\end{center}
\end{table}
```

	no.restart	uniform.restart	qmc.restart
1	1.79	1.79	1.79
2	1.90	1.83	1.83
3	1.81	1.78	1.78
4	1.67	1.67	1.67
5	4.26	1.72	1.72

Chapter 3

The Racing Method

The racing method is an automatic sequential testing procedure for the comparison, configuration or tuning of algorithms. This chapter describes the use of the method in R. For the theory behind the method, the reader is referred to [1, 2, 4].

3.1 Set up

We assume that the R package `race`[1] developed by M.Birattari [1, 2, 3] is installed (see R documentation on how to install packages.

```
> install.packages("race")
```

The package `race` consists of a library and a *wrapper file*. The library defines the engine of the race, that is, the function `race`. The wrapper file defines all details concerning the specific experiment that must be undertaken. Only this latter file must be edited to specify the details of the experiment.

It is worth emphasizing that the function `race` only implements the race in an *unreplicated design*!

It is possible to execute the race in a distributed computing environment. In this case the machine where the experiments are launched is identified as the master and the other machines as the slaves. Running the race in a distributed environment requires to have installed `pvm` and `Rpvm` in the master and the slave machines.

The first step in setting up the race is retrieving the wrapper file from the library of the R installation:

```
> file.path(system.file(package = "race"), "examples", "example-wrapper.R")
[1] "/examples/example-wrapper.R"
```

The library example implements the tuning of a neural network algorithm using a cross validation methodology that divides data in training data and testing data. It might be confusing for the typical cases of optimization. Hence, alternatively, one can use the examples available at www.imada.sdu.dk/~marco/Teaching/Files/.

[1] http://cran.r-project.org/web/packages/race/

The wrapper file contains the following functions that have to be adapted to the specific case:

- `race.init` the initialization function
- `race.wrapper` the interface between `race` and the external optimization program. It is the function called by the library to launch one single run of an algorithm on a single instance.
- `race.info` for reporting purposes
- `race.describe` for reporting purposes.

The two main functions to look at are `race.init` and `race.wrapper`. `race.init` is needed to define all data of the race. In particular the instances and the configurations to test. Both can be either read from an external file or encoded in R:

```
> instances <- scan(file = "u-1000-10-1000.txt", what = as.character(0),
     skip = 0, quiet = TRUE)
> n <- length(instances)
> candidates <- as.data.frame(rbind(c(label = "300787", path = "300787/src",
     command = "Driver -tt 30 -ch 2 -ls 3"), c(label = "100884",
     path = "100884/", command = "dm811e/Forced")))
```

Alternatively, candidates can be generated by a full factorial design by crossing several factors. For example:

```
> candidates <- expand.grid(solver = c("CH", "LS"), alpha = c(0.5,
     1.5), idle = c(100, 300))
> candidates[1:5, ]
  solver alpha idle
1     CH   0.5  100
2     LS   0.5  100
3     CH   1.5  100
4     LS   1.5  100
5     CH   0.5  300
```

The output of the `race.init` function is a list of data. in particular: `no.tasks` is the maximum number of stages in the race. Clearly, in an unreplicated design, this number corresponds to the number of instances. The number of subtasks should be always left to its default value which is 1. Finally, `smpl` is a vector of randomly shuffled integers `1:n` that serves as a mask for deciding the order of examination of the instances.

```
> return(list(name = class, no.candidates = nrow(candidates), no.tasks = n,
     no.subtasks = 1, wd = wd, smpl = smpl, instances = instances,
     candidates = candidates))
```

The `race.wrapper` function strongly depends on the way the external program has been implemented. The function must return one single value which is the result of the run of the algorithm `candidate` on the instance `smpl[task]`. The simplest way is to let the optimization program return one single value and redirect all the rest. For example, with a C program:

```
> command <- paste(data$candidates[candidate, ]$command, " -i ",
     instance, " -t ", time, " -s ", data$smpl[task], " -o ",
     paste(candidate, task, 1, sep = "-"), " 2>/dev/null", sep = "")
> s <- system(command, intern = TRUE, ignore.stderr = TRUE)
```

It might be wise in a debugging phase to print out the full launch command, for example, with `cat(command)`. Further, it is advisable, when running long experiments, to write the outcome of the run in a log file as soon as this result is retrieved.

When the wrapper file is ready it is advisable to run some tests. For example:

```
> D <- race.init()
> D
> race.wrapper(1, 1, D)
```

If the tests run fine then everything is ready to be launched. The following is an example of launch command:

```
> O <- race("wrapper-race.R", maxExp = 5000, stat.test = c("friedman"),
     conf.level = 0.95, first.test = 5, interactive = TRUE, log.file = "race.log",
     no.slaves = 0)
```

See the race documentation (`?race`) for an explanation of the parameters.

When the race is finished it is possible to plot a profile of what happened by means of the function `plot.race` available from www.imada.sdu.dk/ marco/Teaching/Files/plot.race.R.

```
> source("plot.race.R")
> plot.race(O, "wrapper-file.R")
```

It might be necessary to edit the function for layout adjustments.

3.2 An Example

```
> O <- race(wrapper.file="example-wrapper.R",
+           maxExp=3240, ## multiple of number of candidates
+           stat.test=c("friedman"),
+           conf.level=0.95,
+           first.test=5,
+           interactive=TRUE,
+           #log.file=paste(file,".log",sep=""),
+           no.slaves=0)
Racing methods for the selection of the best
Copyright (C) 2003 Mauro Birattari
This software comes with ABSOLUTELY NO WARRANTY

Race name.....................NM for Least Median of Squares
Number of candidates.....................................162
Number of available tasks.....................................45
Max number of experiments.....................................3240
Statistical test.....................................Friedman test
```

```
Tasks seen before discarding.................................5
Initialization function.....................................ok
Parallel Virtual Machine....................................no

                            Markers:
                               x No test is performed.
                               - The test is performed and
                                 some candidates are discarded.
                               = The test is performed but
                                 no candidate is discarded.


+-+-----------+-----------+-----------+-----------+-----------+
| |       Task|      Alive|       Best|  Mean best| Exp so far|
+-+-----------+-----------+-----------+-----------+-----------+
|x|          1|        162|         81|  2.869e-05|        162|
|x|          2|        162|        140|  2.761e-05|        324|
|x|          3|        162|         86|  2.607e-05|        486|
|x|          4|        162|        140|  2.887e-05|        648|
|-|          5|         52|        140|  3.109e-05|        810|
|=|          6|         52|         34|  3.892e-05|        862|
...
|=|         42|         13|         32|   4.76e-05|       1703|
|=|         43|         13|         32|  4.684e-05|       1716|
|=|         44|         13|         32|  4.616e-05|       1729|
|=|         45|         13|         32|   4.55e-05|       1742|
+-+-----------+-----------+-----------+-----------+-----------+

Selected candidate:          32        mean value:   4.55e-05

Description of the selected candidate:
   initial.method max.reinforce alpha beta gamma                     label
32   quasi-random             1   1.5  0.5   1.5 quasi-random-1-1.5-0.5-1.5
```

The race finished after all instances available (45) have been used without a single winner. At the end 13 configurations were still alive, that is, not yet found significantly different. The race returns however a winner decided on the basis of the median rank of its results.

Figure 4.1 visualizes the process. The grey area represents the aggregate computation time effectively used by the race. This corresponds to 1742 runs of the algorithms with the best of them tested on 45 instances. Given the 162 initial algorithm configurations, experimenting all of them on the 45 instance would have required 3240 runs corresponding to the whole area covered by the graph. Alternatively, with the same computation time it would have been possible to experiment only on 20 instances.

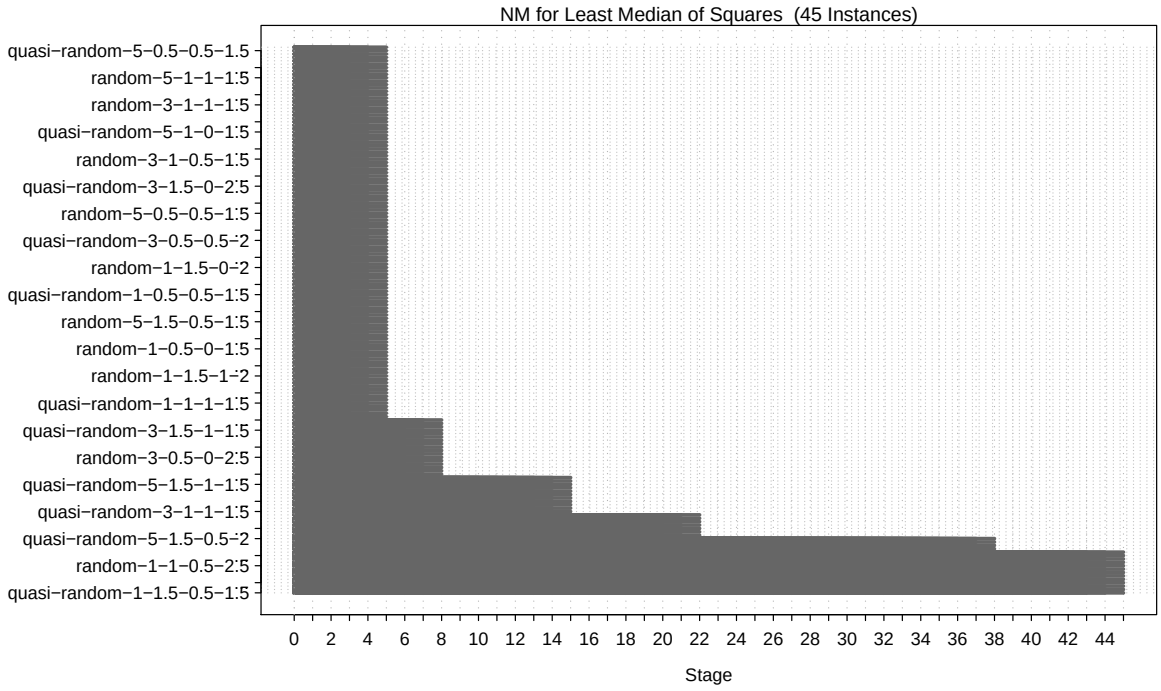


Figure 3.1: Graphical view of the race described in the text.

Chapter 4

ANOVA and Regression Trees

4.1 Regression Tree on Random Restart Nelder-Mead

We test random restart Nelder-Mead method and study five different parameters of this algorithm.

Factor	Type	Levels
initial.method	Categorical	random, quasi-random
max.reinforce	Integer	1,3,5
alpha	Real	0.5,1,1.5
beta	Real	0,0.5,1
gamma	Real	1.5,2,2.5

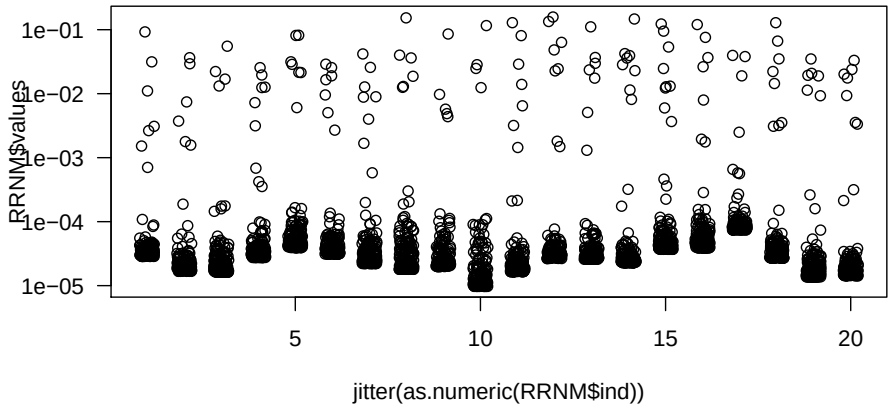
Table 4.1: Random restart Nelder-Mead.

The method for generating initial solutions, uniform random or quasi Monte Carlo Method, the times the Nelder-Mead is run in sequence before restarting. Finally the three scaling parameters for the Nelder-Mead method: 'alpha' is the reflection factor, 'beta' the contraction factor and 'gamma' the expansion factor. The levels tested for each factors are reported in Table 4.1.

We run a full factorial experiment blocking on 20 instances. This yields $162 \times 20 = 3240$ runs of our algorithms. Each run performs 500 restarts. We load the data collected in `RRNM`. Since `max.reinforce` can assume only integer values, we transform it in an ordered factor. We then plot the distribution of results per instance, using a jittering of the data in order to visualize better the results. This is useful to gain insight on how the data are distributed within the instances. We see that the distribution of results is clearly bounded from below indicating that the optimal solutions are very likely found for these instances. We should not however to see normality in these plots because the factors may be responsible for differences in results.

