Computational Biology (Bio 321G) Lecture Notes

Dennis Wylie

August 31, 2020

Contents

Chapter 1

Introduction

1.1 Data and code

Computing generally consists of application of a discrete series of operations to input data hopefully resulting in some desired output.

During a computation (at least the types we will be interested in), data is organized into chunks which our programming languages access through variable names.

The operations we perform on this data are often similarly organized into named chunks which may be called functions, routines, subroutines, methods, subprograms, or (worst of all) "callable units." Depending on who is doing the talking, these terms may be given more specific meanings relating to the details of what sort of operations are being performed, but here I will ignore such distinctions and generally just use the word "function."

Here is a simple computation in R:

```
> exp(1)
[1] 2.718282
```

This applies the predefined function `exp` to the data defined by the expression `1` (which you may be mildly surprised to learn represents the real number 1 in R), and *returns* the result, in this case printing it to the R console.

We can collect the result of this computation into a new variable using one of the *assignment* functions

```
> x = exp(1)   ## or, alternatively,
> x <- exp(1)
```

In case you find it odd to call `=` and `<-` functions, you can confirm that they are indeed such in R by applying the `class` function to them:

```
> class(`=`)
[1] "function"
> class(`<-`)
[1] "function"
```

Because they are a special sort of function (binary operators), they must be surrounded by backticks to use them the way we would an ordinary (non-operator) function:

```
> `=`(x, exp(2))
> x
[1] 7.389056
```

I will generally use `=` for assignment as opposed to `<-` from now on (and promise to use it in its normal binary operator infix syntax!).

By passing the assigment operator as an argument to `class`, we have demonstrated an important feature of R ("functions are first-class objects"). In some cases it is useful to think of functions themselves as data!

1.2 Primitive data types

Consider:

```
> class(1)
[1] "numeric"
> class(1L)       ## gotta put the L after the number to make an integer!
[1] "integer"
> class(TRUE)
[1] "logical"
> class("string")
[1] "character"
```

These—`numeric`, `integer`, `logical`, and `character`—are the most common *primitive data types* we will deal with. Most variables we work with during our computations will be *composites* of these things.

1.3 Vectors

The simplest form of composite object in R is a `vector`, here constructed with the aid of the function `c`:

```
> v = c(0, 2, 1)
> v
[1] 0 2 1
```

You can extract a "slice" of a vector using single-bracket notation:

```
> v[2:3]  ## use single brackets to extract 'slice' of vector
[1] 2 1
```

Or you can extract the individual elements composing the `vector` using double-bracket notation:

```
> v[[1]]  ## use double brackets to extract single element
```

```
[1] 0
> v[[2]]
[1] 2
> v[[3]]
[1] 1
```

All of the elements of a `vector` must be of the same `class`, which can be checked using

```
> class(v)
[1] "numeric"
```

This is actually a reflection of an oddity about R: it doesn't distinguish between a single primitive object (i.e. a single number or string) and a `vector` containing only one such object. In many cases this is actually quite convenient, but there are some reasons why most other programming languages do consider these two cases as being fundamentally different!

1.4 Lists

A more flexible way to join things together in R is provided by `list`s:

```
> list(1, TRUE, "third element", c(0, 2, 1))
[[1]]
[1] 1

[[2]]
[1] TRUE

[[3]]
[1] "third element"

[[4]]
[1] 0 2 1
```

Notice that the fourth element of this list is itself a composite object (a `vector`). It could just as well have been a `list`; in fact, nested `list`s—`list`s containing some elements which are themselves `list`s—are both very useful and very common in R (and many other programming languages).

1.5 Matrices

You've probably run across matrices—composites of $m \times n$ numbers arranged into a tabular layout with m rows and n columns—in a math class at some point. (If not, you'll be learning a bit about them in this class!)

Because of their importance across statistics (and mathematics more generally), R includes the `matrix` as a fundamental data structure.

```
> m = matrix(1:12, nrow=3, ncol=4)
> m
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

You can extract slices and elements from a matrix:

```
> m[1, 2]       ## get element in first row, second column
[1] 4
> m[1:2, 2:3]   ## submatrix w/first two rows, middle two columns of m
     [,1] [,2]
[1,]    4    7
[2,]    5    8
```

As with a `vector`, a `matrix` require that all of their elements have a common `class`, but for a `matrix` this is checked using the `mode` function:

```
> mode(m)
[1] "numeric"
```

Compare this to

```
> class(m)
[1] "matrix" "array"
```

1.6 Data frames

Just like `list`s offer a more flexible alternative for making linear composites of objects, R has a special data type—the `data.frame`—which offers more flexible tabular data layouts:

```
> df = data.frame(A=1:3, B=c("walrus", "penguin", "cod"))
> df
  A       B
1 1  walrus
2 2 penguin
3 3     cod
```

A `data.frame` is actually a kind of `list`: a `list` of `vector`s, with each *column* of the `data.frame` being a `vector`:

```
> df$A       ## extract column named "A" from df
[1] 1 2 3
> df$B
[1] "walrus"  "penguin" "cod"
> df[["B"]]  ## standard list-like syntax for column extraction
[1] "walrus"  "penguin" "cod"
> df[[2]]    ## can select columns by position as well!
```

```
[1] "walrus"  "penguin" "cod"
```

Each column of a `matrix` is a `vector` as well, but unlike a `matrix`, a `data.frame` can contain both `numeric` columns and `character` columns (as does `df` here).

1.7 Naming things

The `data.frame` we just looked at had *named* columns. This can be very useful, as it means we don't have to remember what number goes with which column—and also because the position of a column might change if we add or remove columns from the `data.frame` during our computations.

This same logic frequently applies to elements of `vector`s or `list`s or to the rows and columns of a `matrix` as well. Conveniently, R offers the ability to handle names for indexing all of these objects:

```
> v = c(a=1, b=2)  ## named vector
> v[["b"]]         ## accessing vector element by name instead of position
[1] 2
> names(v)         ## extract the names of the vector
[1] "a" "b"
> names(v) = c("b", "a")  ## reset the names of v
> v[["b"]]         ## now it's the first element instead of second
[1] 1
```

A `list` works pretty much the same way as a `vector` with regard to names:

```
> things = list(a=1:5, b=c("string1", "string2"))
> things[["b"]]
[1] "string1" "string2"
> names(things) = c("b", "a")
> things$b         ## OK, this one is different!
[1] 1 2 3 4 5
```

The last example above shows one difference between how names are handled for a `list` than for a `vector`: the weird dollar-sign syntax introduced for accessing the columns of a `data.frame` above is really `list` syntax inherited by `data.frame`s (since the class `data.frame` is a subclass of `list`).

For a `matrix` it's usually easier to set the names after creation of the object:

```
> rownames(m) = letters[1:3]
> colnames(m) = LETTERS[1:4]
> m
  A B C  D
a 1 4 7 10
b 2 5 8 11
c 3 6 9 12
> m["b", "D"]
```

```
[1] 11
```

The `rownames` and `colnames` functions can be used to get or set the relevant names for a `data.frame` as well:

```
> rownames(df) = c("tusky", "beaky", "scaly")
> df
      A       B
tusky 1  walrus
beaky 2 penguin
scaly 3     cod
> df[c("beaky", "scaly"), "B"]
[1] "penguin" "cod"
```

1.8 Functions

A `function` is a block of code grouping a series of operations to be performed in their own *scope*.

The concept of scope is fundamental in programming, so I'll say a few words about it here. When we access data using a variable name, the computation has to resolve what actual data is being referenced by the indicated name. Given the number of different programs running on a computer—or even the number of different pieces involved in any one program—there is a very high chance that two different chunks of data might have been given the same name (for instance, the name `x` is an ever-popular choice for variables of all sorts).

This is not really a problem specific to programming: you've probably met more than one person named Emily in your life, yet more likely than not this has never been much a problem for you because of scope: If you're having a conversation involving one particular Emily you can usually assume that if someone references an Emily they mean that specific Emily. If you replace the word "conversation" in the last sentence with the phrase "code block" (marked by curly braces in a `function` definition) you can get an intuitive feel for what scopes are in programming.

Let's define a `function` in R:

```
> reverseLookup = function(x) {
      out = names(x)
      names(out) = x
      return(out)
  }
```

`reverseLookup` takes a vector argument, assigns it the name `x` in the scope defined by the curly braces in its definition, and `return`s a new vector (assigned the name `out` within `reverseLookup`'s internal scope) whose elements are the names of `x` and whose names are the elements of `x`.

Let's see an example of how to use this `function`:

```
> geneIdToName = c(
      "ENSG00000244734" = "HBB",
```

```
    "ENSG00000010610" = "CD4",
    "ENSG00000177885" = "GRB2",
    "ENSG00000141510" = "TP53"
  )
> geneNameToId = reverseLookup(geneIdToName)
> geneNameToId[["TP53"]]
[1] "ENSG00000141510"
```

When we used `reverseLookup`, we assigned the `returned` value (which was born named `out` inside the scope of `reverseLookup`) the name `geneNameToId` in the so-called *global scope*. Thus when want to use this object to look up the (Ensembl) gene id corresponding to the gene name TP53, we access it via it's global scope name `geneNameToId`, not the name `out` which it was known by inside `reverseLookup`.

1.9 Operators (including %>%)

I mentioned in the beginning of section 1.1 that the binary operators `=` and `<-` are really just R `functions`. This is equally true of those other popular binary operators `+`, `-`, `*`, and `/`:

```
> c(1, 3, 4) + c(-1, 8, 0)      ## + works elementwise on vectors in R
[1]  0 11  4
> `+`(c(1, 3, 4), c(-1, 8, 0))  ## operator + is also function `+`
[1]  0 11  4
```

Note that `-`, `*`, and `/` also work elementwise on pairs of vectors in R.

That operators are really functions is also true:

- of the logical and operator `&`:

  ```
  > c(TRUE, TRUE, FALSE, FALSE) & c(TRUE, FALSE, TRUE, FALSE)
  [1]  TRUE FALSE FALSE FALSE
  > `&`(c(TRUE, TRUE, FALSE, FALSE), c(TRUE, FALSE, TRUE, FALSE))
  [1]  TRUE FALSE FALSE FALSE
  ```

- of the logical or operator `|`:

  ```
  > c(TRUE, TRUE, FALSE, FALSE) | c(TRUE, FALSE, TRUE, FALSE)
  [1]  TRUE  TRUE  TRUE FALSE
  > `|`(c(TRUE, TRUE, FALSE, FALSE), c(TRUE, FALSE, TRUE, FALSE))
  [1]  TRUE  TRUE  TRUE FALSE
  ```

- as well as of the unary (only one argument) operator `!` (not):

  ```
  > !c(TRUE, TRUE, FALSE, FALSE)
  [1] FALSE FALSE  TRUE  TRUE
  > `!`(c(TRUE, TRUE, FALSE, FALSE))
  [1] FALSE FALSE  TRUE  TRUE
  ```

While there are some occasions where the standard function syntax for such operators is useful, the operator syntax is preferred in most cases. Knowing the relationship between operators and functions is of more general use in that it allows us to define our own new operators:

```
> `%concatenate%` = function(x, y) {paste0(x, y)}
> "Bio" %concatenate% "informatics"
[1] "Bioinformatics"
```

A couple of things to note:

1. The percentage signs on either side of the word concatenate in the definition of `%concatenate%` above are required whenever you want to define new operators in R; only the builtins (`+`, `-`, `*`, `&`, etc.) get to omit these.
2. Unlike many (most?) other programming languages, R does not provide a standard operator for string concatenation (`+` will not do this unless you redefine it, which I'd advise against unless you really know what you're doing!). Instead you can use `paste0` as shown above; `paste` is similar, but by default puts a space between the two strings `x` and `y` when joining them together.

I won't use `%concatenate%` any more in these notes, but I do want to bring to your attention one non-base R operator of great import, `%>%` from the package `magrittr`:

```
> ## install.packages("magrittr")  ## uncomment and run if necessary
> library(magrittr)
> "Bio" %>% paste0("informatics")
[1] "Bioinformatics"
```

What does `%>%` (the so-called *pipe* operator) do? As used above, it simply substitutes its first argument—the expression immediately to the left of it, since it is a binary operator—into the function call made in its second argument (immediately to the right of `%>%`). Thus, the example above is equivalent to `paste0("Bio", "informatics")`.

Admittedly, for just concatenating two character strings, this doesn't seem all that impressive, but consider the following slightly more long-winded example:

```
> "Bioinformatics" %>%
      paste("is one") %>%
      paste("of the rare words") %>%
      paste("in English") %>%
      paste("containing ioi.")
[1] "Bioinformatics is one of the rare words in English containing ioi."
```

compared to

```
> paste(paste(paste(paste("Bioinformatics",
                          "is one"),
                    "of the rare words"),
              "in English"),
        "containing ioi.")
[1] "Bioinformatics is one of the rare words in English containing ioi."
```

I would argue that the piped version of this bit of code—which we might call a *pipeline*—is more elegant, easier to read, and really just plain better than the non-piped version!

Of course, in the case of R's `paste`, I should note that a better solution is to take advantage of its ability to take a variable number of arguments:

```
> paste("Bioinformatics",
        "is one",
        "of the rare words",
        "in English",
        "containing ioi.")
[1] "Bioinformatics is one of the rare words in English containing ioi."
```

One last thing to note about the `x %>% fun(y)` syntax: it has a certain similarity to the `x.fun(y)` syntax common to object-oriented code in languages like C++, Java, and Python and can, in certain situations, offer some of the advantages of this ordering.

1.10 Logical indexing

The logical operators `&`, `|`, and `!` are very useful in selecting sub`vector`s, sub`list`s, sub`data.frame`s, etc. using *logical indexing*. Here's an example:

```
> someIntegers = 1:10
> remainderAfterDividingBy2 = someIntegers %% 2
>  ## the %% operator yields the remainder of its first argument
>  ## when divided by its second argument;
>  ## here the 2 provided as second arg is recycled to match length of first
> remainderAfterDividingBy2
 [1] 1 0 1 0 1 0 1 0 1 0
> isEven = (remainderAfterDividingBy2 == 0)
>  ## the == operator tests equality of its first argument
>  ## with its second argument;
>  ## here the 0 provided as second arg is recycled just as above
> isEven
 [1] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE
> someIntegers[isEven]  ## index someIntegers by logical vector isEven
[1]  2  4  6  8 10
```

The expression `someIntegers[isEven]` here takes two `vector`s: `someIntegers` and `isEven`, the second of which (`isEven`) is a `logical vector`, and returns a new `vector` which consists of only those elements `someIntegers[[i]]` occurring at positions i where `isEven[[i]]` takes the value `TRUE`.

By building up compound expressions using logical operators and comparison operators (like `==`, `<`, `<=`, `>=`, and `>`), we can select just those elements satisfying complex conditions. We will do this a lot.

1.11 `if` statements

Logical values are also useful in R for conditionally controlling which operations described in a chunk of code get evaluated and which do not. This is most simply accomplished using `if` statements, such as:

```
> x = 7
>  ## if keyword should be followed by parenthesized statement which
>  ## evaluates to single TRUE or FALSE value:
> if (x %% 2 == 0) {   ## and then by code block surrounded by { and }:
      cat("x is even.")  ## cat function prints text (without quotes)
  } else if (x %% 2 == 1) {
      cat("x is odd.")
  } else {
       ## no way to get here if x is an integer, since then
       ## x %% 2 is either 0 or 1...
      cat("x is not an integer.")
  }
x is odd.
```

Just as we used curly braces to group together a series of operations to be performed together when a `function` is called, we can also group together a series of operations `...` to run only if the `condition` specified in `if (condition) {...}` evaluates to `TRUE`.

1.12 `while` and `for` loops

An `if` statement only executes the block of code under its control either once if the provided condition is `TRUE` or zero times if it is `FALSE`. By contrast, a `while` *loop* such as

```
> x = 296
> while (x %% 2 == 0) {
      x = x / 2
      cat("x is now", x, "\n")  ## "\n" is programming-speak for newline
  }
x is now 148
x is now 74
x is now 37
```

Of course, we have to be careful with `while` loops: Had we set `x=0` before running the loop shown above, it would keep running until we intervened to stop it, since `x` would be repeatedly reset to the same `0` value and the condition `x %% 2 == 0` would always evaluate to `TRUE`.

The common situation in which one knows that a given block of code will need to be repeated for all values of a particular variable within a particular range has its own special construct, the `for` loop:

```
> for (power in 1:3) {
      eToPower = exp(power)
```

```
      cat("e to the", power, "is", eToPower, "\n")
  }
e to the 1 is 2.718282
e to the 2 is 7.389056
e to the 3 is 20.08554
```

There's nothing special about the name `power` or the use of the colon-defined vector `1:3`:

```
> for (i in c(1, 2, 3)) {
      cat("e to the", i, "is", exp(i), "\n")
  }
e to the 1 is 2.718282
e to the 2 is 7.389056
e to the 3 is 20.08554
```

works just as well.

(Note: if you think `i` should be the imaginary unit and that `exp(i)` should thus evaluate to some complex number, you might try running `exp(1i)`.)

Chapter 2

Modeling and Simulation

2.1 Population growth

2.1.1 Geometric growth

Growth and reproduction—and the limits placed thereon by available food sources—were subject of one of the earliest quantitative models in biology (by Malthus in 1798):

$$n(t+1) = rn(t) \tag{2.1}$$

Eq (2.1), which defines so-called *geometric growth*, says that the population at time $t+1$ is simply a multiple r of the population at time t. Its solution is

$$n(t) = r^t n_0 \tag{2.2}$$

where n_0 is the population at time $t = 0$.

This is a good time to introduce `ggplot`, a popular data visualization package in R. First let's set up some data to visualize, using Eq (2.2) with $r = 2$ and $n_0 = 1$:

```
> t = 0:10  ## equivalent to t = c(0, 1, 2, 3, 4, 5, 6, 7, 8, 10)
> growthData = data.frame(t=t, n=2^t)
> growthData
    t   n
1   0   1
2   1   2
3   2   4
4   3   8
5   4  16
6   5  32
7   6  64
8   7 128
9   8 256
```

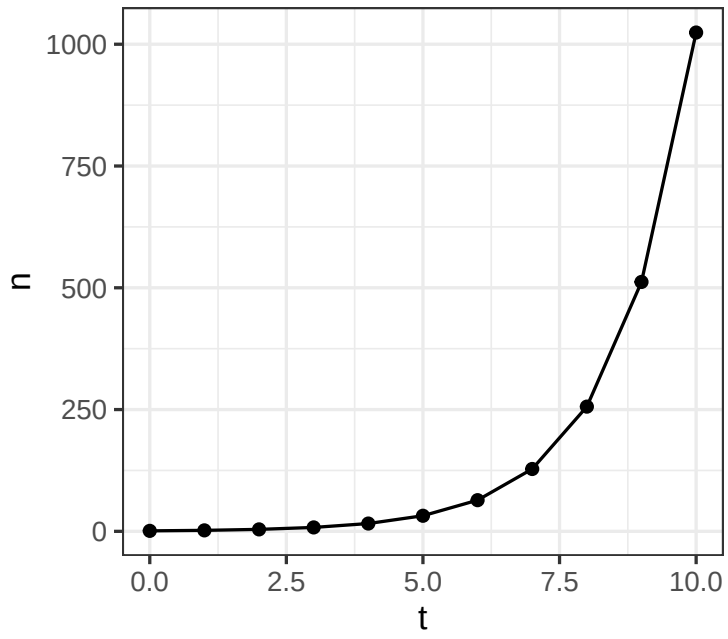
```
10  9  512
11 10 1024
```

I want to emphasize a couple of points here:

1. When creating `growthData`, we use `t` as both a column name in the `data.frame`—when it shows up on the left-hand side of the equals sign of `t=t`—as well as as the variable `t`.
    - The variable `t` in fact shows up twice: once in the right-hand side of `t=t` and once in the right hand side of `n=2^t`.
2. Just as with the other binary operators discussed in section 1.9, the binary operator `^` works elementwise on its second argument (the vector `t`),
    - though here we supplied only a single number (`2`) as its first argument: this number is *recycled* to be used with each element of `t`.

Now, on to plotting!

```
> ## install.packages("ggplot2")  ## uncomment and run if necessary
> library(ggplot2)
> theme_set(theme_bw())  ## tell ggplot to use white background
> gg = ggplot(
      growthData,    ## first argument is data.frame to draw from
      aes(x=t, y=n)  ## aes function maps x, y positions to column names
  )
> gg = gg + geom_point()   ## add points to plot
> gg = gg + geom_line()  ## add lines connecting points
> print(gg)
```



Note that `ggplot` builds up an R object—I named it `gg`—representing the plot which must specify the data set (contained in a `data.frame` object) to be visualized, the "mapping"

(specified using `aes`) between things like x- and y-positions in the plot and the column names of the data set, and finally the `geom`etric objects (points, lines, etc.) to include in the plot. This building up is done using the `+` binary operator; remember that such operators are really just R functions with two arguments and can work differently depending on what types of objects are passed into them as arguments!

2.1.2 The logistic map

The rapid growth implied by Eq (2.2) (as shown in the plot above) is not generally sustainable in a world of limited resources, however. As n grows over time, at some point the rate of increase must slow down. There are many ways in which this might happen, but one simple modification to the geometric model that incorporates such effects is the *logistic map*:

$$n(t+1) = rn(t)\left(1 - \frac{n(t)}{k}\right) \tag{2.3}$$

Eq (2.3) modifies Eq (2.1) by multiplying by $(1 - n(t)/k)$. This factor is approximately 1 for $n(t) \ll k$, so that growth looks geometric starting from very small numbers, but then slows down such that if

$$n(t) = \frac{k(r-1)}{r} = n_* \tag{2.4}$$

then $n(t+1)$ will also be equal to n_*, that is, there will be no continued growth at all! For $n(t) > n_*$, you will in fact obtain $n(t+1) < n(t)$; that is, the population shrinks instead of growing.

We can simplify Eq (2.3) a bit by *nondimensionalizing* it as follows: Define

$$x(t) = \frac{n(t)}{k} \tag{2.5}$$

so that, dividing both sides of Eq (2.3) by k, we obtain:

$$x(t+1) = rx(t)\left(1 - x(t)\right) \tag{2.6}$$

The nontrivial steady state value of x is then

$$x_* = \frac{r-1}{r} \tag{2.7}$$

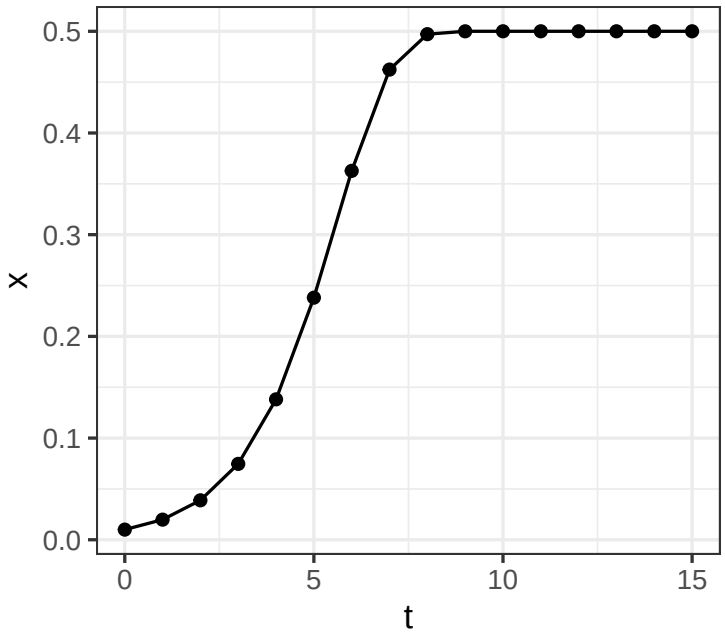
What does population growth under the logistic map look like? Let's define a function

```
> logisticMapTrajectory = function(r, x0, tmax) {
      x = rep(x0, tmax+1)
       ## note: above sets all (tmax+1) elements of vector x to x0;
       ## these values will be replaced below with correct values
      for (t in 1:tmax) {
```

```
        x[[t+1]] = r * x[[t]] * (1-x[[t]])
    }
    return(data.frame(
        t = 0:tmax,
        x = x,
        r = r  ## keep track of which r value was used:
               ## r is recycled tmax+1 times here!
    ))
}
```

which we can use to plot trajectories:

```
> traj_r2 = logisticMapTrajectory(r=2, x0=0.01, tmax=15)
> gg = ggplot(
    traj_r2,       ## traj_r2 is a data.frame with columns t, x, and r
    aes(x=t, y=x)  ## put t column on x-axis, x column on y-axis
) + geom_point() + geom_line()
> print(gg)
```



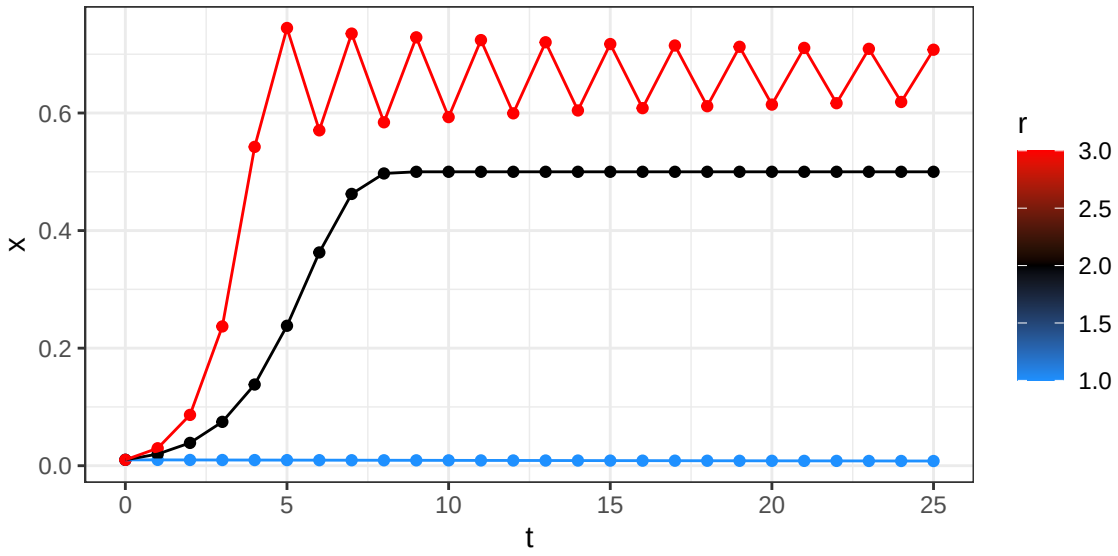
Here we see a trajectory which starts out looking like geometric growth with $r = 2$ but then *saturates* at a population of $x_* = 0.5$.

What do trajectories look like with other values of r?

