UNIVERSITÄT BERN

R COURSE

LAURENT EXCOFFIER

Day 4, Introduction to programming

Outline of Introduction to programming

- Why programming?
- R as a programming language
- Interpreted vs. compiled language
- Scripts
- Elements of syntax
- R variables
- R types
- Flow control (if), loops
- Simple functions
- Scope of variables

Why programming?

- Automatize analyses
- For repetitive tasks
- Reuse invested time
- Perform complex analyses
- Analyze large data sets
- Learn more about computer use

R as a programming language

UNIVERSITÄT BERN

- R has all the necessary features of a programming language

Strong points

- Access to many built-in functions and statistical resources
- Strong graphic abilities
- Quick development time
- Interpreted language
- Platform independent
- Object oriented

Weak points

- Weak debugging tools
- Can be slow, memory hungry
- Weakly typed languages (implicit type changes)
- Interpreted language

When to use R for programming

- For rapid prototyping
- For statistical data analysis
- For data visualization
- For data reformatting, file conversion

When not using it?

- When computation speed is an issue
- When memory needs to be managed efficiently

Other languages

- C/C++: compiled, for scientific computing, speed and large projects
- Python: interpreted, for everything, if speed not an issue
- Perl: interpreted, for bioinformatics, text file and string handling
- java: interpreted and compiled, for almost everything, portable

Interpreted vs. compiled language

R is a language, not an environment.

The R environment reads R commands, line by line, and executes each piece of commands, one after the other.

A compiled program would be read by the compiler at once, and translate all commands into low-level machine instructions.

The R environment is responsible for the translation of the R commands into the low-level machine instructions for different environments (e.g. Windows, Mac, Linux…).

Interpreted languages are platform independent
Compiled languages are executed directly in low-level machine instructions, which leads generally to faster execution times.

A first simple program

UNIVERSITÄT BERN

A program is just a series of instructions, written in a file (an R script file), usually ending with an **.R** extension.

hello.R

```
#This is a comment.
#Program begins

print("hello world")

write(file="", "hello world")

#Program ends
```

```
> source("hello.R") #execution of the program "hello.R" in the R console
[1] "hello world"
hello world
```

Sections in an R script

UNIVERSITÄT BERN

An R script can have different sections and different types of instructions

Variable declarations and initializations

```
> v=vector(mode="double", length=10); v
 [1] 0 0 0 0 0 0 0 0 0 0
> vv=rep(0,10); vv
 [1] 0 0 0 0 0 0 0 0 0 0
```

Flow control and execution block

```
b=3
if (b==2) {
  print(paste("condition is true: ", "b=", b, sep=""))
} else {
  print(paste("condition is false: ", "b=", b, sep=""))
}
```

Elements of syntax

UNIVERSITÄT BERN

Flow control

```
if () {} else {}
```

Loops

```
for () {}
while () {}
```

Arithmetic operators

```
+,-,*,/, ^, **, %%, %/%
```

Logical operators

```
==, !=, <, >, <=, >=, &, |, isTRUE(x)
```

Matrix operators

```
+,-,/,*, %*%, %o%, t(A), …
```

Self-made functions

```
myFunc<-function() {}
```

Pre-existing functions

```
mean(x), crossprod(A,B), eigen(A), rowSums(A), …
```

R variables

R variables can be of several types:

- numeric
- alphanumeric (character)
- vectors, matrices, arrays
- data frame
- time series
- lists (e.g. list(1:10, letters[1:4]))
- formulae (e.g. A ~ B+C)

Simple function

In R, **functions** are a series of commands that can be applied in a generic way by a single call. For instance:

```
myFunc<-function(x) {
  x^2+3*x
}

z=seq(from=-5, to=5, by=0.1)
plot(z, myFunc(z))
```

Syntax:

- **function**: keyword
- (x): parameter(s) passed to the function
- {…} function body
- The result of the **last line of the function is returned**

Function needs to be executed (selection + CTRL-ENTER) as a whole to become active and available. You cannot execute parts of a function

Parameters of a function

UNIVERSITÄT BERN

- Parameters do not exist outside the function: the function is their **scope**
- Parameter type is not specified in the declaration (implicit typing).
  - Can be scalars or vectors (or anything)
  - It is possible to assign default values to parameters: e.g.

```
quadraticFunc<-function(x, a=1, b=1, c=1) {
    a*x^2+b*x+c
}
x=-5:5
plot(x, quadraticFunc(x, a=3), type=”b)
```

Beware:

Function parameters are assigned values with "=" , not with "<-"

When to use functions?

- Split complex tasks into smaller and simpler tasks
- Redo same computations over and over: reusability
- Makes scripts more readable (hiding complexity)

```
standardize<-function(v) {
  s=sd(v)
  if (s>0) {
    norm=(v-mean(v))/s
  } else {
    norm=v
  }
  norm
}
vec=rnorm(n=1000, mean=2, sd=5)
vec.norm=standardize(vec)
plot(vec.norm)
abline(h=mean(vec.norm), lty=1, col="blue")
abline(h=c(m.norm-1.96* sd(vec.norm), m.norm+1.96* sd(vec.norm)),
lty=2, col="blue") #Draws approx 95% CI
```

Functions use lists to return several results

UNIVERSITÄT BERN

```
moments<-function(v) {
  list(mu=mean(v),sigma=sd(v))
}

#Create a vector of random normal numbers
vec=rnorm(n=1000, mean=3, sd=1)
vec.norm=moments(vec)
```

Use $ to access members of the returned list

```
vec.norm$mu
vec.norm$sigma

vec.norm[1]
vec.norm[2]
```

Note that most R functions return several results as lists

Return statements in a function

Rather than returning the last expression of a function, one can use a **return** statement, to return different possible values within a function

```
myCondFun<-function(x, returnMin=T) {
  if (returnMin) {
    return(min(x))
  } else
  return(max(x))
}
vx=sample(1:100, 10, replace=T) ; vx
retMin=T
myCondFun(vx,retMin)
myCondFun(vx)
myCondFun(vx,F)
summary(vx)
```

What makes a script a program

In R, a program is a simple series of instructions.