```
> load("Data/rrnm-ff.RData")
> str(RRNM)
```

```
'data.frame':        3240 obs. of  7 variables:
 $ s.initial.method: Factor w/ 2 levels "random","quasi-random": 1 2 1 2 1 2 1 2 1 2 ...
 $ s.max.reinforce : num  1 1 3 3 5 5 1 1 3 3 ...
```



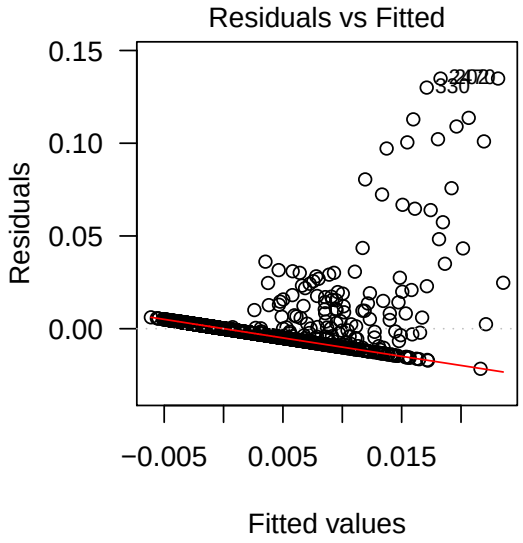

```
 $ s.alpha         : num  0.5 0.5 0.5 0.5 0.5 0.5 1 1 1 1 ...
 $ s.beta          : num  0 0 0 0 0 0 0 0 0 0 ...
 $ s.gamma         : num  1.5 1.5 1.5 1.5 1.5 1.5 1.5 1.5 1.5 1.5 ...
 $ values          : num  3.20e-05 8.43e-05 4.02e-05 9.24e-02 1.08e-04 ...
 $ ind             : Factor w/ 20 levels "V1","V10","V11",..: 1 1 1 1 1 1 1 1 1 1 ...
```

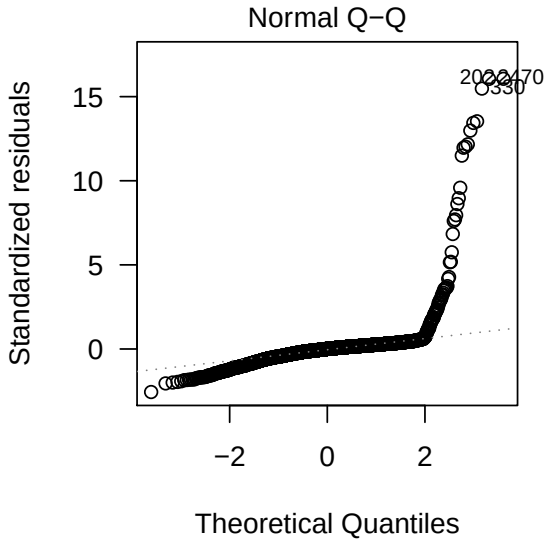
```
> RRNM$s.max.reinforce <- ordered(RRNM$s.max.reinforce)
> plot(jitter(as.numeric(RRNM$ind)), RRNM$values, log = "y", type = "p")
```

Trying to apply ANOVA to this design we encounter two main difficulties. First diagnostic plots on the linear model with terms up to the second order interactions violate saliently the assumptions. Secondly with so many factors we are easily overwhelmed by information with ANOVA.

A way to work around the first of these issues is by applying some transformation to the data. The Box Cox transformation from the package `MASS` finds the best transformation among a family of transformations by maximizing the likelihood. In our case it is still not enough. We then use jointly this transformation and the rank transformation obtaining something mildly acceptable.

```
> l <- lm(values ~ (s.initial.method + s.max.reinforce + s.alpha +
     s.beta + s.gamma + ind)^2 - 1, data = RRNM)
> par(mfrow = c(1, 2))
> plot(l, which = c(1, 2))
```






```
> RRNM$rankval <- unsplit(tapply(RRNM$values, RRNM$ind, rank, ties.method = "average"),
      RRNM$ind)
> l <- lm((rankval^(0.8) - 1)/0.8 ~ (s.initial.method + s.max.reinforce +
      s.alpha + s.beta + s.gamma)^2 - 1, data = RRNM)
> summary(l)

Call:
lm(formula = (rankval^(0.8) - 1)/0.8 ~ (s.initial.method + s.max.reinforce +
    s.alpha + s.beta + s.gamma)^2 - 1, data = RRNM)
Residuals:
   Min     1Q Median     3Q    Max
-48.70 -13.63   2.28  14.97  48.38

Coefficients:
                                                Estimate Std. Error t value
s.initial.methodrandom                            69.943      5.174   13.52
s.initial.methodquasi-random                      96.894      5.174   18.73
s.max.reinforce.L                                  1.277      3.255    0.39
s.max.reinforce.Q                                 -2.557      3.255   -0.79
s.alpha                                          -24.231      4.170   -5.81
s.beta                                            14.112      4.498    3.14
s.gamma                                          -16.272      2.451   -6.64
s.initial.methodquasi-random:s.max.reinforce.L    -3.438      1.125   -3.06
s.initial.methodquasi-random:s.max.reinforce.Q     1.089      1.125    0.97
s.initial.methodquasi-random:s.alpha              -7.214      1.590   -4.54
s.initial.methodquasi-random:s.beta               -9.134      1.590   -5.74
s.initial.methodquasi-random:s.gamma              -9.681      1.590   -6.09
s.max.reinforce.L:s.alpha                          0.707      1.377    0.51
s.max.reinforce.Q:s.alpha                          1.486      1.377    1.08
s.max.reinforce.L:s.beta                          -0.289      1.377   -0.21
s.max.reinforce.Q:s.beta                           2.721      1.377    1.98
s.max.reinforce.L:s.gamma                          0.905      1.377    0.66
s.max.reinforce.Q:s.gamma                          0.430      1.377    0.31
```

```
s.alpha:s.beta                                    -19.977      1.948  -10.26
s.alpha:s.gamma                                    15.810      1.948    8.12
s.beta:s.gamma                                     -0.118      1.948   -0.06
                                                 Pr(>|t|)
s.initial.methodrandom                            < 2e-16 ***
s.initial.methodquasi-random                      < 2e-16 ***
s.max.reinforce.L                                  0.6948
s.max.reinforce.Q                                  0.4321
s.alpha                                           6.8e-09 ***
s.beta                                             0.0017 **
s.gamma                                           3.7e-11 ***
s.initial.methodquasi-random:s.max.reinforce.L     0.0023 **
s.initial.methodquasi-random:s.max.reinforce.Q     0.3329
s.initial.methodquasi-random:s.alpha              5.9e-06 ***
s.initial.methodquasi-random:s.beta               1.0e-08 ***
s.initial.methodquasi-random:s.gamma              1.3e-09 ***
s.max.reinforce.L:s.alpha                          0.6076
s.max.reinforce.Q:s.alpha                          0.2806
s.max.reinforce.L:s.beta                           0.8336
s.max.reinforce.Q:s.beta                           0.0483 *
s.max.reinforce.L:s.gamma                          0.5113
s.max.reinforce.Q:s.gamma                          0.7549
s.alpha:s.beta                                    < 2e-16 ***
s.alpha:s.gamma                                   6.7e-16 ***
s.beta:s.gamma                                     0.9519
---
Signif. codes:  0 âĂŸ***âĂŹ 0.001 âĂŸ**âĂŹ 0.01 âĂŸ*âĂŹ 0.05 âĂŸ.âĂŹ 0.1 âĂŸ âĂŹ 1

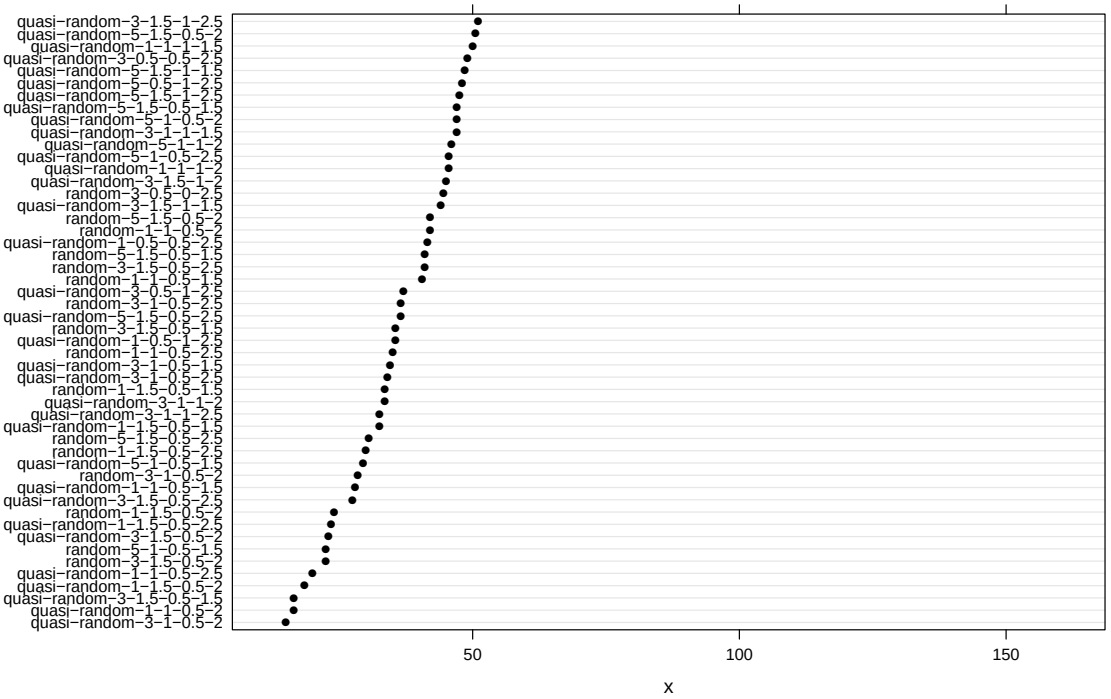
Residual standard error: 18.5 on 3219 degrees of freedom
Multiple R-squared: 0.828,        Adjusted R-squared: 0.827
F-statistic:  740 on 21 and 3219 DF,  p-value: <2e-16
```

The results of this model are however overwhelming, almost every factor and interaction are significant, with the only exception of `max.reinforce`. A dotplot of median rank results over the instances might help us to get an insight. Even from this visualization it is evident that there is no clear pattern arising.

