Introduction to R: the cell is very different from our world

The very basics

R is a powerful environment (programming language) for statistical computations. It runs on all systems and can be used from the console or in a graphical environment called RStudio, which you are typing in now. RStudio is demonstrated live in class, and is relatively easy to get used to.

The most basic use of R is like a calculator, you type an expression and get the answer:

```
1 + 3

## [1] 4

2*3 + 5

## [1] 11

2*(3 + 5)

## [1] 16

2^3

## [1] 8
```

The standard arithmetic operators are `+`, `-`, `*`, and `/` for add, subtract, multiply and divide, and `^` for exponentiation. These operators have the standard evaluation order, with exponentiation taking place first, then multiplication and division, and addition/subtraction at last. You can always control the order of evaluation with parentheses though, and you will want to do so often.

```
2*3 + 5

## [1] 11

2*(3 + 5)

## [1] 16
```

The results of a calculation may be stored in a variable and retrieved by just typing the name of the variable:

```
myvariable <- 2
myvariable

## [1] 2
```

The "arrow", `<-`, is conveniently typed in RStudio by `Alt -`, the Alt key and the hyphen key pressed at the same time. You save one keystroke in that way. (Instead of the arrow, one can also use `=`, but this is not standard style.)

There is also the scientific notation: $a \times 10^b$ is conveniently typed by replacing `*10^` by `e`. So for storing 5×10^{-6} in the variable 'fivemicrons', we type like so:

```
fivemicrons <- 5e-6
fivemicrons

## [1] 5e-06
```

Basic scientific arithmetic

- Imagine you shrink so much, a eucaryotic cell appears to you as big as this classroom (diameterwise). How large does the diameter of a protein then appear to you? Use the scientific notation. *Hint: use Regula de tri: proportionality dictates that (protein diameter)/(cell diameter) = x/(classroom diameter), where x is what we are looking for.*
- Imagine you grow so big, that the earth would look like this classroom. How large would an ordinary human appear?
- If you're so big, one light-year (around 10e12 km) would look to you like this classroom wall-to-wall; how large would then the earth look like?

Variable names may contain letters, numbers or periods, but must start with a letter or period. Warning: R is case sensitive! So, 'mprot' and 'Mprot' are two different things.

The most central thing for you in R are functions. Functions in R work similar to mathematical functions, they take something as an input and give you back an output. They can also have side-effcts, such as producing plots, as you will see. To use ("call") a function you type the name followed by the arguments (the variables given to the function) in parentheses. If the function takes no arguments you just type the name followed by left and right parenthesis (empty argument sequence).

```r
q()      # This will quit your current session!
help()
```

Other basic functions are 'sin()', 'cos()', 'sqrt()', 'exp()' , ... There are thousands of functions in R, which gives it much of its power. You will also learn to write your own functions, they will work like reusable building blocks for you. If you are not sure how a function works 'help(*function-name*)' (or ?*function-name*) has to be typed in the command line. RStudio also offers a help pane, which you will want to get comfortable with.

Functions

- Calculate the sine and the cosine of π (the constant 'pi' is provided by R). What would you expect? Can you explain the difference?
- What happens if you just type 'log' (without parentheses)?

The random walk of proteins

As our application when learnin programming with R, we will look at the diffusion of a protein in a cell. As we will see, in terms of time scales, this is nothing like the world were we live; the microscopic world is very different.

A protein moves into one direction with a certain speed until it changes its direction due to random events in its solvent (the cytosol). One speaks of the protein performing a *random walk*. In the figure below a random walk in two dimensions is illustrated. The red dashed line indicates the distance a protein traveled from a starting point.

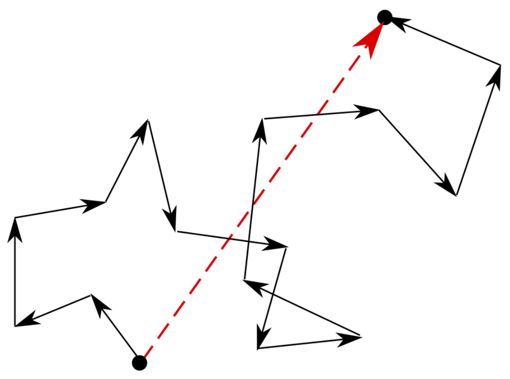


Figure 1: Example of a random walk in two dimensions.