Variables can be declared any time, best just before their first use (as in C). Functions must be defined before their first use in the program, unless they are stored in memory

In our case we can consider a script to be a **program** if it is **a self-contained series of R instructions**, such that it can be called from another program (R script) or executed externally from the command line.

Note that an R script can call other R scripts.

Structure of a program (2)

An R program generally follows the following structure:

- Setting the working directory
- Loading necessary libraries and packages
- Reading some input file(s)
- Function declaration(s)
- Computations and statistical analyses
- Graphic outputs in specific windows or output files (e.g. pdf)
- Writing output files with results in text files

Calling an external script from another script

UNIVERSITÄT BERN

source(file=filename)
E.g.:

```
> source("hello.R") #execution of the program "hello.R" in the R console
[1] "hello world"
hello world
```

Calling an R script from the command line

UNIVERSITÄT BERN

1) Using **Rscript**

```
>"c:\Program Files\R\R-3.4.4\bin\x64\Rscript.exe" hello.R
[1] "hello world"
hello world
```

2) Using **R**

```
>"c:\Program Files\R\R-3.4.4\bin\x64\R.exe" --slave -f hello.R
[1] "hello world"
hello world
```

(**-f**: take input from file; **--slave** : Make R run as quietly as possible)
Option **--vanilla** : do not read or save environment data

Scripts run in this way should only make computations or output results in external file: no graphical output other than graphic files (e.g. pdf, jpg) is possible.

3) Using R BATCH mode

```
>"c:\Program Files\R\R-3.4.4\bin\x64\R.exe" CMD BATCH --vanilla –-slave
hello.R
```

No output on the console. The output is redirected into a file "hello.Rout

hello.Rout

```
[1] "hello world"
hello world
> proc.time()
utilisateur     système      écoulé
       0.21        0.06        0.26
```

Advantages:

1) No console output
2) Timing of the execution, useful for profiling

Scope of variables

Global environment:

- user workspace
- contains all variables declared on the command line

Local environment:

- local variables created within functions
- variables defined within functions are not available outside functions

```
g=10
f<-function() {
  g=4
  x=3
  x+g
}
f()
g
```

But "<<-" operator allows you to modify global variables within functions

Loops

Loops are used to repeat some instruction several times
There are three main ways to make loops
for() loops: repeat a given number of times
while() loops: repeat until a condition is met
repeat loops: repeat until explicit exit of loop
Beware: executing loops are slow in R

Example:

```
x=0
for (i in 1:1e7) {
  x=x+i
}
```

takes much more time than

```
y=1:1e7
z=sum(as.numeric(y))
```

Evaluate speed gain with function system.time()

for loops

Syntax:

for(var in seq) expr

Eg:

```
for (day in c("Saturday", "Sunday")) {
  print(day)
}

for (i in seq(1, 11, 3)) print(i)

for (i in seq(0, 1, length=10)) print(i)

numtype=c("even", "odd")
for (j in 1:10) {
  modulo2=j%%2
  print(paste(j,"is an", numtype[modulo2+1], "number"))
}
```

while loops

Syntax:

while(cond) expr

Eg:

```
#Predefined number of iterations to do, like for (i in 1:3) print(i)
i=1
while (i<4) {
  print(i)
  i=i+1
}

#undefined number of repetitions
rn=1; i=1
while (rn>0.05) {
  print(paste (i, ":", rn<-runif(1)))
  i=i+1
}
```

How to stop loops

loops can be stopped by the **break** command

```
rn=1; i=1
repeat {
  print(paste (i, ":", rn<-runif(1)))
  if (rn<0.05) {
    break  #break needed with repeat loops
  }
  i=i+1
}
#another example of the use of break in a while loop
i=1
while(TRUE) {
  print(paste (i, ":", rn<-runif(1)))
  if (rn<0.05) {
    break
  }
  i=i+1
}
```

Vectorization vs. loops

R is very efficient at dealing with vectors, and R programs are much more efficient when performing operations on vectors, rather than repeating operations on all the elements of a vector
This is because a low level language (e.g. C++) is used for vector operation, whereas R commands are interpreted in loops
Eg:

```
v1=1:10
v2=3*v1 #operator * is vectorized
#better than
for (i in 1:length(v1)) v3[i]=3*v1[i]

m1=mean(v1);m1 # mean is vectorized
#much better than
m2=0
for (i in 1:length(v1)) m2=m2+v1[i]
m2=m2/length(v1);m2
```

Use of outer function vs. double for loops

The function outer (for outer product) is a powerful way to avoid writing double for loops

Creates a matrix from the outer product of two vectors (beware of memory use)

E.g.:

```
x=seq(-10,10,length=30)
y=x
my.f<-function(x,y) 10*sin(sqrt(x^2+y^2))/sqrt(x^2+y^2)
z=outer(x,y,my.f)
persp(x,y,z,theta=30,phi=30,expand=0.5)
```

