Statistical Analysis of Optimization Algorithms with R

T. Bartz-Beielstein, Mike Preuss, Martin Zaefferer

CIplus (Cologne University of Applied Sciences) and
Algorithm Engineering (TU Dortmund)

July 2012

GECCO'12 Companion, July 7–11, 2012, Philadelphia, PA, USA.

ACM 978-1-4503-1178-6/12/07.

Agenda

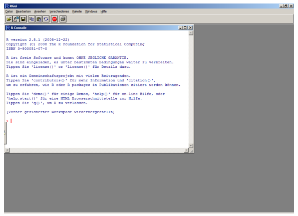
Your Instructors Today

- Dr. Thomas Bartz-Beielstein is a professor for Applied Mathematics at Cologne University of Applied Sciences. He has published more than several dozen research papers, presented tutorials about tuning, and has edited several books in the field of Computational Intelligence.
- Mike Preuss is research associate at the Computer Science Department, TU Dortmund. His main fields of activity are EAs for real-valued problems and their application in numerous engineering domains
- Martin Zaefferer is a research assistant at Cologne University of Applied Sciences. His research interests include computational intelligence, applications of knowledge discovery and sequential parameter optimization.

Goals

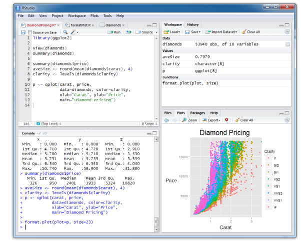
- Most effective approach for learning how to design, conduct, and analyze experiments that optimize performance in algorithms
- Show how to use statistically designed experiments to
  - Obtain information for characterization and optimization of algorithms
  - Improve their performance
  - Design and develop new operators and algorithms
- Learn how to evaluate algorithm alternatives, improve their field performance and reliability
- Conduct experiments effectively and efficiently
- Hands-on tutorial which
  - demonstrates how to analyze results from real experimental studies, e.g., experimental studies in EC
  - gives a comprehensive introduction in the `R` language
  - introduces the powerful GUI "rstudio" (`http://rstudio.org`)
  - will be held in an interactive manner, i.e., the analyses will be performed in real time.

Pure R



- Windows version comes with a simple build-in GUI

RStudio



- Powerful productivity tools
  - Syntax highlighting, code completion, and smart indentation
  - Execute R code directly from the source editor
  - Easily manage multiple working directories using projects
  - Quickly navigate code using typeahead search and go to definition

The Famous Iris Data Set

- Four features were measured from each sample
- Length and width of sepal and petal, respectively

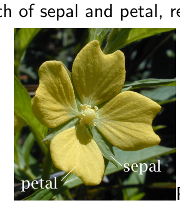


Photo by[7]

The Famous Iris Data Set: Iris Setosa, Virginica, and Versicolor

- Based on the combination of the four features, Fisher [5] developed a linear discriminant model to determine which species from these four measurements
- Used as a typical test for many other classification techniques

- Iris setosa [3], iris virginica [10], and iris versicolor [8]

The Famous Iris Data Set: Hands-on Exercises

- How to generate a scatter plot of Fisher's Iris data with pure R code
- First, we load the data frame:

```
> data(iris)
```

- Next, we have a quick look at the data (here, only the first three rows are shown)

```
> iris[1:3,]
```

```
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
```


The Famous Iris Data Set: Hands-on Exercises

- The `summary()` command gives a quick overview

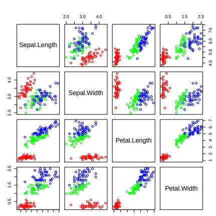
```
> options(width=70)
> summary(iris)
```

```
  Sepal.Length    Sepal.Width     Petal.Length    Petal.Width   
 Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100  
 1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300  
 Median :5.800   Median :3.000   Median :4.350   Median :1.300  
 Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199  
 3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800  
 Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500  
       Species  
 setosa    :50  
 versicolor:50  
 virginica :50  
```


The Famous Iris Data Set: Hands-on Exercises

- Finally, a scatter plot is generated with the `pairs()` function.

```
> pairs(iris[,1:4], col=c("red", "green",  "blue")[as.numeric(iris$Species)])
```



A Gentle Introduction to R

- Some of the following examples are based on [9]
- R can be used as a calculator

```
> 2+2
[1] 4
> 5*3*4
[1] 60
```

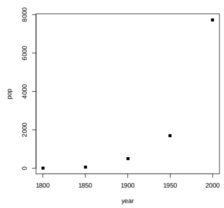
- Data entry

```
> year <- c(1800,1850,1900,1950,2000)
> pop <- c(18,54,500,1701,7731)
```

A Simple plot

Data entry:

```
> print( c(year,pop))
 [1] 1800 1850 1900 1950 2000   18   54  500 1701 7731
> plot(pop~year, pch=15)
```



R sessions

The working directory.

```
> getwd()
[1] "C:/Users/bartz/Documents/workspace/SvnSpot.d/trunk/publications/Gecco2012Tutorial.d/Slides2012.d"
```

Use `ls` to list contents of R's workspace:

```
> ls()
[1] "iris" "mygd" "pop"  "year"
```

Quitting: Note, `q()` is a function and can be used if R should be quit.

Expressions, Objects, and Methods

- Standard interaction mode in R is as follows:
  - Users enter an expression, which is evaluated by the R system. Result is printed on the screen
- Expressions work on *objects*
- Each object has a *class* attribute, which is a character vector

```
> x <- 10
> class(x)
[1] "numeric"
```


Logical Operators and Vectors in R

- R implements the following logical operators
  - `&` , the logical "and",
  - `|` , the logical "or", and
  - `!`, the logical "not" operator
- R commands to generate vectors:
  - `c()` ("concatenate"),
  - `seq()` ("sequence"), and
  - `rep()` ("replicate")
- Modes: logical, numeric, character, or list

```
> x <- c(1,2)
> y <- c(3,4)
> z <- c(x,y)
> x ==y
[1] FALSE FALSE
```

Vectors in R

- R's repeat command `rep()` can be used in two variants
  - To repeat the numerical value one ten times, we use

```
> rep(1,10)
 [1] 1 1 1 1 1 1 1 1 1 1
```

  - The second argument to the `rep()` command can be a vector.

```
> v <- c(1,2,4)
> letters <- c("a","b","c")
> rep(letters, v)
[1] "a" "b" "b" "c" "c" "c" "c"
```

- Here, the first element "a" is repeated once, the second element "b" twice, and the third element "c" four times

Vectors in R: Sequences

```
> seq(from=5, to=22, by=3)
[1]  5  8 11 14 17 20
```

Short form

```
> seq(5,22,3)
[1]  5  8 11 14 17 20
```

Default step size is one. Short form with colon

```
> seq(0,10) == 0:10
 [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
```

