PROGRAM STRUCTURE IN R, AND SOME PLOTTING

Modelling Biological Systems

Lund University

Pedro Rosero

A few facts (from the internet…)

- On average, a developer creates 70 bugs per 1000 lines of code
- 15 bugs per 1,000 lines of code find their way to the customers
- Fixing a bug takes 30 times longer than writing a line of code
- 75% of a developer's time is spent on debugging

How to avoid bugs, and write readable code

- Comment your code! (# this is a comment)
- Write pseudocode ('human-style' code), and keep it as comments.
  - In this way, you know what the code is supposed to do, which simplifies the interpretation tremendously.
- Structure! (divide and conquer!)
  - Solve big problems by dividing them into many small problems
  - Test your small solutions, on their own
- Test simplified versions of your code, then add more functionality step by step (extreme programming, agile programming)

random_walk.R

```
# A program that generates a random
# walk and plots it

# Input the length of the walk
reply <- readline('How long walk?')
n <- as.integer(reply)

# Generate a sample time series
x <- rep(0,n)
for (i in 2:n) {
  x[i] <- x[i-1] + rnorm(1)
}

# Plot it as a solid line
plot(1:n,x,type='l')
```

SCRIPTS AND FUNCTIONS

<u>Scripts</u>

- A sequence of commands
- Executed by source(*filename*)
- Use the main workspace

<u>Functions</u>

- A sequence of commands
- Must be defined before executed
- Executed by *name*()
- Separate workspace

WHY SCRIPTS?

- Repeated tasks
- Documentation (what did I do?)
- Focusing on one task at a time (scripts calling scripts). Divide and conquer!

Example from “A Beginner’s Guide to R”

```
setwd("C:/RBook/")
ISIT <- read.table("ISIT.txt", header = TRUE)
library(lattice) #Load the lattice package
#Start the actual plotting
xyplot(Sources  SampleDepth | factor(Station), data = ISIT,
xlab = "Sample Depth", ylab = "Sources”, strip = function(bg = ’white’, ...)
strip.default(bg = ’white’, ...),panel = function(x, y) {
        #Add grid lines
        #Plot the data as lines (in the colour black)
        panel.grid(h = -1, v = 2)
        I1 <- order(x)
        llines(x[I1], y[I1], col = 1)
})
```

FUNCTIONS

- **Syntax:**

```
functionName <- function(arg_1, arg_2, ...) expression
```

Input variables

- **Example:**

Use curly brackets for functions longer than one line

This is the name of the function. Function names have the same rules as variable names.

```
add2 <- function(a) {
    aplus2 <- a + 2
    return(aplus2)
}
```

Don't forget!

The value of the function, what it *returns.*

USING FUNCTIONS

The file add2.R

```
add2 <- function(a) {
    aplus2 <- a + 2
    return(aplus2)
}
```

- A function has to be *defined* before usage. Function definition is similar to variable assignment.

R console session:

```
> source("add2.R")          # the function is defined,
                              i.e. a function object is created
> ls()
[1] "add2"                  # it now exists as a function object

> add2
function(a) {               # viewing the object
    aplus2 <- a + 2
    return(aplus2)
}
>
> x <- add2(3)              # calling the function
> x
[1] 5
> x <- add2(x)
> x
[1] 7
>
```

WHY FUNCTIONS?

- Separate workspace, local variables
- Better defined interface (input, output)
- Enforces and enhances structure, i.e. subdivided, self-contained tasks
- + all benefits of scripts

Variable *scope* – when and where variables are accessible

Write this in a file printSum.R

```
printSum <- function(a,b) {
     sumab <- a + b
     cat("The sum is:", sumab)
}
```

`a`, `b` and `sumab` are *local* variables

`a` and `b` are copied from the the input

R console session:

```
> source('printSum.R')

> printSum(2,3)
The sum is: 5

> sumab
Error: object 'sumab' not found
```

Variables local to a function are not accessible outside the function.
A function has its own *workspace*, where local variables are stored. The workspace is created each time the function is called, and is deleted when the function finishes.

```
> sumab <- 8
> printSum(2,-1)
The sum is: 1
> sumab
[1] 8
```

Even if `sumab` is defined outside the function, it remains a local variable.
In this example, there are two variables with the same name, in different workspaces.

