Contents

Implementing Needleman-Wunsch in R

IMPORTANT NOTES

1. Read the whole of this document *before* you start writing code in `R`. Some of the code snippets are included to aid the *explanation* of key concepts and some are *example* codes that you can use to perform sequence alignment. It will be easier for you to distinguish these if you read the whole document first.
2. This document includes equations; some of you may find these difficult and possibly even a bit scary. I have done my best to explain the equations, but if you do not understand them, then ask *now*, not tomorrow or at some undefined time in the future.
3. Don't be discouraged by not finishing the whole practical *today*; the purpose of the practical is *not* to produce a working sequence alignment program, but for you to learn how to solve problems using code.

The practical

In this and following practicals you will (hopefully):

1. Implement the Needleman-Wunsch algorithm in R. That is, you will write a pairwise sequence alignment program.
2. Play around with parameter settings and different sequences and see how the parameters affect the alignment.

Note that we will not cover the use of *affine* gap penalties in this practical, as that is a little bit more complicated. If you have time, then please think about how this can be done. I will provide additional material (later) giving you one of the possible solutions.

R isn't really a 'natural' language for writing sequence alignment programs with, and your alignment programs will be (relatively) slow. However, this exercise should make you familiar with:

1. How to handle (encode) different types of data in R.
2. How to write functions.
3. How to iterate over elements using the `for` loop.

These skills are pretty much transferrable across different computing languages and so it doesn't matter too much that we are using R here. These skills are also fairly central to the use of R and so the exercise serves two roles.

The algorithm

The Needleman-Wunsch algorithm (NWA from here onwards) can divided into two steps:

1. Determining the values in the the score and pointer tables. The score table contains the alignment scores from the first residue of each sequence up to the position of the cell; the pointer table contains values that can be used to construct the alignment giving rise to the score in each cell. You can think of the pointer table as containing a load of arrows pointing either to the left, up or diagonal up-left.
2. Extracting the aligned sequences by following the pointers from the bottom right position of the table to the top left (i.e. backwards).

Today we will only implement the first part of the NWA. But if you are fast you may consider how to extract the aligned residues from the pointer table.

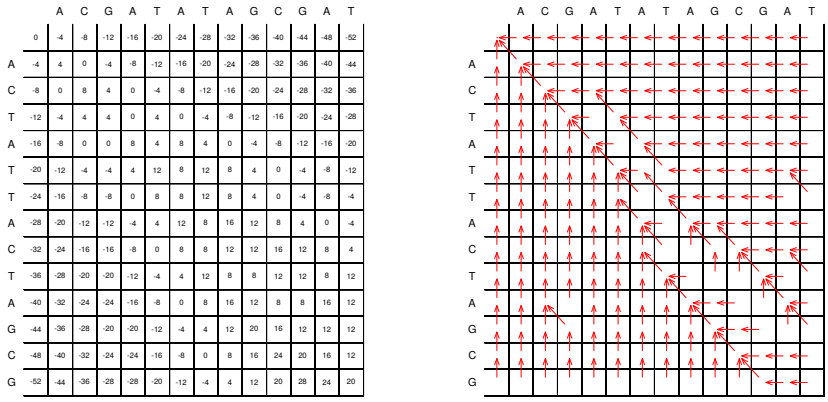

Figure 1: Score (left) and pointer (right) tables. Here the pointers were encoded using the values 1-3, indicating up, left and diagonal arrows respectively.

The scores

For a pair of sequences `A` and `B` of lengths l_a and l_b respectively, the score table (called S from here onwards) needs to have $l_a + 1$ rows and $l_b + 1$ columns[1]. The extra row and column are needed to represent the insertion of gaps at the beginning of each sequence.