Character vectors: vectors of text strings, entries are specified in quotes

```
> c("one","two","three")
[1] "one"   "two"   "three"
```


Calculations with Vectors in R

- Calculations with vectors of the same length (like ordinary numbers)

```
> x <- c(1,2,3)
> y <- c(1,2,4)
> x+y
[1] 2 4 7
```

- Relational expressions can be evaluated as follows

```
> x < y
[1] FALSE FALSE  TRUE
```

- If vectors do not have the same length, the shorter vector is recycled

```
> y <- c(5,6)
> x+y
[1] 6 8 8
```

- Vector v modified to $(5, 6, 5)$, i.e., the first element is added at the end. Both vectors have the same size and can be added

Calculating the mean of vectors in R

Consider the vector

```
> x<- c(2,3,5,7)
```

To calculate its mean,

$$\overline{x} = \frac{1}{n} \sum_{i=1}^{n} x_i,$$

we can proceed as follows:

```
> sum(x)/length(x)
[1] 4.25
```

Calculating the standard deviation of vectors in R

To calculate its standard deviation,

$$\mathrm{sd}(x) = \sqrt{\sum_{i=1}^{n} \frac{(x_i - \overline{x})^2}{n-1}},$$

we can proceed as follows:

```
> xbar <- sum(x)/length(x)
> sqrt( sum(  (x - xbar)^2 / (length(x)-1) ) )
[1] 2.217356
```

Alternatively, we can use the build-in commands

```
> mean(x)
[1] 4.25
> sd(x)
[1] 2.217356
```


Subsets, and Indexing

Brackets are used to access certain elements of a vector. To select the i-th entry of the vector v, e.g., the third entry v_3, we can use the command

```
> v <- c(10,20,30,40,50,60,70,80,90,100)
> v[3]
[1] 30
```

This procedure is referred to as *indexing* in the following. To select a subset, we can index with a vector.

```
> v[ c(3,4,5)]
[1] 30 40 50
```

Use negative subscripts to omit elements in nominated subscript positions

```
> v[-c(2,3)]
[1]  10  40  50  60  70  80  90 100
```


Conditional Selection

- To modify elements of an vector, we can use the assignment operator

```
> v[1] <- -10
> v
 [1] -10  20  30  40  50  60  70  80  90 100
```

- Conditional selection can be performed as follows

```
> v>50
 [1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
> v[v>50]
[1]  60  70  80  90 100
```

- Logical operators can be used to combine several subset selection conditions.
- For example, to select entries, which are larger than 55 and smaller than 79, we can use the following command

```
> v[ v > 55 & v < 79]
[1] 60 70
```


Missing Values. The Symbol NA

```
> y <- c(1, NA, 3, 0, NA)
> y
[1]  1 NA  3  0 NA
```

Any operation that involves `NA` generates `NA`. The following does not work as expected, all values remain unchanged:

```
> y[y==NA] <- 0
> print(y)
[1]  1 NA  3  0 NA
```

To replace `NA` by 0, use `is.na()`:

```
> y[is.na(y)] <- 0
> print(y)
[1] 1 0 3 0 0
```

Some functions, e.g., `mean()` take the argument `"na.rm=T"`.

Factors

Categorical data such as gender can be "female" or "male", respectively. Categorical data should be specified as factors.

```
> gender <- c(rep("female", 3), rep("male", 5))
```

To generate a factor, use R's `factor()` command

```
> gender <- factor(gender)
```

Now: internally 3 1s are followed by 5 2s. We can use the function `as.numeric()` to extract the numerical coding as numbers "1" and "2".

```
> gender
[1] female female female male   male   male   male   male
Levels: female male
> as.numeric(gender)
[1] 1 1 1 2 2 2 2 2
```

A factor has set of levels. "female" and "male" are the levels of the factor gender:

```
> levels(gender)
[1] "female" "male"
```


Matrices and Arrays

Matrices and arrays are vectors with dimensions.

```
> x <- 1:20
> dim(x) <- c(5,4)
> x
     [,1] [,2] [,3] [,4]
[1,]    1    6   11   16
[2,]    2    7   12   17
[3,]    3    8   13   18
[4,]    4    9   14   19
[5,]    5   10   15   20
```

We can also use the `matrix()` command.

```
> matrix(1:20, nrow= 5)
     [,1] [,2] [,3] [,4]
[1,]    1    6   11   16
[2,]    2    7   12   17
[3,]    3    8   13   18
[4,]    4    9   14   19
[5,]    5   10   15   20
```


Matrices and Arrays

To fill the matrix rowwise, we can use the argument `byrow=T`. To label the rows of a matrix, we can use the command `rownames()`.

```
> A <- matrix(1:20, nrow= 5, byrow=T)
> rownames(A) <- LETTERS[1:nrow(A)]
> colnames(A) <- 1:ncol(A)
> A
   1  2  3  4
A  1  2  3  4
B  5  6  7  8
C  9 10 11 12
D 13 14 15 16
E 17 18 19 20
```


Combining Matrices and Vectors

`cbind()` and `rbind()` combine objects such as matrices or vectors columnwise or rowwise, respectively

```
> A <- matrix(1:4, nrow= 2, byrow=T)
> B <- matrix(10*(1:4), nrow= 2, byrow=T)
> cbind(A,B)
     [,1] [,2] [,3] [,4]
[1,]    1    2   10   20
[2,]    3    4   30   40
> rbind(A,B)
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]   10   20
[4,]   30   40
```

Lists

Many R functions return results as a list. Flexible structures to store heterogeneous data, e.g., numerical or boolean values.

```
> l <- list(  c("a", "b", "c"), 1:4, c(TRUE, TRUE, FALSE))
> l

[[1]]
[1] "a" "b" "c"

[[2]]
[1] 1 2 3 4

[[3]]
[1]  TRUE  TRUE FALSE
```

List `l` has three elements: 1) three strings, 2) numbers from one to four, and 3) three boolean values.