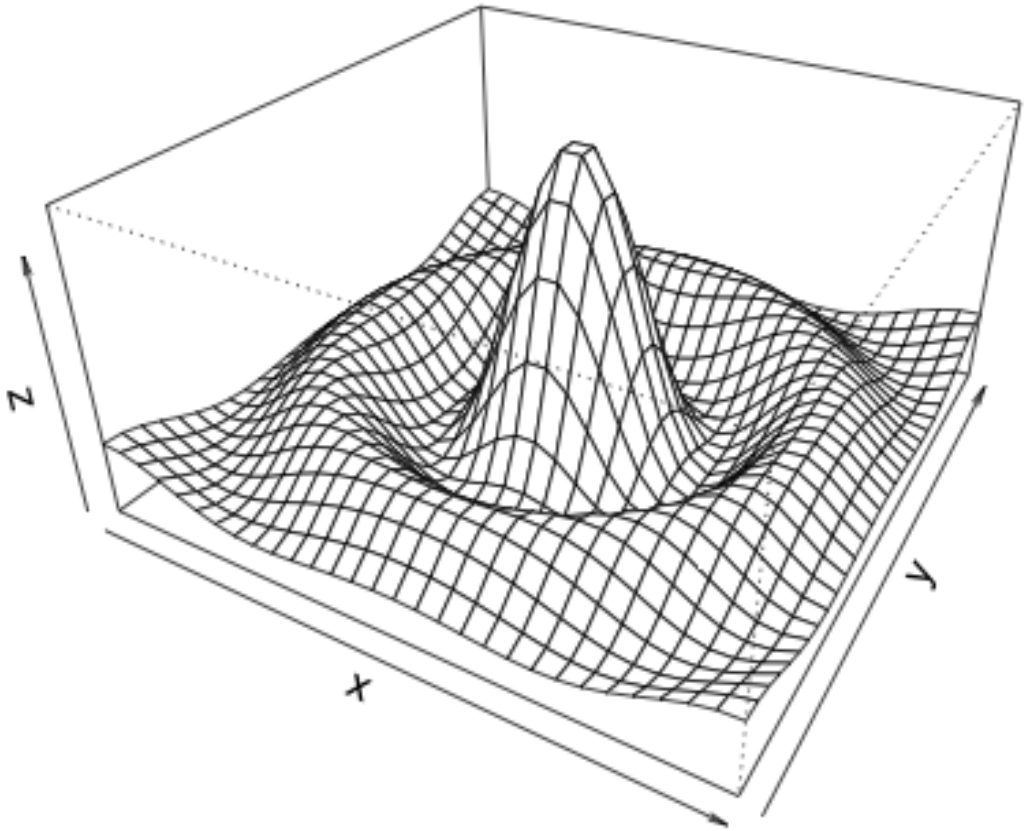
Use of outer function vs. double for loops

The function outer (for outer product) is a powerful way to avoid writing double for loops

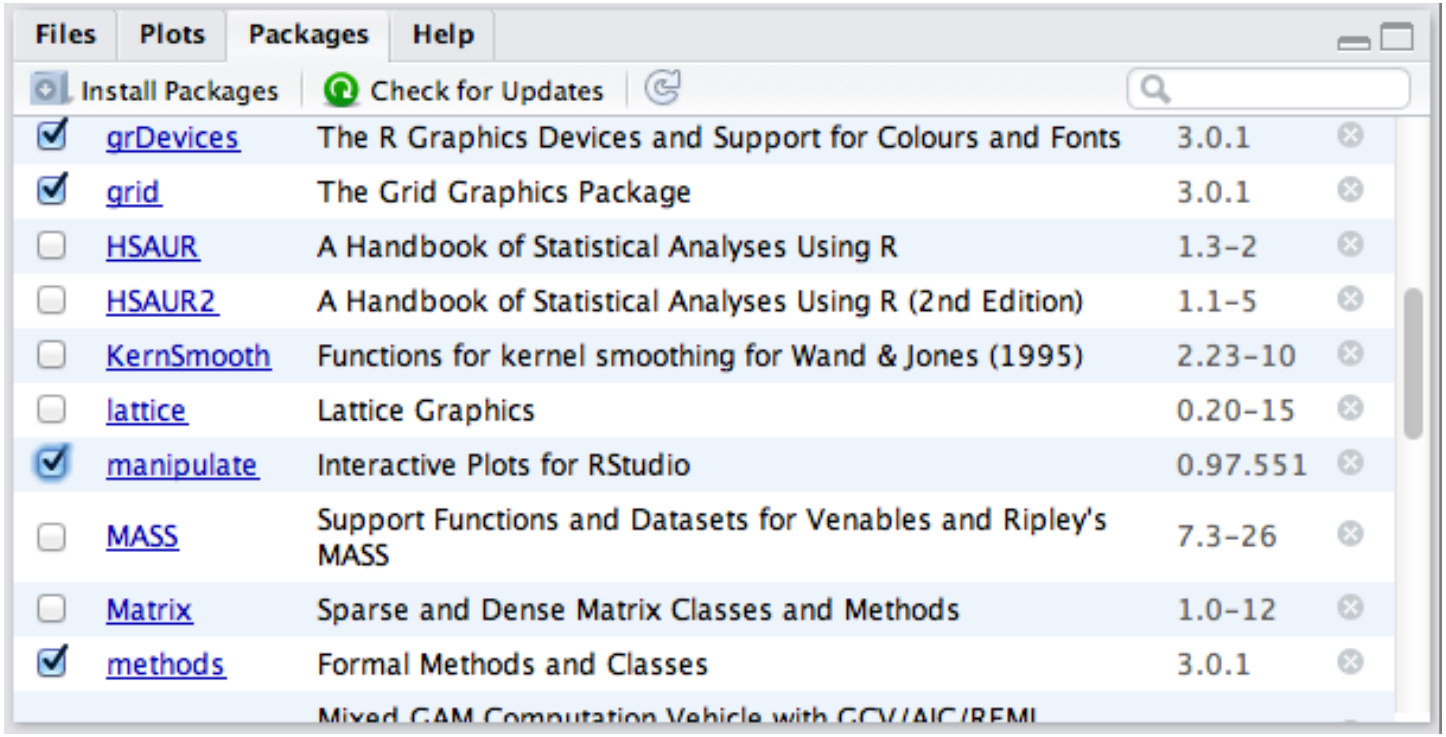
Creates a matrix from the outer product of two vectors (beware of memory use)

E.g.:

```
x=seq(-10,10,length=30)
y=x
my.f<-function(x,y) 10*sin(sqrt(
z=outer(x,y,my.f)
persp(x,y,z,theta=30,phi=30,expa
```



