1 Matlab

Matlab a very common mathematical program package that is quite easy to use. The program has many built in functions and, as it is widely spread, one can search the internet for additional free software libraries.

When using Matlab, one should get used to employ its built in help function. That help function can be reached simply by writing `help subject`. For example, `help stats` gives a list of all function in the statistical toolbox, while `help hist` gives help on the function `hist`. In addition, one can make use of the commands `helpdesk` and `helpwin` (or going to **Help** → **Product help**) for easy access to help.

When using Matlab, one should make use of an editor, rather than writing commands directly in the Matlab window. The reason for this is that, when writing long programs, it is convenient to have everything in "one place", easily accessible and editable.

One may use the built in editor of Matlab. To run a Matlab program, written with emacs, save the code to a m-file, called `foo.m`, say.[1] Notice that one have to save the file to the directory that Matlab is run from, in order to avoid specifications of paths.

When plotting, it is nice to have colourfull graphs. However, when you print those graphs, they will usually be black and white. Hence it is a good practice to design the graphs, so that they work well also in black and white. One example of this is to use solid, dotted and dashed lines, etc. to distinguish different graphs, rather than different colours only. It can be quite useful, and is simple, to add a Matlab *legend* to a plot.

Sometimes, Matlab can be really slow especially when one uses loops. Always try to avoid loops, when there is an alternative! For example, sometimes a (slow) loop can be replaced with a (quicker) vector multiplication.

2 Mathematica

Mathematica is a very versatile mathematics program package. As in virtually any program, there is a helpbrowser, which is activated with punching *help*: One should get used to make use of this helpbrowser!

All Mathematica commands begin with a capital letter. Arguments of Mathematica functions are given within brackets, for example, the sample mean of `data` is calculated with `Mean[data]`. A command is executed with *shift+return* (alternatively the "right" *return* that is found on the Numpad).

A very important tip is to always use the command `Clear[Symbol]`, before defining `Symbol`, in order to avoid that old conflicting definitions "disturb" new definition. Sometimes, a simple Clear is not enough and one will need to use `Remove["Global‘*"]`.

Mathematica can fabricate plots in *.eps/.ps format* in several ways: One way is to mark the image by left clicking on it, and right click in the frame that occurs around the image. Select *File* in the menu, and then *Print Selection*. If you do not want headers or footers on

[1]If a Matlab m-file is a function, then one will not have access to objects created within that function, as they are well-defined "locally" only, inside that function.

the image, they can be removed by clicking *File* followed by *Headers and Footers*, and then select *No header on first page* or *No footers on first page*, respectively. Another way is to name the plot, *name=Plot[...]*, and then just type *Export["image.eps",name]*.

Here is an example of a Mathematica program:

```
In[1]:= Clear[w]; w:={w1,w2,w3}
In[2]:= Clear[f]; f[w_]:=Sum[Sqrt[w[[i]]],{i,1,3}]
In[3]:= NMaximize[Flatten[{f[w],Table[w[[i]]>=0,{i,1,3}],Sum[w[[i]],
            {i,1,3}]==1}],w]
Out[3]= {1.73205,{w1->0.333333,w2->0.333333,w3->0.333333}}
```

which could also have been written

```
In[4]:= Clear[f]; f[w1_,w2_,w3_]:=Sqrt[w1]+Sqrt[w2]+Sqrt[w3]
In[5]:= NMaximize[{f[w1,w2,w3],w1>=0,w2>=0,w3>=0,w1+w2+w3==1},
            {w1,w2,w3}]
Out[5]= {1.73205,{w1->0.333333,w2->0.333333,w3->0.333333}}
```

Also, if one would like to manipulate matrices the function *Table[...]* is very useful.

3 *R*

R is a program language for general statistic data treatment, including simulation. It is one of the most versatile statistical program packages available. It combines powerful traditional statistics that challenge the "colosses" of the market, such as, for example, *SAS* and *SPSS*, with a general programing function, for example, for stochastic simulations, that is superior to those of SAS and SPSS.