The score in the first cell (row 1, column 1 and here referred to as (1,1)) is set to 0. The scores following these in the first row and column represent gap insertions and as such we simply insert the cost of inserting gaps into these. For the first row, the scores will be:

$$S_{1,j} = S_{1,j-1} + P_g$$

Where $S_{1,j}$ refers to the score in row 1 and column j, and P_g is the gap penalty (a negative value). Remember that here j represents columns and only takes values from 2 to $l_b + 1$, since no score is defined for $S_{1,0}$. Note that if S is a matrix in `R`, then $S_{i,j}$ is equivalent to `S[i,j]` and $S_{i,j-1}$ is the same as `S[i,j-1]`.

Similarly, the scores in the first column will be set to:

$$S_{i,1} = S_{i-1,1} + P_g$$

with i taking values from 2 to $l_a + 1$.

All other scores in the table are set to the maximum of:

$$S_{i,j} = \max \begin{cases} S_{i-1,j} + P_g \\ S_{i,j-1} + P_g \\ S_{i-1,j-1} + M \end{cases}$$

where

$$M = \begin{cases} m, & \text{if } A_{i-1} = B_{j-1} \\ mm, & \text{otherwise} \end{cases}$$

And where A_{i-1}, B_{j-1}, m and mm refer to the residues at the indicated positions in sequence A and B, and the match and mismatch penalties[2] respectively (m should be positive and mm negative). Note that we compare the residues at

[1] Obviously you could transpose these values (i.e. swap the number of rows and columns around). But it is useful to specify in your mind how the table dimensions relate to the sequences.

[2] Yes, I know it is somewhat silly to consider the score allocated for a match as a *penalty*, but it makes it neater to not have to think of awards and penalties.

$i-1$ and $j-1$ rather than i and j since the table contains one more row and column than the lengths (l_A, l_B) of the two sequences.

Alternatively if you have a substitution matrix which is indexed by the possible residue values then we could use that to simply look up the penalty associated with a given pair of residues:

$$S_{i,j} = \max \begin{cases} S_{i-1,j} + P_g \\ S_{i,j-1} + P_g \\ S_{i-1,j-1} + M_{A_{i-1},B_{j-1}} \end{cases}$$

This latter case is necessary when dealing with amino acid sequences, but for today we will simply use the simple case of match or mismatch.

To calculate any given score we first have to calculate the scores of the three preceding positions in the table whose alignments can be extended by a single step to include the the given position. We normally do this by simply filling up row after row in the table (this will be shown later).

The pointers

At each position of the table we calculate the score:

$$S_{i,j} = \max \begin{cases} S_{i-1,j} + P_g \\ S_{i,j-1} + P_g \\ S_{i-1,j-1} + M \end{cases}$$

The purpose of the pointer table is to record which of the three alternatives gave rise to the maximum score. If the maximum score was obtained from the first alternative; $S_{i-1,j} + P_g$ then this means that the alignment leading to the cell above (i, j) was extended by adding one residue from A and a gap from B (a downwards movement in the table). We can visualise this as an upward pointing arrow. Similarly if the maximum score was obtained from the last alternative $S_{i-1,j-1} + M$, this would mean that the alignment leading to the cell one to the left and above, was extended by adding a residue from each of the two sequences.

If we record each choice it becomes possible to find the best scoring alignment leading to any cell of the table by tracing the 'arrows' backwards to the top left corner of the table. We can't really store arrows as such, but we can encode the arrows as numbers, eg:

- up: 1
- left: 2
- diagonal: 3

The pointer table has exactly the same dimensions as the score table, and we can define the value of each cell of a table T (for traceback):

$$T_{i,j} = \begin{cases} 1, & \text{if case 1 is max} \\ 2, & \text{if case 2 is max} \\ 3, & \text{otherwise} \end{cases}$$

This looks a bit horrible expressed like this, but is actually rather clearer when expressed in `R`.

It doesn't make sense for the first cell in the pointer table to have a value (as it does not represent any alignment). Hence we set this to 0. All other values in the first row and column would be set to 2 (left) and 1 (up) respectively.