Addressing Lists

Single square brackets return a list.

```
> l[1]

[[1]]
[1] "a" "b" "c"
```

Here, `l[1]` returns a list of length one, whereas `l[2:3]` returns a list of length two.

```
> l[2:3]

[[1]]
[1] 1 2 3 4

[[2]]
[1]  TRUE  TRUE FALSE
```

List elements can be addressed by double square brackets

```
> l[[1]]

[1] "a" "b" "c"

> l[[1]][2]

[1] "b"
```


Adding and Deleting List Elements

List elements can be deleted by setting their values to `NULL`

```
> l

[[1]]
[1] "a" "b" "c"

[[2]]
[1] 1 2 3 4

[[3]]
[1]  TRUE  TRUE FALSE
```

Delete the first element

```
> l[[1]]   <- NULL
> l

[[1]]
[1] 1 2 3 4

[[2]]
[1]  TRUE  TRUE FALSE
```

To delete multiple list elements, we can use the minus sign

```
> l <-  c(l,l)
> l <- l[-c(3,4)]
> l

[[1]]
[1] 1 2 3 4

[[2]]
[1]  TRUE  TRUE FALSE
```


Length of Lists

`length()` to determine the length of a list

```
> length(l)

[1] 2
```

Adding a new element at the end using `length()`

```
> l[[length(l)+1]] <- c("x","y")
> l

[[1]]
[1] 1 2 3 4

[[2]]
[1]  TRUE  TRUE FALSE

[[3]]
[1] "x" "y"
```

Naming List Elements

We can use the `names()` function to add names to list elements.

```
> names(l) <- c("numbers", "booleans")
> l

$numbers
[1] 1 2 3 4

$booleans
[1]  TRUE  TRUE FALSE

$<NA>
[1] "x" "y"
```


Data Frames

Data frames can be used for grouping data. A data frame is a list of vectors of the same length.

```
> year <- c(1800,1850,1900,1950,2000)
> pop <- c(18,54,500,1701,7731)
> demography <- data.frame(y=year, p =pop)
> demography

     y    p
1 1800   18
2 1850   54
3 1900  500
4 1950 1701
5 2000 7731
```


Implicit Loops Using Apply

- `apply(x, margin, fun)` returns a vector or array or list of values obtained by applying a function to margins of an array or matrix
- Margin: vector giving the subscripts which the function will be applied over
- For example, for a matrix 1 indicates rows, 2 columns, c(1, 2) rows and columns, or in matrices named dimnames
- Implicit Loops Using sapply(), lapply(), and tapply()
  - `sapply()` returns a simplified result (vector or matrix),
  - `lapply()` returns a list, and
  - `tapply()` creates a table

Implicit Loops Using lapply()

List of car data:

```
> cars <- list(speed=c(180, 250, 300),
+ price = c(10.5, 55.6, 76.0),
+ consumption=c(5, 7.1, 12.5))
> cars

$speed
[1] 180 250 300

$price
[1] 10.5 55.6 76.0

$consumption
[1]  5.0  7.1 12.5
```

Consider the `sum()` function.

```
> lapply(cars, sum)

$speed
[1] 730

$price
[1] 142.1

$consumption
[1] 24.6
```

Implicit Loops Using Anonymous Functions

An anonymous function can be used as well

```
> lapply(cars, function(x) return(x[2]))

$speed
[1] 250

$price
[1] 55.6

$consumption
[1] 7.1
```


sapply()

Given a list structure x, the function `unlist()` simplifies it to produce a vector. In order to obtain a vector of mean values instead of a list using the `lapply()` function, the following command can be used.

```
> unlist(lapply(cars, mean))

     speed       price consumption
 243.33333    47.36667     8.20000
```

Using `sapply()`, the same result can be obtained directly.

```
> sapply(cars, mean)

     speed       price consumption
 243.33333    47.36667     8.20000
```

For example, to apply `mean()` to each of the iris data set columns:

```
> data(iris)
> sapply(iris, mean)

Sepal.Length  Sepal.Width Petal.Length  Petal.Width      Species
    5.843333     3.057333     3.758000     1.199333           NA
```


Sorting

We can use the `sort()` function to sort a vector.

```
> sort(iris$Sepal.Length)[1:10]

 [1] 4.3 4.4 4.4 4.4 4.5 4.6 4.6 4.6 4.6 4.7
```

The `order()` function generates a vector of the indices of the sorted values.

```
> a <- c(20,-1,4,3)
> order(a)

[1] 2 4 3 1
```

Here, the smallest value "1" is at position 2, the next at position 4, whereas the largest value is at position 1.

Sorting Using order()

```
> a

[1] 20 -1  4  3

> i <- order(a)
> i

[1] 2 4 3 1

> a[i]

[1] -1  3  4 20
```

Sorting a set of variables according to the values of some other variables:

```
> i <- order(iris$Sepal.Length)
> options(width=70)
> iris[i,][1:4,]

   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
14          4.3         3.0          1.1         0.1  setosa
9           4.4         2.9          1.4         0.2  setosa
39          4.4         3.0          1.3         0.2  setosa
43          4.4         3.2          1.3         0.2  setosa
```

Simple Formulas

```
> celsius<-(0:4)*10
> fahrenheit <- 9/5*celsius+32
> conversion <- data.frame(c=celsius, f=fahrenheit)
> print(conversion)

   c   f
1  0  32
2 10  50
3 20  68
4 30  86
5 40 104
```


Control Structures

Try using functions from the `apply(x)` family instead of loops. `system.time()` returns CPU (and other) times used by process. First, an implementation without loops.

```
> require(stats)
> x <- 1:1000000
> system.time(y <- x^2)

   user  system elapsed
   0.02    0.01    0.03
```

Next, the `for()` function to perform calculation with loops.

```
> x <- 1:1000000
> system.time(
+ for( i in 1:length(x)) y[i] <- x[i]^2
+ )

   user  system elapsed
   2.49    0.02    2.53
```


Loops with for() and while()

Loops can be generated with the `for()` function as follows.

```
> for( i in 1:5) print(i)

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

We can also use the `while()` command.

```
> x<-1
> while(x <= 5){
+ print(x)
+ x <- x+1
+ }

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


Loops with repeat()

Alternatively, the `repeat()` command can be used in combination with the `break()` command.

```
> x <- 1
> repeat{
+ print(x)
+ x<-x+1
+ if (x > 5) break
+ }

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

R Functions and Arguments

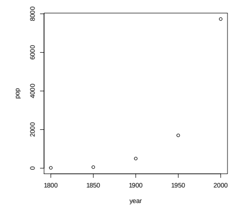
R has many pre-defined functions, e.g., `mean`, `sum`, or `range`.

```
> year <- c(1800,1850,1900,1950,2000)
> pop <- c(18,54,500,1701,7731)
> range(year)
[1] 1800 2000
> summary(year)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   1800    1850    1900    1900    1950    2000
```


Arguments and Positional Matching

R uses positional matching, i.e., the n-th argument corresponds to the n-th function variable. For example, `plot()` assumes: 1st argument (year) corresponds to x, whereas 2nd (population) corresponds y

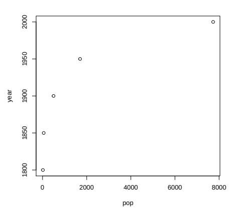
```
> plot(year, pop)
```



Named Actual Arguments

- Positional matching: very simple concept, becomes unhandily if many arguments occur
- R can handle *named actual arguments*, i.e., names are matched against their formal arguments
- Output from `plot()` with x and y values exchanged:

```
> plot(y=year, x=pop)
```



Writing R Functions

Using R's `function()` function, we can write our own functions. Note, %*% denotes matrix multiplication

```
> norm <- function(x) sqrt(x%*%x)
> norm(1:4)
         [,1]
[1,] 5.477226
```

Curly braces can be used to define the body of the function.