To model the random walk you will need some parameters and constants which you can save as variables. In the lines below, you will also see comments which can be very useful in describing source code. Comments start with '#' and are not executed. **Use comments liberally when writing computer code. They will be invaluable notes to self.**

```r
mprot <- 1e-23 # mass of a typical protein in kg
k <- 1.38e-23  # Boltzmann's constant
T <- 310       # body temperature in Kelvin
kT <- k*T
kT

## [1] 4.278e-21
```

In order to characterize diffusive spreading, we will reduce the problem and consider the motion of a protein along one axis. The proteins execute a random walk according to the following rules:

1. Each protein steps to the right or to the left once every τ seconds, moving at velocity v_x a distance $v_x \cdot \tau$.
2. The probability of going into one direction at each step is $1/2$. Successive steps are independent.

The formula for kinetic energy in one dimension (Einstein) is

$$m_{\text{prot}} \times v^2/2 = kT/2$$

which we can use to calculate the velocity of the protein.

Speed of a protein

- Calculate the speed of a typical protein and save your result in a variable called 'vprot'.
- Assuming that the step rate, i.e. the number of times direction is decided per time, is 1×10^{12} per second, calculate the distance with which a protein moves in one step and save your result in a variable 'steplength'.

Letting things change: flow control for simulating protein movement

So far, we have only used functions that are already available in R. You can also write your own functions, which is especially helpful if you plan to reuse parts of your script. We will now make our own function that determines whether a protein goes left (-1) or right ($+1$) with a probability of $1/2$.

```
stepdir <- function() {
  rn <- runif(1)
        if (rn > 0.5) {
                dir <- 1
        }
        else {
                dir <- -1
        }
  return(dir)
}
```

The name of our function is 'stepdir', so that's the first thing we type. This works the same way as defining a variable. Then we tell R we want to define this as a function by typing '¡- function'. In the parentheses we can now add arguments that we want passed to the function. We don't need any for the function 'stepdir', so we will leave that empty. Everything between the curly brackets will be part of the newly-defined function. Within the function three different things happen:

- a random number is chosen from a uniform distribution between 0 ... 1,
- a test whether the random number is greater $1/2$ or not is carried out,
- and a value is returned.

In the first part, we see the power of reusable functions. Random number generation is a very complicated procedure, but someone has already taken care of this common taks for us, and has implemented a reusable function, `runif()`. This works like a building block. Just as a construction worker building a house does not have to fabricate all materials from scratch, a computer programmer does not have to program each procedure, but resorts to pre-programmed library functions as often as possible. This saves time and reduces errors.

The second part is a way of controlling the order in which statements/commands are executed. The three basic types of flow controls are depicted below. In the simplest case, you execute one command after the other in a sequential way (as you have done so far). If you want to program a conditionality (think: IF the light is green, you can cross the street ELSE you stop), you will need to test for a certain fact. In the case of choosing a direction, we use the value of a random number to decide, whether we turn left (-1) or right (1).

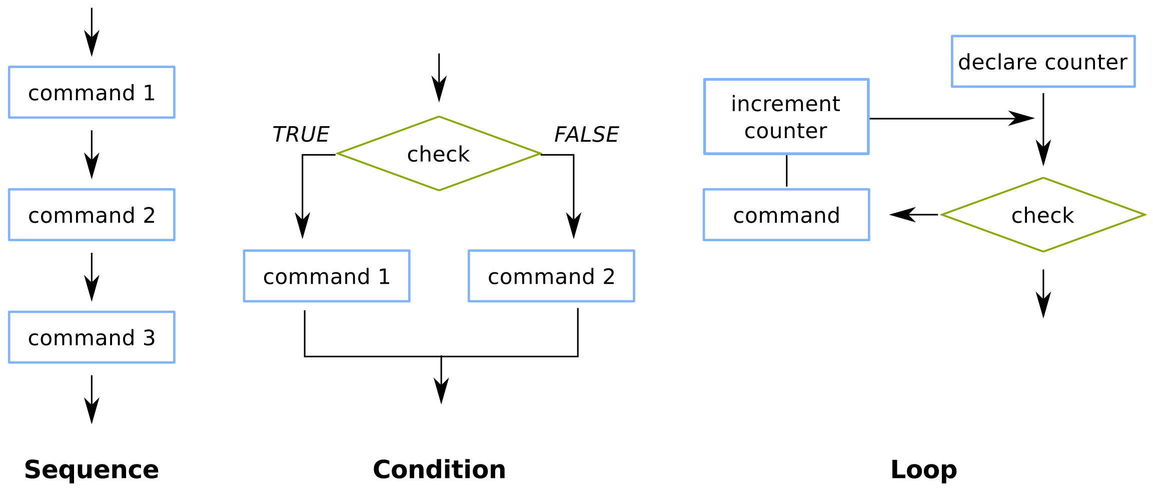


Figure 2: The three basic flow control constructs.