The traceback

Having obtained the score and pointer tables we construct the alignment by following the arrows from the bottom right hand corner until we get to the top left arrow. We can describe this in pseudocode:

1. $A_{al} \leftarrow$ "";$B_{al} \leftarrow$ ""
2. $i \leftarrow l_a + 1; j \leftarrow l_b + 1$
3. do while $i > 1$ or $j > 1$
   - if $T_{i,j} = 1$
     a. $A_{al} \leftarrow A[i-1].A_{al}$
     b. $B_{al} \leftarrow$ "$-$".B_{al}
     c. $i \leftarrow i - 1$
   - if $T_{i,j} = 2$
     a. $A_{al} \leftarrow$ "$-$".A_{al}
     b. $B_{al} \leftarrow B_{j-1}.B_{al}$
     c. $j \leftarrow j - 1$
   - if $T_{i,j} = 3$
     a. $A_{al} \leftarrow A[i-1].A_{al}$
     b. $B_{al} \leftarrow B_{j-1}.B_{al}$
     c. $i \leftarrow i - 1; j \leftarrow j - 1$

If you found the pseudocode difficult to read then you are not alone. It is however, frequently used because it can be both compact and non-ambiguous[3]. In the code above I have used $\leftarrow$ to indicate assignment of values to variables (as done in `R`). I have also used a full stop (`.`) to define the concatenation of two strings (as used in `perl`). Unfortunately there is no standard form of pseudocode, and what I have used here is just what I have cobbled together and may not make that much sense to others. I've included it here as it's often wortwhile to try and express procedures in some form of pseudocode.

[3]Well, at least if the syntax is explained carefully.

Note that when we extract the alignment we need to create two sequences; one from each of A and B and with gaps inserted in the appropriate positions. In the first step above we simply create two empty strings (empty text objects) by setting them to `""`. We then gradually add characters from the sequences or gaps to these depending on the values in the matrix `T`.

We start the traceback from the bottom right hand corner (`i` and `j` get set to the lengths of the sequences plus one), and then decrement `i` and `j` when needed until both `i` and `j` are set to 1.

Implementation in R

The first thing you have to consider when writing any program is how you are going to represent the data in the program. In this case we need to represent the following entities:

1. At least two sequences
2. Penalties for match, mismatch and gaps.
3. Two tables, one for the scores and one for the pointers

Penalties, scores and pointers

R is very convenient for handling numeric values and tabular data. The penalties we use can be represented as numeric vectors and the tables as matrices.

To set the value of a variable you use the `<-` assignment operator. To set the value of a variable called `gap` to -8:

```
gap <- -8
```

To access (use) the value of the variable in an equation simply type the variable name (see below).

Values within matrices are accessed by their row and column using a square bracket notation. Eg. to access the value in row 6 and column 8 of the table `S` we can write:

```
S[6,8]
```

If you have such a table and you execute this code in R all that would happen is that the value[4] would be printed to the console.

That on it's own is not terribly useful. But you can also use this notation to modify the value at a given position:

```
S[6,8] <- S[5,8] + gap
```

Here I am setting the value in `S[6,8]` to that of the cell above it plus the value of a variable called `gap`. For this to work, the value of `gap` must have been set;

[4] Well, at least if the value in that cell had been defined; otherwise R would print `NA` to the console.

otherwise the result would be undefined. Hopefully you will see the similarity of the above expression with the NWA equations described above.

Matrices can be created with a function called `matrix` which provides a number of different ways to define a new matrix. We will see this later on.

Sequences of characters (strings)

String handling in R is rather inefficient and troublesome. Strings can be held in both vectors and matrices (tables). There is no conceptual difference and here I will only describe how to create short vectors of strings. To define two strings representing sequences you can:

```
seq <- c("ACTAGACGAT", "TAGAGACGTTA")
```

Here we have used the `c` (concatenate) function to create a vector of strings. Vectors are ordered lists of values, where all the values are of the same *type*. This means that you can have a vector of strings or numeric values, but not one containing both. There is a special type of vector called a `list` in R. `lists` are special in that they can hold different types of data (including complex data structures). We will come across lists later on.