Variable *scope* – when and where variables are accessible

This is the file printA.R

```
printA <- function() {
    print(A)
}
```

printA is defined in the main workspace

R console session:

```
> source('printA.R')

> A <- runif(4)
> printA()
[1] 0.8332588 0.4045777 0.3406681
0.7679895
```

Special R feature: A function has access to objects (variables) in the workspace where it is defined (and above).

If no locally defined variables can be found, R looks in 'higher' workspaces.

Keep in mind that *assignments* always creates a local variable, even if one already exists with the same name in higher workspaces.

Is this good or bad???

In-parameters in R

Example:

```
add_ab <- function(a,b) {
    aplusb <- a + b
    return(aplusb)
}
```

- Functions can take as many in-parameters as you like.
- The call
  `add_ab(x,5)`
  copies the values x and 5 to the local variables `a` and `b`.
- The order of the input values determines which parameter gets which value, *unless* you specify:
  `add_ab(b=x,a=5)`
- The call
  `add_ab(5)`
  creates an error unless you specify default values:
  ```
  add_ab <- function(a=0,b=0) {
    aplusb <- a + b
    return(aplusb)
  }
  ```

Out-parameters (return-values) in R

Example:

```
add_ab <- function(a,b) {
    aplusb <- a + b
    return(aplusb)
}
```

- **Functions can only return one value in R**
- **The return-value is specified with the** `return()` **statement, which also ends execution of the function.**
- **If there is no return-statement (or none is executed), the value of the last statement of the function is returned instead.**

```
add_ab <- function(a,b) {
  a + b
}
```

- **If more than one return-values are needed, put them in a list and return the list:**

```
add_ab <- function(a,b) {
  aplusb <- a + b
  output <- list(a=a, b=b, absum=aplusb)
  return(output) # alternative: return(list(a=a,b=b,absum=aplusb))
}
```

WORKFLOW

- Write your function in 'pseudocode', describing what it is supposed to do and how.
- Write the actual code, in a separate file or in a script where the function is used (perhaps some testing code).
- *Define* your function by running its definition, using `source`
- Test your function, i.e. *call* it, using its name (+ input variables)
- After debugging or other changes, save and *define it again!* The function object in workspace specifies what the function does, not the text file itself.

ANOTHER EXAMPLE:
THE BUBBLE SORT ALGORITHM

Bubble sort is an algorithm for sorting objects, usually numbers. It is memory-efficient (very little extra memory is required), but has otherwise few advantages except for being simple. We here use it as an example.

Given a vector of numbers, do the following:

1. For each number in the vector, compare it to the next number.
2. If they are in the wrong order, swap them.
3. Repeat steps 1-2 until no swaps have been made in the entire vector.

The name 'bubble-sort' comes from the fact that small numbers 'bubble up' from the end of the vector to the beginning and large numbers 'bubble down'.

9 4 3 8 1 starting sequence
4 9 3 8 1
4 3 9 8 1
4 3 8 9 1
4 3 8 1 9
3 4 8 1 9
3 4 1 8 9
3 1 4 8 9
1 3 4 8 9 sorted sequence

https://en.wikipedia.org/wiki/File:Bubble_sort_animation.gif

A BUBBLE-SORT SOLUTION IN R

```
# the bubble-sort function
# sort the input vector x in ascending order
# return the sorted vector
bubble.sort <- function(x) {
  repeat {
    done <- TRUE; # set a flag
    for (i in 1:(length(x)-1)) {
      if( x[i] > x[i+1] ) { # wrong order?
        # swap
        swapx <- x[i];
        x[i] <- x[i+1];
        x[i+1] <- swapx;
        done <- FALSE;
      }
    }
    if (done) break; # no swaps have been made, we’re done!
  }
  return(x) # return the sorted vector
}
```