Use of the Rstudio manipulate function

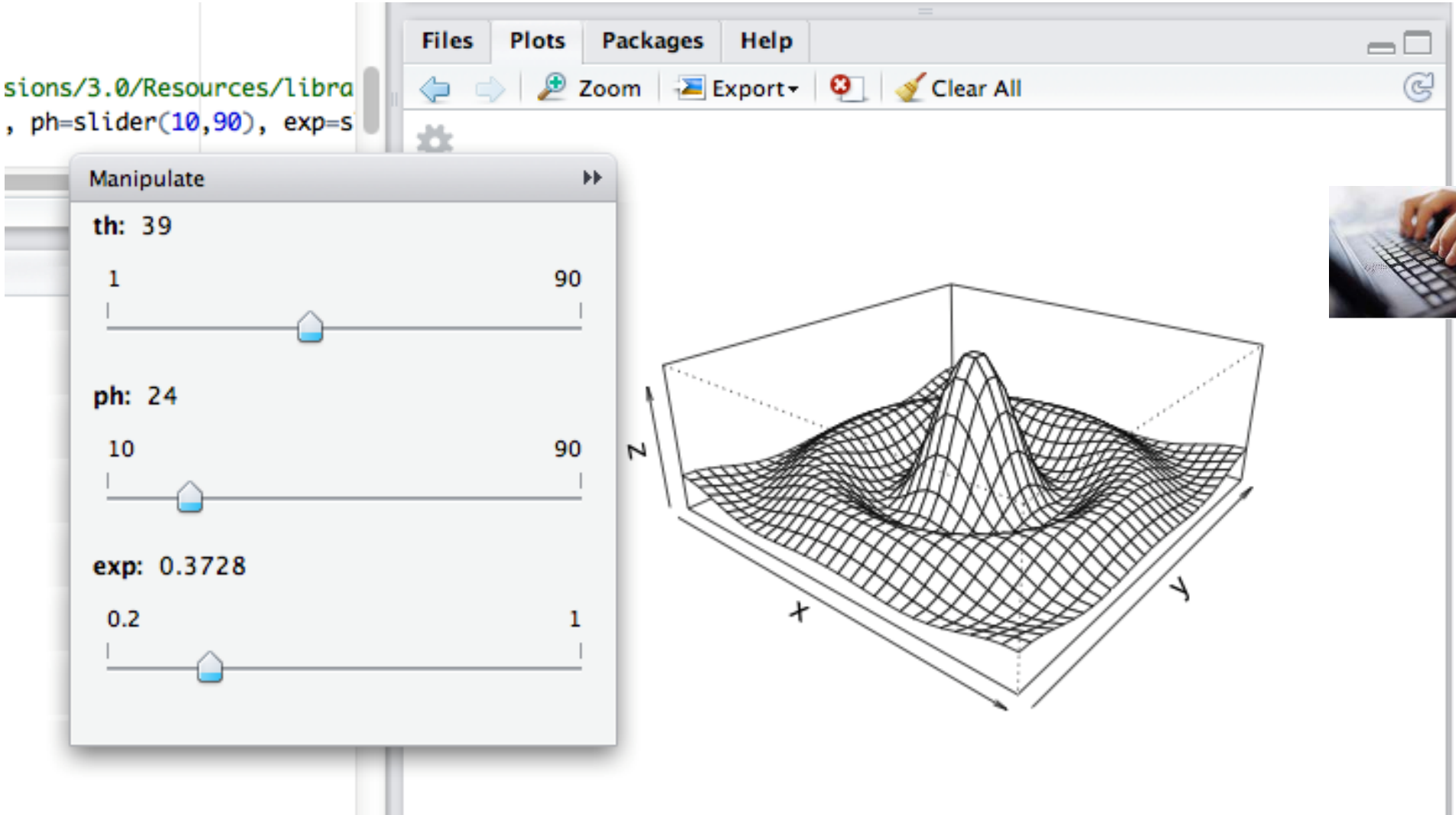



```
install.packages("manipulate")
library("manipulate")
```

Use of the Rstudio manipulate function

UNIVERSITÄT BERN

```
manipulate(persp(x,y,z,theta=th,phi=ph,expand=exp),th=slider(1,90),
ph=slider(10,90), exp=slider(0.2,1))
```



Generating random numbers

R can efficiently generate random numbers from various distributions

3.4.2 Random sequences

law	function
Gaussian (normal)	`rnorm(n, mean=0, sd=1)`
exponential	`rexp(n, rate=1)`
gamma	`rgamma(n, shape, scale=1)`
Poisson	`rpois(n, lambda)`
Weibull	`rweibull(n, shape, scale=1)`
Cauchy	`rcauchy(n, location=0, scale=1)`
beta	`rbeta(n, shape1, shape2)`
'Student' (t)	`rt(n, df)`
Fisher–Snedecor (F)	`rf(n, df1, df2)`
Pearson (χ^2)	`rchisq(n, df)`
binomial	`rbinom(n, size, prob)`
multinomial	`rmultinom(n, size, prob)`
geometric	`rgeom(n, prob)`
hypergeometric	`rhyper(nn, m, n, k)`
logistic	`rlogis(n, location=0, scale=1)`
lognormal	`rlnorm(n, meanlog=0, sdlog=1)`
negative binomial	`rnbinom(n, size, prob)`
uniform	`runif(n, min=0, max=1)`
Wilcoxon's statistics	`rwilcox(nn, m, n)`, `rsignrank(nn, n)`

Probability distributions

R can not only draw random numbers from many distributions, but it can also compute several connected function:

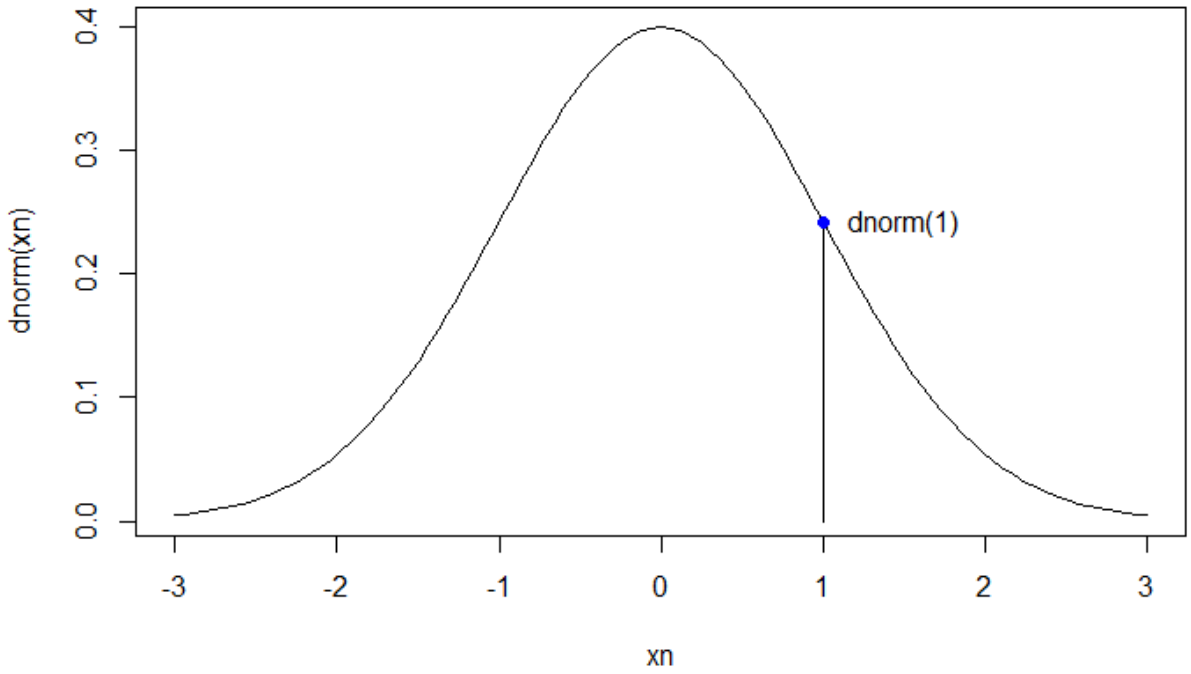
- the probability distribution function (pdf or *density*)
- cumulative distribution function (cdf or *cumulative density*)
- quantiles

Probability distribution function (density)

E.g.

Density of the **normal distribution** evaluated at x=1

```
>dnorm(1)
[1] 0.2419707
```



Probability distribution function (density)

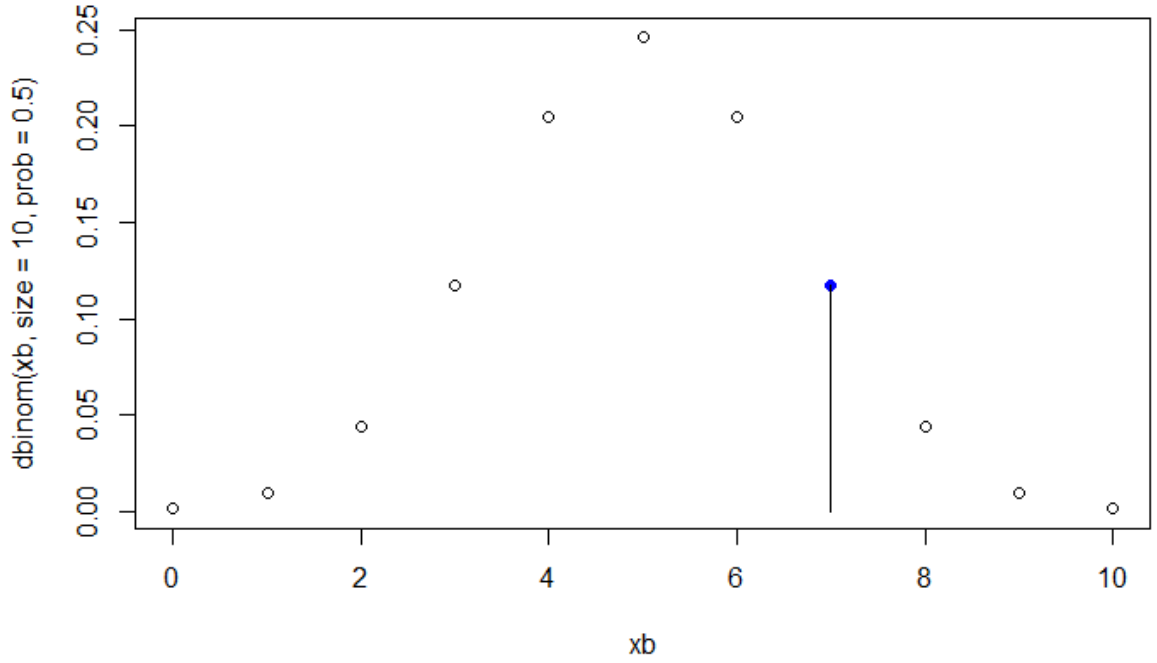
UNIVERSITÄT BERN

E.g.

Binomial distribution

Describes the probability of *x* successes among *n* trials, when the probability of a success is *p*.