Strings can be combined with each other using the `paste` function and compared with the equality operator (`==`). To create a single string combining the two defined above you could:

```
seq2 <- paste(seq[1], seq[2], sep="")
```

The `paste` function takes lots of options which we will not cover here.

There is unfortunately no direct way to access individual characters of a string. Instead you have to either split the string into a vector of single character strings or use the `substring` function to extract parts of the string one at a time. In this class we will use the `strsplit` function to split the sequences into vectors of single characters. This wastes a lot of memory, but it makes the accessing of individual elements easier to express.

The strsplit function takes a vector of strings as its first argument; it returns a `list` of vector of strings. As I mentioned above a `list` in `R` is a special kind of vector that can hold elements of different and complex types. To access single elements of a `list` you must use the double bracket `[[]]` notation. Again, lists are ordered sequences of elements, so you can access elements by their position in the list.

To split the two sequences in `seq` defined above into single nucleotides we would simply do:

```
## seq.n to indicate that it holds the nucleotides of seq
seq.n <- strsplit(seq, "")
```

The second argument of `strsplit` defines the pattern that it uses to split the strings. Here we have specified an empty string (`""`); this causes strsplit to split at every character. The result of this operation:

```
seq.n
[[1]]
 [1] "A" "C" "T" "A" "G" "A" "C" "G" "A" "T"

[[2]]
 [1] "T" "A" "G" "A" "G" "A" "C" "G" "T" "T" "A"
```

Note how the `seq.n` is split into two parts denoted with square brackets. We can access the first and second elements using this double bracket notation:

```
seq.n[[1]]
 [1] "A" "C" "T" "A" "G" "A" "C" "G" "A" "T"
seq.n[[2]]
 [1] "T" "A" "G" "A" "G" "A" "C" "G" "T" "T" "A"
```

To access individual characters we simply add another set of single square brackets:

```
seq.n[[1]][3]
[1] "T"
## we can access more than a single character at a time:
seq.n[[1]][1:3]
[1] "A" "C" "T"
```

In this practical we will simply define the sequences directly in the `R` code as shown above. However in any real application we would read the sequences from a file and then handle them as shown above.

A function that calculates the score of any cell

The heart of the NWA is the equation that defines the scores for each cell in the score table:

$$S_{i,j} = \max \begin{cases} S_{i-1,j} + P_g \\ S_{i,j-1} + P_g \\ S_{i-1,j-1} + M \end{cases}$$

In order to perform this calculation we need to know:

1. The value of i and j. That is, the position of the score whose value should be calculated.
2. The scores at the preceding and neighbouring cells (the ones immediately above and to the left of the current cells).
3. The sequences that represent the rows and columns of the score matrix. These are needed to determine the value of M in the above equation.

4. The penalties associated with matches, mismatches and gaps.

Although it might seem a bit like the wrong way around, we often write functions like these *before* we have created the structures that will be passed to the functions. This is actually quite natural as the function should not care about *what* data will be passed to it. However, we will define some rules for what should be passed to the function (i.e. its arguments). In this case we can say that the following must be true:

1. S should be passed as a (numeric) matrix holding the scores (called `scores` in the implementation below).
2. The values of S at $(i-1, j)$, $(i, j-1)$ and $(i-1, j-1)$ must be defined.
3. The sequences should be represented by vectors of characters where:
   a. The first sequence represents rows
   b. The second sequence represents columns
4. The length of the sequences should be one less than the number of rows and columns of the table respectively.

Furthermore we should define what types of data types the function accepts. This can be written as comments above the function definition.