A MORE STRUCTURED SOLUTION

```
# the swap function
# swaps elements i1 and i2 in vector v,
# and returns the new vector
swap <- function(v,i1,i2) {
   vswap <- v[i1];
   v[i1] <- v[i2];
   v[i2] <- vswap;
   return(v)
}

# the bubble-sort function
# sorts a vector x in ascending order,
# and returns the sorted vector
bubble.sort <- function(x) {
   repeat {
      done <- TRUE; # set a flag
      for (i in 1:(length(x)-1)) {
         if( x[i] > x[i+1] ) {
            x <- swap(x,i,i+1)
            done <- FALSE;
         }
      }
      if (done) break; # no swaps have been made, we're done!
   }
   return(x) # return the sorted vector
}
```

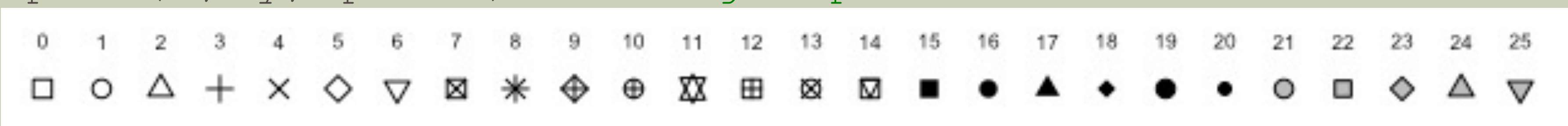
EXERCISE

- Write a function `reverse.vector()` (in a separate file!) that takes a vector `v` as input and
  - Creates a new vector with the elements of `v` in reverse order
  - Prints the new vector in the console (use `cat() or print()`)
  - *Returns* the reverse vector

- Write a *script* "test.reverse.R" that
  - Defines the reverse.vector() function using a source command
  - Generates a vector `u` of 10 random numbers
  - Calls `reverse.vector(u)`
  - Plots the reverse vector

Graphics: the plot command

```
plot(x, y) # x and y are vectors

plot(x, y, pch=6) # change 'plot character'
```




```
plot(x, y, type='b')
```

'p' for **p**oints, 'l' for **l**ines, 'b' for **b**oth, …

```
plot(x, y, lty='solid')
```

'solid', 'dashed', 'dotted', 'dotdash', 'longdash', 'twodash'

```
plot(x, y, col='orange')
```

Try colors() to see a <u>long</u> list of colors

```
plot(x, y, xlab='x-axis', ylab='this is the y')
```

Add custom labels to the axes

See also 'Using the plot command.pdf' and the R documentation (?plot, ?plot.default)