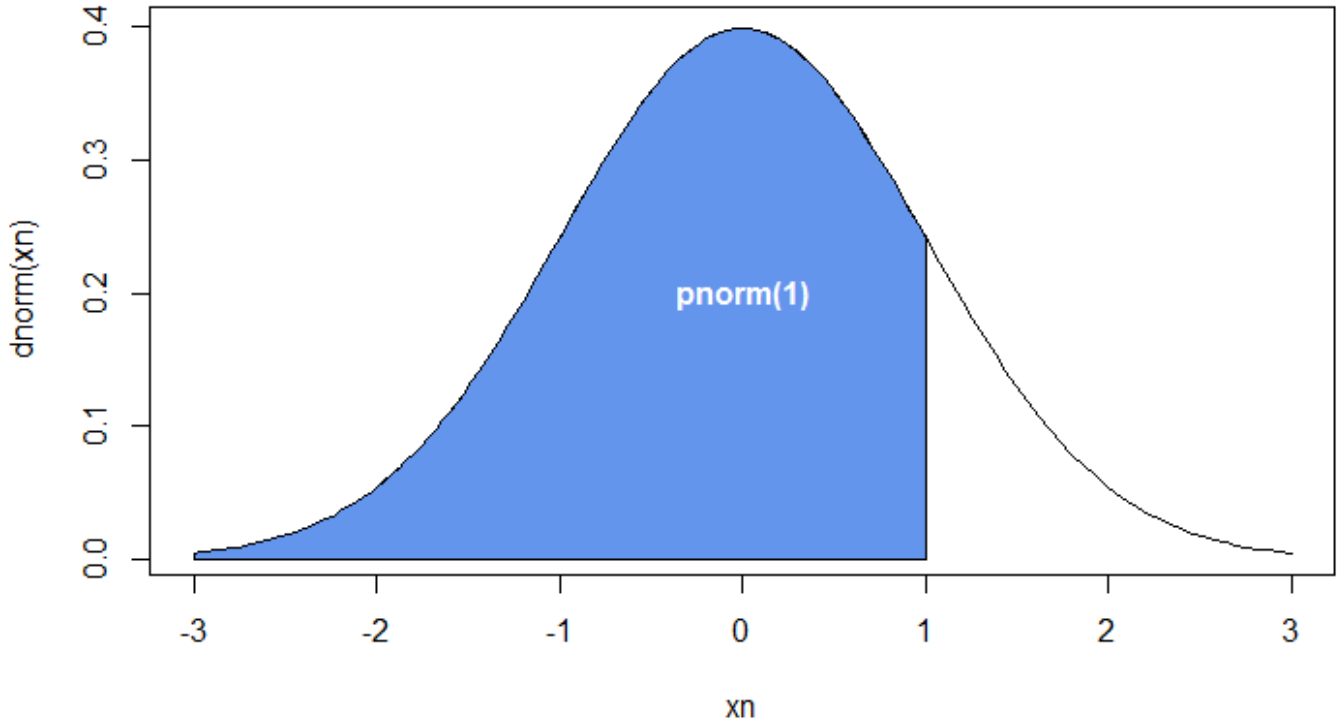
```
> dbinom(x=7, size=10, prob=0.5)
[1] 0.1171875
```



Normal **cumulative density**

```
> pnorm(1)
[1] 0.8413447
```

It corresponds to the area under the curve from -infinity to 1



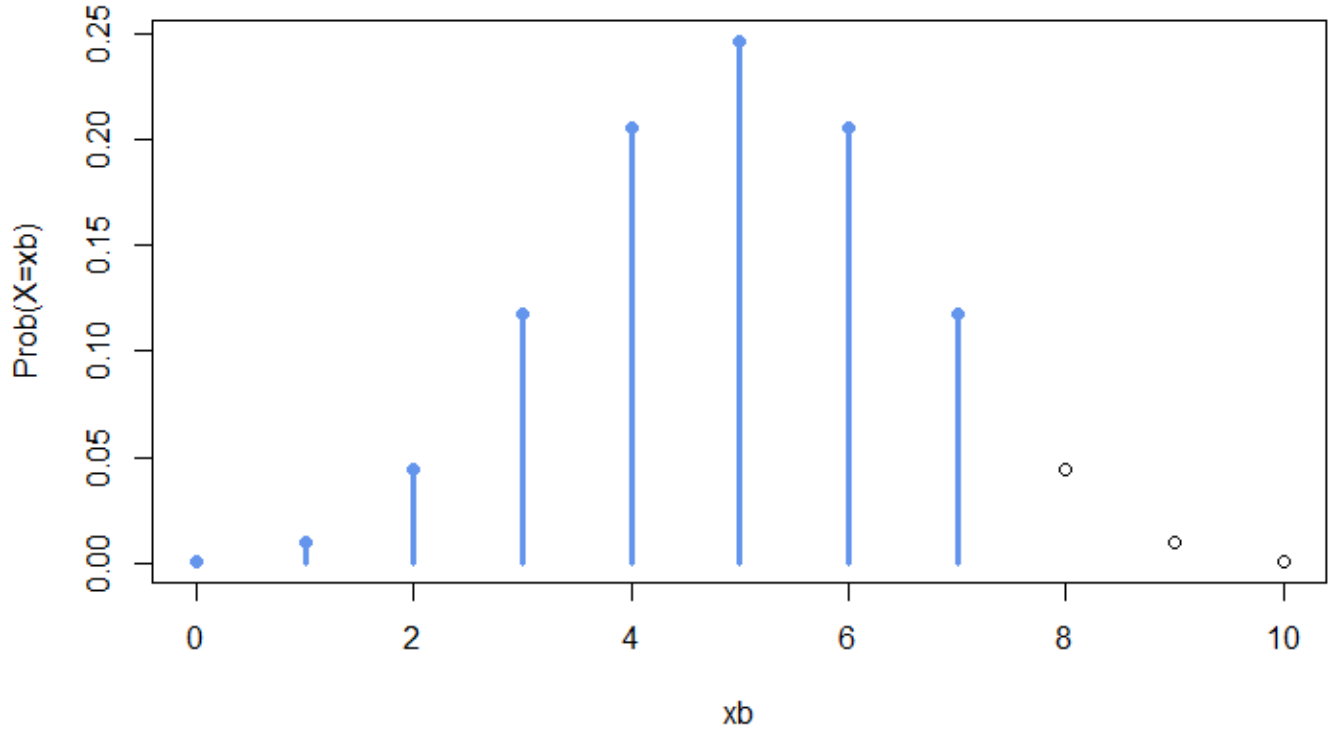
Binomial **cumulative density**

Binomial cumulative density

```
> pbinom(q=7, size=10, prob=0.5)
[1] 0.9453125
```