On Linux systems *R* is started by entering

```
dali> R
```

in the Terminal window.

R has two kinds of basic commands; expressions which are evaluated directly

```
> sqrt(2)
[1] 1.414214
```

and the operators <- and -> to assign values

```
> x <- 3^0.5; x
[1] 1.732051
```

Notice that several commands can be given on the same row, if separated by ;

```
> x <- log(2); exp(x) -> y; x; y; as.integer(x); as.integer(y)
[1] 0.6931472
```

```
    [1] 2
    [1] 0
    [1] 2
```

Incomplete commands result in the prompt + in stead of >

```
> sin(pi/4)*
+ 2^(-1/2)
[1] 0.5
```

A compiled assistance for a function is obtained via `args(function)`

```
    > args(logb)
    function(x, base)
    NULL
    > logb(3, 3)
    [1] 1
```

More thorough assistance is obtained with `help(function)` or `?  function`

```
    > help(gamma)
    Gamma Function (and its Natural Logarithm)
    DESCRIPTION:
    Returns the gamma function or the log of the gamma function.
    ...
```

To leave `help`, use `q` (for ”quit”).

R is a vector language. Functions of *R* operate on vectors, which, for example, can be created according to

```
    > c(pi/2, pi/4) -> x; y <- c(1, 1/sqrt(2)); z <-1:15*2
    > sin(x); y; sin(x)+y; cos(x)*y; z
    [1] 1.0000000  0.7071068
    [1] 1.0000000  0.7071068
    [1] 2.0000000  0.7071068
    [1] 6.123234e-17  5.000000e-01
    [1] 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30
```

(Note the priorities between : and ∗!) More examples of vector operations are

```
    > x <- c(rep(1, times=5), rep(2, times=5), rep(3, times=5); x
    [1] 1 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3
    > y <- x[1:5]; z <- c(x[1:4],x[10:11]); y; z
    [1] 1 1 1 1 1
    [1] 1 1 1 1 2 3
    > x[-7]
    [1] 1 1 1 1 1 2 2 2 2 3 3 3 3 3
```

A vector x can be reinterpreted as a matrix y, and *R* functions also works on matrices, componentvise (!!), as the following example on multiplications of *R* matrices illustrates:

```
> x <- c(1,2,3,4,5,6); matrix(x,2,3) -> y; y^2 -> z; y; z
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
     [,1] [,2] [,3]
[1,]    1    9   25
[2,]    4   16   36
```

Notice that the order in which the elements from the vector x appear in the matrix y is not what one typically would expected: This is a common source of programming errors in R!

It is easy to refer to single elemenst in vectors and matrices:

```
> x[4]; y[2,2]; z[2,1]
[1] 4
[1] 4
[1] 4
```

An element in an *atomic vector* is a number, character string or a logical value. *Recursive vectors*, more oftenly called *lists*, have some element or elements which in turn are vectors or matrices themselves. Lists can be created with `list`, and one can test whether a vector is recursive or atomic:

```
> vekt1 <- c(10,TRUE); vekt2 <- c(5,FALSE)
> lista <- list(vekt1,vekt2)
> is.atomic(vekt1); is.atomic(lista); is.recursive(lista)
[1] T
[1] F
[1] T
```

The logical functions and and or are written in the following way:

```
> (1<2) & (2<1); (1<2) | (2<1)
[1] F
[1] T
```

There are also logical equivalent versions of these logical operator, as follows:

```
> (1<2) && (2<1); (1<2) || (2<1)
[1] F
[1] T
```

When `&` is used, the logical value of both of the logical expressions which `&` are applied to are computed. Then the logical value under `&` is determined. When `&&` is used, the logical value of the expression to the left of `&&` is computed. If that value is false, the value of `&&` is set to false directly, without checking the values of the expression to the right of `&&`, which thus need not even be well-defined. Only if the expression on the left of `&&` is true, the expression on the right is called for.