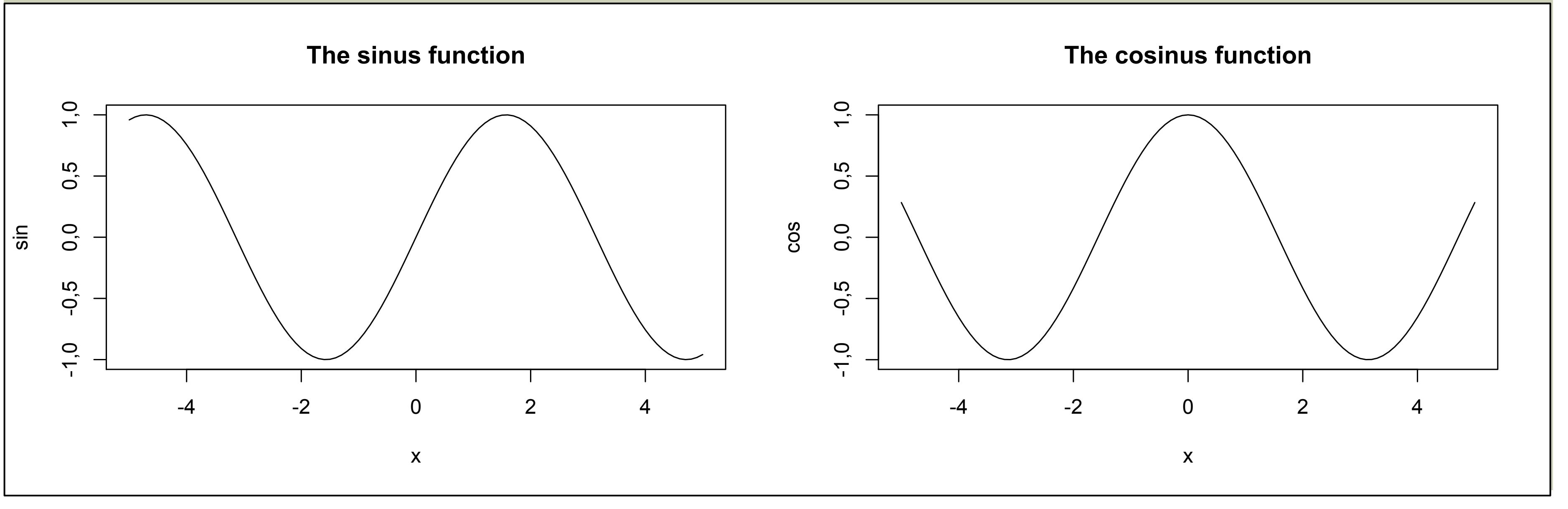
lines and points

- **The commands `lines` and `points` adds to the current plot.**
- **They take much the same input parameters as does the plot command.**

Several plots in the same window

```
op <- par(mfrow=c(1, 2))  # The function par() is used to change graphics settings. Here,
                          we want two figures in the same window organized in one row and
                          two columns.
                          We assign the return value of par() to an object op. By doing so
                          the "old par" setting is saved and we can restore the plotting
                          frame to normal afterwards (see last row in the following chunk
                          of code)
x <- seq(-5,5,length.out=100)
plot(x, sin(x), type='l', main='The sinus function')
plot(x, cos(x), type='l', main='The cosinus function' )
par(op) # restore old settings (otherwise, all following plots will be in a 1x2 arrangement)
```

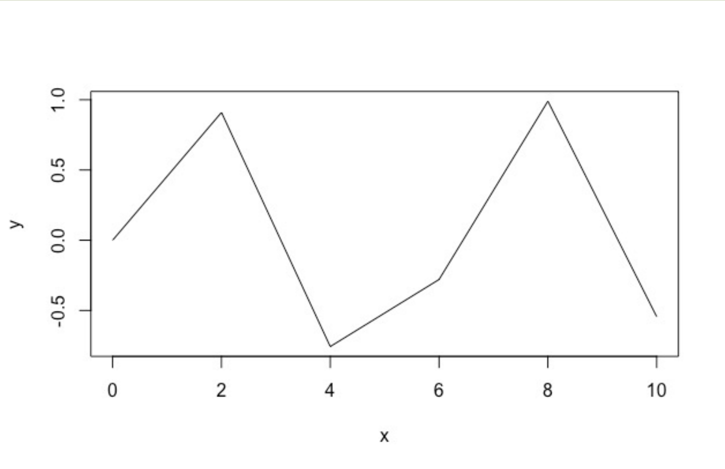
use `mfcol` to plot column by column



Plotting a function in R

- Task: We want to plot a function, such as sin(x), x^2, or our own function.
- General idea:
  - create a vector of x-values of the desired range, with the desired resolution
  - calculate the corresponding y-values, f(x)
  - Use plot(x,y), with extra input variables to control the design
- Example:

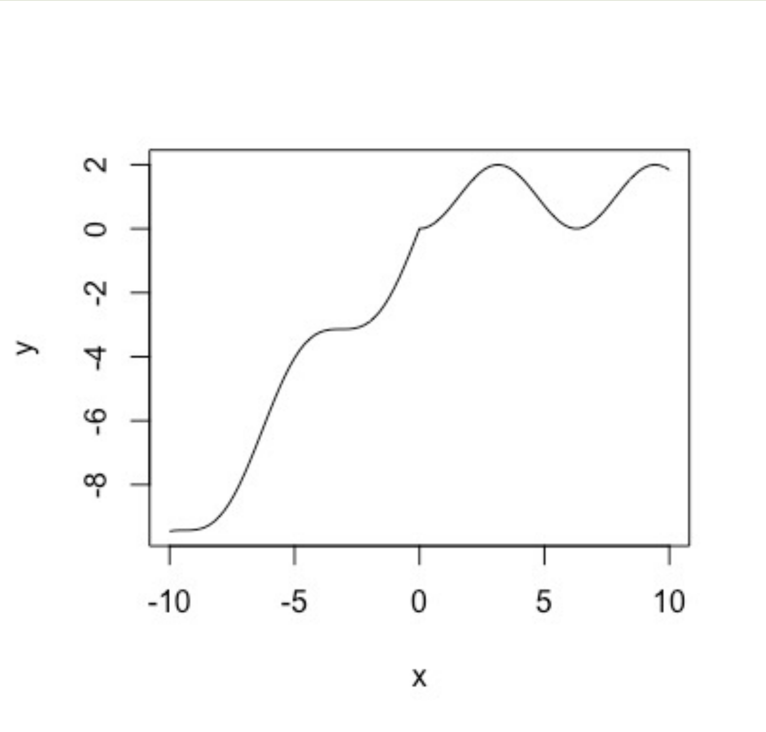
```
> x <- seq(0,10,by=2)
> y <- sin(x)
> plot(x,y,type='l')

> x
[1]  0  2  4  6  8 10
> y
[1]  0.0000000  0.9092974 -0.7568025 -0.2794155  0.9893582 -0.5440211
```