Biased brownian motion
Write a function `stepdir2()` that returns 1 with 30% probability and returns -2 otherwise.

Loops and vectors to build our protein movement simulation

We need to discuss loops and so-called data structures (here: vectors and matrices) before coming back to proteins. Loops are useful if you want to execute a command over and over, and this is one thing that a computer is quite a bit faster at than we are. A simple is taking the square root of the many numbers (here only 1, 2, and 3 are printed to save space).

```
sapply(1:3, sqrt)

## [1] 1.000000 1.414214 1.732051

manynumbers <- sapply(1:1e6, sqrt)
```

Here, we 'applied' the function `sqrt()` to each number between 1 and 3 first, then each number between 1 and 1000000. For this, we used (implicitly) the concept of a vector. So far we have worked with scalars (single numbers) but R is designed to work with vectors as well. The colon operator ':' above is used to generate a sequence of numbers.

An alternative way for expressing loops is via the keyword `for` together with a dummy variable, like so:

```
for (dummyvariable in 1:3) {
  print(sqrt(dummyvariable))
}

## [1] 1
## [1] 1.414214
## [1] 1.732051
```

Back to vectors: The function 'c', which is short for concatenate (or combine if you prefer) can be used to create vectors from scalars or other vectors:

```
x <- 1:4
x

## [1] 1 2 3 4

y <- c(1,3,5,7)
y

## [1] 1 3 5 7
```

Usually, the functions in R are also vectorized, that means they work element by element. So, for example is `sqrt()`, and rather than using `apply` or `for`, we can actually – and this is faster – do:

```
sqrt(1:3)

## [1] 1.000000 1.414214 1.732051

manynumbers <- sqrt(1:1e6)
```

The number of elements of a vector is returned by the function 'length'. Individual elements are addressed using subscripts in square brackets, so x[1] is the first element of x, x[2] is the second, and x[length(x)] is the last. The subscript can be a vector itself, so x[1:3] is a vector consisting of the first three elements of x. A negative subscript excludes the corresponding element, so x[-1] returns a vector with all elements of x except the first one.

Vectors

- Calculate $x + y$.
- Multiply x with 1.5.
- What happens if you add the first three elements of x to y?

R also have extensive support for matrices and higher dimensional arrays. The following function creates a 3 by 4 matrix and fills it by columns with the numbers 1 to 12:

```
M = matrix(1:12, 3, 4)
M[1, 1] # first element of matrix

## [1] 1

M[2, ]  # second row

## [1]  2  5  8 11

t(M)    # transpose M

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

Building the simulation

Now you have all the building blocks needed to put together a simulation of the random walk of a protein. Starting from position $x = 0$, we will simulate 10000 steps of a random walk and calculate the distance a protein moves from its starting position. Here is a quite verbose first implementation of this:

```
x <- 0
nrSteps <- 1e4

for (step in 1:nrSteps) {
        x <- x + stepdir()*steplength
}

x

## [1] -1.902866e-09
```

A more practical method in R is the use of the function 'replicate'. The function is commonly used for repeated evaluation of an expression involving random numbers. The first argument is the number of repeats and the second argument is the expression which is going to be evaluated. Here, we calculate the 10000 steps at once and save the result in a vector. Then we can use the ready-made building block function `sum()` to calculate the distance traveled by summing up all elements in this vector:

```
steps <- replicate(nrSteps, stepdir()*steplength)
dist <- sum(steps)
dist

## [1] -1.075533e-09
```

Traveled distance
Write a function `simNsteps(nrSteps)` that takes an argument 'nrSteps'. Simulate 'nrSteps' steps, sum them up and return the distance traveled.

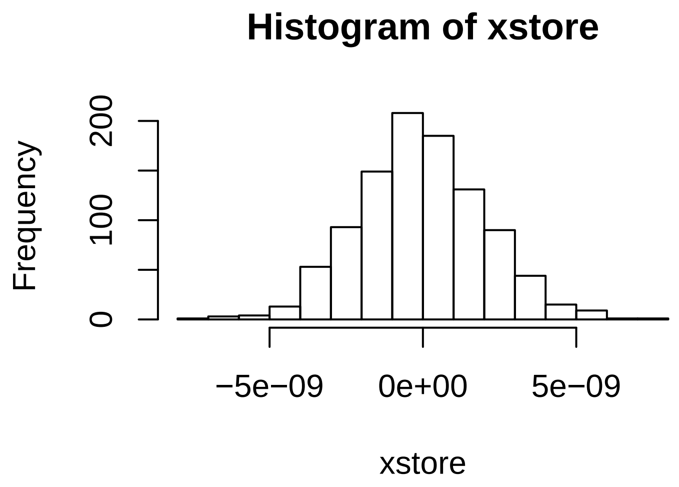
```
simNsteps(1e6)