```
> RRNM$label <- paste(RRNM$s.initial.method, RRNM$s.max.reinforce,
      RRNM$s.alpha, RRNM$s.beta, RRNM$s.gamma, sep = "-")
> K <- aggregate(RRNM$rankval, list(label = RRNM$label), median)
> library(lattice)
> print(dotplot(factor(label, levels = c(label[order(K$x)[1:50]])) ~
      x, data = K))
```

4.2 Regression Tree

```
> Tr <- ctree(rankval ~ (factor(s.initial.method) + ordered(s.max.reinforce) +
      s.alpha + s.beta + s.gamma)^2, data = RRNM)
> plot(Tr, type = "simple")
```



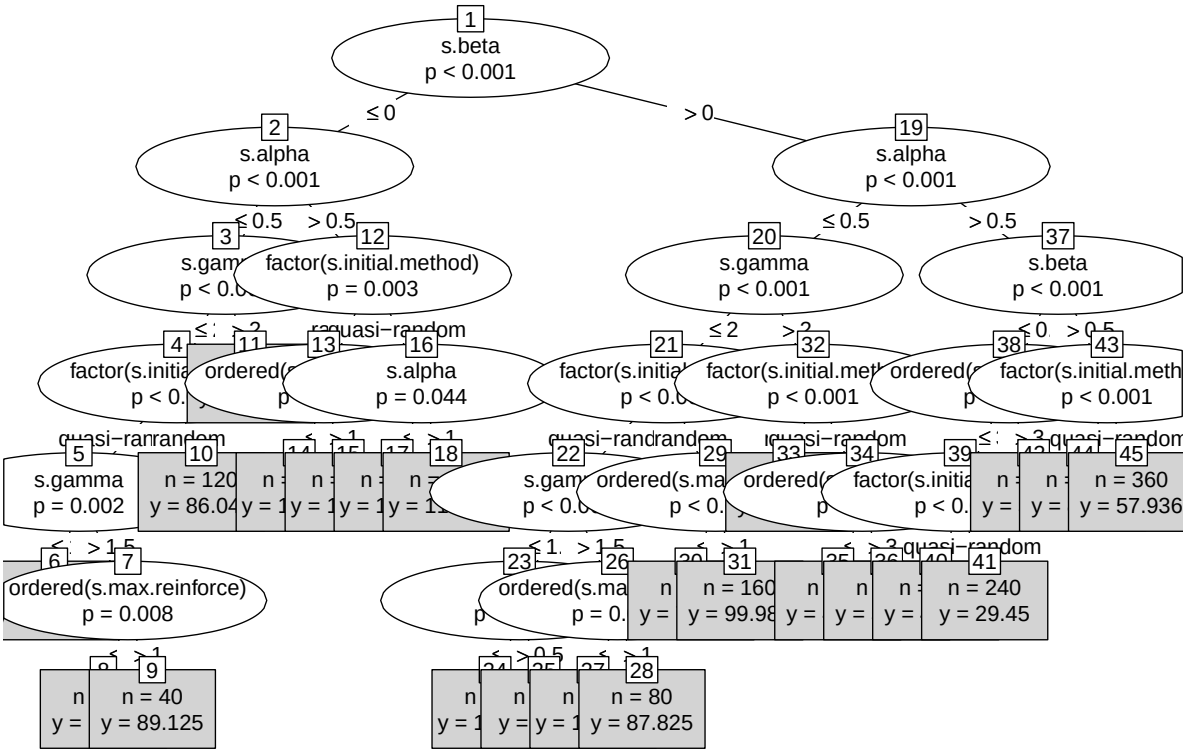
4.3 Race

```
> O <- race(wrapper.file="example-wrapper.R",
+           maxExp=3240, ## multiple of number of candidates
+           stat.test=c("friedman"),
+           conf.level=0.95,
+           first.test=5,
+           interactive=TRUE,
+           #log.file=paste(file,".log",sep=""),
+           no.slaves=0)

Racing methods for the selection of the best
Copyright (C) 2003 Mauro Birattari
This software comes with ABSOLUTELY NO WARRANTY

Race name.....................NM for Least Median of Squares
Number of candidates.....................................162
Number of available tasks.....................................45
Max number of experiments.....................................3240
Statistical test.................................Friedman test
Tasks seen before discarding.................................5
Initialization function.....................................ok
Parallel Virtual Machine....................................no

                          Markers:
                             x No test is performed.
                             - The test is performed and
                               some candidates are discarded.
```



= The test is performed but
no candidate is discarded.

```
+-+-----------+-----------+-----------+-----------+-----------+
| |       Task|      Alive|       Best|  Mean best| Exp so far|
+-+-----------+-----------+-----------+-----------+-----------+
|x|          1|        162|         81|  2.869e-05|        162|
|x|          2|        162|        140|  2.761e-05|        324|
|x|          3|        162|         86|  2.607e-05|        486|
|x|          4|        162|        140|  2.887e-05|        648|
|-|          5|         52|        140|  3.109e-05|        810|
|=|          6|         52|         34|  3.892e-05|        862|
...
|=|         42|         13|         32|   4.76e-05|       1703|
|=|         43|         13|         32|  4.684e-05|       1716|
|=|         44|         13|         32|  4.616e-05|       1729|
|=|         45|         13|         32|   4.55e-05|       1742|
+-+-----------+-----------+-----------+-----------+-----------+

Selected candidate:          32        mean value:   4.55e-05

Description of the selected candidate:
   initial.method max.reinforce alpha beta gamma                       label
32   quasi-random             1   1.5  0.5   1.5 quasi-random-1-1.5-0.5-1.5
```

The race finished after all instances available (45) have been used without a single winner. At the end 13 configurations were still alive, that is, not yet found significantly different.

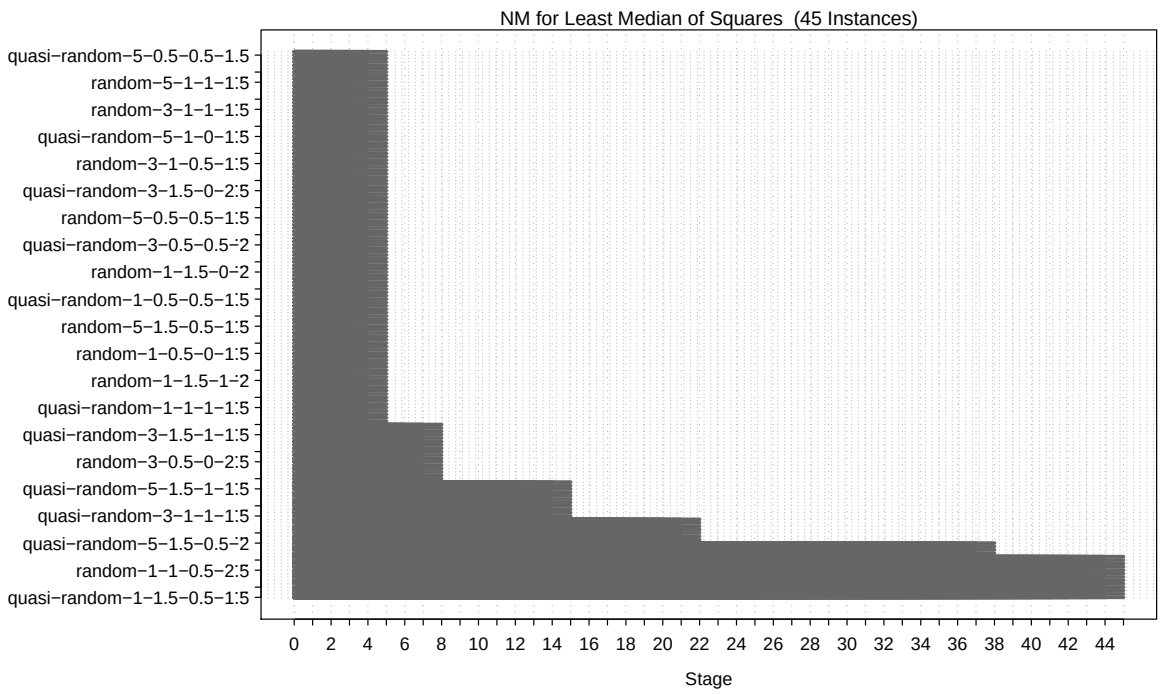


Figure 4.1: Graphical view of the race described in the text.

Looking closer at the results we can see that

The main advantage of the race is that with much less experiments than the full factorial, namely 1742 against 3240 we have been able to test the best configurations on 45 instances rather than 20.

4.4 Screening on Differential Evolution

We use the implementation of DE available from the package DEoptim but we add to the implementation the possibility to stop the run if a number of iterations have elapsed without improvement (idle iterations).

We intend to study in a screening experiment the factors indicated in Table 4.2.

	Factor	Type	Low (−)	High (−)
NP	Number of population members	Int	20	50
F	weighting factor	Real	0	2
CR	Crossover probability from interval	Real	0	1
initial	An initial population	Cath.	Uniform	Quasi MC
strategy	Defines the DE variant used in mutation	Cath.	rand	best
idle iter	Number of idle iterations before terminating	Int.	10	30

Table 4.2: Factors under study for DE.

A full factorial experiment with blocking on 5 instances would amount to $2^6 * 5 = 320$

experiments.

Fractional factorial designs allow to reduce considerably the number of experiments. They are obtained by fractionating the design matrix. In our case we choose a 2^{6-2}_{IV} fractional factorial design. This means that we will have 16 runs and a resolution of 4 which is the maximum attainable for a 2^{6-2}. Resolution 4 implies that No main effects are confounded with any 2-factor interactions; main effects are confounded with 3-factor interactions. The defining relation is $0 = 1235 = 2346 = 1456$ [8].

It is possible to construct fractional factorial designs by means of the method `ffDesMatrix` of the package `BHH2`. Alternatively, a catalogue of designs can be found at http://www.itl.nist.gov/div898/handbook/pri/section3/pri3347.htm