```
> h <- function(x){
+  if (x<0)  -1
+  else 1}
> h(1)
[1] 1
```

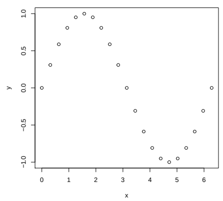
Note, we can use the command `ifelse()` as well.

```
> heaviside <- function(x) ifelse(x<0,-1,1)
> heaviside(1)
[1] 1
```

Graphics: The Basic Plot Command `plot(x,y)`

- The basic plot command is `plot(x,y)`
- Alternatively, `plot(y ~ x)` can be used

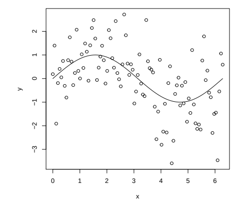
```
> x <- (0:20)*pi/10
> y <- sin(x)
> plot(y~x)
```



Combining Plots

- Add lines to this plot using the function `lines()`

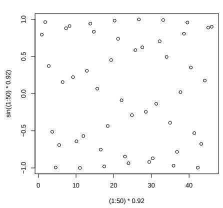
```
> n<-100
> x <- (0:n)*2*pi/100
> y <- sin(x)+rnorm(n+1)
> plot(y~x)
> lines(x,sin(x))
```



Modifying the Layout

- `par()` modifies layouts, e.g., margin sizes, line widths and types, colors, clipping, character sizes and fonts

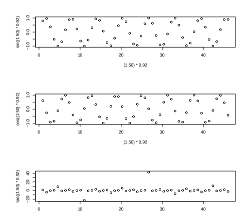
```
> plot( (1:50)*0.92, sin( (1:50)*0.92))
```



Modifying the Layout

- Now we plot the same figure again with a modified layout. In addition, two new figures are plotted

```
> par(mfrow=c(3,1))
> plot( (1:50)*0.92, sin( (1:50)*0.92))
> plot( (1:50)*0.92, cos( (1:50)*0.92))
> plot( (1:50)*0.92, tan( (1:50)*0.92))
```



Importing from Text Files

- `read.table()` is an easy to use method to importing data from a simple text file
- Simple test file, say "simple.txt":

```
x y
1 2
2 4
3 6
4 8
```

```
> df.simple <- read.table("simple.txt", header = TRUE)
> df.simple
```

```
  x y
1 1 2
2 2 4
3 3 6
4 4 8
```

- The result of the `read.table()` is a data frame.

Exporting to Text Files

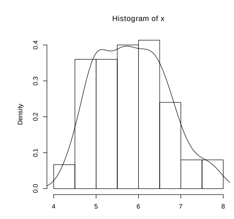
- `write.table()` prints its required argument x (after converting it to a data frame if it is not one nor a matrix) to a file or connection
- `write.csv()` and `write.csv2()` provide convenience wrappers for writing CSV files

Exploratory data analysis (EDA): Overview

- Idea: *let the data speak for themselves*
- Use of human brain's abilities as a pattern recognition device
- Reveal new information ("playing trumpet to the tulips")
- Ways how explore data prior to a formal analysis
- Standard tools:
  - Histograms and density plots
  - Stem-and-leaf plots
  - Scatter plots
  - Lattice: lowess smoother, trellis graphics
- Histograms: graphical representations of the frequency distribution of sets of data
- Areas of the plotted rectangles proportional to the number of observations with values within rectangle width
- Add density curves, they do not rely on breakpoints

Histograms and Density Plots

```
> data(iris)
> x <- iris$Sepal.Length
> dens <- density(x)
> hist(x,freq=F)
> lines(dens)
```



Stem-and-leaf plots

- The stem is on the left, leaves are on the right
- Smallest value reads 42. The value 44 appears four times

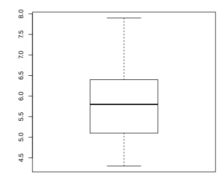
```
> stem(iris$Sepal.Length)
The decimal point is 1 digit(s) to the left of the |

42 | 0
44 | 0000
46 | 000000
48 | 00000000000
50 | 0000000000000000000
52 | 00000
54 | 0000000000000
56 | 00000000000000
58 | 0000000000
60 | 000000000000
62 | 0000000000000
64 | 000000000000
66 | 0000000000
68 | 0000000
70 | 00
72 | 0000
76 | 00000
78 | 0
```


Boxplots

- Boxplots summarize graphically the following information:
  - Outliers
  - Smallest and largest value (outliers excluded)
  - Lower and upper quantile
  - Median

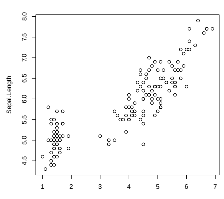
```
> boxplot(iris$Sepal.Length)
```



Scatterplots

- Simple but effective tool for the analysis of pairwise relationships

```
> plot(Sepal.Length~Petal.Length,data=iris)
```



What to Look for in Plots: Outliers

- Points that appear to be isolated from the main region of the data are called outliers
- Outliers can distort models to be fit to the data
- But there is no general definition for outliers
- This definition depends on our view of the data
- Boxplots are useful to detect outliers in one dimension, scatterplots are useful in two dimensions
- However, sometimes outliers will be apparent only in three or more dimensions.

What to Look for in Plots

- Asymmetry
  - Most asymmetric distributions are positively or negatively skewed
  - Positively skewed distributions can be characterized as follows: There is a long tail to the right, values near the minimum are bunched up together, and the largest values are widely dispersed
- Different variabilities
  - Sometimes variability increases as data values increase
  - Then the logarithmic transformation can be helpful
- Clustering
  - Outliers can be considered as a special form of clustering
  - Clusters may suggest structures in the data which may or may not have been expected
  - Scatterplots can be useful to detect clusters.
- Non-linearity
  - Linear models should not be fitted to data where relationships are non-linear

Distributions

Example (Binomial distribution)

- Determine five random numbers following a binomial $(100, 1/5)$ distribution

```
> set.seed(123)
> rbinom(5, size = 100, p=1/5)
[1] 18 23 19 25 26
```

- Hundred samplings with replacement from a box with 64 black and 16 red balls. Probability of drawing a red ball is $p = 16/(64 + 16) = 1/5$.

- Probability that ten red balls are drawn, i.e., $P(X = 10)$

```
>  dbinom(10,100,1/5)
[1] 0.00336282
```

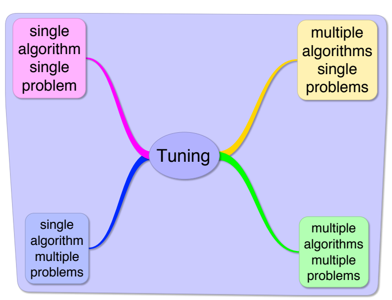
- CDF, i.e., compute $P(X \leq 10)$