Plotting a function in R

- **Sometimes the function is more complicated, and can't handle vector input. In such cases we need a for loop.**

```
x <- seq(-10,10,by=0.2)
myfun <- function(x) {
  if (x<0) {
    y <- sin(x) + x
  } else {
    y <- 1-cos(x)
  }
  return(y)
}
y <- rep(0,length(x))
for (i in 1:length(x)) {
  y[i] <- myfun(x[i])
}
plot(x,y,type='l')
```



Exercise

- Write a function called `moving_average` that takes a single vector as input, and returns a *moving average* vector.
- The moving average is calculated from the input such that each element *i* is the average of three neighbouring elements.
  - Input (example):

12	3	42	4	53	4	23	-3	...

  - Output:

7.5	19	16.3	...	...	...	...	...	...

- Write a script that
  - Defines the `moving_average` function
  - Generates a random vector of length 50
  - Calculates the moving average, using the above function
  - Plots the vector together with the moving average
  - Smooths the vector further by repeatedly calling the `moving_average` function, and plots the result

GGPLOT2 AND RGL

- ggplot2:
  - http://ggplot2.org
  - http://r-statistics.co/Complete-Ggplot2-Tutorial-Part1-With-R-Code.html
  - https://cedricscherer.netlify.app/2019/08/05/a-ggplot2-tutorial-for-beautiful-plotting-in-r/
  - ggplot2 uses a 'grammar of graphics' to step-wise add features to a graph. One can also store graph objects in variables, and reuse them later, adding features when needed.

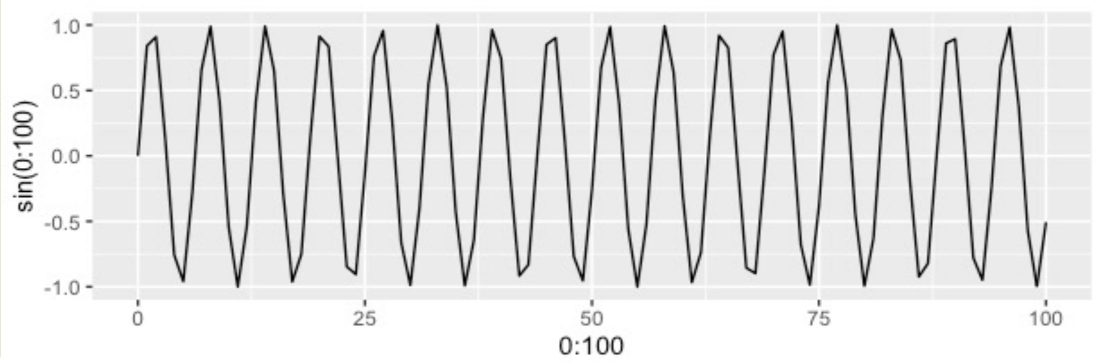
- rgl:
  - http://rgl.neoscientists.org
  - https://cran.r-project.org/web/packages/rgl/rgl.pdf
  - http://www.sthda.com/english/wiki/a-complete-guide-to-3d-visualization-device-system-in-r-r-software-and-data-visualization
  - rgl has several powerful 3D visualization tools.