The difference between `|` and `||` is analogous to that between `&` and `&&`.

Usually, `&` and `|` are used for common logical computations, while `&&` and `||` are used, for example, in connection with conditions to stop while sloops (see below), where occasionally only the expressions on the left of `&&` or `||` is well-defined.

Elements of lists can be referred to with double bracket paranthesises:

```
> lista[[1]]; lista[[1]][1]
[1] 10 1
[1] 10
```

Notice that `TRUE` has been written as 1.

Elements of lists can be given names in the following manner:

```
> ny <- list(no1=vekt1,no2=vekt2); vekt <- ny$no2; vekt[1]; vekt[2]
[1] 5
[1] 0
```

Many *R* functions, for example, linear regression `lm`, have a list as their output values. Using `$`, one may refer to different list elements, for example, `lm$coef` gives the regression coefficients.

Files can be read to *R* with `scan`. For example, a file *D4-fil1.dat*, with contents

```
1 4
9 16 25
```

can be read in the following way:

```
> x <- scan("D4-fil1.dat"); sqrt(x)
[1] 1 2 3 4 5
```

Observe that " " is used around the file name!

Of course, for files which are not placed in the directory from which *R* was started, a path to the correct directory has to be specified: For example, a file *D4-fil2.dat*, with contents

```
1 4
9 16
```

placed in the directory */users/mdstud/stada/lab_grupper/stada-??*, can be read with

```
> y <- scan("/users/mdstud/stada/lab_grupper/stada-??/D4-fil2.dat")
> sqrt(y)
[1] 1 2 3 4
```

The function `punif(x,min,max)` gives the distribution function for a uniform distribution over the interval `[min,max]`, and `qunif(x,min,max)` its invers. Further, `runif(n,min,max)` generates n independent observations of such a random variable.

```
> punif(0.7,0,1); qunif(0.7,0,2); runif(5,1,2)
[1] 0.7
[1] 1-4
[1] 1-76225 1.927495 1.659281 1.727141 1.077847
```

In the same way, `pnorm(x,mean,sd)`, `qnorm(x,mean,sd)` and `rnorm(n,mean,sd)` give a normal distribution function, its inverse, and a normal distributed random variable, respectively.

See the *R*-manual on other examples of distributions that can be obtained with the commands `p.(.)`, `q.(.)` and `r.(.)`.

One can define ones own functions in S^+, in the following way:

```
my.func <- function(x,y,z) sin(x)*(y**2)*exp(-z); my.func(pi/2,1,0)
[1] 1
```

R offers an array of common statistical routines (disregard eventual warnings they tend to give you!):

```
> N <- rnorm(25,5,2); N
[1] 5.878507 5.722384 4.906421 6.930634 5.791734 3.180767 2.026125
[8] 7.274999 4.111786 5.673488 5.396242 2.414764 1.793812 6.724480
[15] 4.036401 4.533922 0.690289 5.12308 9.059758 8.625384 6.650143
[22] 3.799066 6.201977 2.6770640 8.795774
> mean(N); var(N); median(N)
[1] 5.120513
[1] 4.948436
[1] 5.396242
> min(N); quantile(N,0.25); quantile(N,0.5); quantile(N,0.75); max(N)
[1] 0.690289
25% 3.799066
50% 5.396242
75% 6.650144
[1] 9.059758
```

It is easy to write your own programs in *R*:

```
> x<-NULL; for (i in 1:10) {n<-1; while (n<i) {
+    n<-n+1; fac<-fac*n}; x<-c(x,fac)}; x
[1] 1 2 6 24 120 720 5040 40320 362880 3628800
```

Two of the most common programing errors in *R* are due to the fact that `:` does not function as typically is expected:

```
> 1:2*5; 1:(-1)
[1] 5 10
[1] 1 0 -1
```

Notice that the operation `:` has the highest priority! And that the list `m:n` is not empty when $m > n$.