```
> sum(dbinom(0:10, 100, 1/5))
[1] 0.005696381
> pbinom(10,100,1/5)
[1] 0.005696381
```


A Taxonomy of Algorithm and Problem Designs

- Classify parameters
- Parameters may be *qualitative*, like for the presence or not of an recombination operater or *numerical*, like for parameters that assume real values
- Our interest: understanding the contribution of these components
- Statistically speaking: parameters are called *factors*
- The interest is in the effects of the specific *levels* chosen for these factors

Problems and Algorithms



- How to perform comparisons?
- Adequate statistics and models?

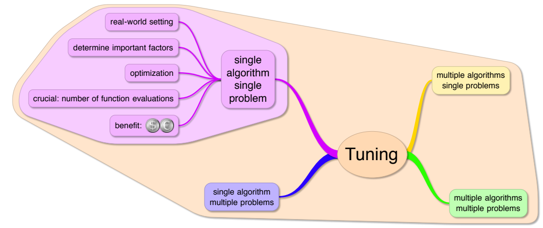
SASP: Algorithm and Problem Designs

- Basic design: assess the performance of an *optimization algorithm* on a single problem instance π
- Randomized optimization algorithms $\Rightarrow$ performance Y on one instance is a random variable
- Experiment: On an instance π algorithm is run r times $\Rightarrow$ collect sample data $Y_1, \dots, Y_r$ (independent, identically distributed)
- One instance π, run the algorithm r times $\Rightarrow$ r replicates of the performance measure Y, denoted by $Y_1, \dots, Y_r$
- Samples are conditionally on the sampled instance and given the random nature of the algorithm, independent and identically distributed (i.i.d.), i.e.,

$$p(y_1, \dots, y_r|\pi) = \prod_{j=1}^{r} p(y_j|\pi). \tag{1}$$

- Y might be described by a probability density/mass function $p(y|\pi)$

SASP – Single Algorithm, Single Problem



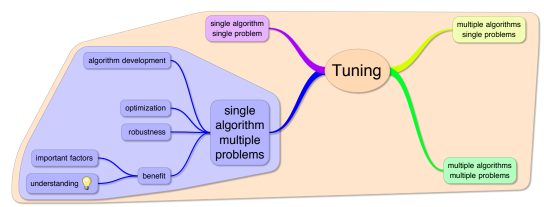
SAMP: Algorithm and Problem Designs

- Multiple problem instances occur if optimization problems have a set of input data which instantiate the problem
- Experiment: collect sample data $Y_1, \dots, Y_R$ (independent, identically distributed)
- Goal: Drawing conclusions about a certain *class* or *population* of instances Π
- Single algorithm, multiple problems: performance Y of the algorithm on the class Π is described by the probability function

$$p(y) = \sum_{\pi \in \Pi} p(y|\pi)p(\pi), \tag{2}$$

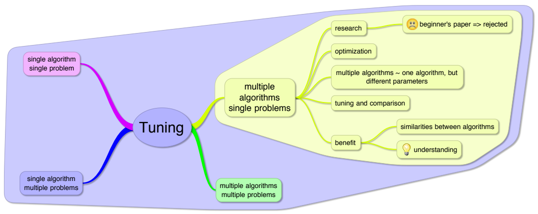
with $p(\pi)$ being the probability of sampling instance π
- In other terms, we are interested in the distribution of Y marginalized over the population of instances

SAMP – Single Algorithm, Multiple Problems



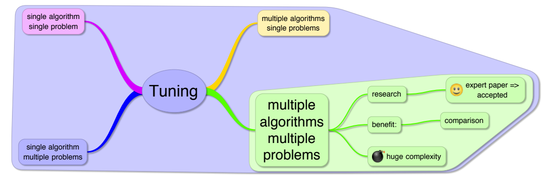
MASP: Algorithm and Problem Designs

- Several optimization algorithms are compared on one fixed problem instance π
- Experiment: collect sample data $Y_1, \ldots, Y_R$ (independent, identically distributed)
- Goal: comparison of algorithms on one (real-world) problem instance π



MAMP: Fixed Algorithm and Problem Designs

- Typically:
  - Take a few, *fixed* instances for the problem at hand
  - Collect the results of some runs of the algorithms on these instances
- Statistically, instances are also *levels of a factor*
- Instances treated as *blocks*
- All algorithms are run on each single instance
- Results are therefore *grouped* per instance



MAMP: Randomized Problem Designs

- Sometimes, several hundred (or even more) problem instances to be tested $\Rightarrow$ interest not just on the performance of the algorithms on a few specific instances, but rather on the generalization of the results to the entire population of instances
- Procedure: instances are chosen at random from a large set of possible instances of the problem
- Statistically, instances are also *levels of a factor*
- However, factor is of a different nature from the fixed algorithmic factors described above
- Levels are chosen at random and the interest is not in these specific levels but in the population from which they are sampled
- $\Rightarrow$ levels and the factor are *random*
- This leads naturally to a mixed model [4]

MAMP: Randomized Problem Designs

- Organize our presentation in different cases according to the number and type of factors involved
- Identify the cases with the following notation:

$$\left\langle \begin{array}{c}\text{algorithm}\\ \text{factors}\end{array}, \begin{array}{c}\text{number of}\\ \text{instances}\end{array} \left(\begin{array}{c}\text{instance}\\ \text{factors}\end{array} \right), \begin{array}{c}\text{number of}\\ \text{runs}\end{array} \right\rangle .$$

- Lower-case letters when referring to the number of factors, upper-case letters when referring to the number of levels
- Dash (-) indicates absence of fixed factors, round parenthesis indicates nesting
- Example: $\langle N, q(M), r \rangle$ means N algorithmic factors, q instances sampled from each combination of M instance factors, and r runs of the algorithm per instance

MAMP: Nested Linear Mixed Models

- In statistics, the effects described are modeled as linear combinations, and mathematical theory has been developed to make inferences about the populations on the basis of the results observed in the samples.
- The mixed nature of the factors leads to so-called *nested linear mixed models*
- Nontrivial designs, go beyond the classical multifactorial ANOVA, where all factors are instead treated as fixed
- Mathematical formula involved and the inference derived are different in the case of mixed-effects models and this may lead to a different inference
- [4] give an example where this difference clearly arises

Summary: A Taxonomy of Algorithm and Problem Designs

- Taxonomy combining ideas from [1] and [4]
- Experimental design notation:

$$\left\langle \begin{array}{c}\text{algorithm}\\ \text{factors}\end{array}, \begin{array}{c}\text{number of}\\ \text{instances}\end{array} \left(\begin{array}{c}\text{instance}\\ \text{factors}\end{array} \right), \begin{array}{c}\text{number of}\\ \text{runs}\end{array} \right\rangle.$$

- Case $\langle$-, q(-), $r\rangle$: Random-Effects Design: one algorithm is evaluated on q instances randomly sampled from a class Π
- Case $\langle N, q(\text{-}), r\rangle$: Mixed-Effects Design: h *algorithms* are evaluated on q *instances* randomly sampled from a class Π
- Case $\langle 1, 1(1), r\rangle$: Fixed-Effects Design: one algorithm is evaluated r times on one fixed instance π
- ...

Algorithm and Problem Designs

- Classify parameters
  - Continuous, categorical, etc.
- Designs
  - Factorial, fractional factorial, space filling, etc.
- Models
  - ANOVA, regression, kriging, tree-based models, etc.
- R packages for experimental designs: Groemping's CRAN Task View: Design of Experiments (DoE) & Analysis of Experimental Data `http://cran.r-project.org/web/views/ExperimentalDesign.html`

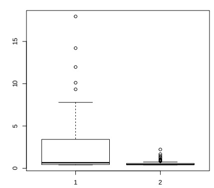
Comparison of Two Simulated Annealing Parameter Settings

- Case $\langle 2, 1(1), r\rangle$: Fixed-Effects Design: one *algorithm* is evaluated on one *instance* π (fixed), i.e., SASP