- First install the package, most easily done from the menus [Tools | Install package…]
- Next, load it: `library(ggplot2)`
- The basic command is ggplot(). It usually takes a dataframe as first input parameter, but you can do without, as long as you specify x- and y-values. Create a plot object and store it in a variable:

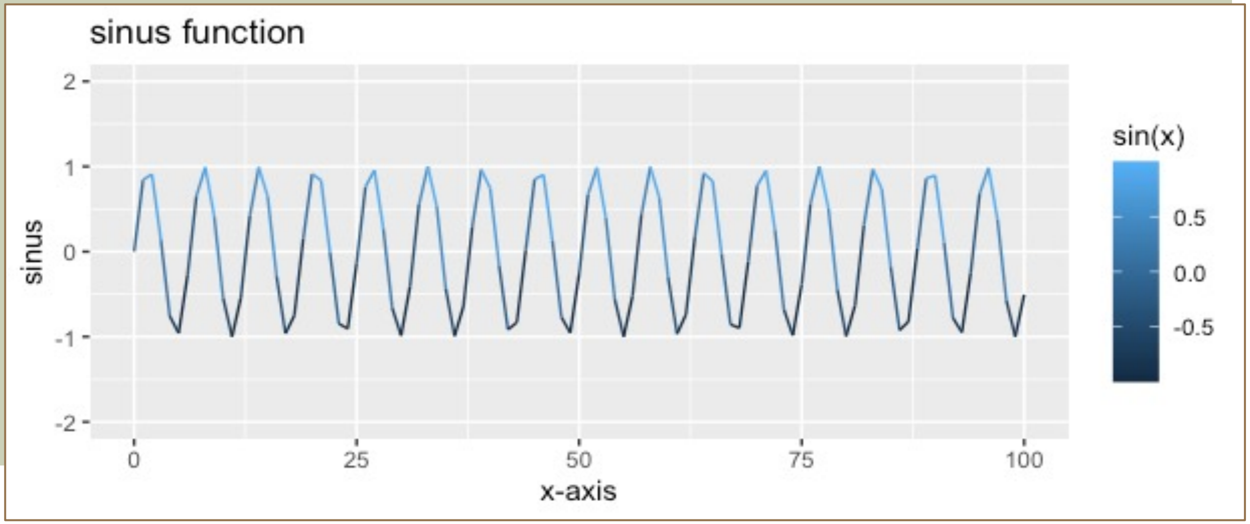
```
x <- 0:100
my_g <- ggplot(NULL, aes(x=x,y=sin(x)))
```

- The aes (for aesthetics) function is used to add features or modify the plot. You get used to it.
- Next, decide how the data should be plotted, by adding ‘geometries’:

```
my_g + geom_line()
```



- **Colours, fixed**
```
my_g <- my_g + geom_line(col='red')
```
  or based on values (inside aesthetics)
```
my_g <- my_g + geom_line(aes(col=sin(x)))
```
- **Labels**
```
my_g <- my_g + labs(title="sinus function", x="x-axis", y="sinus")
```
- **Axis limits**
```
my_g <- my_g + xlim(c(0,100)) + ylim(c(-2,2))
```
- **Finally, 'print' the result**
```
print(my_g)
```
  Alternatively, in the console:
```
> my_g
```



Read more here:
http://r-statistics.co/Complete-Ggplot2-Tutorial-Part1-With-R-Code.html
https://cedricscherer.netlify.app/2019/08/05/a-ggplot2-tutorial-for-beautiful-plotting-in-r/

Plotting in 3D

- If a function takes two input variables, it can not be plotted with an ordinary plot command.
- Example:

```
myfun <- function(x,y) {
  z <- sin(x) - cos(y)
  return(z)
}
```

- The idea is to generate a whole matrix of function values corresponding to underlying *x* and *y* coordinates.
- The R-function `outer` does exactly that.