R graphics are displayed in a special graphic window. For example, two vectors x and y, of the same length, are ploted against each other with the command `plot(x,y)`.

The empirical distribution function and histogram for N can be plotted with

```
plot(sort(N),1:25/25); plot(sort(N),1:25/25,type=l''); hist(N)
```

figure=bild3-2.ps,width=0.525
figure=bild3-3.ps,width=0.525
figure=bild3-4.ps,width=0.525

In *R* one can use the command `postscript(``filename.eps'')` to save a graph to a file *filename.eps*. The syntax is as follows:

```
> postscript("filename.eps") ; hist(N); dev.off()
```

One can make comments in an *R* program, in the following way:

```
> command 1; command 2; ...; # comment
```

Everything on a command line that follows after `#` is neglected when running *R*.

An *R* session is ended with the command `q()`.

As with any other language working only with the command window is impossible. Instead, longer programs are written in a separate file with a text editor (*emacs, gedit,...*) and saved as **filename.r**. To run the code, write

```
> source("/users/math/mdstud/stada/lab_grupper/stada-??/filename.r")
```

in the *R* window. Shorter programs can be copied and pasted from the editor to the *R* window.

4 C

To learn C basics, we start with a very simple program, that is stored in a file called *hello.c*, say.

```
#include <stdio.h>
main()
{
   printf("Hello, world!\n");
   return 0;
}
```

When executed, this program will just produce the text *Hello, world!* on the screen.

The first line of the above program includes a *library*, with prefabricated C functions. Other examples of important libraries are *math.h, stdlib.h* and *time.h.*

The *main program*, of the above program, begins on the second line. The code of the main program is surrounded by { and }, and is ended with *return 0;*.

For another example, in order to estimate the expected value of a normal $\mathrm{N}(0,1)$-distributed random variable, by means of a sample mean based on 10000000 observations, the code, that is stored in a file called *normmean.c*, say, would look something like this:

```
/*Program for cacluating mean of a normal random variable
  using monte carlo simulation */
#include <math.h> //for mathfunctions
#include <stdio.h>
#include <stdlib.h>
#include <time.h> // for timing

//function for generate normal random variable (mu,sigma) (Box-Muller)

double normrnd(double mu,double sigma)
{
   double chi,eta,ret;
   chi=drand48();
   eta=drand48();
   ret=mu+sigma*sqrt(-2*log(chi))*cos(2*M_PI*eta);
   //M_PI is a constant in math.h
   return ret;
}

main()
{
   int i,nmbrofsim;
   double tid,est,mu,sigma;
   clock_t ticks1, ticks2;
   unsigned short myseed;

   myseed=750908;
   srand48(myseed);  //Make a new seed for the random number generator

   mu=0.0;
   sigma=1.0;
   est=0.0;

   ticks1=clock()/CLOCKS_PER_SEC;

   nmbrofsim=10000000;
   for(i = 0; i < nmbrofsim; i = i + 1)
   {
      est+=normrnd(mu,sigma); //est=est+normrnd;
   }

   est*=1.0/nmbrofsim;//est=est*1.0/nmbrofsim;
   ticks2=clock()/CLOCKS_PER_SEC;
   tid=difftime(ticks2, ticks1);

   printf("Time is %f\n", tid);
   printf("Est is %f\n", est);
   return 0;
}
```

The above program is compiled with the Linux command *gcc -o normmean normmean.c -lm*. Here *gcc* is the command that invokes the C compiler, *-o* is a flag that indicates the we want to have the compiled program in the file *normmean*. Further, *-lm* is a flag that have to be included when we use the *math.h* library. The program is executed with the UNIX command *./normmean*.

5 Emacs

There is a feature available called ”Automatic spell checking” found in **Tools** → **Spell Checking**. Please, feel free to use it.