```
> traj_r1 = logisticMapTrajectory(r=1, x0=0.01, tmax=25)
> traj_r2 = logisticMapTrajectory(r=2, x0=0.01, tmax=25)
> traj_r3 = logisticMapTrajectory(r=3, x0=0.01, tmax=25)
> all3trajectories = rbind(traj_r1, traj_r2, traj_r3)
>  ## rbind row binds together data.frames into one tall data.frame
```

Let's try plotting all of these together using `ggplot`:

```
> gg = ggplot(
      all3trajectories,  ## pull from all 3 trajectories
      aes(x=t, y=x, group=r, color=r)
  ) + geom_point() + geom_line()
> gg = gg + scale_color_gradientn(colors=c("dodgerblue", "black", "red"))
> print(gg)
```

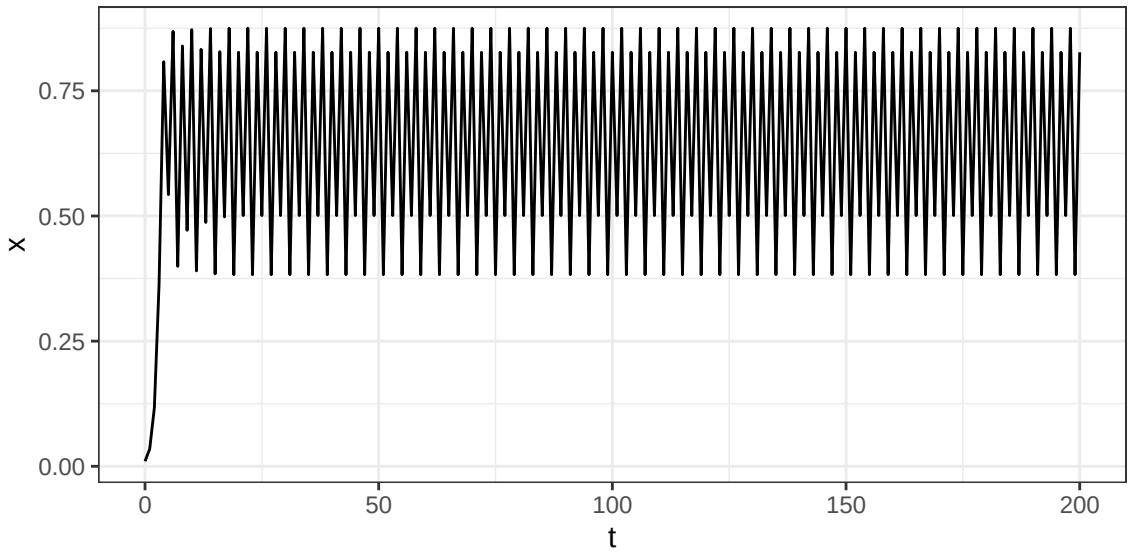


For $r \leq 1$, the trajectories just die off toward 0; this isn't so surprising since our nontrivial steady state $x_* = \frac{r-1}{r} \leq 0$ when $r \leq 1$.

But what about $r = 3$? In this case the trajectory looks almost periodic, though if you follow it long enough the amplitude of the fluctuations shrinks and the trajectory ultimately settles down to the stable value x_*.

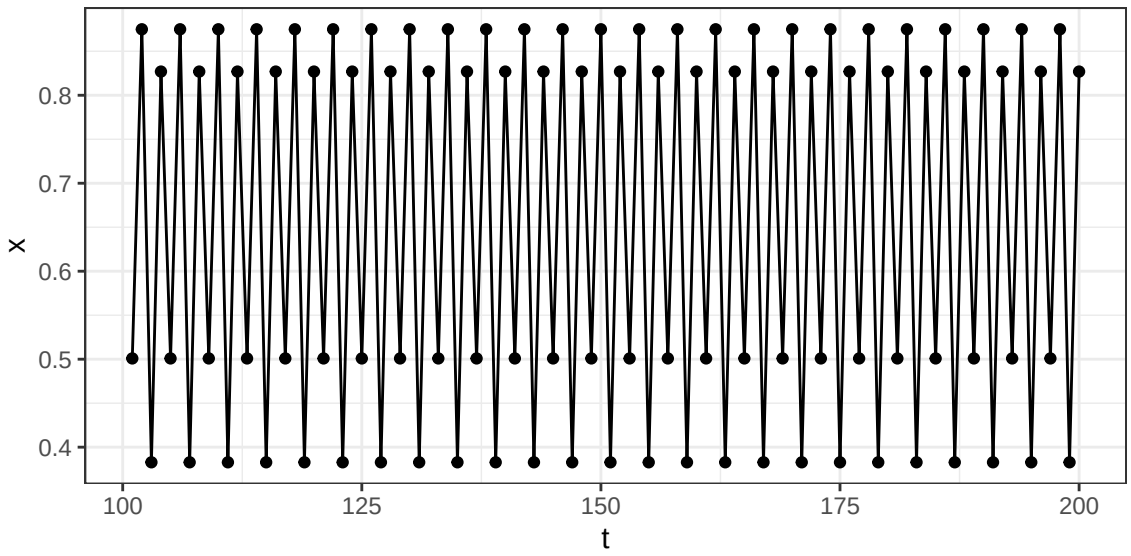
Things get more interesting for $r > 3$:

```
> traj_r3p5 = logisticMapTrajectory(r=3.5, x0=0.01, tmax=200)
> ggplot(traj_r3p5, aes(x=t, y=x)) + geom_line()
```



Here we see a trajectory which, after an initial transient period, settles into periodic behavior with a period of 4 time units. For reasons which will become apparent shortly, let's chop off the initial transient:

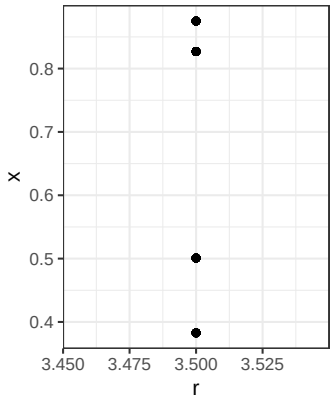
```
> traj_r3p5_last100 = traj_r3p5[traj_r3p5$t > 100, ]
>  ## - line above uses logical indexing to keep only rows of
>  ##   traj_r3p5 for which t-column has value > 100
>  ## - all columns (t, x, and r) are retained b/c nothing put
>  ##   between the comma and end-square-bracket
> ggplot(traj_r3p5_last100, aes(x=t, y=x)) + geom_point() + geom_line()
```



If we're only interested in this long-term behavior, we might collapse the whole trajectory by plotting the growth rate parameter r on the x-axis instead of the time t and omitting the

lines connecting the points:

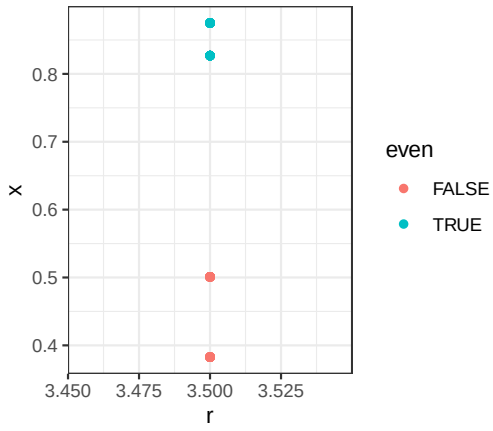
```
> ggplot(traj_r3p5_last100, aes(x=r, y=x)) + geom_point()
```



Keep in mind that there are 100 points in the plot above, but that since they take only 4 distinct values, and since we have removed any information about time t from the plot, they get plotted on top of each other.

We can keep at least a minimal amount of information of the order in which the points appear by keeping track of whether we're on an even or odd time point:

```
> timeIsEven = (traj_r3p5_last100$t %% 2 == 0)
>  ## %% operator returns remainder upon dividing by right-hand-side
>  ## even numbers are those with 0 remainder when dividing by 2
> traj_r3p5_last100$even = timeIsEven
>  ## now have column with value TRUE for even t and FALSE for odd t
>  ## in data.frame traj_r3p5_last100
> ggplot(traj_r3p5_last100, aes(x=r, y=x, color=even)) + geom_point()
```



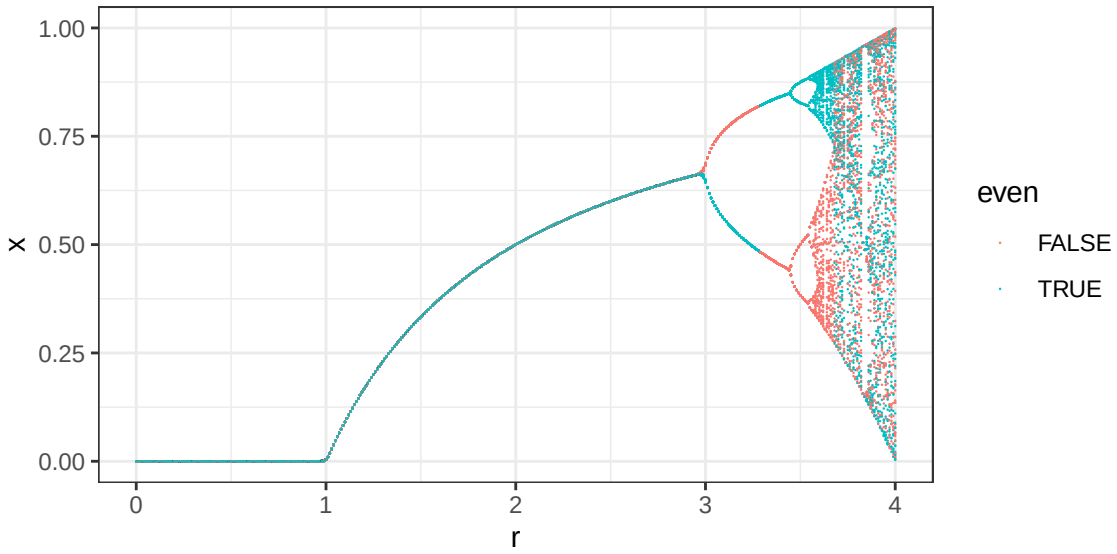
This shows us that the period four trajectory bounces back and forth between higher values (at even times when starting from $x_0 = 0.01$) and lower values (at odd times), though the height of the higher values (and the depth of the lower values) varies from one period to the next!

That lonely x-axis suggests we should include some other values of r as well: let's do that...

```
> lotsOfTrajectories = list()  ## start with empty list
> for (r in seq(from=0, to=4, by=0.01)) {
       ## seq function creates vector of values ranging from
       ## the 'from' value to the 'to' value in steps of the 'by' value
      lotsOfTrajectories[[length(lotsOfTrajectories)+1]] =
             logisticMapTrajectory(r, x0=0.01, tmax=200)
       ## note lack of anything to right of '=' sign two lines
       ## above tells R to look at next line for value to assign
  }
```

Now we have long list of `data.frame`s we'd like to plot together. We used `rbind` to do this with 3 trajectories before, but that required specifying each `data.frame` as a separate argument to `rbind`. Don't want to write a function call to `rbind` by typing out hundreds of arguments! Luckily there's a way to to tell R to call a function with a whole `list` of arguments:

```
> roundTrajectories = do.call(rbind, lotsOfTrajectories)
>  ## do.call says to call the function specified by first argument
>  ## - rbind in this case
>  ## with a provided list of arguments
>  ## - lotsOfTrajectories in this case
>  ## now roundTrajectories is single data.frame containing
>  ## all of the trajectories
>  ## - perfect for ggplot!
> roundTrajectories$even = (roundTrajectories$t %% 2 == 0)
> ggplot(
      roundTrajectories[roundTrajectories$t > 100, ],
      aes(x=r, y=x, color=even)
  ) + geom_point(size=0.1, shape=16)  ## shape=16 allows small points
```



Now we can see how the logistic map trajectories behave over a wide range of growth rates r (May (1976)):

- $r \leq 1$: population dies away
- $1 < r \leq 3$: population ultimately stabilizes at x_*
- $3 < r < 1 + \sqrt{6}$: oscillation with period 2
- $1 + \sqrt{6} < r < 3.54409\ldots$: oscillations with period 4
- $3.54409\ldots < r < 3.56995\ldots$: period-doubling cascade
- $3.56995\ldots < r \leq 4$: chaos!

2.1.3 Deterministic chaos

Both the geometric growth model and the logistic map are examples of *dynamical models*: models which describe the evolution of the state of a system over the course of time. Moreover, both of these models provide a rule which, in principle, allows one to determine exactly the state of the system at all future times given knowledge of the current system state. Such systems are referred to as *deterministic* systems (and can be contrasted with so-called *stochastic* systems, in which there is a random component to the rules for evolving the system forward in time).

Deterministic dynamical systems have been heavily used in scientific modeling for a long time now, especially since Newton showed that a few simple physical laws can be used to derive deterministic models simultaneously leading to highly accurate predictions of both terrestrial and celestial phenomena. The ongoing success of these models over the following century, and then of similarly constructed deterministic differential equation models of electromagnetic fields, led to a widespread belief that deterministic modeling would ultimately allow for the prediction of the course of virtually all of nature's operations.

However, starting in the late 19th century, certain limitations in this scientific program started to appear. While strictly speaking still deterministic, some differential equation models were found to exhibit "sensitivity to initial conditions" (colloquially known as the "butterfly effect"): Any small change to the state of the system at time t will become exponentially magnified as time moves forward (Strogatz (2015)). Thus *any* imprecision in the measurement of the current state of such a system will render predictions in the relatively near future subject to very great uncertainty. This is the main ingredient in what is known as *deterministic chaos*.

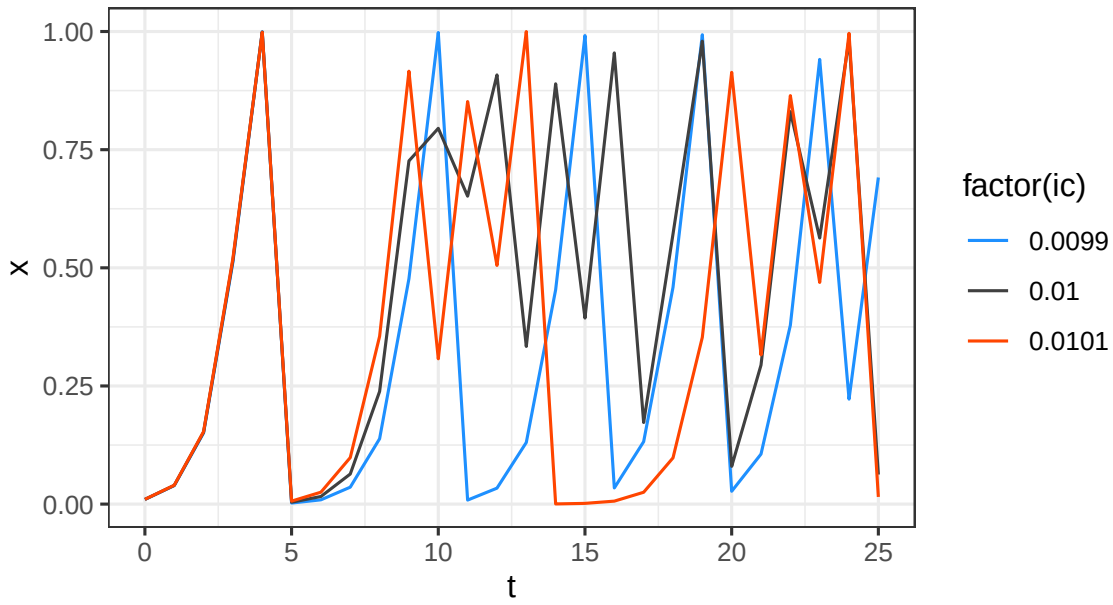
In section 2.1.2 I indicated that the logistic map model with r at or just below 4 it will exhibit chaotic behavior. We can demonstrate this by considering three different $r = 4$ logistic map trajectories starting from the very nearby initial conditions $x_0 \in \{0.0099, 0.01, 0.0101\}$:

```
> r4trajectories = list()
> for (initialCondition in seq(from=0.0099, to=0.0101, by=0.0001)) {
      r4trajectories[[length(r4trajectories)+1]] =
              logisticMapTrajectory(r=4, x0=initialCondition, tmax=25)
       ## add column ic indicating what the initial condition was in
       ## so that we can keep track of which simulation was which
       ## after we rbind all of these trajectories together
       ## into one data.frame:
      r4trajectories[[length(r4trajectories)]]$ic = initialCondition
```

```
      ## (since initialCondition is just a single number, the column
      ##  ic will just end up being 201 copies of this same number)
  }
>  ## use do.call with rbind to reformat list into data.frame again;
>  ## this time we'll just redefine r4trajectories itself:
> r4trajectories = do.call(rbind, r4trajectories)
```

Having run the three simulations starting from the different initial conditions 0.0099, 0.01, and 0.0101 and packaged them up into a single `data.frame` object, we can use `ggplot` to visualize the three resulting trajectories:

```
> gg = ggplot(r4trajectories,
              aes(x = t,
                  y = x,
                  group = ic,  ## group aesthetic tells geom_line
                               ## to treat rows of r4trajectories
                               ## with different ic values as part
                               ## of separate lines
                   ## now set color aesthetic to categorical,
                   ## or factor, version of ic column
                   ## (easier to deal with coloring small number of
                   ##  values in ggplot if they are treated as discrete
                   ##  values instead of continuous numbers):
                  color = factor(ic)))
> gg = gg + geom_line()
> gg = gg + scale_color_manual(
      values = c("dodgerblue", "gray25", "orangered")
  )
> print(gg)
```

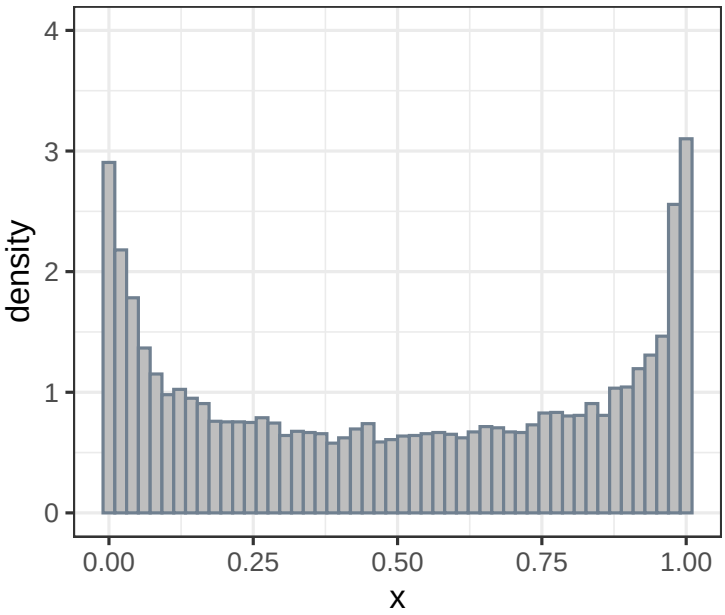


From the plot we can clearly see that an error in the 4th decimal in the estimation of the initial state of the logistic map trajectory leads to totally incorrect prediction of what the trajectory looks like after as few as 10 iterations!

This certainly doesn't mean that there is nothing to be learned from modeling chaotic systems. For one thing, you can see that the immediate future is still predictable even if the present is subject to the inevitable measurement error: for the first few time steps the blue, gray, and orange trajectories move together just as one would expect given the similarity in their initial states. But beyond this, even if attempts to predict all of the local details of a particular trajectory of a chaotic system may be doomed, there are larger scale properties that may be understood.

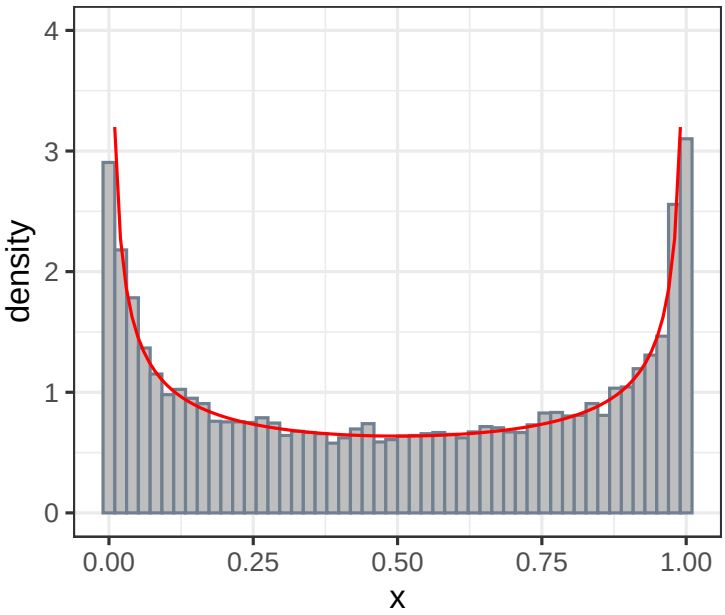
For instance, considering Eq (2.6) with $0 \leq x(0) = x_0 \leq 1$ and $r = 4$, it is not hard to see that it must also be true that $0 \leq x(1) \leq 1$, and thus by the same logic that $0 \leq x(2) \leq 1$, and so on—that is, a trajectory governed by Eq (2.6) which starts with $x_0 \in [0, 1]$ will remain in that interval for all times. But for $r = 4$ we can actually say something much more interesting about the trajectories constricted to this range: Most of them can be characterized *probabilistically*. To illustrate this, let's return to our favorite initial condition of $x_0 = 0.01$, run it for 10,000 steps, and examine the resulting distribution of $x(t)$ values without worrying about the time indices t:

```
> r4trajLong = logisticMapTrajectory(r=4, x0=0.01, tmax=10000)
>  ## use ggplot to histogram this data:
> gg = ggplot(r4trajLong)
> gg = gg + geom_histogram(
       ## can specify aes (aesthetic) mappings as argument to individual
       ## geoms as well as in the initial ggplot function call:
      aes(x=x, y=..density..),  ## y=..density.. here says to scale y-axis
                                ## so that total *area* of all bars together
                                ## sums to 1, like a probability distribution
      bins = 50,
      fill = "gray",        ## inner fill color of the histogram bars
      color = "slategray"   ## the outline color of the bars
  )
>  ## set limits of y-axis to between 0 and 4:
> gg = gg + ylim(0, 4)
> print(gg)
```



Aside from a few exceptional cases like the trajectory $x(t) = 0$ for all t, almost all trajectories of the $r = 4$ version of the logistic map system will result in this exact same distribution of time spent visiting various different x values over the long haul! The functional form of this distribution for $r = 4$ turns out to be $\frac{1}{\pi\sqrt{x(1-x)}}$ (Berliner (1992)), which is the probability density function associated with the arcsin distribution (so-named because if has *cumulative* distribution function proportional to arcsin, the functional inverse of sin):

```
> ## install.packages("VaRES")  ## uncomment and run if necessary
> library(VaRES)  ## has arcsine distribution functions
>                 ## (we want darcsine,
>                 ##  the probability *d*ensity function for *arcsin*)
>  ## use stat_function to add the plot of a function to a ggplot object:
> gg = gg + stat_function(fun=darcsine, color="red")
> print(gg)
```



While this is not necessarily the only—or perhaps even most important—reason probabilistic modeling plays as important a role as it does in biological modeling, the fact that chaotic behavior is quite common in the type of models which seem most plausible for many biological phenomena is worth noting. It would seem to suggest that if we want to make quantitative predictions regarding the the response of such systems to experimental stimuli, we may often be limited to probabilistic statements.

2.2 Continuous time and differential equations

The models we've studied so far treat time as a discrete quantity which advances in integer units. For some biological problems this makes sense, but in many cases more realistic models involve treating time as a continuous variable.

The most common deterministic continuous time models applied to biological (or really much of any) dynamical systems are *differential equations*.

2.2.1 Exponential growth

Differential equations compose a major branch of modern mathematics of which I will give only the briefest glimpse, but we can get a good start by revisiting Malthus's geometric growth model in continuous time. The differential equation

$$\frac{dx}{dt} = rx(t) \tag{2.8}$$

says that, for very small time intervals dt,

$$x(t + dt) \approx x(t) + rx(t)dt = (1 + rdt)x(t) \tag{2.9}$$

I've switched notation from n, which is often used as a variable name for integer-valued variables, to x, since a differential equation model generally implies continuous valued variables. Thus we should think of x in this model as representing some sort of approximatated population level applicable when the numbers involved are large; we will see models which can more directly handle truly integer-valued variables in continuous time later.

Eq (2.9) suggests that if we want to know what the value of $x(t)$ is starting from $x(t_0)$, where $t - t_0$ is not necessarily a "small" value, we might try dividing it into a very large number $\frac{t-t_0}{dt}$ of small increments dt and thus obtain

$$x(t) \approx (1 + rdt)^{\frac{t-t_0}{dt}} x(t_0) \tag{2.10}$$

Eq (2.10) can be directly applied computationally to approximately solve Eq (2.8) by choosing a small enough value of dt and cranking through the calculation. This is an example of the "Euler method" for numerically solving a differential equation. While this is the most straightforward approach to solving a differential equation, it is rarely used in practice because more efficient—though much more complicated—methods are available these days.

Eq (2.10) can also be solved exactly using the methods of calculus: Taking the limit with the length of the small intervals $dt \to 0$, the factor multiplying $x(t_0)$ becomes the exponential function

$$x(t) = e^{r(t-t_0)} x(t_0) \tag{2.11}$$

equivalently written as

$$x(t) = \exp\left[r(t - t_0)\right] x(t_0) \tag{2.12}$$

Fitting exponential growth to data

The simple exponential growth model has the virtue of being relatively easy to fit to data using simple linear regression, since, taking the (natural) logarithm of both sides of Eq (2.12):

$$\log\left[x(t)\right] = r(t - t_0) + \log\left[x(t_0)\right] \tag{2.13}$$

However, while simple linear modeling is an easy way to fit an exponential growth model to data, we should still think about whether it is truly appropriate. In particular, we should consider whether the following assumptions of linear modeling are really satisfied:

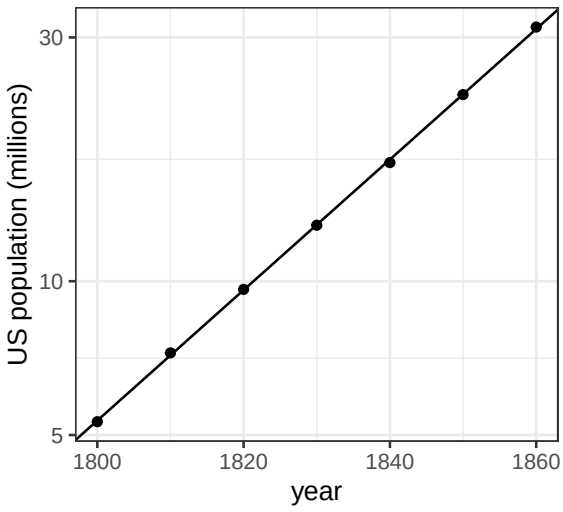
1. **Linearity** of relationship: this is confirmed by virtue of Eq (2.13).
2. **Homoskedasticity** variance of residual is constant independent of predictor value t; will discuss this further when we get to stochastic models.
3. **Independence** of observations: do we really think population at time t_2 is independent of population at t_1? A more theoretically sound approach will be described when we get to stochastic models.

4. **Normality** of residuals; will again discuss this further when we get to stochastic models.

Despite the fact that the independence assumption seems problematic at best, we'll go ahead and use simple linear regression to fit an exponential model using the ordinary least squares (OLS) method.

As an example data set, here are the census values for the population of the United States from 1800 to 1860, packaged into an R `data.frame`:

```
> year = seq(1800, 1860, by=10)
> population = c(5.31, 7.24, 9.64, 12.87, 17.07, 23.19, 31.44)
>  ## population numbers are in millions of people
> usPop = data.frame(year=year, population=population)
> usPop
  year population
1 1800       5.31
2 1810       7.24
3 1820       9.64
4 1830      12.87
5 1840      17.07
6 1850      23.19
7 1860      31.44
> linModel = lm(log10(population) ~ year, data=usPop)
> coef(linModel)  ## extract vector of linModel regression coefficients
 (Intercept)         year
-22.26412125   0.01277302
> library(ggplot2)
> theme_set(theme_bw())
> gg = ggplot(usPop, aes(x=year, y=population))
> gg = gg + scale_y_log10()  ## semilog plot: population is shown in log-scale
> gg = gg + geom_point()
>  ## now add geom_abline to include regression line from linModel;
>  ## note that this only makes sense with the inclusion of scale_y_log10()
> gg = gg + geom_abline(
      intercept = coef(linModel)[["(Intercept)"]],
      slope = coef(linModel)[["year"]]
  )
> gg = gg + ylab("US population (millions)")
> print(gg)
```



2.2.2 Logistic growth model

As discussed in passing from the geometric growth to logistic map models for the discrete time scenario, exponential growth can only continue for so long in a world of scarce resources. The continuous time analog of the logistic map, often referred to as the logistic equation (Verhulst (1838)), is

$$\frac{dx}{dt} = rx(t)\left(1 - \frac{x(t)}{k}\right) \tag{2.14}$$

By formulating the expression for small-time advancement of this model we can see that it is actually fairly different from the discrete-time logistic map:

$$x(t+dt) \approx x(t) + rx(t)\left(1 - \frac{x(t)}{k}\right)dt \tag{2.15}$$

$$= (1 + rdt)\, x(t) - \frac{r}{k}dt\,[x(t)]^2 \tag{2.16}$$

and we can also see from Eq (2.16) that there is a critical difference from the exponential model in that a term proportional to $x(t)^2$ has shown up, making this a *non-linear* differential equation.

Non-linear differential equations are in general very difficult to solve exactly, but in the case of this particular equation it turns out there is a relatively simple analytic solution:

$$x(t) = \frac{k}{1 + \exp\left[-r(t - t_*)\right]} \tag{2.17}$$

Fitting logistic growth models to data

Let's fit logistic growth curves to a some data taken from Gause (1932). First we need to read the data in from a tab-delimited text file (sometimes referred to as a "tab-separated values" or .tsv file):

```
> gause1932 = read.table("gause1932data.tsv", sep="\t", header=TRUE)
> gause1932[c(1:2, 8:9, 15:16, 21:22, 28:29, 35:36), ]
                     culture experiment  age volume
1   Saccharomyces cerevisiae          1  6.0   0.37
2   Saccharomyces cerevisiae          1 16.0   8.87
8   Saccharomyces cerevisiae          2  7.5   1.63
9   Saccharomyces cerevisiae          2 15.0   6.20
15          Mixed population          1  6.0   0.50
16          Mixed population          1 16.0   6.83
21          Mixed population          2  7.5   1.33
22          Mixed population          2 15.0   4.80
28 Schizosaccharomyces kefir          1 16.0   1.00
29 Schizosaccharomyces kefir          1 29.0   1.70
35 Schizosaccharomyces kefir          2 15.0   1.27
36 Schizosaccharomyces kefir          2 31.5   2.33
```

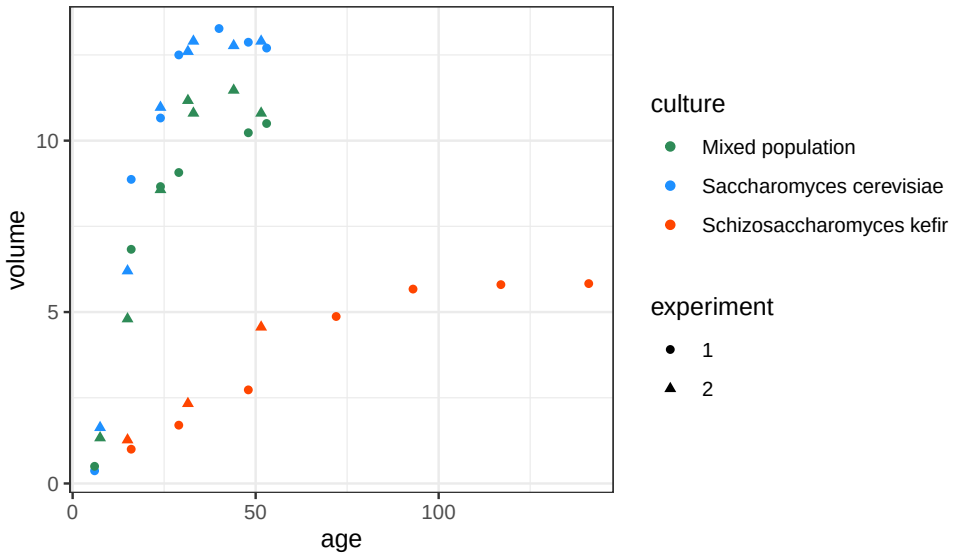
The argument `sep="\t"` tells `read.table` that the fields of the file are tab (`"\t"`) delimited, while `header=TRUE` is pretty self-explanatory: the first line of the file should be interpreted as a header containing the column names, not as data to be included within the resulting `data.frame` output.

For `ggplot` purposes, we're going to let R know that the columns `culture` and `experiment` in `gause1932` should be treated as `factor`s. `factor` is a special class in R for encoding categorical data. This is especially for the column `experiment`, since otherwise this column looks like `integer` data and would be treated as such by R.

```
> gause1932$culture = factor(gause1932$culture)
> gause1932$experiment = factor(gause1932$experiment)
```

Before doing any fitting let's examine the data visually:

```
> gg = ggplot(
      gause1932,
      aes(x=age, y=volume, shape=experiment, color=culture)
  )
> gg = gg + geom_point()
> gg = gg + scale_color_manual(
      values=c("seagreen", "dodgerblue", "orangered")
  )
> print(gg)
```



So the volume of each yeast culture does appear to grow rapidly at first before stabilizing at a higher level, just as one would expect for a population following a logistic growth curve. It does appear that the different cultures should be fit to different curves, however!

Using `nls` to fit nonlinear least squares models

First let's use the `nls` (nonlinear least squares) function to fit a logistic growth curve to the data for `Saccharomyces cerevisiae`:

```
>  ## logical indexing to select desired rows of gause1932 data.frame:
> cerevisiaeData = gause1932[
      gause1932$culture == "Saccharomyces cerevisiae",
  ]
> cerevisiaeFit = nls(
      formula = volume ~ k / (1 + exp(-r * (age-tstar))),
      data = cerevisiaeData,
      start = list(k = max(cerevisiaeData$volume),
                   r = 1,
                   tstar = median(cerevisiaeData$age))
  )
> cerevisiaeFit
Nonlinear regression model
  model: volume ~ k/(1 + exp(-r * (age - tstar)))
   data: cerevisiaeData
      k       r   tstar
12.7421  0.2586 14.6317
 residual sum-of-squares: 4.946

Number of iterations to convergence: 14
Achieved convergence tolerance: 5.471e-06
```

Here's what the arguments to `nls` mean:

1. The `formula` argument `volume ~ k / (1 + exp(-r * (age-tstar)))` tells R to fit a logistic growth curve model using the column `volume` from `cerevisiaeData` as the x variable in Eq (2.17) and the column `age` for t.
2. `data = cerevisiaeData` tells `nls` to pull data from columns in `cerevisiaeData`.
3. The `start` argument takes a named `list` which must provide starting values for all named quantities appearing in the formula provided as the first argument to `nls` which are model parameters needing to be fit (as opposed to columns to be pulled from the `data.frame` provided via the `data` argument).

The phrase "starting values" is used above because nonlinear least squares fitting is done via an iterative procedure: start with whatever parameter values are given by the `start` argument and then apply a fancy algorithm to improve the values so as to shrink the sum of squared residuals. Stop when the algorithm converges—that is, when the sum of squared residuals is no longer shrinking appreciably.

The `nls` output stored in `cerevisiaeFit` stores (among other things) the logistic growth curve model parameters r, k, and t_*, represented in R by `r`, `k`, and `tstar`. If you just want to extract these parameters, you can use the `coef` function (defined for many different types of models in R, including those produced by `nls`):

```
> coef(cerevisiaeFit)
         k          r      tstar
12.7421389  0.2586376 14.6316830
```

Before fitting logistic curves to the other two yeast cultures, let's pause to reflect on a few aspects of our usage of `nls` to fit the nonlinear logistic growth model:

- I didn't explicitly go through the assumptions underlying this model the way I did for the linear model applied to fit the exponential growth model, but they are essentially the same with the obvious exception of linearity.
  - Linearity assumption here replaced by assumption that the shape of the logistic growth curve is appropriate for the data, which we did examine graphically.
  - The independence assumption is still problematic!
- We were aided in fitting the logistic growth model by the exact solution provided by Eq (2.17) (it provided the `formula` for `nls`).
  - Exact solutions are not available for most nonlinear differential equations!
  - Fitting such models to data is considerably more difficult and computationally demanding (though certainly possible).

Using `lapply` instead of a `for` loop

Getting back to logistic growth: To fit all three cultures, I'm going to introduce a couple of useful new (to this course, anyway) R functions, `split` and `lapply`:

```
> gauseSplitByCulture = split(gause1932, gause1932$culture)
>  ## gauseSplitByCulture is now a (named) list of data.frames:
> names(gauseSplitByCulture)  ## names are the unique culture values
```

```
[1] "Mixed population"          "Saccharomyces cerevisiae"
[3] "Schizosaccharomyces kefir"
>  ## now use lapply (list apply) to apply a function to each element
>  ## of a list; in this case the function will be nls:
> logisticFits = lapply(
      X = gauseSplitByCulture,  ## arg X is list to apply function to
      FUN = nls,                ## arg FUN is function to apply to X
       ## all remaining args (must be named) fill in arguments to FUN
       ## - these arguments will be included each time FUN is called
       ##   (that is, once for each element of the list X)
      formula = volume ~ k / (1 + exp(-r * (age-tstar))),
      start = list(k = max(cerevisiaeData$volume),
                   r = 1,
                   tstar = median(cerevisiaeData$age))
  )
>  ## notice that we did NOT provide data argument needed by nls:
>  ## - since it is first "empty" argument slot, it will be filled in using
>  ##   the elements of the list supplied to lapply's first argument
>  ##   (here gauseSplitByCulture)
```

Let's take advantage of R's interactive REPL to learn a bit about what `lapply` produces:

```
>  ## what is logisticFits?
> class(logisticFits)
[1] "list"
>  ## what are the names of this list?
> names(logisticFits)
[1] "Mixed population"          "Saccharomyces cerevisiae"
[3] "Schizosaccharomyces kefir"
>  ## use lapply again (in conjunction with coef function) to extract
>  ## the parameters (coefficients) of each fit:
> lapply(logisticFits, coef)
$`Mixed population`
         k          r      tstar
10.6908350  0.2116552 15.5992366

$`Saccharomyces cerevisiae`
         k          r      tstar
12.7421389  0.2586376 14.6316830

$`Schizosaccharomyces kefir`
          k           r       tstar
 5.88023359  0.05744204 41.47229207
```

I could have used a `for` loop instead of `lapply`, but `lapply` is a bit more elegant as it avoids the need to intialize the output list or to explictly name the thing being looped over (yeast culture here) and automatically synchronizes the names of the output list to those of the input list provided to `lapply` through its first argument.

In order to fully understand what's going on with the `lapply` statement above, here is the `for` loop version of the same code:

```
> gauseSplitByCulture = split(gause1932, gause1932$culture)
>  ## initialize empty list to be populated by for loop:
> logisticFits = list()
>  ## now loop through the *names* of gauseSplitByCulture
>  ## so that we can use these as the names for the list
>  ## logisticFits as well:
> for (culture in names(gauseSplitByCulture)) {
      logisticFits[[culture]] = nls(
           ## the first argument here is the one filled in by
           ## the elements of the X argument in lapply above:
          gauseSplitByCulture[[culture]],
           ## remaining arguments were directly specified in
           ## lapply version above:
          formula = volume ~ k / (1 + exp(-r * (age-tstar))),
          start = list(k = max(cerevisiaeData$volume),
                       r = 1,
                       tstar = median(cerevisiaeData$age))
      )
  }
> class(logisticFits)
[1] "list"
> names(logisticFits)
[1] "Mixed population"          "Saccharomyces cerevisiae"
[3] "Schizosaccharomyces kefir"
> lapply(logisticFits, coef)
$`Mixed population`
         k          r      tstar
10.6908350  0.2116552 15.5992366

$`Saccharomyces cerevisiae`
         k          r      tstar
12.7421389  0.2586376 14.6316830

$`Schizosaccharomyces kefir`
          k           r       tstar
 5.88023359  0.05744204 41.47229207
```

Let me emphasize that both the `lapply` and `for` loop approaches work just fine and produce exactly the same output. The `for` loop version is a bit more explicit, while the `lapply` approach is a bit more streamlined and elegant as discussed above. I recommend becoming familiar with the `lapply` methodology—I'm going to use it here and there in these notes, for one thing—but it's fine if you prefer to `for` loops in your own code.

Using `stat_function` to plot functions with `ggplot`

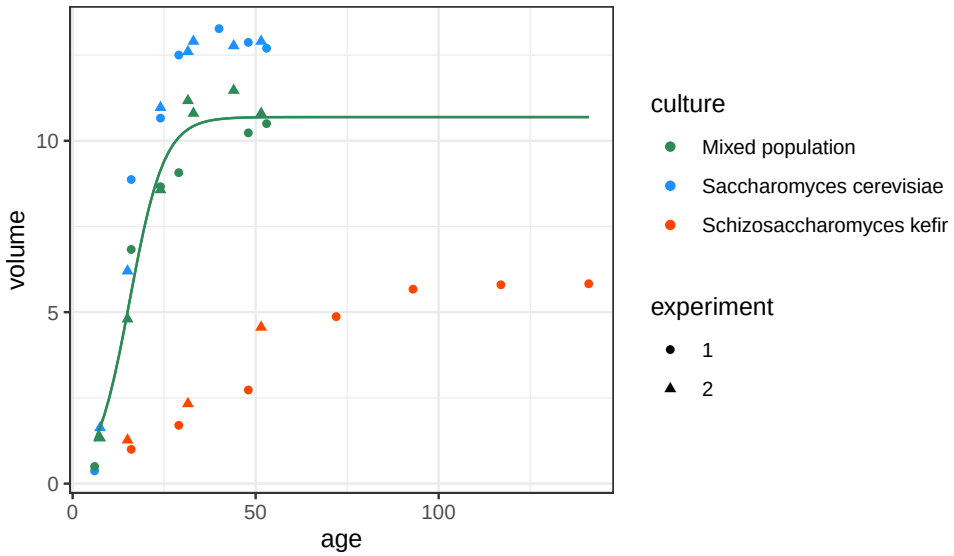
We'd like to visually assess the quality of the `logisticFits`. We can use the `predict` method in R to find the value one of these models predicts at a certain set of age values:

```
>  ## need to package the age value for which we want to get
>  ## predicted value up into data.frame with column named age
>  ## to use predict:
> predict(
       ## first argument to predict: model to use to make prediction:
      logisticFits[["Mixed population"]],
       ## second argument to predict: data.frame with columns named
       ## in same fashion as was used on RHS of model fitting formula
       ## (here: a single column named age):
      data.frame(age = c(10, 20, 30, 40, 50))
  )
[1]  2.503100  7.669265 10.206497 10.630077 10.683480
```

So we can see that, for instance, the `"Mixed population"` fit predicts a `volume` value of 10.2 for a culture `aged` 30 days (the third output value).

We can add these to the plot `gg` we had previously started using the `ggplot2` function `stat_function` method, but we'll need to define an *adaptor function* allowing us to call predict with a single argument:

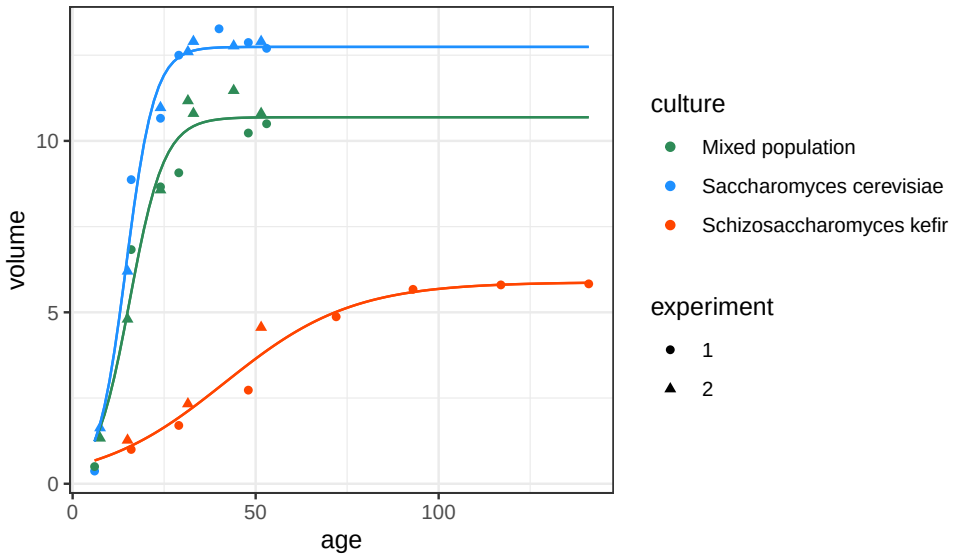
```
> predictMixed = function(age) {
      predict(logisticFits[["Mixed population"]],
              newdata = data.frame(age=age))
  }
>  ## stat_function takes fun argumet
> gg = gg + stat_function(fun=predictMixed, color="seagreen")
> print(gg)
```



We'll go ahead and add curves for the other two cultures in precisely the same manner:

```
> predictSacCer = function(age) {
```

```
      predict(logisticFits[["Saccharomyces cerevisiae"]],
              newdata = data.frame(age=age))
  }
> gg = gg + stat_function(fun=predictSacCer, color="dodgerblue")
> predictSchizKef = function(age) {
      predict(logisticFits[["Schizosaccharomyces kefir"]],
              newdata = data.frame(age=age))
  }
> gg = gg + stat_function(fun=predictSchizKef, color = "orangered")
> print(gg)
```



Looking at these plots we can see that the logistic growth model summarizes the shape of the curves represented by these data pretty well.

2.2.3 Lotka-Volterra equation for predator-prey Interaction

Unlike the discrete-time logistic map, the continuous-time logistic growth model will produce neither periodic solutions nor chaotic behavior—regardless of the parameter values r or t_*—as can be seen by considering the general solution Eq (2.17).

More complicated continuous-time differential equation systems can produce both periodic and chaotic behavior, however. As an example, we'll consider a two-species *predator-prey* model which has been used to explain the observed periodicity in the population of Canadian lynx and hares:

```
> lynxhare = read.table("lynxhare.tsv",
                        sep="\t", row.names=NULL, header=TRUE)
>  ## keep only those columns of lynxhare data.frame we plan to analyze:
> lynxhare = lynxhare[ , c("year", "hare", "lynx")]
> head(lynxhare)
  year  hare lynx
1 1845 28000   NA
```

```
2 1846 25000   NA
3 1847 25000   NA
4 1848 25000   NA
5 1849 12000   NA
6 1850 26000   NA
```

Note that many of the rows of `lynxhare` have missing values (`NA`) for one or the other of the columns `hare` or `lynx`. We'll use logical indexing, along with the function `is.na`, to filter the data set down to just those rows which have values for at least one of the two:

```
>  ## use logical indexing to keep only those rows of data.frame
>  ## which do not have NA values for both hare and lynx column
> lynxhare = lynxhare[
      !(is.na(lynxhare$hare) & is.na(lynxhare$lynx)),
  ]
> head(lynxhare)
  year  hare lynx
1 1845 28000   NA
2 1846 25000   NA
3 1847 25000   NA
4 1848 25000   NA
5 1849 12000   NA
6 1850 26000   NA
```

While `ggplot` wants every point plotted to have its own line in the `data.frame` given to it, `lynxhare` has data for both lynx and hare populations in a given year in the same row. We can reorganize the `data.frame` by *pivoting* it into longer form using the `tidyr` function `pivot_longer`:

```
> ## install.packages("tidyr")  ## uncomment and run if necessary
> library(tidyr)
> lynxhare = lynxhare %>%
      pivot_longer(-year,
                   names_to = "species",
                   values_to = "number")
>  ## arguments to pivot_longer:
>  ## - arg 1 is lynxhare data.frame, passed from %>% operator
>  ## - arg 2 is -year: this says leave the year column alone
>  ## - arg names_to is "species": create new column named species
>  ##   will take values lynx and hare, since those are columns of
>  ##   lynxhare which were not left alone by arg 2
>  ## - arg values_to is "number": create new column named number
>  ##   in which values from lynx and hare columns will go
> head(lynxhare)
# A tibble: 6 x 3
   year species number
  <int> <chr>    <int>
1  1845 hare     28000
2  1845 lynx        NA
3  1846 hare     25000
```

```
4  1846 lynx        NA
5  1847 hare     25000
6  1847 lynx        NA
```

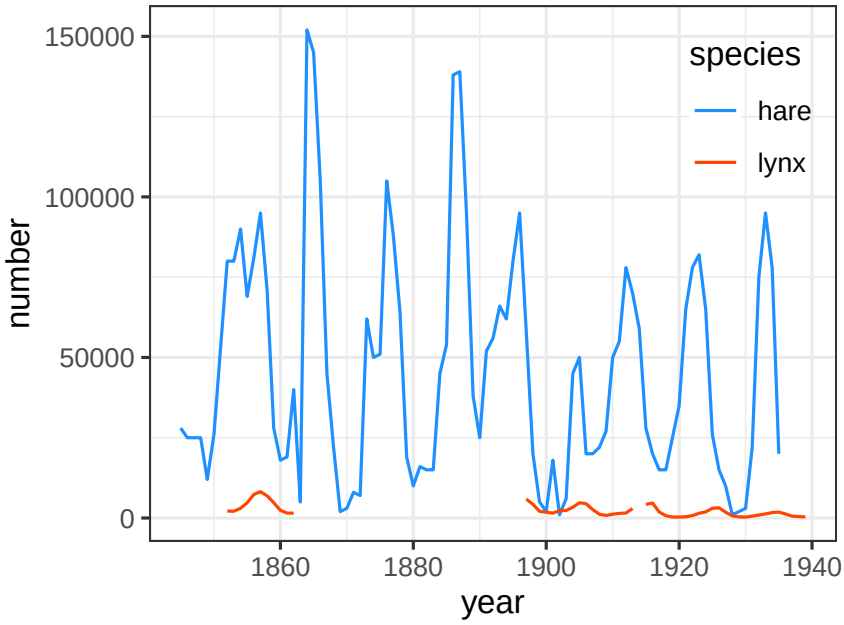
So the output `pivot_longer` here has two rows for each row in the input (hence "longer"), one for the hare column and one for the lynx column of the input.

You can also see that `pivot_longer` took in `lynxhare` as a `data.frame` but returned back out something called a `tibble`—this is an alternative version of a `data.frame` used by `tidyr` and various other packages composing the so-called "tidyverse."

For the most part you can treat `tibble`s in just the same way you do `data.frame`s; probably the most important difference from the point of view of this class is that functions that output `tibble`s tend to strip `row.names` off of any `data.frame`s that they take in.

Now we can use `ggplot` on the `pivot_longer`ed version of `lynxhare`:

```
> gg = ggplot(lynxhare,
              aes(x=year, y=number, group=species, color=species))
> gg = gg + geom_line()
> gg = gg + scale_color_manual(values=c("dodgerblue", "orangered"))
> gg = gg + theme(legend.position=c(0.875, 0.8),
                  legend.background=element_rect(fill="transparent"))
> print(gg)
```



There does appear to be a periodicity to the hare population level with peaks around the middle of each decade especially from the 1850s to the 1890s and troughs around the years ending in 0. While the lynx data is missing for large chunks of this data set—and is on a much lower overall scale—there do appear to be peak lynx populations showing up right near the end of the hare peak periods in the late 1850s, the late 1890s, and right around 1905, 1915, and (to a lesser degree) 1925 and 1935.

Many modelers have suggested that this periodicity could be explained by periodic population

explosions of hares, leading to good times for the lynx until they eventually decimate the hare population—and are in turn themselves decimated by starvation.

This makes a nice story as is, but from a modeling perspective it would make a nicer story if adorned with some equations. Here are the equations making up the Lotka-Volterra system (Lotka (1920)):

$$\begin{aligned} \frac{dx}{dt} &= \alpha x - \beta xy \\ \frac{dy}{dt} &= \delta xy - \gamma y \end{aligned} \tag{2.18}$$

Here x represents the number of prey (e.g., hares) while y represents the number of predators (e.g., lynx). The parameter

α represents the net reproduction rate of hares,

β is the rate at which hares are killed by lynx,

δ is the per-hare rate at which lynx are able to reproduce, and

γ is the death rate of lynx.

Unlike the exponential or logistic growth models, there is no simple analytic formula for the solution to the Lotka-Volterra system, so we will use a more general numeric approach to solving them with the `deSolve` package. This will require defining an R function representing the differential equation system Eq (2.18) which have a particular argument structure:

```
> lotkaVolterra = function(t, state, par) {
       ## extract x and y from the state vector:
      x=state[["x"]]; y=state[["y"]]
       ## extract a(lpha), b(eta), g(amma), and d(elta) from par vector:
      a=par[["alpha"]]; b=par[["beta"]]; g=par[["gamma"]]; d=par[["delta"]]
       ## return list with one component for each equation
       ## corresponding to rates of change in the state variables
       ## (indicated here by prefixing the variable name with "d"):
      return(list(c(
          dx = a*x - b*x*y,
          dy = d*x*y - g*y
      )))
  }
```

The arguments to `lotkaVolterra` must be

t the time point at which to evaluate the differential equations; this is not actually needed for `lotkaVolterra` but must still be included as an argument for use with `deSolve`,

state a numeric vector (may be named, as it is here) which encodes the state variables (x and y here) which are changing over time, and

par a numeric vector (may be named, as it is again here) encoding the model parameters (α, β, δ, and γ here)—these do not change over time.

Now that we have encoded Eqs (2.18) in the R function `lotkaVolterra` with the special argument structure required by `deSolve`, we compute a trajectory solving the Lotka-Volterra equations with a particular set of **parameter values**

- $\alpha = 0.9$, $\beta = 2.5$, $\gamma = 0.75$, and $\delta = 0.25$

and starting from a particular **initial condition**

- we'll take $x(0) = y(0) = 0.2$

over a particular range of **times**

- t starting at 0 and going up to 50, with values recorded every 0.1 time units

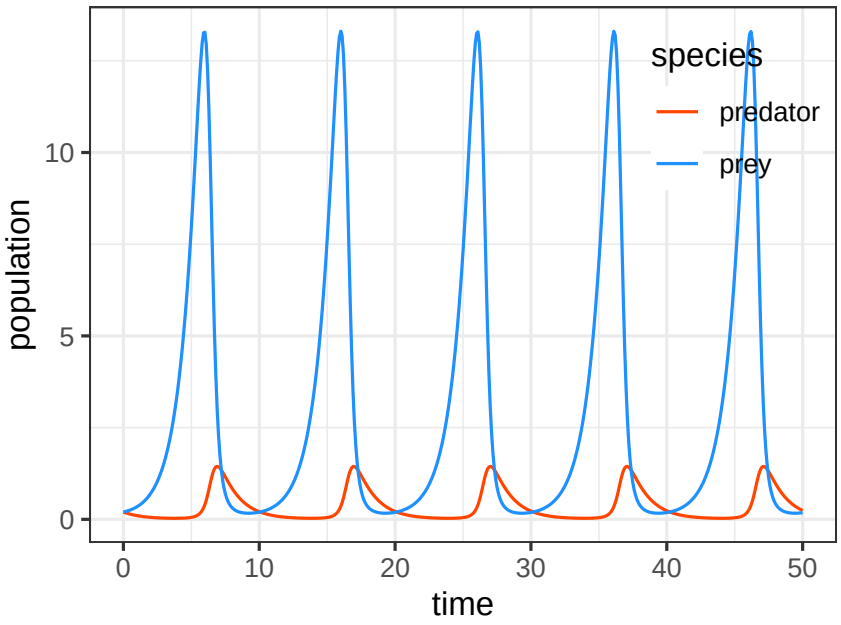
The trajectory is computed using the `ode` function from the `deSolve` package:

```
> ## install.packages("deSolve")  ## uncomment and run if necessary
> library(deSolve)
>  ## define parameters and initial condition for simulation:
> parameters = c(alpha=0.9, beta=2.5, gamma=0.75, delta=0.25)
> initialCondition = c(x=0.2, y=0.2)
>  ## as well as time points to record data for:
> times = seq(0, 50, by=0.1)
>  ## numerically solve the system of differential equations:
> trajectory = ode(initialCondition,
                   times,
                   lotkaVolterra,
                   parameters)
>  ## re-package ode solution as data.frame:
> trajectory = data.frame(trajectory)
> head(trajectory)
  time         x         y
1  0.0 0.2000000 0.2000000
2  0.1 0.2085187 0.1864975
3  0.2 0.2181083 0.1739472
4  0.3 0.2288306 0.1622823
5  0.4 0.2407561 0.1514424
6  0.5 0.2539658 0.1413710
```

The rows of `trajectory` indicate the numbers of prey x and predator y at the requested times t. It is easier to digest this information by plotting it:

```
>  ## ggplot still wants one data.frame row per plotted data point,
>  ## so use pivot_longer again to wrangle data into plottable form:
> plotData = trajectory %>%
      pivot_longer(-time, names_to="species", values_to="population")
>  ## let's rename "x" and "y" to "prey" and "predator", respectively:
> plotData$species = c(x="prey", y="predator")[plotData$species]
>  ## now set up ggplot with time on x-axis, population on y-axis:
> gg = ggplot(plotData, aes(x=time, y=population, color=species))
> gg = gg + geom_line()
> gg = gg + scale_color_manual(values=c("orangered", "dodgerblue"))
```

```
> gg = gg + theme(legend.position=c(0.865, 0.8),
                  legend.background=element_rect(fill="transparent"))
> print(gg)
```



With this particular combination of `parameters` and `initialConditions`s, we're able to qualitatively reproduce the periodic behavior observed in the actual `lynxhare` data set—though obviously the real data shows a lot more complexity than just this periodicity!

A more principled approach here would be to actually fit the parameter values to the data, as we did with the exponential and logistic growth models above. This is complicated here, both because the differential equation doesn't have as simple of a solution and because the data set is incomplete—we don't have lynx data for many years—so we aren't going to try it out in this class.

2.3 Stochastic models

Deterministic chaos, like that resulting from the logistic map with r just below 4, can generate trajectories that look random given imperfect knowledge of initial conditions in the ways we discussed earlier. Beyond that, our best theories of how the physical world works on the smallest scales currently suggest that the laws of nature are likely not fully deterministic anyway!

Stochastic models build randomness in explicitly. Random components can be added to all of the deterministic models we have considered so far, resulting in various discrete time or continuous time stochastic models. Such models can become mathematically very complicated very quickly, however, so will restrict ourselves to a couple of very simple stochastic models: a simple form of *random walk* and a stochastic generalization of the exponential growth model.

2.3.1 Random walk

One of the simplest possible stochastic models consists of random variables X_t for each discrete time point $t \in \{0, 1, 2, \ldots\}$ with conditional probability distributions:

$$\mathbb{P}\left(X_t = x_t \mid X_{t-1} = x_{t-1}\right) = \begin{cases} \frac{1}{2} & \text{if } x_t = x_{t-1} - 1 \text{ (“step to left”),} \\ \frac{1}{2} & \text{if } x_t = x_{t-1} + 1 \text{ (“step to right”),} \\ 0 & \text{for any other value of } x_t. \end{cases} \tag{2.19}$$

Eq (2.19) just says that at each time point t, x_t must be either $x_{t-1} - 1$ or $x_{t+1} - 1$, each with probability 50%. If you visualize x_t as the position of a walker on a number line, this can be thought of as the walker randomly choosing to step either to the left or to the right by one unit at each time point t.

A word regarding notation here: I'm using upper-case X_t for random variables—which are not themselves specific numbers—and lower-case x_t for specific numeric values which could be taken in a simulation generated from the model. The equal sign in an expression like $\mathbb{P}(X = x) = p$—usually read as “X takes value x with probability p”—does *not* mean that X is the same as x: They can't be because X is a special thing called a random variable while x is just a number.

This (upper- vs. lower-) case difference is a common notational scheme used in probability and statistics literature, but if you're fuzzy on the exact meaning of a “random variable,” don't worry! For this class, you can think of the random variables X_t as just a notational device used in writing down probability formulae like Eq (2.19).

One final point worth noting from the the theory of stochastic processes: Eq (2.19) is a conditional probability distribution for the distribution of the random variable X_t given that we know the specific value of the position x_{t-1} immediately preceding time t. The standard random walk model makes the additional assumption that the distribution of X_t given x_{t-1} is the same regardless of the values x_{t-k} for $k > 1$—that is, that no matter how the random walk ended up at x_{t-1} at time $t - 1$ it has the same chance of moving to any particular position x_t at time t. Models satisfying this assumption that history doesn't matter if you have a *full* knowledge of the state of things right now are known as **Markov** models. The Markov property may be precisely formulated in probabilistic terms:

$$\mathbb{P}(X_t = x_t \mid X_1 = x_1, \ldots, X_{t-1} = x_{t-1}) \;=\; \mathbb{P}(X_t = x_t \mid X_{t-1} = x_{t-1}) \tag{2.20}$$

We will consider simulating a simple Markovian random walk described by Eq (2.19) together with Eq (2.20). Simulation of such a process is straightforward since, by virtue of Eq (2.20), we need only know to draw random numbers distributed according to to the distribution indicated in (2.19) in order to simulate step t once we've done steps 1 through $t - 1$.

Before we get to the simulation itself, we'll use the R function `set.seed` to specify the (pseudo-)random number generator *seed*. This enables us to get the same sequence of “random” numbers—and thus the same results for any analysis applied to the simulated data—if we want to run the code again later.

```
>  ## set.seed before generating random walk below (for reproducibility):
> set.seed(123)
```

We can simulate a random walk very easily in R using the `sample` function, which allows us to randomly pick numbers with equal probability from a supplied vector—which we will set here to `c(-1, +1)`. If we set `replace=TRUE` in the `sample` call, we can repeat this random sampling however many times we want; the second argument to `sample` controls this number:

```
>  ## randomly pick one million (1e6) steps,
>  ## each either -1 or +1 with equal chance,
>  ## using R's built-in sample function:
> steps = sample(c(-1, +1), 1e6, replace=TRUE)
>  ## use cumsum (cumulative sum) function to calculate location
>  ## x_t (represented by R vector location) at each time t:
> location = c(0, cumsum(steps))
```

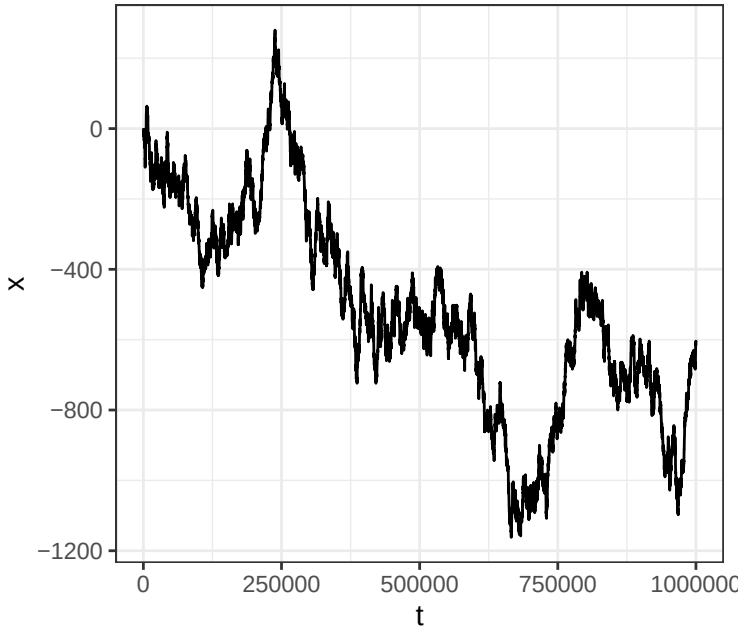
The function `cumsum` is very useful in R (and if you use Python at all, you can find a similar function in `numpy`); if you were to write the same code using a `for` loop in R (or Python) it would run much, much slower. For clarity as to exactly what `cumsum` does, consider the following comparison (and note that you could replace `3737` with any number up to `length(steps)`):

```
>  ## compare:
> sum(steps[1:3737])
[1] -49
>  ## with:
> cumsum(steps)[3737]
[1] -49
```

So now `location[i]` corresponds to x_{i-1}—the offset -1 required because we define the positions x_t to start at $t = 0$ while R indexes vectors starting from `i=1`. In order to get a sense of what a random walk looks like, let's use `ggplot` to plot the `location` values (or x) against the step index (corresponding to t):

```
> library(ggplot2)
> theme_set(theme_bw())
>  ## use seq function to select only every 100th point from
>  ## location so as to keep plot from getting overly large:
> plotData = data.frame(t = seq(0, 1e6, by=100),
                        x = location[1+seq(0, 1e6, by=100)])
>  ## let's take a quick look at plotData to make sure it looks
>  ## like we want it to:
> head(plotData)
    t   x
1   0   0
2 100 -14
3 200  -6
4 300  -8
5 400 -22
6 500 -22
>  ## set up ggplot object: note that here we are setting the
>  ## y-axis of the plot to use the column named "x" in plotData:
```

```
> gg = ggplot(plotData, aes(x=t, y=x))
>  ## connect points by lines in plot using geom_line:
> gg = gg + geom_line()
> print(gg)
```



Note that while the t scale goes from 0 to one million, the x scale is measured in hundreds to a few thousand. This is one of the most characteristic features of random walks: the distance of a random walker from its starting point after t steps is generally around $\sqrt{t}$ step-lengths! (While we won't stop to prove it here, this result holds for random walks in more than one spatial dimension as well.)

We can make this observation much more precise by considering many independent random walks: Let's start by encoding the procedure for generating a random walk in a function:

```
> randomWalk = function(nSteps) {
      steps = sample(c(-1, +1), nSteps, replace=TRUE)
      return(c(0, cumsum(steps)))
  }
```

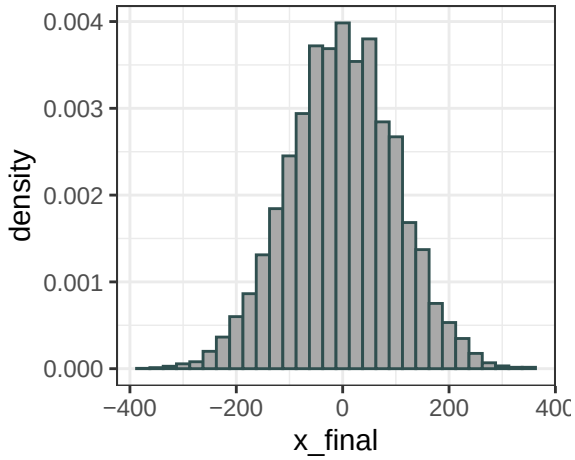
Now we'll use `lapply` to generate lots of random walks:

```
> nSteps = 1e4
> nRandomWalks = 1e4
> randomWalks = lapply(1:nRandomWalks, function(i) {randomWalk(nSteps)})
>  ## note that argument to randomWalk is always nSteps,
>  ## independent of the value i indexing the nRandomWalks random walks!
>  ## ---
>  ## Now use lapply again to extract locations where each walk ended;
>  ## in this case the function applied *does* depend on the argument rw,
>  ## which will be one full vector of random walk positions:
> finalPositions = lapply(randomWalks, function(rw) {rw[length(rw)]})
>  ## so the body of the lapplied function rw[length(rw)] just says
```

```
>  ## to pull out the final value of the vector rw, i.e.,
>  ## the position at which the random walk rw ends.
```

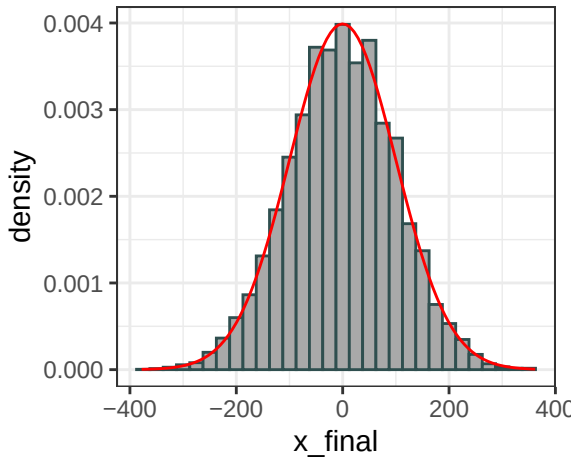
Once we've generated `nRandomWalks` independent random walks and extracted the `finalPositions` of all of them, we can examine the distribution of where they all ended up with a histogram using `ggplot`:

```
> gg = ggplot(data.frame(x_final=unlist(finalPositions)),
              aes(x=x_final))
> gg = gg + geom_histogram(aes(y=..density..),
                           binwidth = 25,
                           fill = "darkgray",
                           color = "darkslategray")
> print(gg)
```



If you think that looks a lot like a Gaussian bell-curve (a.k.a., the "normal" distribution), you're right! Let's use `stat_function` to plot a Gaussian/normal curve with standard deviation `sqrt(nSteps)` on top of the histogram (incidentally, the weird-looking `aes(y=..density..)` provided as the first argument to `geom_histogram` above was necessary to set the y-axis scale to be consistent with the output of the **d**ensity-of-**norm**al-distribution function `dnorm` used here):