Given this we can easily define a function that takes a suitable set of arguments:

```
## scores should be a matrix of numeric values
## seq should be a list of character vectors holding a single character each
## i, j are the row and column of the score to be calculated.
## match, mismatch and gap are the respective alignment parameters
## note that nrow(scores) = length(seq[[1]]) + 1
##      and  ncol(scores) = length(seq[[2]]) + 1
## match > 0, mismatch < 0 and gap < 0
##
## The function returns a vector of length 2 holding:
## 1. The index of the maximum score (this can be used as a pointer value)
## 2. The maximum score
NWscore <- function(scores, seq, i, j, match, mismatch, gap){
    ## make a vector of scores in order to allow us to use
    ## which.max()
    ## construct a vector of three elements that holds numeric values
    alt.scores <- vector(mode='numeric', length=3)
    ## ifelse is a convenient function that returns one of two values
    ## depending on a a conditional statement
    ## ifelse( <conditional statement>, value 1, value 2)
    ## Here ifelse will return match if
    ## seq[[1]][i-1] is the same as seq[[2]][j-1]
    ## and mismatch otherwise. Remember that match and mismatch
    ## contain the match and mismatch scores.
    M <- ifelse( seq[[1]][i-1] == seq[[2]][j-1], match, mismatch )
    alt.scores[1] <- scores[i-1, j] + gap
```

```
    alt.scores[2] <- scores[i, j-1] + gap
    alt.scores[3] <- scores[i-1, j-1] + M
    ## which.max is a convenient function which does what you might expect
    max.i <- which.max(alt.scores)
    ## in R functions return the evalution of the last statement
    ## Hence the following statement will return a vector of two values:
    ## 1. The index associated with the maximal score.
    ## 2. The maximum score itself.
    c(max.i, alt.scores[max.i])
}
```

To use this function we specify its arguments in brackets like for any other function. The *values* of the arguments that are given in the function call are copied[5] to the function and available within the function by the names given in the first line of the function definition (`function(scores, seq, i, j, match, mismatch, gap)`). The function definition defines the arguments that the function takes and is sometimes called the function *prototype*.

To use the above function you could do[6]

```
cell.sc <- NWscore( sc.table, seq, 5, 6, 4, -4, -8 )
```

But note that in real code you would not write actual numbers, but make use of variables holding those numbers (we will see this later).

The above lines are almost sufficient to implement the first part of the NWA. Apart from this all we really need to do is to initialise the various data structures that we need.

A function that initialises the required matrices

We can make a function that initialises the score and pointer tables. All this function needs to know are:

1. The length of the two sequences
2. The pointer values associated with left and up arrows
3. The gap penalty

For the second of these (the pointer values) we could define the values as part of the init function. That might be a good idea as it would ensure that we do not use different values each time we use the code. However, the correct values of these are implicitly defined by the order of `alt.scores` in the `NWscore` function, and these two functions *must* be consistent for the alignment to work.

[5] Well R *pretends* to copy the value. For now it is fine to believe in this pretence. It only becomes important when you are handling very large data sets which we will not do here. The fact that the arguments are copied means that modifying the copy in the function does not have any effect on the data that was passed to the function.

[6] Well, only if you have defined the arguments first; `sc.table` being a `matrix` and `seq` being a `list` containing the sequences.

To initialise the values of the matrices we will need to set the values of the first row and column. To do this here I will use a *for each* loop style construct. Although `R` does provide a more efficient way of doing this, the loop construct is fairly universal across languages. In `R` a loop usually looks something like this:

```
for(i in 1:10){
    ## do something with i. Maybe just print it to the console:
    ## paste allows us to combine numbers and text.
    print(paste("i:", i))
}
```

The above code can be divided into two parts. The *loop definition*, `for(i in 1:10)` and the *loop body*, everything enclosed in the curly brackets (`{...}`). The code in the loop body is executed once for every element specified in the second part of the loop definition (`1:10`). In R `1:10` evaluates to a vector of elements (`1,2,3,4,5,6,7,8,9,10`) and the loop will be executed once for each of these with the value of `i` set to the value of the element.

Note that there is nothing special about the names I've used above. For example:

```
v <- 1:10
for(k in v){
    print(paste("i:", k))
}
```

will do exactly the same as the previous code.