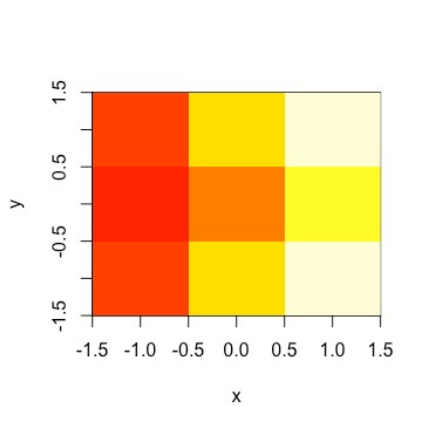
```
x <- -1:1
y <- -1:1
z <- outer(x,y,myfun)
x
[1] -1  0  1
> y
[1] -1  0  1
> z
           [,1]      [,2]       [,3]
[1,] -1.3817733 -1.841471 -1.3817733
[2,] -0.5403023 -1.000000 -0.5403023
[3,]  0.3011687 -0.158529  0.3011687
```

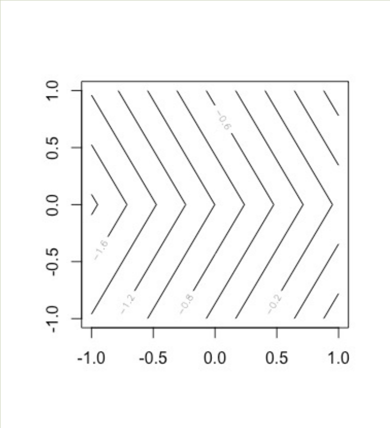
```
z[i,j] = myfun(x[i],y[j])
```

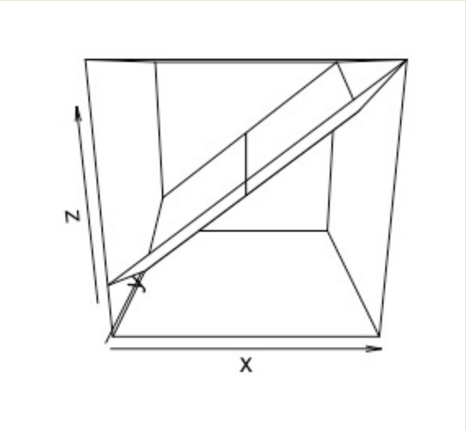
Plotting in 3D

- Once we have the matrix of function values, it can be plotted using `image`, `contour`, or `persp`.

```
> image(x,y,z)
> contour(x,y,z)
> persp(x,y,z)
```