```
> load("Data/fractional-DE-5.RData")
> DE[1:16, ]
```

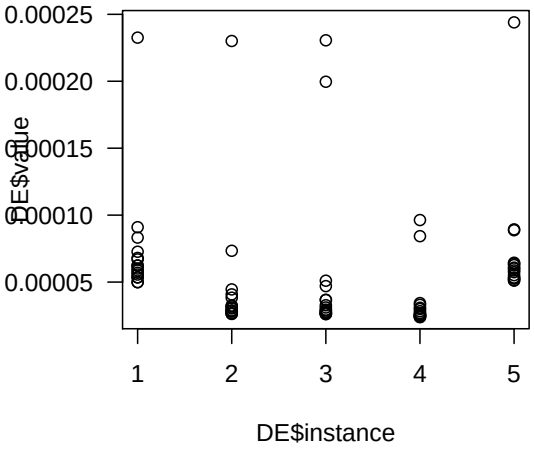
	instance	NP	F	CR	initial	strategy	idleiter	value	time	nfeval
1	1	-1	-1	-1	-1	-1	-1	5.36e-05	0.216	440
2	1	1	-1	-1	-1	1	-1	5.56e-05	0.448	880
3	1	-1	1	-1	-1	1	1	6.80e-05	0.660	1240
4	1	1	1	-1	-1	-1	1	6.23e-05	1.308	2480
5	1	-1	-1	1	-1	1	1	4.99e-05	0.652	1240
6	1	1	-1	1	-1	-1	1	4.99e-05	1.305	2480
7	1	-1	1	1	-1	-1	-1	5.87e-05	0.228	440
8	1	1	1	1	-1	1	-1	6.69e-05	0.448	880
9	1	-1	-1	-1	1	-1	1	5.70e-05	0.676	1240
10	1	1	-1	-1	1	1	1	7.27e-05	1.308	2480
11	1	-1	1	-1	1	1	-1	2.33e-04	0.220	440
12	1	1	1	-1	1	-1	-1	9.10e-05	0.452	880
13	1	-1	-1	1	1	1	-1	8.32e-05	0.228	440
14	1	1	-1	1	1	-1	-1	6.02e-05	0.460	880
15	1	-1	1	1	1	-1	1	6.24e-05	0.668	1240
16	1	1	1	1	1	1	1	5.35e-05	1.352	2480

We fit the model using the cell means formula and including second order interactions. We also include the instance factor as a blocking variable. The diagnostic plots of the model are not completely satisfactory. We therefore seek for a transformation of data that might improve the situation. The `boxcox()` method from the package MASS [13] search for a transformation of the kind:

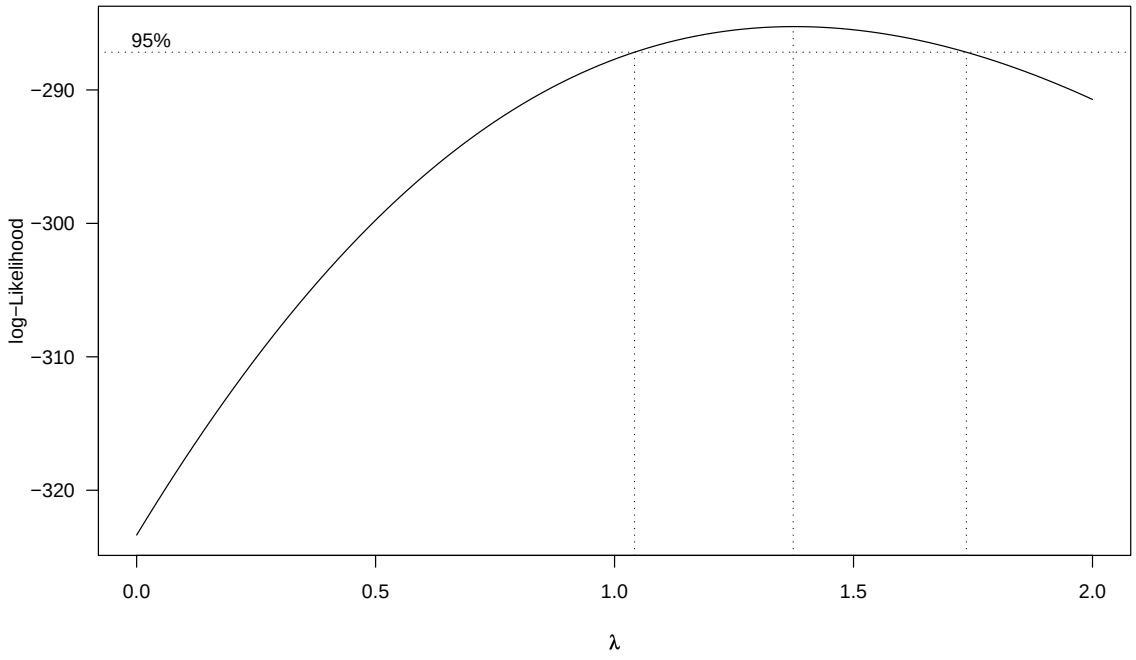
$$y(\lambda) = \begin{cases} (y^\lambda - 1)/\lambda & \lambda \neq 0 \\ \log y & \lambda = 0 \end{cases}$$

by maximizing the likelihood of the model.

```
> par(mfrow = c(1, 2))
> plot(DE$instance, DE$value)
> DE$rank <- unsplit(tapply(DE$value, DE$instance, rank), DE$instance)
> plot(DE$instance, DE$rank)
```




```
> library(MASS)
> boxcox(rank ~ (NP + F + CR + initial + strategy + idleiter +
      instance)^2 - 1, data = DE, singular.ok = TRUE, lambda = seq(0,
      2, 0.1))
```



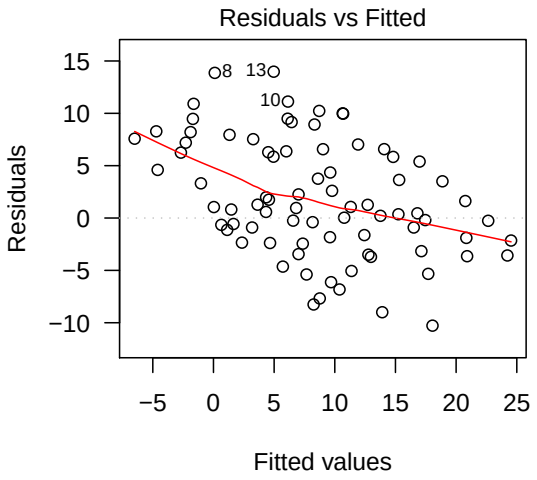
a value of 1.2 seems the best choice, and indeed the diagnostic plots are this time more satisfactory.

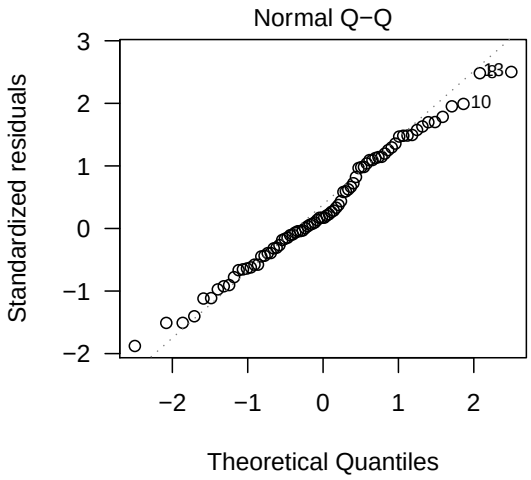
```
> l <- lm((rank^(1.2) - 1)/1.2 ~ (NP + F + CR + initial + strategy +
     idleiter + instance)^2 - 1, data = DE)
> par(mfrow = c(1, 2))
> plot(l, which = c(1, 2))
> summary(l)

Call:
lm(formula = (rank^(1.2) - 1)/1.2 ~ (NP + F + CR + initial +
    strategy + idleiter + instance)^2 - 1, data = DE)
Residuals:
   Min     1Q Median     3Q    Max
-10.28  -1.96   1.06   6.42  13.98

Coefficients: (8 not defined because of singularities)
                  Estimate Std. Error t value Pr(>|t|)
NP                 -1.3245     1.7677   -0.75    0.457
F                   3.4064     1.7677    1.93    0.059 .
CR                 -2.2118     1.7677   -1.25    0.216
initial             2.4763     1.7677    1.40    0.166
strategy            1.4755     1.7677    0.83    0.407
idleiter           -1.8129     1.7677   -1.03    0.309
instance            2.8501     0.2273   12.54   <2e-16 ***
NP:F               -1.8449     0.7538   -2.45    0.017 *
NP:CR              -1.9201     0.7538   -2.55    0.013 *
NP:initial         -0.6288     0.7538   -0.83    0.407
NP:strategy        -0.9669     0.7538   -1.28    0.205
NP:idleiter         0.5465     0.7538    0.73    0.471
NP:instance         0.4639     0.5330    0.87    0.388
F:CR                    NA         NA      NA       NA
F:initial          -0.2920     0.7538   -0.39    0.700
F:strategy              NA         NA      NA       NA
F:idleiter         -0.6186     0.7538   -0.82    0.415
F:instance          0.0182     0.5330    0.03    0.973
CR:initial              NA         NA      NA       NA
CR:strategy             NA         NA      NA       NA
CR:idleiter             NA         NA      NA       NA
CR:instance        -0.1230     0.5330   -0.23    0.818
initial:strategy        NA         NA      NA       NA
initial:idleiter        NA         NA      NA       NA
initial:instance   -0.2990     0.5330   -0.56    0.577
strategy:idleiter       NA         NA      NA       NA
strategy:instance  -0.2858     0.5330   -0.54    0.594
idleiter:instance   0.0571     0.5330    0.11    0.915
---
Signif. codes:  0 âĂŸ***âĂŹ 0.001 âĂŸ**âĂŹ 0.01 âĂŸ*âĂŹ 0.05 âĂŸ.âĂŹ 0.1 âĂŸ âĂŹ 1

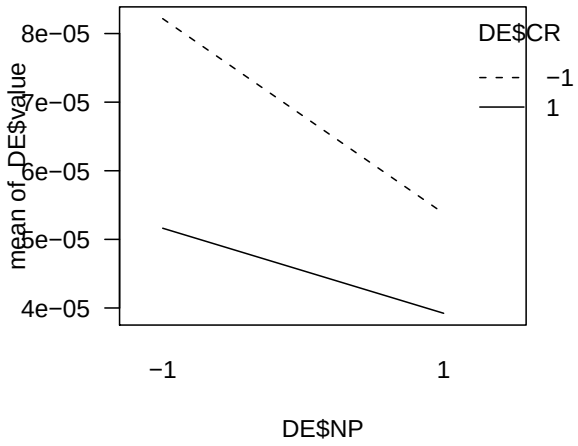
Residual standard error: 6.74 on 60 degrees of freedom
Multiple R-squared: 0.784,        Adjusted R-squared: 0.712
F-statistic: 10.9 on 20 and 60 DF,  p-value: 2.86e-13
```

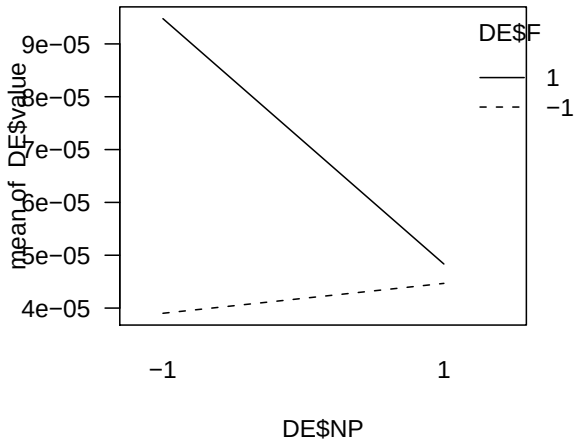




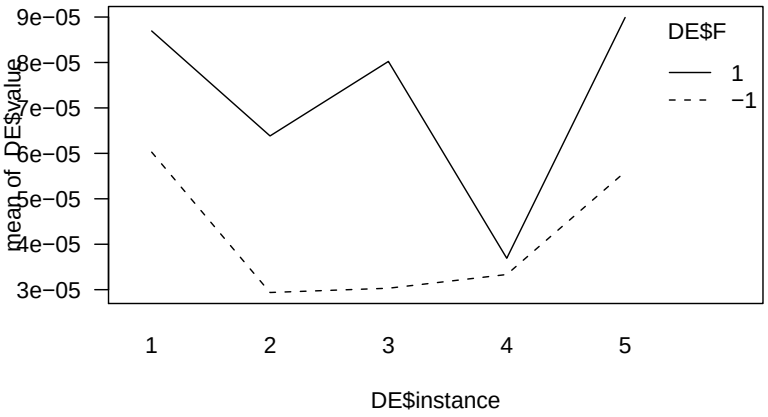
We can finally proceed with the analysis:

```
> par(mfrow = c(1, 2))
> interaction.plot(DE$NP, DE$CR, DE$value)
> interaction.plot(DE$NP, DE$F, DE$value)
```

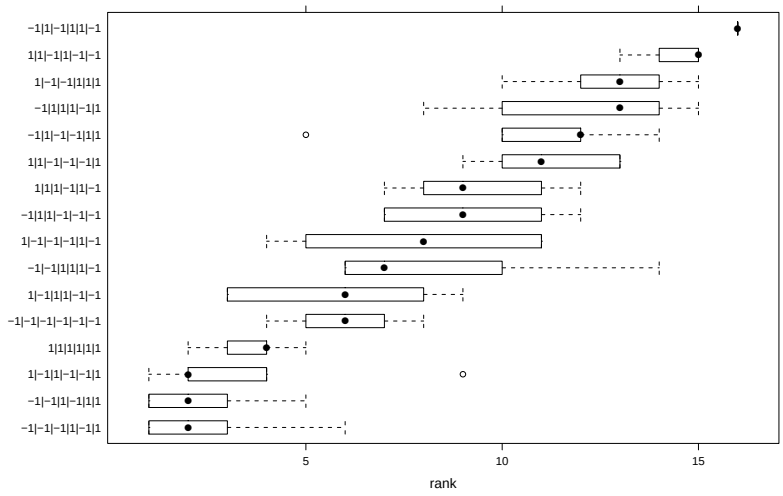