Given the above (and maybe after reading the ouput of `?matrix`) we can write the initialising function as:

```
NWinit <- function(l1, l2, gap){
    scores <- matrix(nrow=l1+1, ncol=l2+1)
    ## to make the pointer table we can simply copy the scores
    ## table as it has exactly the same dimensions
    ptr <- scores
    ## set the first value to 0
    scores[1,1] <- 0
    ptr[1,1] <- 0
    ## set the values of the first column
    ## remember that the nrow() and ncol() functions return
    ## the number of rows and columns of a matrix.
    for(i in 2:nrow(scores)){
        scores[i,1] <- scores[i-1,1] + gap
        ptr[i,1] <- 1
    }
    ## and then the values of the first row
    for(i in 2:ncol(scores)){
        scores[1,i] <- scores[1,i-1] + gap
        ptr[1,i] <- 2
```

```
    }
    list('scores'=scores, 'ptr'=ptr)
}
```

Since we want to create and initialise two matrices (tables) of values we will want to return these as an `R list` datatype. Here I am returning the matrices as a *named* list. The elements of a named list can be accessed using names as well as the positions of the elements.

Putting together the first part of the NWA

We can now combine the above elements into a script that performs the alignment of two sequences specified in the script. Note that we will use a loop to go through each element of the score and pointer tables.

Note that for the following code to work you will first need to have created the `NWinit` and `NWscore` functions. To do so you need to *execute* the code that defines the functions (shown in previously in this document). One way to do this is to put that code into a second file and then *sourcing* the file (i.e. `source(<file_name>)`, where `<filename>` should be replaced with the name of the file containing the code).

```
## If we have defined NWinit and NWscore in a file called
## "nw_functions.R", then we should source that file:
source("nw_functions.R")

## first let us define some variables:
match <- 4
mismatch <- -4
gap <- -8
seq <- c("ACTAGACGAT", "TAGAGACGTTA")
seq.n <- strsplit(seq, "")

## remember:
## this will return a list of two tables
nw.tables <- NWinit(length(seq.n[[1]]), length(seq.n[[2]]), gap)

## use a nested loop (a loop within a loop) to set
## the scores and pointers that were not set by
## the initialisation function.
for(i in 2:nrow(nw.tables$scores)){
    for(j in 2:ncol(nw.tables$scores)){
        tmp <- NWscore(nw.tables$scores, seq.n, i, j, match, mismatch, gap)
        ## assign the new values into the two tables:
        nw.tables$scores[i, j] <- tmp[2]
        nw.tables$ptr[i, j] <- tmp[1]
    }
```

```
}

## part one done.
```

We can write some custom code to visualise the score and pointer matrices. But we will leave that for another time.

If you have the time please consider how you could write code that extracts the alignment. At some point you will probably need to use `paste()` and the concatenate (`c()`) function.

How to organise your code

I would recommend that you create two files:

1. One containing the function definitions (`NWinit` and `NWscore`). You could call this something like 'NWfunctions.R'. Do not execute code from this file directly.
2. One containing the code defining the sequences and calling the functions. This is your `main` code. This should call `source` on the file containing the functions before you attempt to use the functions. You can call this something like 'NWmain.R'.

And **REMEMBER** to start a new `R` session that is associated with a new directory. Like before this is most easily done by downloading the source to this file (`pairwise_alignment_practical.md`) to a suitable directory and then opening it with `RStudio`.

Three final notes of caution:

1. **Do not** copy and paste from the pdf file you have been provided with. pdf files **do not** only contain text but also lots of formatting code that may be pasted in to your source code without it being visible. Be aware that you may need to use a hex editor to inspect the binary form of your code if you use copy and paste.
2. Be aware that I have not actually run the code in this file. It is likely to contain mistakes. That is by design, as if the code was perfect you would be able to copy it and get something working without understanding anything. Working out where you and others have made mistakes is key to understanding.
3. You will probably make mistakes when writing code that you *cannot* see when looking at the code. You will be convinced that you have done `A` when in fact your have done `B`. This is normal and happens to everyone frequently. There are two ways to overcome problems like this. Either take a break of a day or so and then look at the code again, or, simply ask a friend; they will probably see the mistake quickly.