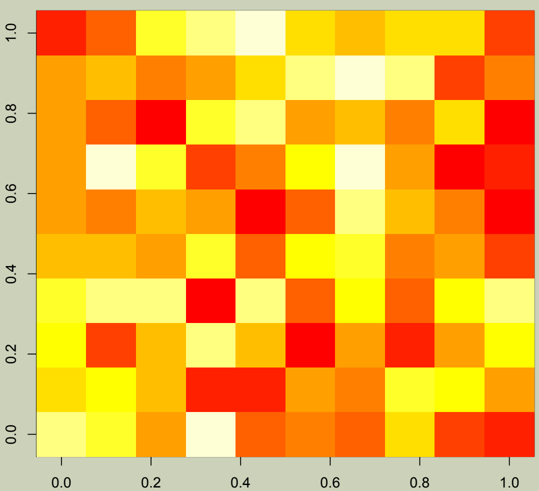


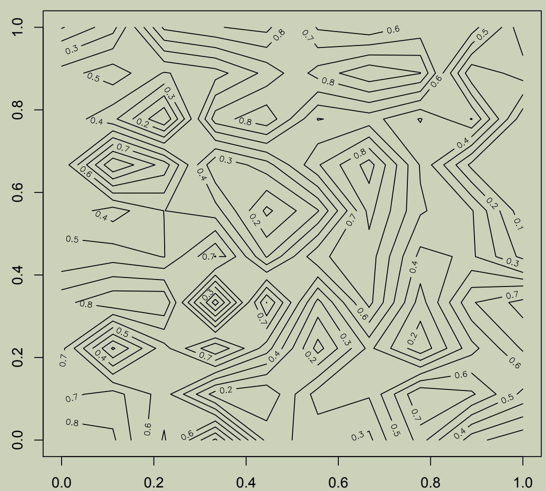


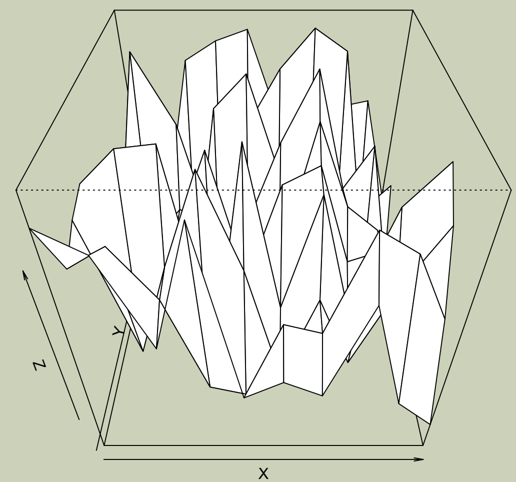
3D Graphics: image, contour and persp

```
> X <- matrix(runif(100),10,10)
> image(X) # see also levelplot
> contour(X) # see also contourplot and filled.contour
> persp(X, phi=40)
```

Angle 'above horizon', in degrees



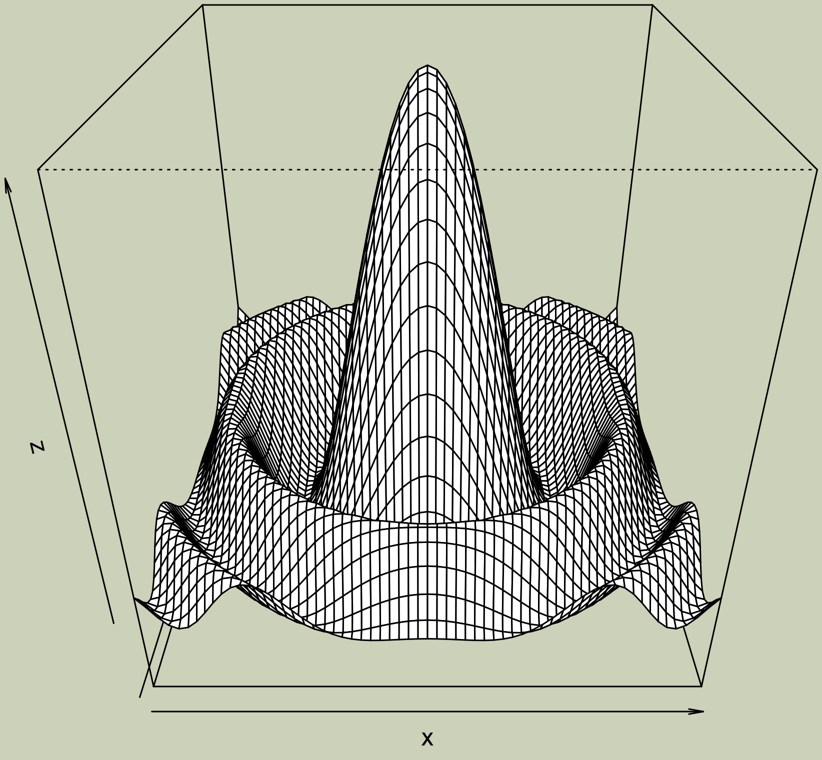




See also additional packages, such as **rgl** or **ggplot2**

A 3D EXAMPLE

```
x <- seq(-3,3,by=.1)
y <- seq(-3,3,by=.1)
myfun <- function(x,y) cos(x^2+y^2)/(x^2+y^2+1)
z <- outer(x,y,myfun)
persp(x,y,z,phi=30)
```



A few useful rgl commands

- open3d()
- close3d()
- plot3d(x,y,z,…)
- lines3d(), points3d()
- spheres3d(x,y,z,radius=…)
- quads3d(x,y,z,…) #
- polygon3d(x,y,z,…)
- decorate3d(xlim, ylim, zlim, xlab, ylab, zlab, box [T/F], axes [T/F], main, sub, top, aspect, expand)
- persp3d(x,y,z,…)

Another exercise

- Write a function that calculates a 2D moving average of a matrix, such that each element of the output matrix is the average of the corresponding surrounding 9 elements of the input matrix.
- Edges and corners are treated similarly to the vector moving average earlier.
- Write a script that creates a random matrix of size 100x100 and uses the moving average function above repeatedly (5 times?), and finally plots the result using one of `image`, `contour` or `persp`. Alternatively, use the rgl package.