## [1] -9.14203e-09
```

Now, we want to simulate N steps many times to get the distribution of the distance walked. That means, we need to run the function `simNsteps` multiple times and store all distances in one parameter.

Distribution of distances
Do it.
Note: The simulation might take a while.

With R you can also visualize your results. In this case we can plot the histogram of distances using

```
hist(xstore)
```



As you can see, the proteins go nowhere on average. In addition, one can show that the standard deviation of the traveled distance is proportional not to the time, but to the square-root of the time. This is because the variance of a sum equals the sum of the individual variances. Thus, the *variance* is proportional to time, and the standard deviation to the square-root of time. This is a deep relationship that additionally helps explain why the larger eukaryotes developed extensive transport systems, since relying on diffusion would take to much time.

Mean and standard deviation

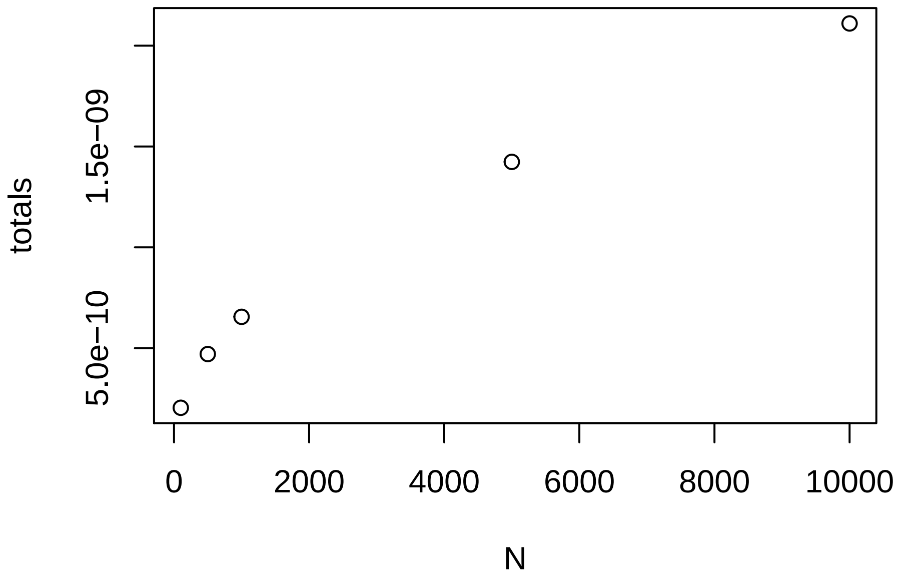
- Calculate the mean and the standard deviation of your distribution of distances 'xstore'. What happens if you use the function 'summary()'?
- Remember the function 'stepdir2()'? If you do the same analysis using 'stepdir()', what is mean distance traveled, and what is the standard deviation?
- Are the mean distances the same? Which test would you use?

In order to see for ourselves that the statement about the standard deviation of the traveled distance is true, we will need to simulate a reasonable amount of random walks for different number of steps. Then, we want to plot different numbers of steps (N) against the standard deviation of the travelled distance. First we store the calculation of the standard deviation in its own function:

```
sdDistance <- function(N, Nsim) {
  xstore <- replicate(Nsim, simNsteps(N))
        return(sd(xstore))
}
```

The function 'sdDistance' takes two arguments: The first is the number of steps (which we want to vary) and the second is the number of simulations (which will be kept fixed). If we want to run the function for different values of N, we can use `sapply()`, introduced above in the early `sqrt()` example – remember? – which is similar to the function 'replicate'. The first argument is a vector and each element should be used for calculating an expression. The second argument is the expression (or function) name and the last arguments are additional parameters for the function (in our case 'sdDistance').

```
N <- c(100, 500, 1000, 5000, 1e4)
totals <- sapply(N, sdDistance, Nsim)

plot(N, totals)
```



Now, when we plot the standard deviation of the traveled distance against the time, we can see for ourselves that the relation between time and standard deviation is not linear, but follows the square root of time.