It corresponds to the sum of probabilities of x being smaller or equal to 7

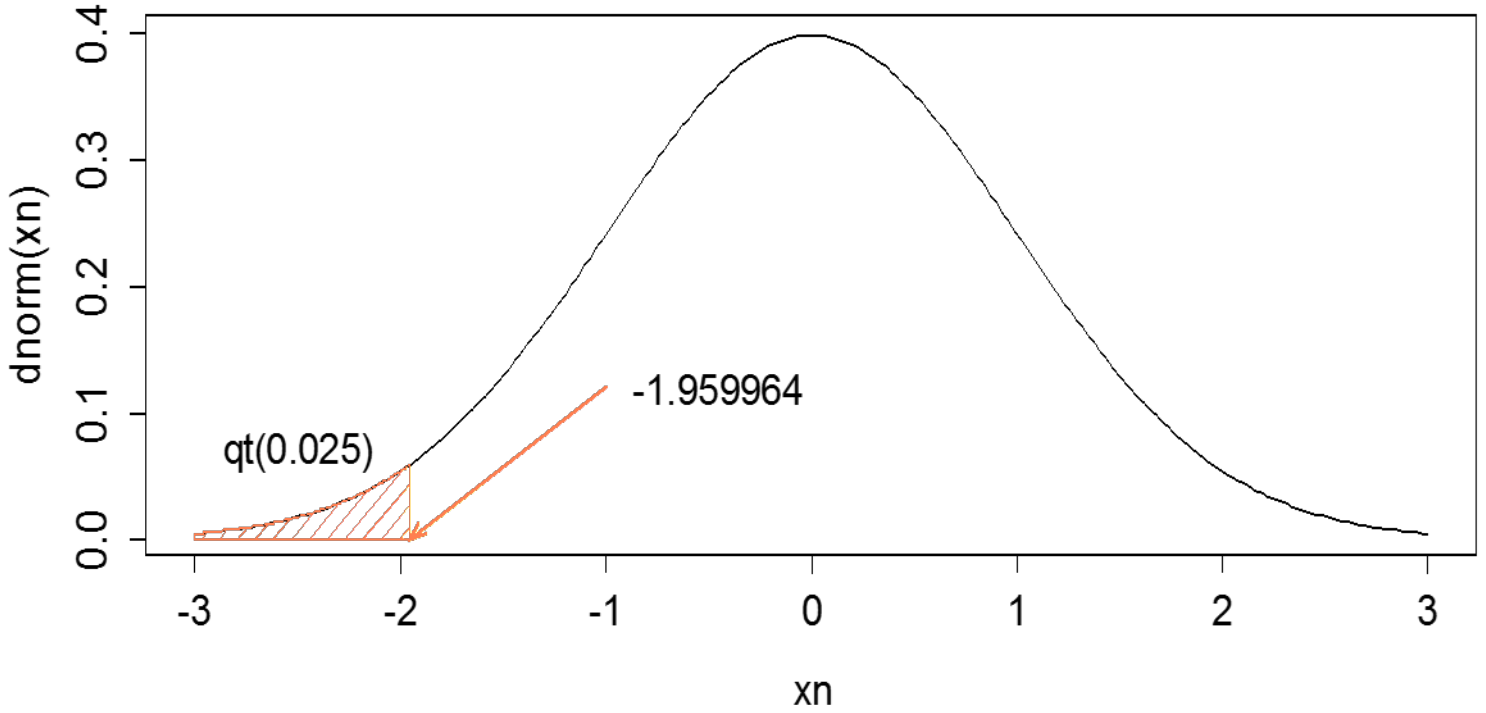
```
> sum(dbinom(0:7,size=10, 0.5))
[1] 0.9453125
```



Quantiles

R allows you to compute the quantiles of probability distributions

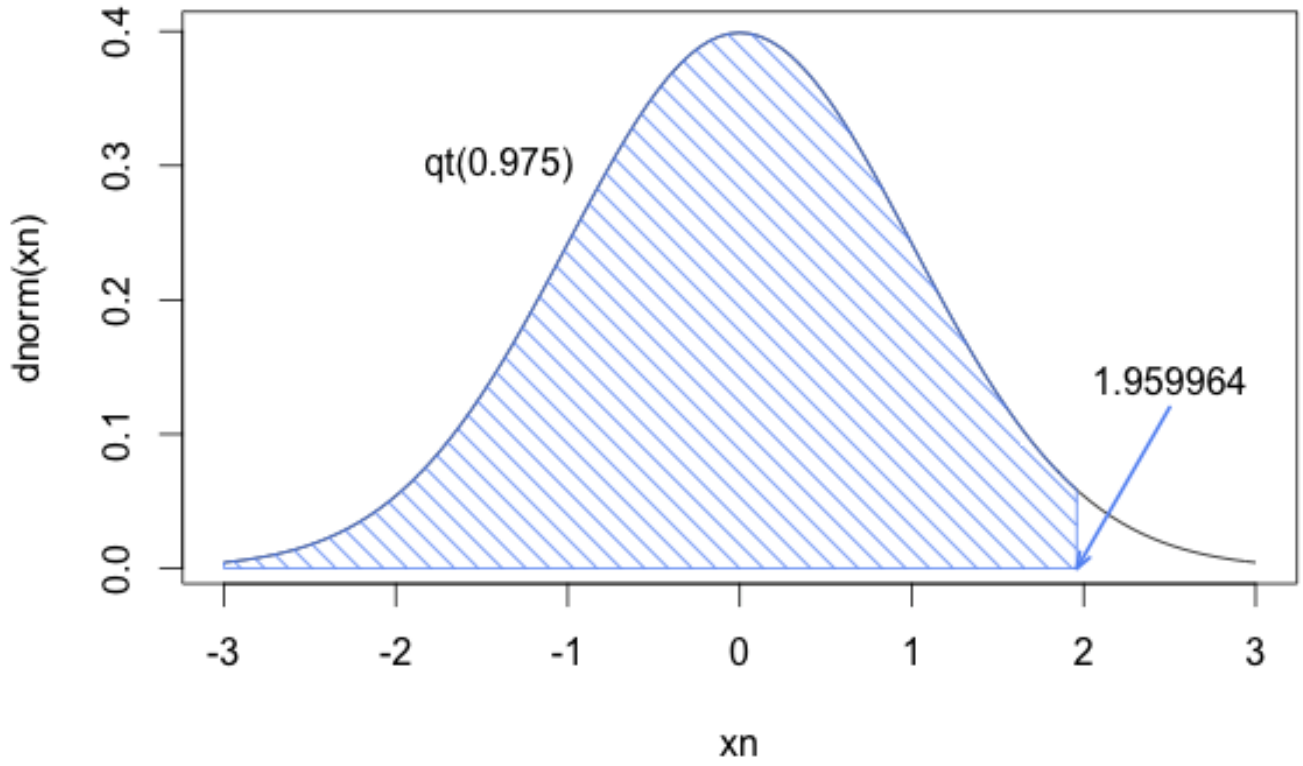
```
> qnorm(c(0.025, 0.975))
[1] -1.959964  1.959964
```