```
> interaction.plot(DE$instance, DE$F, DE$value)
```




```
> library(lattice)
> DE$label <- paste(DE$NP, DE$F, DE$CR, DE$initial, DE$strategy,
      DE$idleiter, sep = "|")
> O <- aggregate(DE$rank, list(DE$label), median)
> print(bwplot(factor(label, levels = c(O$Group.1[order(O$x)])) ~
      rank, data = DE))
```



Note that an analysis conducted only on one single instance provided much different results. For an example of this kind of analysis see [**?**]. The analysis here resemble [8, page 1233].

Test for lack of fit:

```
> l2 <- lm((rank^(1.2) - 1)/1.2 ~ factor(label) + instance - 1,
      data = DE)
> anova(l, l2)

Analysis of Variance Table
Model 1: (rank^(1.2) - 1)/1.2 ~ (NP + F + CR + initial + strategy + idleiter +
```

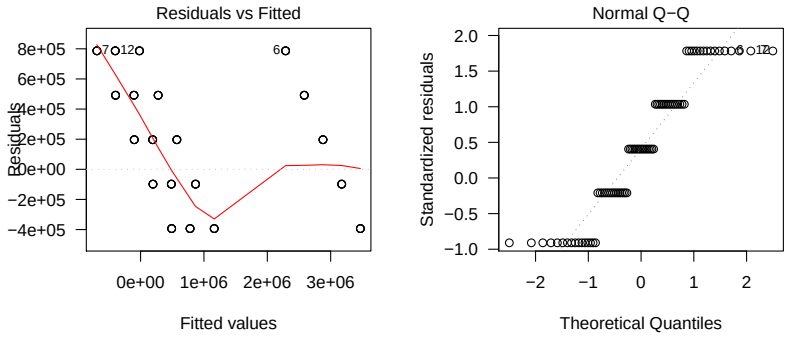
```
    instance)^2 - 1
Model 2: (rank^(1.2) - 1)/1.2 ~ factor(label) + instance - 1
  Res.Df  RSS Df Sum of Sq F Pr(>F)
1     60 2727
2     63  804 -3      1923
```

```
> l <- lm((nfeval^2 - 1)/2 ~ (NP + F + CR + initial + strategy +
     idleiter + instance)^2 - 1, data = DE)
> par(mfrow = c(1, 2))
> plot(l, which = c(1, 2))
> summary(l)
```

```
Call:
lm(formula = (nfeval^2 - 1)/2 ~ (NP + F + CR + initial + strategy +
    idleiter + instance)^2 - 1, data = DE)
Residuals:
    Min      1Q  Median      3Q     Max
-393454  -98364  196727  491818  786909

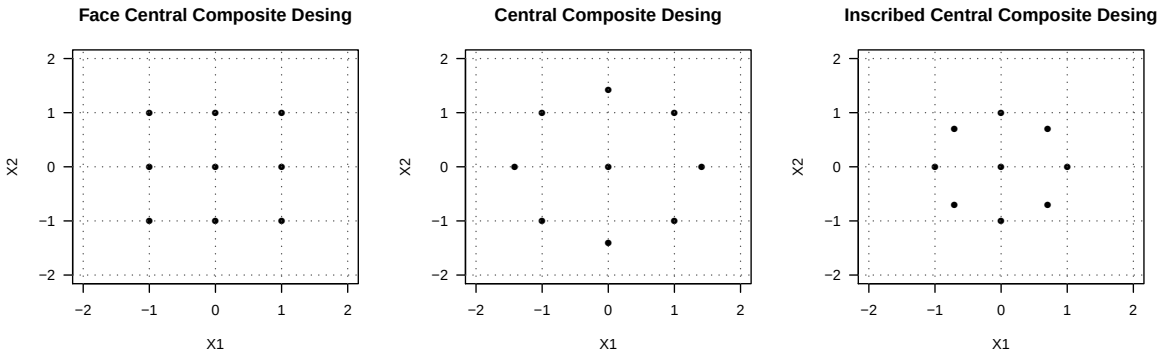
Coefficients: (8 not defined because of singularities)
                  Estimate Std. Error  t value Pr(>|t|)
NP                6.49e+05   1.40e+05     4.65  1.9e-05 ***
F                 4.41e-11   1.40e+05  3.2e-16        1
CR                3.01e-11   1.40e+05  2.2e-16        1
initial          -1.24e-11   1.40e+05 -8.9e-17        1
strategy         -8.73e-11   1.40e+05 -6.2e-16        1
idleiter          8.40e+05   1.40e+05     6.01  1.2e-07 ***
instance          2.95e+05   1.80e+04    16.43  < 2e-16 ***
NP:F             -5.73e-11   5.96e+04 -9.6e-16        1
NP:CR            -2.62e-11   5.96e+04 -4.4e-16        1
NP:initial        4.69e-11   5.96e+04  7.9e-16        1
NP:strategy       4.54e-12   5.96e+04  7.6e-17        1
NP:idleiter       5.04e+05   5.96e+04     8.46  8.0e-12 ***
NP:instance       6.65e-11   4.21e+04  1.6e-15        1
F:CR                    NA         NA       NA       NA
F:initial        -3.14e-11   5.96e+04 -5.3e-16        1
F:strategy              NA         NA       NA       NA
F:idleiter        1.43e-11   5.96e+04  2.4e-16        1
F:instance       -1.65e-11   4.21e+04 -3.9e-16        1
CR:initial              NA         NA       NA       NA
CR:strategy             NA         NA       NA       NA
CR:idleiter             NA         NA       NA       NA
CR:instance      -1.61e-11   4.21e+04 -3.8e-16        1
initial:strategy        NA         NA       NA       NA
initial:idleiter        NA         NA       NA       NA
initial:instance -7.15e-12   4.21e+04 -1.7e-16        1
strategy:idleiter       NA         NA       NA       NA
strategy:instance 4.79e-11   4.21e+04  1.1e-15        1
idleiter:instance 3.73e-11   4.21e+04  8.9e-16        1
---
Signif. codes:  0 âĂŸ***âĂŹ 0.001 âĂŸ**âĂŹ 0.01 âĂŸ*âĂŹ 0.05 âĂŸ.âĂŹ 0.1 âĂŸ âĂŹ 1

Residual standard error: 533000 on 60 degrees of freedom
Multiple R-squared: 0.917,        Adjusted R-squared: 0.889
F-statistic:   33 on 20 and 60 DF,  p-value: <2e-16
```



4.5 Response Surface

```
> alpha <- sqrt(2)
> mp <- c(-1, 1)
> x1 <- c(-1, 1, 0, 0, 0)
> x2 <- c(0, 0, -1, 1, 0)
> fccd <- expand.grid(X1 = c(-1, 0, 1), X2 = c(-1, 0, 1))
> ccd <- expand.grid(X1 = mp, X2 = mp)
> ccd <- rbind(ccd, cbind(X1 = x1 * alpha, X2 = x2 * alpha))
> iccd <- expand.grid(X1 = mp/alpha, X2 = mp/alpha)
> iccd <- rbind(iccd, cbind(X1 = x1, X2 = x2))
> par(mfrow = c(1, 3), pch = 16)
> plot(fccd, xlim = c(-2, 2), ylim = c(-2, 2), main = "Face Central Composite Desing")
> grid(col = "grey30")
> plot(ccd, xlim = c(-2, 2), ylim = c(-2, 2), main = "Central Composite Desing")
> grid(col = "grey30")
> plot(iccd, xlim = c(-2, 2), ylim = c(-2, 2), main = "Inscribed Central Composite Desing")
> grid(col = "grey30")
```



We design a RSM on Simulated Annealing considering the quantitative factors and their high and low levels described in Table 4.3.

We use an inscribed central composite design with 4 replicates at the center. We collect 10 replicates for each of the 18 points of the design blocking on 10 different instances. We use

the design with encoded variables $-1, 0, 1$. The corresponding actual levels can be obtained by:

$$A = \frac{\max\{A\} + \max\{A\}}{2} + X_i \frac{\max\{A\} - \min\{A\}}{2}$$

```
> iccdmp
```

```
       X1     X2     X3
1  -0.707 -0.707 -0.707
2   0.707 -0.707 -0.707
3  -0.707  0.707 -0.707
4   0.707  0.707 -0.707
5  -0.707 -0.707  0.707
6   0.707 -0.707  0.707
7  -0.707  0.707  0.707
8   0.707  0.707  0.707
9  -1.000  0.000  0.000
10  1.000  0.000  0.000
11  0.000 -1.000  0.000
12  0.000  1.000  0.000
13  0.000  0.000 -1.000
14  0.000  0.000  1.000
15  0.000  0.000  0.000
16  0.000  0.000  0.000
17  0.000  0.000  0.000
18  0.000  0.000  0.000
```

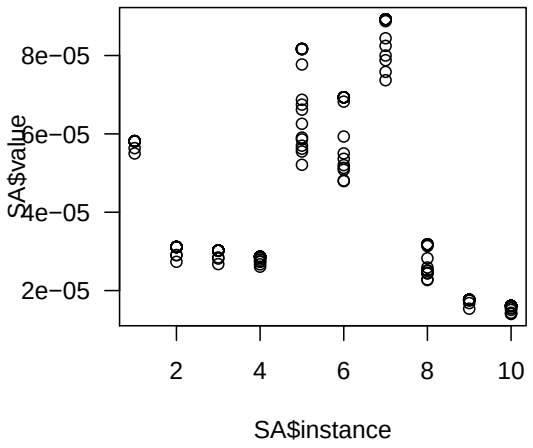
```
> load("Data/rsm.SA.RData")
> str(SA)
```

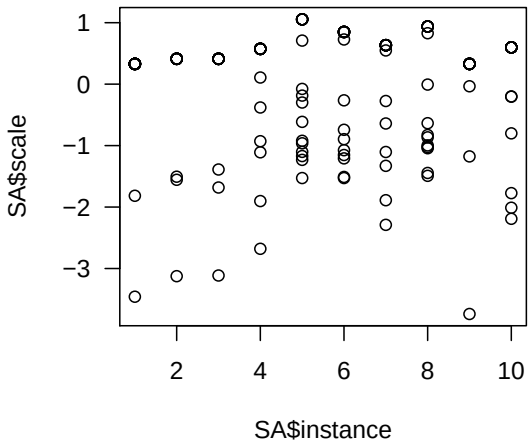
```
'data.frame':        180 obs. of  6 variables:
 $ instance: num  1 1 1 1 1 1 1 1 1 ...
 $ Eval    : num  -0.707 0.707 -0.707 0.707 -0.707 ...
 $ Temp    : num  -0.707 -0.707 0.707 0.707 -0.707 ...
 $ Tmax    : num  -0.707 -0.707 -0.707 -0.707 0.707 ...
 $ value   : num  5.81e-05 5.81e-05 5.81e-05 5.81e-05 5.81e-05 ...
 $ time    : num  2.7 5.52 2.63 5.55 2.64 ...
```

```
> par(mfrow = c(1, 2))
> plot(SA$instance, SA$value)
> SA$scale <- unsplit(tapply(SA$value, SA$instance, scale, scale = TRUE,
      center = TRUE), SA$instance)
> plot(SA$instance, SA$scale)
```

	Factor	Low $(-)$	High $(-)$
Eval	Max number of evaluations	100000	300000
Temp	Starting temperature for the cooling schedule	5	15
Tmax	number of function evaluations at each temperature	50	150

Table 4.3: Factors under study for SA.