```
> set.seed(123)
> library(SPOT)
> fn <- spotBraninFunction #test function to be optimized by SANN
> x0 <- c(-2,3) #starting point that SANN uses when optimizing Branin
> maxit <- 100 #number of evaluations of Branin allowed for SANN
> temp <- 10
> tmax <- 10
> n <- 100
> y <- rep(1,n)
> y0<-sapply(y, function(x) x<-optim(par=x0, fn=fn, method="SANN"
+                                   , control=list(maxit=maxit,
+                                                  temp=temp, tmax=tmax))$value)
> temp <- 4
> tmax <- 62
> y <- rep(1,n)
> y1<-sapply(y, function(x) x<-optim(par=x0, fn=fn, method="SANN"
+                                   , control=list(maxit=maxit,
+                                                  temp=temp, tmax=tmax))$value)
```

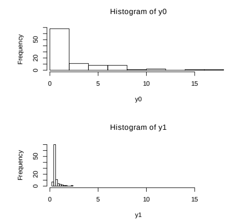
Comparison: Simple EDA Using Boxplots

```
> summary(y0)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 0.3984  0.4444  0.6587  2.2770  3.4020 17.9600
> summary(y1)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 0.3985  0.4150  0.4439  0.5609  0.5736  2.2250
> boxplot(y0,y1)
```



Comparison: Simple EDA Using Histograms

```
> par(mfrow=c(2,1))
> hist(y0,xlim = c( min(y0,y1), max(y0,y1)))
> hist(y1,xlim = c( min(y0,y1), max(y0,y1)))
> par(mfrow=c(1,1))
```

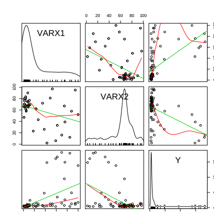


Simple EDA: Let the Data Speak

```
> df1 <- read.table("Data.d/NULL.res",header=T)
> options(width=80)
> df1[1:10,]
                 Function XDIM YDIM STEP SEED CONFIG VARX1 VARX2          Y
1  UserSuppliedFunction    2    1    0 1234      1    23    62  0.5183426
2  UserSuppliedFunction    2    1    0 1235      1    23    62  0.4020790
3  UserSuppliedFunction    2    1    0 1234      2    62    33  3.5002149
4  UserSuppliedFunction    2    1    0 1235      2    62    33 16.6525805
5  UserSuppliedFunction    2    1    0 1234      3    38    26  4.7735424
6  UserSuppliedFunction    2    1    0 1235      3    38    26  0.3987177
7  UserSuppliedFunction    2    1    0 1234      4    70    16  1.4725001
8  UserSuppliedFunction    2    1    0 1235      4    70    16 18.1272253
9  UserSuppliedFunction    2    1    0 1234      5     8    90  0.5871467
10 UserSuppliedFunction    2    1    0 1235      5     8    90  0.6200017
```


Analysis: Simple EDA Using Scatterplots

```
> library(car)
> scatterplotMatrix(~VARX1+VARX2+Y, reg.line=lm, smooth=TRUE,
+  spread=FALSE, span=0.5, diagonal = 'density', data=df1)
```



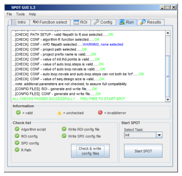
Sequential Parameter Optimization SPO

Use statistical techniques and methods from design of experiment to solve optimization problems.

1. Take initial samples from design space and evaluate on target function/algorithm
2. Build surrogate model (Linear, Tree-based, Kriging, ...) based on known evaluations
3. Determine promising new solutions with model
4. Evaluate new solutions
5. If termination criterion not reached: go to **2.**
6. Summarize Results / Create Report

SPO Toolbox (SPOT)

- Currently maintained and developed as an R-Package
- Interfaces to several other R-packages
- Provides Demos and Documentation
- Graphical User Interface
- Alternative version is available for matlab



SPOT: Installation, Help, Demos

- Install from CRAN:
  `> install.packages("SPOT")`
- Load package to Workspace:
  `> require("SPOT")`
- Get help on some spot functions
  `> ?spot`
  `> ?spotOptim`
- Get a list of SPOT demos
  `> demo(package="SPOT")`
- Run a SPOT demo
  `> demo("spotDemo18ForresterOptim",ask=F)`
- Start the GUI
  `> spotGui()`

Applications: algorithms tuned by SPOT

- Several types of evolution strategies
- Time series prediction and anomaly detection
- Classification
- Symbolic Regression
- Simulated Annealing
- For more applications see [2]

Simulated Annealing SANN

- Randomized optimization algorithm
- Two parameters: starting temperature TEMP and number of function evaluations at each temperature TMAX
- implementation used: optim, part of R-base

```
> #Find minimum of 2D-sphere function with SANN
> fn<-function(x){return(sum(x^2))}
> result<-optim(par=c(2,-4),fn,method="SANN")
> result$value
```