Quantiles

R allows you to compute the quantiles of probability distributions

```
> qnorm(c(0.025, 0.975))
[1] -1.959964  1.959964
```



Setting the seed of the random number generator

Setting a fixed seed allows one to have reproducible results

Simply write set.seed(n) #n should be an integer

```
set.seed(20) #20 is our seed
sample(12)
[1] 11  9  3  5  8  7  1  6  2  4 10 12
```

One should always obtain this sequence of numbers with this seed

It is good practice to define and record the seed used when generating random numbers

Generating random numbers

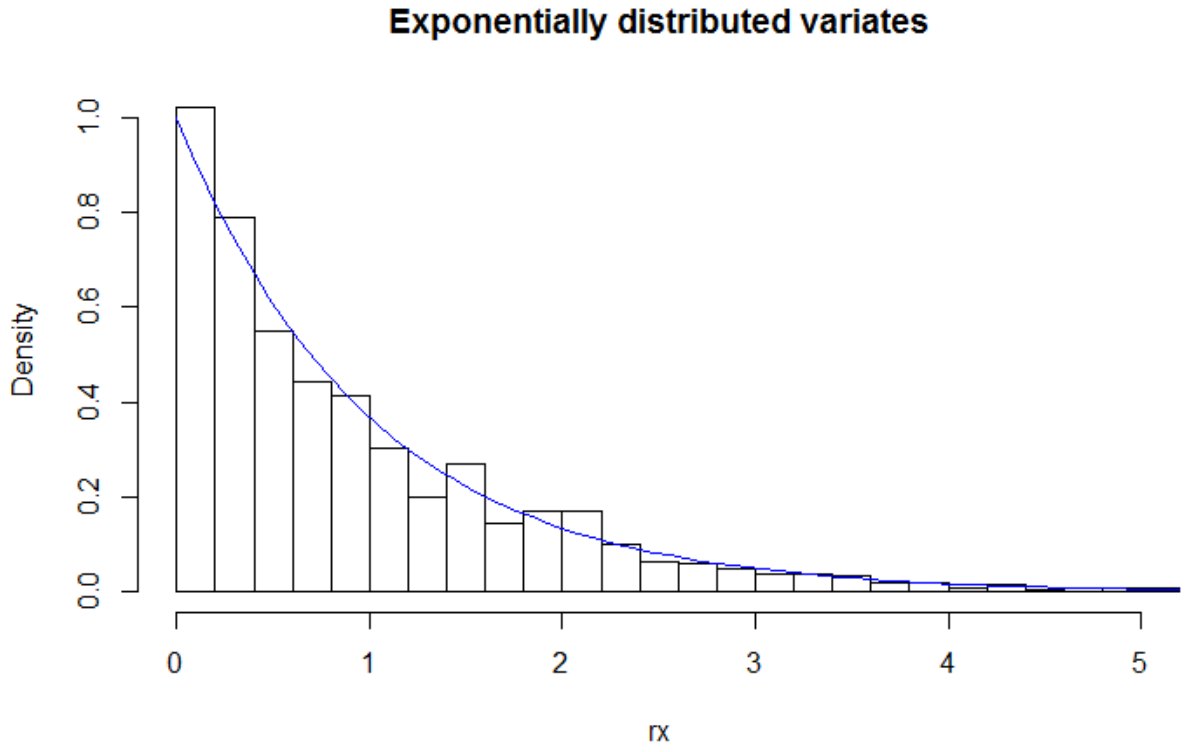
UNIVERSITÄT BERN

Exponentially distributed numbers

```
rx=rexp(1000) #generates 1000 numbers at once
hist(rx, freq=F, xlim=c(0,5), breaks=20)
```

Check with probability distribution (pdf) dexp

```
x=seq(0,8,length.out=101)
lines(x,dexp(x), add=T, col="blue")
```



Generating random numbers

UNIVERSITÄT BERN

Gamma distributed numbers

```
#Gamma distributed numbers
rgx1=rgamma(10000, shape=1, scale=1) #like an exponential distribution
rgx05=rgamma(10000, shape=0.5, scale=1)
rgx5=rgamma(10000, shape=5, scale=1)
plot(density(rgx05, from=0, to=10), xlim=c(0,10), col="blue", main =
      paste(numg,"gamma distributed numbers"), xlab="x")
lines(density(rgx1, from=0, to=10), col="black")
lines(density(rgx5), col="red")
```