```
> sa.q <- stepAIC(lm(scale ~ ((Eval * Temp * Tmax) + I(Eval^2) +
     I(Eval^3) + I(Temp^2) + I(Temp^3) + I(Tmax^2) + I(Tmax^3)),
     data = SA), trace = FALSE)
> sa.q$anova

Stepwise Model Path
Analysis of Deviance Table
Initial Model:
scale ~ ((Eval * Temp * Tmax) + I(Eval^2) + I(Eval^3) + I(Temp^2) +
    I(Temp^3) + I(Tmax^2) + I(Tmax^3))

Final Model:
scale ~ Temp + I(Eval^2) + I(Temp^3) + I(Tmax^2)


               Step Df Deviance Resid. Df Resid. Dev    AIC
1                                     166        150  -5.28
2       - I(Temp^2)  1   0.0116       167        150  -7.27
3       - I(Tmax^3)  1   0.4920       168        150  -8.68
4       - I(Eval^3)  1   0.9777       169        151  -9.51
5  - Eval:Temp:Tmax  1   1.3687       170        152  -9.89
6       - Temp:Tmax  1   0.2157       171        153 -11.63
7       - Eval:Tmax  1   0.3453       172        153 -13.22
8           - Tmax   1   1.0912       173        154 -13.95
9       - Eval:Temp  1   1.1770       174        155 -14.58
10          - Eval   1   0.5332       175        156 -15.96

> sa.t <- stepAIC(lm(time ~ ((Eval * Temp * Tmax) + I(Eval^2) +
     I(Eval^3) + I(Temp^2) + I(Temp^3) + I(Tmax^2) + I(Tmax^3)),
     data = SA), trace = FALSE)
> sa.t$anova

Stepwise Model Path
Analysis of Deviance Table
Initial Model:
time ~ ((Eval * Temp * Tmax) + I(Eval^2) + I(Eval^3) + I(Temp^2) +
```

```
    I(Temp^3) + I(Tmax^2) + I(Tmax^3))

Final Model:
time ~ Eval + I(Eval^2) + I(Tmax^2)


               Step Df Deviance Resid. Df Resid. Dev  AIC
1                                     166       5.03 -616
2  - Eval:Temp:Tmax  1 0.000794       167       5.03 -618
3       - I(Temp^3)  1 0.001239       168       5.04 -620
4       - I(Tmax^3)  1 0.002004       169       5.04 -622
5       - Eval:Tmax  1 0.002040       170       5.04 -624
6       - I(Eval^3)  1 0.006201       171       5.05 -625
7       - Temp:Tmax  1 0.006266       172       5.05 -627
8            - Tmax  1 0.000549       173       5.05 -629
9       - Eval:Temp  1 0.007144       174       5.06 -631
10           - Temp  1 0.000113       175       5.06 -633
11      - I(Temp^2)  1 0.013764       176       5.07 -634


> lq <- lm(scale ~ Temp + I(Eval^2) + I(Temp^3) + I(Tmax^2), data = SA)
> summary(lq)

Call:
lm(formula = scale ~ Temp + I(Eval^2) + I(Temp^3) + I(Tmax^2),
    data = SA)
Residuals:
   Min     1Q Median     3Q    Max
-3.408 -0.458  0.265  0.610  1.679

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   -0.332      0.122   -2.72   0.0071 **
Temp          -0.796      0.365   -2.18   0.0307 *
I(Eval^2)      0.479      0.211    2.27   0.0243 *
I(Temp^3)      1.089      0.517    2.11   0.0366 *
I(Tmax^2)      0.516      0.211    2.45   0.0154 *
---
Signif. codes:  0 âĂŸ***âĂŹ 0.001 âĂŸ**âĂŹ 0.01 âĂŸ*âĂŹ 0.05 âĂŸ.âĂŹ 0.1 âĂŸ âĂŹ 1

Residual standard error: 0.944 on 175 degrees of freedom
Multiple R-squared: 0.0834,        Adjusted R-squared: 0.0624
F-statistic: 3.98 on 4 and 175 DF,  p-value: 0.00408

> lq$coefficient

(Intercept)        Temp   I(Eval^2)   I(Temp^3)   I(Tmax^2)
     -0.332      -0.796       0.479       1.089       0.516

> par(mfrow = c(1, 2))
> plot(lq, which = c(1, 2))
> lt <- lm(time ~ Eval + I(Eval^2) + I(Tmax^2), data = SA)
> summary(lt)

Call:
lm(formula = time ~ Eval + I(Eval^2) + I(Tmax^2), data = SA)
Residuals:
    Min      1Q  Median      3Q     Max
```

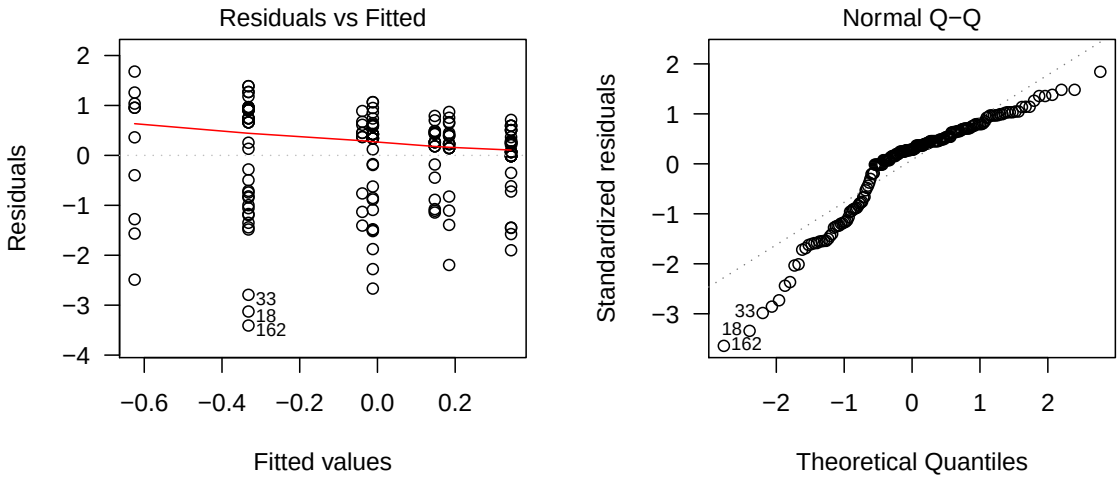
```
-0.2457 -0.0771 -0.0330  0.0126  0.8953

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   4.1377     0.0219   188.8   <2e-16 ***
Eval          2.0281     0.0219    92.5   <2e-16 ***
I(Eval^2)    -0.0571     0.0380    -1.5    0.134
I(Tmax^2)    -0.0683     0.0380    -1.8    0.074 .
---
Signif. codes:  0 âĂŸ***âĂŹ 0.001 âĂŸ**âĂŹ 0.01 âĂŸ*âĂŹ 0.05 âĂŸ.âĂŹ 0.1 âĂŸ âĂŹ 1

Residual standard error: 0.17 on 176 degrees of freedom
Multiple R-squared: 0.98,        Adjusted R-squared: 0.98
F-statistic: 2.86e+03 on 3 and 176 DF,  p-value: <2e-16
```

```
> lt$coefficient
```

```
(Intercept)        Eval   I(Eval^2)   I(Tmax^2)
     4.1377      2.0281     -0.0571     -0.0683
```



In order identify the optimal operating condition with respect to both variables, time and quality, in general one can use the desirability function approach. This works by defining a desirability function $d_i(Y_i) : \mathbf{R} \mapsto [0, 1]$ as follows:

$$d_i(Y_i) = \begin{cases} 1 & \text{if } \widehat{Y}_i(x) < T \text{ (target value)} \\ \frac{\widehat{Y}_i(x) - U_i}{T_i - U_i} & \text{if } T_i \leq \widehat{Y}_i(x) \leq U_i \\ 0 & \text{if } \widehat{Y}_i(x) > U_i \end{cases}$$

and then minimize $\left(\prod_{i=1}^{k} d_i\right)^{1/k}$. The landscape of the desirability function is non-convex. We use the Nelder-Mead algorithm to solve it.

```
> lqf <- function(Eval, Temp, Tmax) {
      Y <- (lq$coefficient[1] + lq$coefficient[2] * Temp + lq$coefficient[3] *
          Eval^2 + lq$coefficient[4] * Temp^3 + lq$coefficient[5] *
```

```
        Tmax^2)
    if (Y < 0)
        return(0)
    else return(Y)
 }
> ltf <- function(Eval, Tmax) {
    Y <- (lt$coefficient[1] + lt$coefficient[2] * Eval + lt$coefficient[3] *
        Eval^2 + lt$coefficient[4] * Tmax^2)
    if (Y < 0)
        return(0)
    else return(Y)
 }
> desirability <- function(e) {
    return(sqrt(lqf(Eval = e[1], Temp = e[2], Tmax = e[3]) *
        ltf(Eval = e[1], Tmax = e[3])))
 }
> OPT <- optim(c(-1, 0, 0), desirability, control = list(trace = 6,
    reltol = sqrt(.Machine$double.eps), abstol = 0, maxit = 500,
    alpha = 1, beta = 0.5, gamma = 2), method = "Nelder-Mead")

  Nelder-Mead direct search function minimizer
function value for initial parameters = 0.550199
  Scaled convergence tolerance is 8.19861e-09
Stepsize computed as 0.100000
BUILD              4 0.559653 0.357529
Exiting from Nelder Mead minimizer
    6 function evaluations used