```
[1] 0.0002277956
```

```
> result$par
```

```
[1] 0.006771081 0.013488814
```


Tuning SANN: Define Problem to solve

- Target function: Branin-Function (2-D function with three global minima)

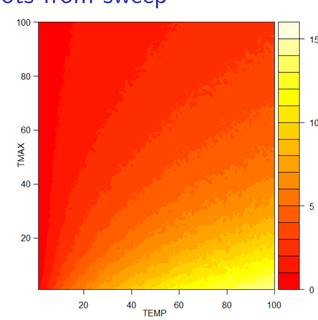
```
> require(SPOT)
> fn <- spotBraninFunction #test function to be optimized by SANN
> x0 <- c(-2,3) #starting point that SANN uses when optimizing Branin
> maxit <- 100 #number of evaluations of Branin allowed for SANN
> testalgorithm <- function(pars,x0,fn,maxit){
+  temp<-pars[1]
+  tmax<-pars[2]
+  y <- optim(x0, fn, method="SANN",
+  control=list(maxit=maxit,
+  temp=temp, tmax=tmax))
+  return(y$value)
+ }
```

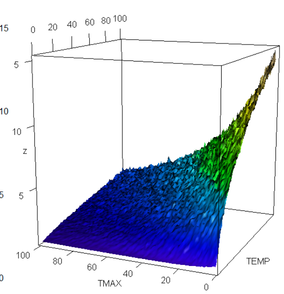

SANN sweep

- Since this is a simple test problem: Complete sweep
- Understand underlying fitness shape
- 1000 repeats for each setting (takes rather long)

```
> target <- function(x,y,x0,fn,maxit){
+  zz<-matrix(0,length(x))
+  repeats=1000
+  for(i in 1:repeats){
+  set.seed(i)
+  zz =zz + apply(cbind(x,y),1,testalgorithm,x0=x0,fn=fn,maxit=maxit)
+  }
+  return(zz/repeats)
+ }
> x <- seq(1, 100, length.out = 100)
> y <- x
> z <- outer(x, y, target,x0=x0,fn=fn,maxit=maxit)
> filled.contour(x, y, z, color.palette=heat.colors,xlab="temp",ylab="tmax")
> pal <- topo.colors(100)
> require(rgl)
> persp3d(x,y,z,col=pal[cut(z,100)],xlab="TEMP",ylab="TMAX")
```


Plots from sweep





Tuning SANN: Configure SPOT

- ROI: Region of interest, in which parameters are tuned
- Surrogate: Kriging based on Forrester et. al. [6]
- Settings are minimalistic (uses a lot of default values)

```
> roi<-spotROI(c(1,1),c(100,100),type=c("INT","INT"))
> config<-list(alg.func=testalgorithm,
+  alg.roi=roi,
+  init.design.size=20,
+  seq.predictionModel.func="spotPredictForrester",
+  seq.predictionOpt.func="spotPredictOptMulti",
+  seq.predictionOpt.method="cmaes",
+  seq.predictionOpt.budget=1000,
+  report.func="spotReportSens",
+  spot.fileMode=T,
+  io.verbosity=3,
+  auto.loop.nevals=100)
```


Tuning SANN: Run SPOT

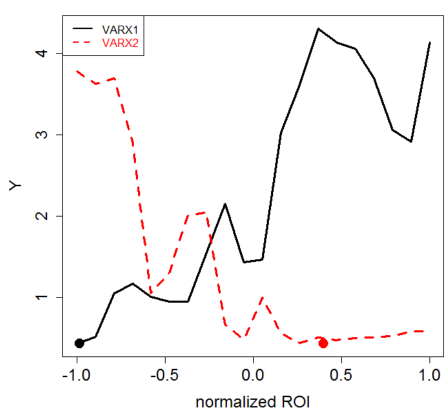
- Pass configuration to SPOT
- Pass additional parameters to SPOT, needed by target function

```
> res<-spot(spotConfig=config,x0=x0,fn=fn,maxit=maxit)

Sensitivity plot for this ROI:
      lower upper type     BEST
VARX1     1   100  INT  1.84744
VARX2     1   100  INT 70.36899

Best solution found with 103 evaluations:
           Y   VARX1    VARX2 COUNT CONFIG
245 0.409769 1.84744 70.36899     5     24

Standard deviation of best solution:
0.409769033349987 +- 0.0112398099327513
```

Tuning SANN: Raw results

- Result file, logged information separated by space

```
Function XDIM YDIM STEP SEED CONFIG VARX1 VARX2 Y
UserSuppliedFunction 2 1 0 1234 1 23 62 0.518342556082896
UserSuppliedFunction 2 1 0 1235 1 23 62 0.402079045134601
UserSuppliedFunction 2 1 0 1234 2 62 33 3.50021485407806
```

- Results in R command line

```
str(res$alg.currentResult)

'data.frame':   103 obs. of  9 variables:
 $ Function: Factor w/ 1 level "UserSuppliedFunction": 1 1 1 1 1 1 1 1 1 1 ...
 $ XDIM    : num  2 2 2 2 2 2 2 2 2 2 ...
 $ YDIM    : int  1 1 1 1 1 1 1 1 1 1 ...
 $ STEP    : int  0 0 0 0 0 0 0 0 0 0 ...
 $ SEED    : num  1234 1235 1234 1235 1234 ...
 $ CONFIG  : int  1 1 2 2 3 3 4 4 5 5 ...
 $ VARX1   : num  23 23 62 62 38 38 70 70 8 8 ...
 $ VARX2   : num  62 62 33 33 26 26 16 16 90 90 ...
 $ Y       : num  0.518 0.402 3.5 16.653 4.774 ...
```

Tuning SANN: Other report functions

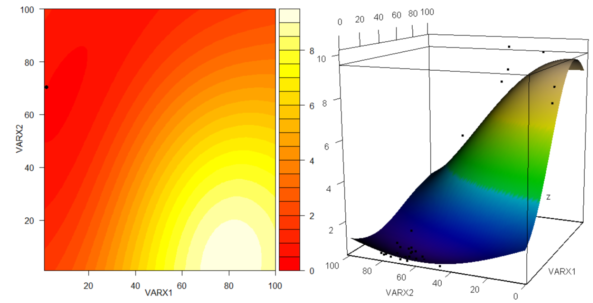
- Other reports/graphics can be created
- spotReportContour for a contour plot

```
> spot(spotConfig=append(list(
+  report.func="spotReportContour",
+  report.interactive=F),
+  res),
+  spotTask="rep")
```

- spotReport3d for 3d plot

```
> spot(spotConfig=append(list(
+  report.func="spotReport3d",
+  report.interactive=F),
+  res),
+  spotTask="rep")
```


Plots from SPOT



Existing features

- Single and multi criteria optimization
- Automated tuning, or manual steps
- modular concept: Use different combinations of models / methods
- Available surrogate models: Linear, Tree, Kriging, Support Vector Machine, Random Forest, ...
- Tuning real valued parameters as well as factors (i.e. with tree-based models)
- User can use custom models
- Different means of budget allocation
- Logging and Report generation

Development

- Extend report functions
- Implementation of ensembles of surrogate models
- Improve multi criteria optimization
- Adaptive ROI
- New test problems or applications

Overview

We will focus on:

- Available optimization algorithms
- Benchmarking resources

But first a very short glimpse on our targets. . .

The Adaptability Perspective

When adapting algorithms to a problem (or multiple), two things are of basic interest [11]:

- How good do we get?
- How long does it take to get there?

What to do with that?

- We can expect that different algorithms have different properties
- It depends on the optimization context which one is more important (algorithm selection problem)
- We encourage to further look at these aspects (together)

Optimization Algorithms

CRAN Task View: Optimization and Mathematical Programming
`http://cran.r-project.org/web/views/Optimization.html`

- Huge list of available algorithms
- Also: Mathematical programming solvers
- We focus on (some) general purpose continuous solvers
- You can also deliver your implementations there (to Stefan Theussl)

Evolutionary Methods Packages

cmaes	Covariance matrix adaptation evolution strategy
genalg	Genetic algorithm
rgenoud	GA plus quasi-Newtonian approach hybridization
pso	Particle swarm optimization
DEoptim	Differential evolution

Other Interesting Methods

optim (built-in function of the stats package) Broyden-Fletcher-Goldfarb-Shanno (BFGS) method, bounded BFGS, conjugate gradient, Nelder-Mead, and simulated annealing (SANN)

optimx new common frame for optim() methods and many more, e.g. bobyqa, uobyqa, and newuoa

nloptr supports several global optimization routines (e.g. DIRECT), local derivative-free and gradient-based (e.g. BFGS) methods used as subroutines

And many more, even interfaces to solvers (COIN-OR, CPLEX)

Benchmarking: BBOB

Black-Box Optimization Benchmarking (BBOB) 2012 library
`http://coco.gforge.inria.fr/doku.php?id=bbob-2012`
(see the GECCO workshop)

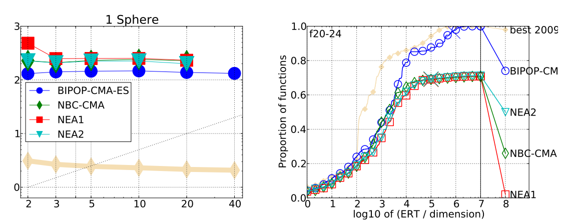
- 24 selected problems
- Interfaces from Matlab, C, Java, R, Python
- Lots of already existing results to compare with (BBOB 2009, BBOB 2010)
- Very powerful visualization for free (Python-based post-processing)
- You can also just use the problems

BBOB Function Overview

Function groups:

- Separable (sphere, ellipsoidal, Rastrigin, Büche-Rastrigin, linear slope)
- Low or moderate conditioning (attractive sector, step ellipsoidal, Rosenbrock original, Rosenbrock rotated)
- High conditioning, unimodal (ellipsoidal, discus, bent cigar, sharp ridge, different powers)
- Multi-modal with global structure (Rastrigin, Weierstrass, Schaffers F7, Schaffers F7, moderately ill-conditioned, Composite Griewank-Rosenbrock F8F2)
- Multi-modal with weak global structure (Schwefel, Gallagher's Gaussian 101-me Peaks, Gallagher's Gaussian 21-hi Peaks, Katsuura Function, Lunacek bi-Rastrigin)

BBOB Sample Graphics



Real-World Problems

Noisy real-world test cases (as e.g. used in [12])
http://ls11-www.cs.tu-dortmund.de/rudolph/kriging/applications

Currently available:

- Gaming related: Car setup optimization (related to the former competition)
- Hydrogeologic Testcase: well placement
- More to come (hopefully)
- If you have other interesting problems, let us know

Acknowledgments

- This work has been supported by the Federal Ministry of Education and Research (BMBF) under the grants FIWA (AIF FKZ 17N1009) and CIMO (FKZ 17002X11)

Thomas Bartz-Beielstein.
Experimental Research in Evolutionary Computation—The New Experimentalism.
Natural Computing Series. Springer, Berlin, Heidelberg, New York, 2006.

Thomas Bartz-Beielstein.
Sequential parameter optimization—an annotated bibliography.
CIOP Technical Report 04/10, Research Center CIOP (Computational Intelligence, Optimization and Data Mining), Cologne University of Applied Science, Faculty of Computer Science and Engineering Science, April 2010.

Radomil Binek.
Iris setosa (photo).
Retrieved March 24, 2001, from Wikimedia Commons Web site: http://en.wikipedia.org/wiki/File:Kosaciec_szczecinkowaty_Iris_setosa.jpg, 2005.
Licensed under the GFDL.

Marco Chiarandini and Yuri Goegebeur.

Mixed models for the analysis of optimization algorithms.
In Thomas Bartz-Beielstein, Marco Chiarandini, Luís Paquete, and Mike Preuss, editors, *Experimental Methods for the Analysis of Optimization Algorithms*, pages 225–264. Springer, Germany, 2010.
Preliminary version available as *Tech. Rep.* DMF-2009-07-001 at the The Danish Mathematical Society.

Ronald A. Fisher.
The use of multiple measurements in taxonomic problems.
Annals Eugen., 7:179–188, 1936.

Alexander Forrester, Andras Sobester, and Andy Keane.
Engineering Design via Surrogate Modelling.
Wiley, 2008.

Eric Guinther.
Primrose willowherb ludwigia octovalvis(photo).
Retrieved April 25, 2012, from Wikimedia Commons Web site: http://en.wikipedia.org/wiki/File:Petal-sepal.jpg, 2012.

Licensed under the Creative Commons Attribution-Share Alike 2.0 Generic license.

Danielle Langlois.
Iris versicolor (photo).
Retrieved March 24, 2001, from Wikimedia Commons Web site: `http://en.wikipedia.org/wiki/File:Iris_versicolor_3.jpg`, 2005.
Licensed under the Creative Commons Attribution-Share Alike 3.0 Unported license.

J. Maindonald and J. Braun.
Data Analysis and Graphics using R—an Example-based Approach.
Cambridge University Press, Cambridge UK, 2003.

Frank Mayfield.
Iris virginica (photo).
Retrieved March 24, 2001, from Wikimedia Commons Web site: `http://en.wikipedia.org/wiki/File:Iris_virginica.jpg`, 2007.

Tutorial: Statistical Analysis of Optimization Algorithms with R
└ Acknowledgments

Licensed under the Creative Commons Attribution-Share Alike 2.0 Generic license.

Mike Preuss.
Adaptability of algorithms for real-valued optimization.
In Mario Giacobini et al, editor, *Applications of Evolutionary Computing, EvoWorkshops 2009. Proceedings*, volume 5484 of *Lecture Notes in Computer Science*, pages 665–674. Springer, 2009.

Mike Preuss, Tobias Wagner, and David Ginsbourger.
High-dimensional model-based optimization based on noisy evaluations of computer games.
In *LION 6, Learning and Intelligent OptimizatioN Conference*, Paris, France, 2012. Springer LNCS.

Bartz-Beielstein, Preuss, Zaefferer | Tutorial: Statistical Analysis of Optimization Algorithms with R | 106/106