> OPT$par

[1] -0.9  0.1 -0.2
```

The results of Nelder Mead are very dependent on the starting solution. The results provided by its optimization are therefore not very helpful. We try to gain more insights by looking at the regression coefficients and at the plot of computational time, that being modelled by only two variables can still be drawn in a 3D plot. As expected the time grows fast with `Eval`. A value of -1 would be preferable according to time plot. However looking at how Eval appear in the function of solution quality we see that a value of zero yields the best choice. We therefore fix Eval to zero and explore visually the model of solution quality.

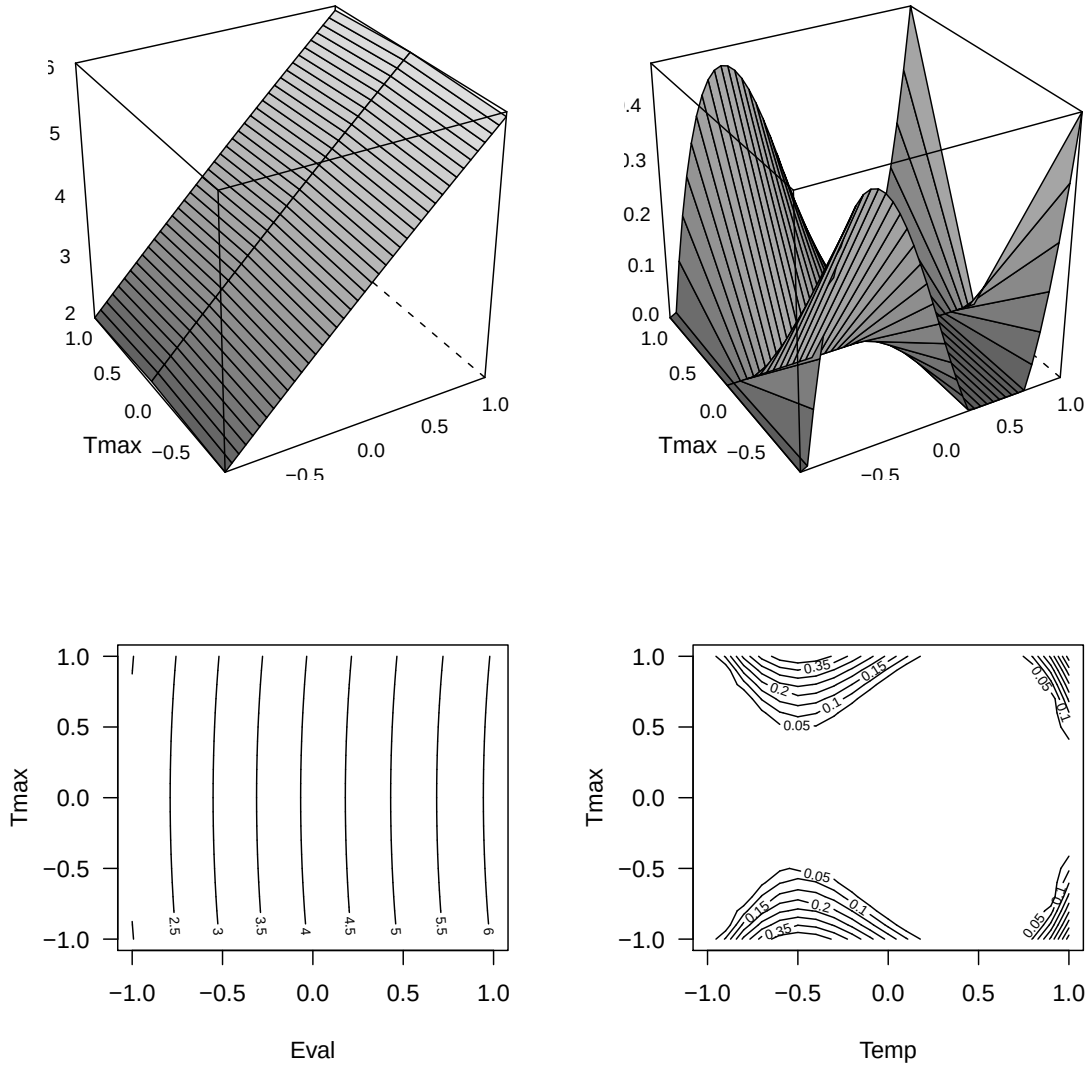


Figure 4.2: Visualization of the response surfaces of computation time (left) and solution quality (right) using 3D plots (top) and contour plots (bottom). The plots on the right are obtained fixing Eval to zero.

Chapter 5

Performance Modelling

5.1 Modelling Run Time Distributions

Modelling of run time distributions has roots in the field of statistics that handle lifetime data, namely survival analysis [9].

In this context the models are typically represented by a probability density function f (assuming time is a continous variable). It is then typical to derive the survivor function

$$S(t) = Pr(T \geq t) = \in_t^\infty f(x)dx = 1 - F(t)$$

that represents the probability that an individual survives till time t, and the hazrd function, given by

$$h(t) = \lim_{\Delta t \leftarrow 0} \frac{Pr(t \leq T < t + \Delta t \mid T \geq t)}{\Delta t}$$

The hazard function specifies the instantaneous rate of death or failure at time t given that the individual survives up till t.

In the analysis of the algorithms, the sense of terms is reversed and death means finding a solution while surviving means not having found yet a solution. The survivor functiuon is therefore not very useful, while the cumulative distribution fucntion F has a more intuitive meaning. The profile of the hazard function might be useful for qualitative information about the capability of an algorithm to solve an instance. It can have different patterns. A monotonoe decreasing hazard function indicates a decrease of the chances of solving an instance with time. An constant hazard function is typical os memory less distributions that are important in this kind of analysis.

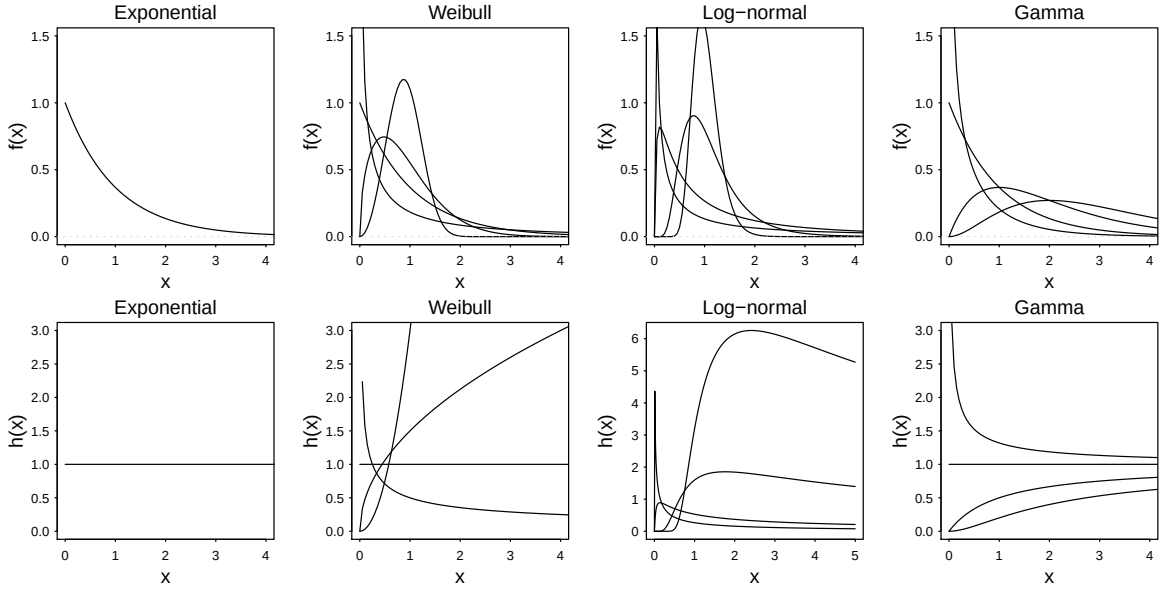
5.2 Some important models

In the following we illustrate four parametric univariate models that are occupy a central role because of their demonstrated usefulneses in a wide range of situations. For a thorough treatment of probability distributions we refer to [7].

We remark that all models are presented without the inclusion of an initialization threashold paramters. Briefly, this is a time $\mu \geq 0$ before which it is assumed that an algorithm cannot

find a solution. The distributiuons can be extended to include such a parameter by merely replacing the soltuiono time t by $t' = t - \mu$, with t' satisfyiong the restriction $t' \geq 0$.

The exponential distribution has the nice feature of being memory less. The Weibull distributiuon is very flexible and appeling for the semplicity of its analytical formula of distribtuiin function, cumulative distribtuion function and hazard function. The log-normal distribution is used when the logarithm of time is distributed as a normal distribution. Moreover it has the feature that its hazard function is decreasing with $t \to \infty$ which might be the case for several algorithms, as we might often have experienced that if an algorithm does not find a solution soon it may time long time before it finds one (for this feature the log-normal distribution is instead often ruled out in survival analysis). The gamma distribution is also very flexible, moreover it has the nice mathematical property that a gamma distribution is a sum of indepenendt indentically distributed exponential random variables.



We analyse the runs of tow algorithms for solving the Hadamard problem. This problem can be formulated as a constraint satistfaction problem. It consists in finding a vector of values $\{-1, 1\}$ that generates a Hadamard matrix. The empirical cumulative distribution functions of the time to find a solution for the two algorithms are depicted in Figure 5.1. Each algorithm was run 50 tines on a sigle instance of this problem, corresponding to Hadamard matrices of size 68.

We search for linearity in plots with appropriate scales. For example, for an exponential distribution, it is:

$$\log S(t) = -\lambda t, \qquad \text{where} S(t) = 1 - F(t) \;\; \text{survivor function}$$

hence the plot of $\log S(t)$ against t should be linear. Similarly, for the Weibull distribution, the cumulative hazard function is linear on a log-log plot. In Figure **??** we observe that a Weibull distribution seem appropriate. We proceed therefore to fit this model.

```
> par(mfrow = c(1, 2))
> plot(t, fun = "log", ylab = "log S(t)", xlab = "t", main = "linear => exponential",
     panel.first = grid(nx = NULL, ny = NULL, col = "grey60",
```

```
> load("Data/r33.RData")
> require(survival)
> t <- survfit(Surv(time, event) ~ case, data = R33, type = "kaplan-meier",
      conf.type = "plain", conf.int = 0.95, se.fit = T)
> plot(t, conf.int = F, panel.first = grid(nx = NULL, ny = NULL,
      col = "grey60"), xlab = "Time to find a solution", col = c("grey50",
      "black"), lty = c(1, 1), ylab = "ecdf", fun = "event", ylim = c(0,
      1))
```

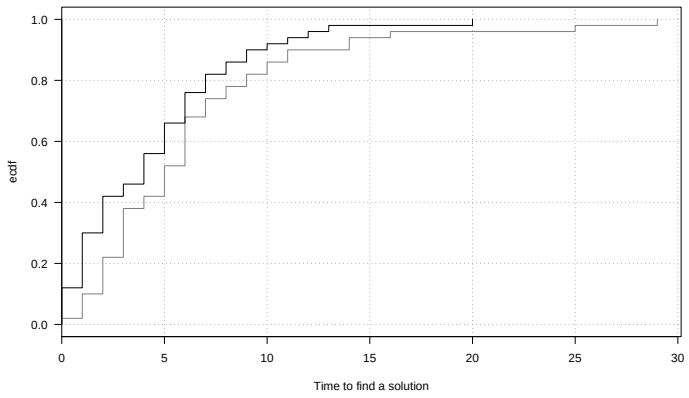
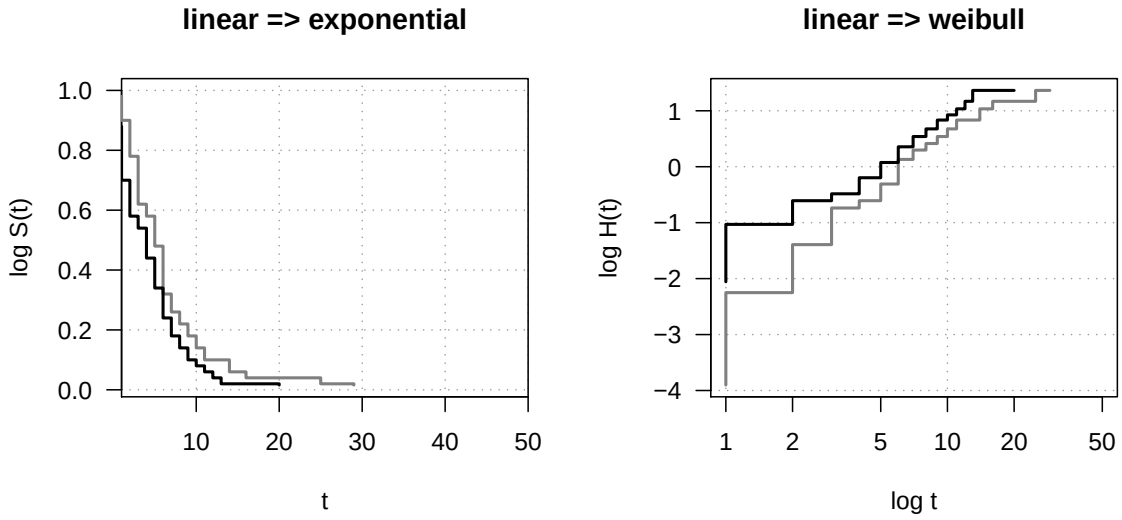


Figure 5.1: Emprical cumulative distribution functions on the Hadamard case.

```
        equilogs = TRUE), xlim = c(1, 50), col = c("grey50",
        "black"), lty = c(1, 1), lwd = c(2, 2))
> plot(t, fun = "cloglog", ylab = "log H(t)", xlab = "log t", main = "linear => weibull",
     panel.first = grid(nx = NULL, ny = NULL, col = "grey60",
        equilogs = TRUE), xlim = c(1, 50), col = c("grey50",
        "black"), lty = c(1, 1), lwd = c(2, 2))
```




```
> require(MASS)
> par(mar = c(3, 4, 1, 1))
> plot(t, conf.int = F, lty = c(1, 1), col = c("grey50", "black"),
     panel.first = grid(nx = NULL, ny = NULL, col = "grey60"),
     xlab = "Time to find a solution", ylab = "ecdf", fun = "event",
     ylim = c(0, 1))
> fm <- fitdistr(R33[R33$case == 2, ]$time, "exponential")
> fm$loglik

[1] -125

> ks.test(R33[R33$case == 2, ]$time, "pexp", rate = fm$estimate["rate"])

        One-sample Kolmogorov-Smirnov test
data:  R33[R33$case == 2, ]$time
D = 0.132, p-value = 0.3470
alternative hypothesis: two-sided

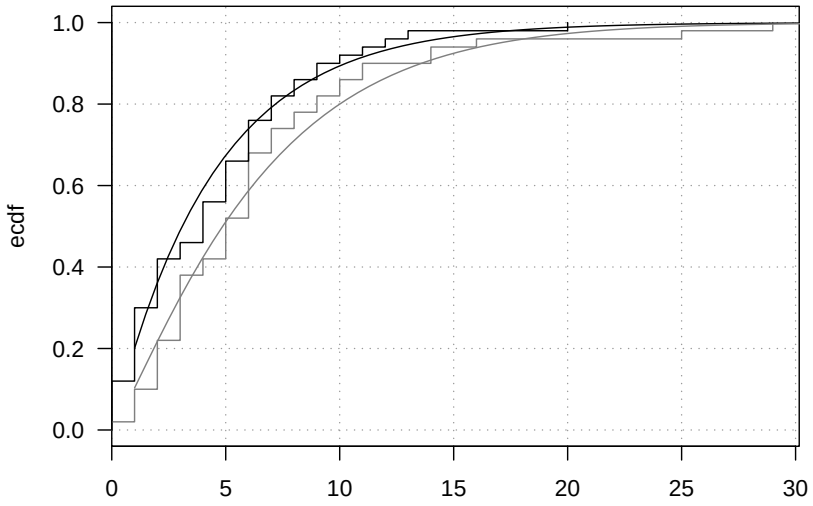
> curve(pexp(x, rate = fm$estimate["rate"]), from = 1, to = 50,
     add = TRUE)
> fm <- fitdistr(R33[R33$case == 1, ]$time + 0.001, "weibull")
> fm$loglik

[1] -141

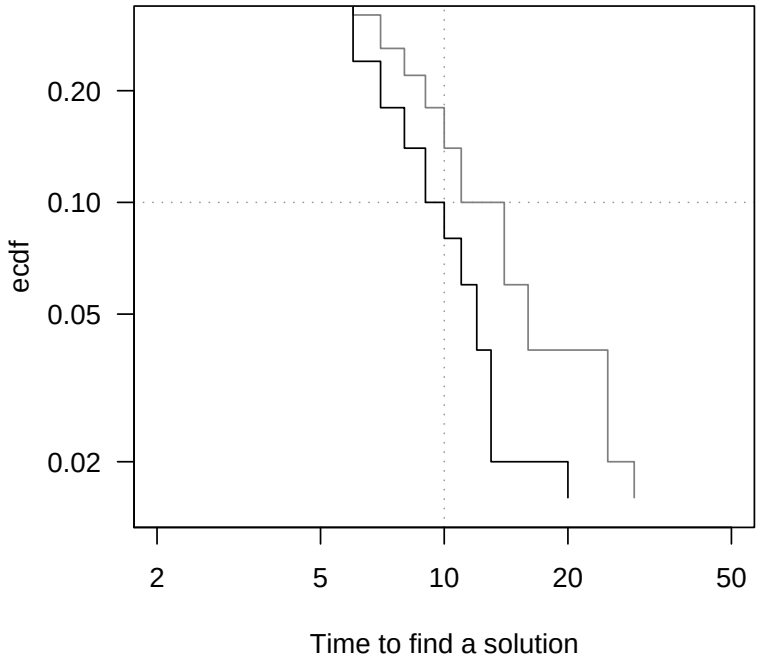
> ks.test(R33[R33$case == 1, ]$time + 0.001, "pweibull", scale = fm$estimate["scale"],
     shape = fm$estimate["shape"])
```

```
        One-sample Kolmogorov-Smirnov test
data:  R33[R33$case == 1, ]$time + 0.001
D = 0.117, p-value = 0.4955
alternative hypothesis: two-sided
```

```
> curve(pweibull(x, shape = fm$estimate["shape"], scale = fm$estimate["scale"]),
      from = 1, to = 50, add = TRUE, col = "grey50")
```




```
> plot(t, log = "xy", conf.int = F, col = c("grey50", "black"),
      lty = c(1, 1), xlim = c(2, 50), ylim = c(0.015, 0.3), panel.first = grid(nx = NULL,
          ny = NULL, col = "grey60", equilogs = TRUE), xlab = "Time to find a solution",
      ylab = "ecdf")
```



Model fitting for censored distributions We may have two types of censoring. In *Type I censor sampling* one decides a cutoff time t_c and stops experiments that exceed that cutoff. Using indicator function δ_i the likelihood to maximize in order to fiind the parameters of a tentive model f is

$$L(T|\theta) = \prod_{i=1}^{k} f(T_i|\theta)^{\delta_i} \Big(\int_{t_c}^{\infty} f(\tau|\theta) d\tau \Big)^{1-\delta_i}$$

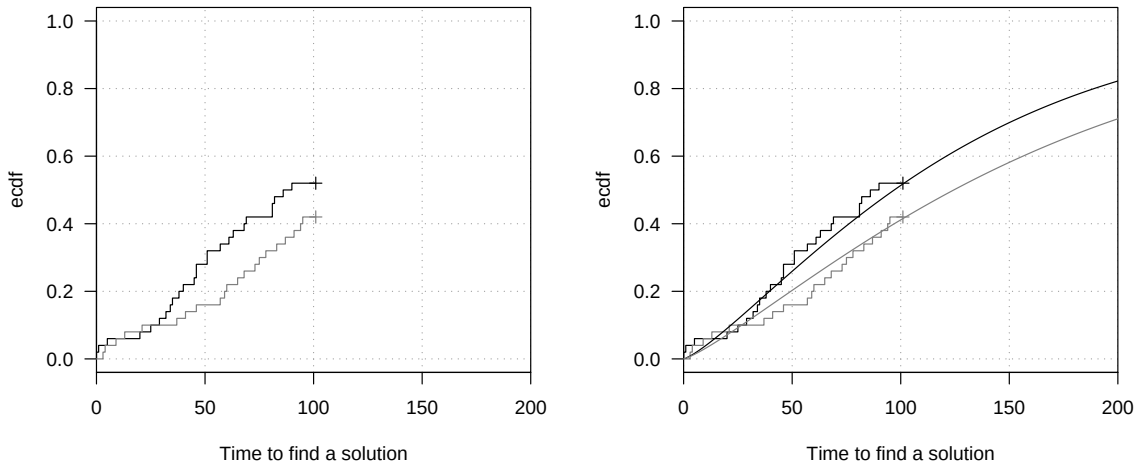
In *Type II censor sampling*, r experiments are run in parallel and stopped whenever u uncensored samples are obtained. Thus, $c = (r-u)/r$ are set in advance, and t_c is equal to time of uth fastest.

Below we report our experience on the Hadamard problem when we have a type I censoring for a restart tabu search algorithm run with two different parameter settings. The time limit was imposed to be 100 seconds. Many runs did not finished with a solution foudn, and are therefore to be considered censored.[1]

We fit a Weibull distribution to both algorithms. The fitting of other models can be conducted similarly.

```
> require(survival)
> load("Data/r37.RData")
```

[1]Note that in statistics literature a difference is made between *censored*, for which a value exists but it is not observed and *truncated*, for which a value does not exist because the one reached is a threshold.




```
> y <- R37[R37$case == 1, ]$time
> y <- y[y > 0]
> censored.weibull <- function(param, t, threshold = 100) {
     ifelse(t < threshold, dweibull(t, scale = param[1], shape = param[2]),
         pweibull(threshold, scale = param[1], shape = param[2],
             lower.tail = FALSE))
 }
> ll <- function(x, y) -sum(log(censored.weibull(x, y, 100)))
> opt.w1 <- optim(c(100, 10), ll, y = y, method = "Nelder-Mead")
> curve(pweibull(x, shape = opt.w1$par[2], scale = opt.w1$par[1]),
     from = 1, to = 200, add = TRUE)
> y <- R37[R37$case == 2, ]$time
> y <- y[y > 0]
> opt.w2 <- optim(c(100, 10), ll, y = y, method = "Nelder-Mead")
> curve(pweibull(x, shape = opt.w2$par[2], scale = opt.w2$par[1]),
     from = 1, to = 200, col = "gray50", lty = 1, add = TRUE)
```

Bibliography

[1] Birattari, M.: The race package for R. racing methods for the selection of the best. Tech. Rep. TR/IRIDIA/2003-37, IRIDIA, Université Libre de Bruxelles, Brussels, Belgium (2003) (Cited on page 32.)

[2] Birattari, M.: The Problem of Tuning Metaheuristics, as seen from a machine learning perspective. Ph.D. thesis, Université Libre de Bruxelles, Brussels, Belgium (2004) (Cited on page 32.)

[3] Birattari, M.: race: Racing methods for the selection of the best (2005), http://cran.r-project.org/web/packages/race/, R package version 0.1.56 (Cited on page 32.)

[4] Birattari, M.: Tuning Metaheuristics, A Machine Learning Perspective, Studies in Computational Intelligence, vol. 197. Springer (2009) (Cited on page 32.)

[5] Bratley, P., Fox, B.L., Niederreiter, H.: Algorithm-738 - programs to generate niederreiters low-discrepancy sequences. ACM Transactions On Mathematical Software 20(4), 494–495 (Dec 1994) (Cited on pages 23 and 24.)

[6] Deepayan, S.: Lattice Multivariate Data Visualization with R. Springer, New York (2007), iSBN 978-0-387-75968-5 (Cited on page 5.)

[7] Johnson, N.L., Kotz, S.: Distributions in statistics. Wiley series in probability and mathematical statistics, New York (1970) (Cited on page 57.)

[8] Kutner, M.H., Nachtsheim, C.J., Neter, J., Li, W.: Applied Linear Statistical Models. McGraw-Hill, fifth edn. (2005) (Cited on pages 44 and 48.)

[9] Lawless, J.F.: Statistical Models and Methods for Lifetime Data. Wiley Series in Probability and Mathematical Statistics, jws (1982) (Cited on page 57.)

[10] Leisch, F.: Sweave: Dynamic generation of statistical reports using literate data analysis pp. 575–580 (2002), http://www.stat.uni-muenchen.de/~leisch/Sweave, iSBN 3-7908-1517-9 (Cited on page 8.)

[11] Leisch, F.: Sweave user manual (2008), http://www.statistik.lmu.de/~leisch/Sweave/ (Cited on page 8.)

[12] Nelder, J.A., Mead, R.: A simplex method for function minimization. The Computer Journal 7(4), 308–313 (1965), an Errata has been published in The Computer Journal 1965 8(1):27 (Cited on page 21.)

[13] Venables, W.N., Ripley, B.D.: Modern Applied Statistics with S. Springer, Berlin, fourth edn. (2002) (Cited on page 44.)