```
> gg = gg + stat_function(
      fun = function(x) {dnorm(x, sd=sqrt(nSteps))},
      color = "red"
  )
> print(gg)
```



That's about as good of agreement between theory and observation as you'll ever see! In fact, if you recall learning about the *central limit theorem* in a probability or statistics class, it's worth noting that since the position of a random walker is just the sum of many independent, identically distributed (**iid**) individual steps, it should not surprise you at all to see a normal distribution emerge for a set of walkers who've spent a long time stepping around.

After all this, you may be wondering why anyone cares about random walks. In fact many real-world processes are modeled using some form of random walk, including famously stock prices, but in biology specifically both the processes of *genetic drift* and of *molecular diffusion* are frequently modeled as forms of random walks.

Molecular diffusion, which refers to the thermal motion of particles, is one of the primary modes of transport for biomolecules within and between cells. The fact that diffusion is efficient for transporting molecules over relatively short distances but inefficient for longer distances—since the distance traversed scales with $\sqrt{t}$—has important ramifications for cell size, organization, and physiology (Schavemaker *et al.* (2018) discusses many of these issues): for instance limiting the distances between various cellular components such as chromosomes and membranes.

2.3.2 Stochastic population growth

Most stochastic models are trickier to work with than random walks. But the general strategy of studying the properties of stochastic models by considering many independent simulations, just as we did with the random walk model above, works well in many other situations as well.

We'll consider just one example: Returning to the phenomenon of exponential population growth, recall that when I introduced the differential equation model $\frac{dx}{dt} = rx$ I mentioned that such models necessarily assume that the population x can be treated as a continuous number.

A more realistic model of population growth—though one which still excludes many, many important details—returns to modeling the (integer) number of individuals $n(t)$ in continuous time t subject to two distinct types of "reaction" (the language used to describe these models tends to reflect their origin in chemistry):

birth in a short time interval dt, there is a probability $bn(t)\,dt$ that the population grows by 1, going from n to $n+1$; and

death in the same interval dt, there is a simultaneous probability $dn(t)\,dt$ that the population shrinks by 1, declining from n to $n-1$.

While such models can be simulated "directly" by looping over many small time steps dt and checking whether either a birth or death happens in each, there is necessarily an approximation made with any value $dt > 0$ because we have to neglect the possibility that there might be multiple births and/or deaths in the same small time interval. Moreover, in trying to minimize the inevitable errors caused by this approximation by making the time step size dt very small, we end up with a very computationally inefficient algorithm, since most time steps won't change $n(t)$ at all if both probabilities $bn\,dt$ and $dn\,dt$ are very small.

The *Gillespie algorithm* (Doob (1945); Gillespie (1977)) is a clever way to simulate this system—and many other similar systems—**exactly**. It starts by observing (as can be proved mathematically if you're so inclined) that the definition in terms of birth and death reactions above is equivalent to the following reformulation:

When is next reaction? If we know population is $n(t)$ at time t, it turns out that the time $t + u > t$ at which the next change in population (either birth or death) will be exponentially distributed with probability density function:

$$\frac{1}{du}\,\mathbb{P}(u \le U < u + du) \;=\; \frac{1}{b+d}\exp[-(b+d)u] \tag{2.21}$$

Knowing this distribution for the time elapsed u before the next birth or death, we can simulate u and jump all the forward from t to $t + u$, at which point we must decide:

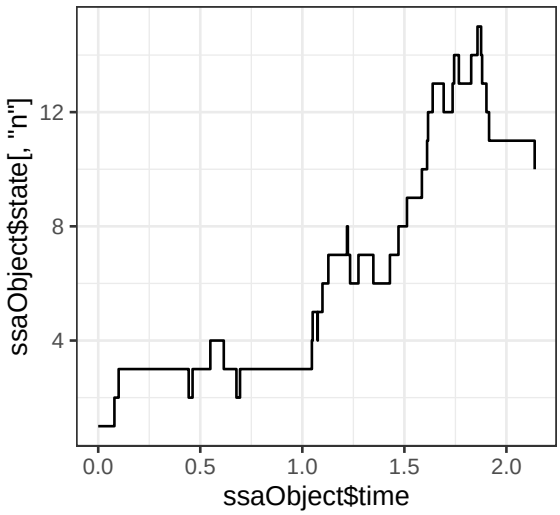
What is next reaction? This is even easier: since births happen at rate bn and deaths at rate dn, the chance that the next birth happens before the next death is given by

$$\frac{bn}{bn+dn} = \frac{b}{b+d} \tag{2.22}$$

So we just draw a random number between 0 and 1; the next reaction is a birth—so that n goes up to $n+1$—if the random number is less than $\frac{b}{b+d}$ and a death otherwise (in which case we decrement n to $n-1$.

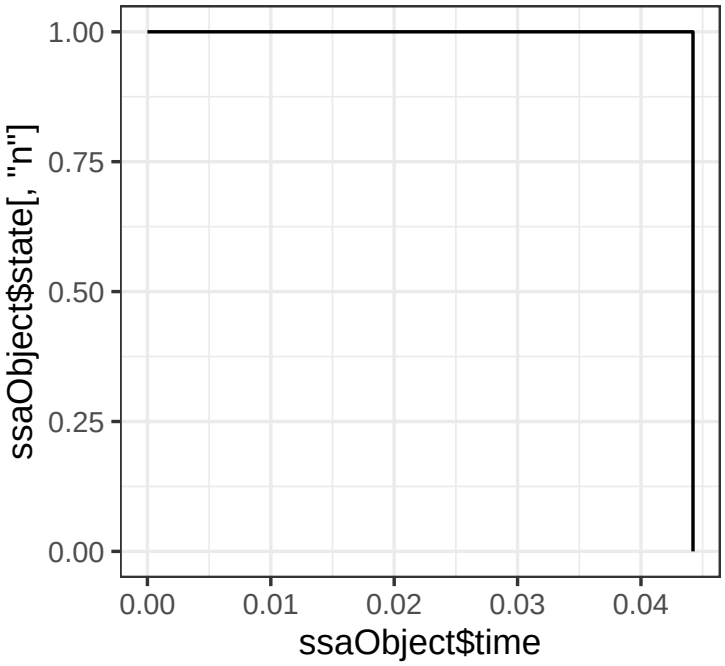
While it is not particularly difficult to implement this algorithm directly in R, it is pretty inefficient, since R `for` loops are very slow relative to those in lower-level languages like C or Java. Thus we will turn to the R package `GillespieSSA2`, which offers a function `ssa` that—after a bit of simulation system setup invoking another function, `reaction`, from the package—can generate such lower-level code for us, run it, and return the results to us in R:

```
> ## install.packages("GillespieSSA2")  ## uncomment and run if necessary
> library(GillespieSSA2)
>  ## need to assemble model parameters into *named* vector:
> params = c(b=2, d=1)
>  ## how long (in simulation time, not real time) will simulations run?
> finalTime = 2
>  ## what should initial value of state variables (here just n) be?
>  ## use *named* vector:
> state0 = c(n=1)
>  ## for Gillespie simulations: time is continuous, but events are
>  ## discrete occurrence of one of a finite list of possible reactions:
> reactions = list(
       ## reaction constructor function from GillespieSSA2 needs 2 args:
       ## - propensity: a character string representation of the
       ##   function of params (b or d) and state variables (n)
       ##   which multiplies dt to yield probability reaction
       ##   happens in infinitesimal time interval of length dt, and
       ## - effect: a *named* vector indicating how an occurrence
       ##   of the reaction changes the values of the state variable(s)
      reaction("b * n", c(n = +1)),
      reaction("d * n", c(n = -1))
  )
>  ## set random number generator seed to obtain reproducible trajectory:
> set.seed(123)
>  ## use ssa function from GillespieSSA2 to simulate trajectory:
> ssaObject = ssa(initial_state = state0,
                  reactions = reactions,
                  params = params,
                  final_time = finalTime)
>  ## let's plot the trajectory with qplot using geom_step:
> qplot(ssaObject$time, ssaObject$state[ , "n"], geom="step")
```



As you can see, stochastic trajectories of population growth models look quite different from deterministic ones! Beyond that, the stochastic population growth model we're considering introduces the possibility of population *extinction*, even when the birth rate parameter b exceeds the death rate parameter d:

```
>  ## try a different random number generator seed to get a different
>  ## trajectory from the same system:
> set.seed(12345)
> ssaObject = ssa(initial_state = state0,
                  reactions = reactions,
                  params = params,
                  final_time = finalTime)
> qplot(ssaObject$time, ssaObject$state[ , "n"], geom="step")
```



As we can see from the code above, `ssaObject$t` tracks the times at which events occur, while `ssaObject$state` tracks the system state immediately after the occurrence of each of these events.

It would be useful to be able to extract from `ssaObject` what the value of $n(t)$—the number n of individuals in the simulated trajectory at time t—for any particular time t we might be interested in. Here is a function for doing this:

```
> stateAtTimeT = function(ssaObject, t) {
       ## find the time of the last event to occur before time t:
      lastEventTimeBeforeT = max(which(ssaObject$time - t <= 0))
       ## report the state at that time
       ## (which will also be the state at t, since nothing happens
       ##  in the intervening time):
      as.numeric(ssaObject$state[lastEventTimeBeforeT, ])
  }
```

Let's check it out:

```
> stateAtTimeT(ssaObject, 0.04)
[1] 1
> stateAtTimeT(ssaObject, 0.05)
[1] 0
```

In this trajectory, there is only one event: the death of the single initial individual at time $t_{\text{extinction}}$ a bit above 0.04. After that, n remains at 0 forever and ever, as the probability of birth $bn\,dt = 0$ in any time interval dt—and similarly the probability of death $dn\,dt = 0$—when $n = 0$. Deterministic differential equation-based population models can produce trajectories which asymptotically shrink to 0 as $t \to \infty$ when death rates exceed birth rates, but they generally cannot account for the possibility of extinction of small populations by random chance ("bad luck") in the way that stochastic models can.

This brings up an important point about modeling using stochastic models: Unlike the case with deterministic models, where once we have settled on a set of model parameters we just need to solve the differential equations to find the trajectory produced by a given initial condition, we must generally simulate many trajectories from the same initial state using the same set of parameters to get an idea as to what a stochastic model indicates is likely to happen.

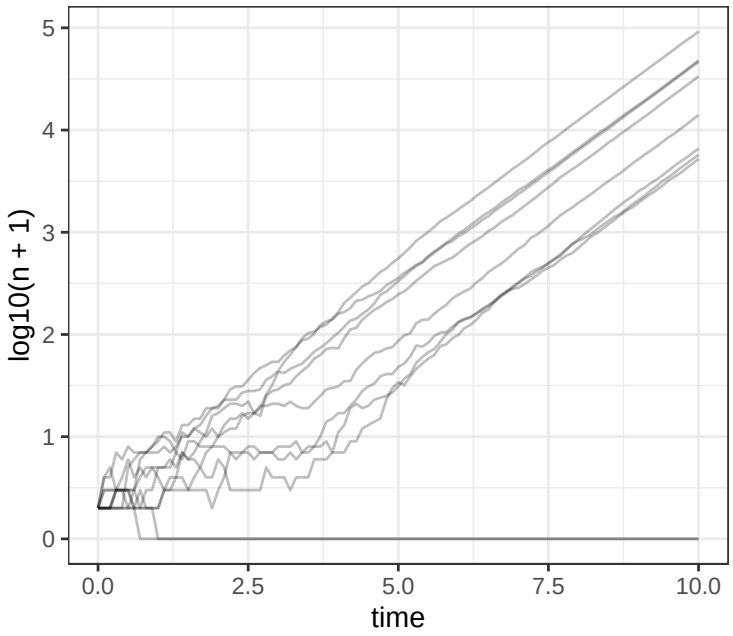
```
> set.seed(123)
>  ## generate a vector of distinct random number generator seeds
>  ## for all of the simulations we plan to do
> seeds = sample(1e6, 10)  ## will generate 10 seeds, each selected
>                          ## from between 1 and 1e6=1000000
>  ## let's let the simulations run for a bit longer (simulated time)
>  ## this time:
> finalTime = 10
>  ## use lapply to iterate through vector random number generator seeds,
>  ## generating one trajectory for each such seed:
> simulations = lapply(seeds, function(seed) {
      set.seed(seed)
      ssa(initial_state = state0,
          reactions = reactions,
          params = params,
          final_time = finalTime)
  })
```

Let's plot all of the simulated trajectories embodied in the R object `simulations`—which is a `list` of `ssa` objects—using `ggplot`. We'll use our `stateAtTimeT` function to extract the states of the systems at a set of common times into a `data.frame`, and then use `pivot_longer` from `tidyr` just as we did previously in section 2.2.3 for plotting the `lynxhare` data, to put the `data.frame` into longer form compatible with `ggplot`:

```
>  ## set up vector of common time points to collect system states at:
> ts = seq(0, finalTime, 0.1)
>  ## use lapply to iterate through time points in ts,
>  ## converting output list into data.frame:
> plotData = data.frame(lapply(ts, function(t) {
       ## use nested inner lapply to iterate through simulations;
       ## convert output list into vector using unlist function:
      unlist(lapply(simulations, stateAtTimeT, t=t))
  }))
>  ## set columns of plotData to be string representations of
>  ## time points at which simulation states were extracted:
> colnames(plotData) = as.character(ts)
>  ## add "rep" column to plotData tracking which replicate
>  ## (simulation index) each row represents:
> plotData$rep = 1:nrow(plotData)
>  ## now convert to longer form using pivot_longer from tidyr:
> library(tidyr)
> plotData = plotData %>%
      pivot_longer(-rep, names_to="time", values_to="n")
>  ## arguments to pivot_longer similar to lynxhare example above:
>  ## - arg 1 is plotData data.frame, passed from %>% operator
>  ## - arg 2 is -rep: this says leave the rep column alone
>  ## - arg names_to is "time": create new column named time
>  ##   will take values from string representation of ts,
>  ##   since these are the columns of plotData (excluding rep)
>  ## - arg values_to is "n": create new column named n
>  ##   in which values from various time point columns will go
> plotData$time = as.numeric(plotData$time)  ## want numeric time points
```

Now that we've got `plotData` set up, we're ready for `ggplot`. However, here I'm going to plot `log10(n+1)` on y-axis instead of just plain `n`; `aes` supports this like so:

```
>  ## aes function allow binding of visual aesthetics
>  ## (x position, y position, color or shape of points, etc.)
>  ## to functions of columns in plotData---like log10(n+1):
> gg = ggplot(plotData, aes(x=time, y=log10(n+1), group=rep))
>  ## we'll use geom_line instead of geom_step since we're not
>  ## plotting every event but instead only the states of the
>  ## systems at the specific time points in ts:
> gg = gg + geom_line(alpha=0.25)
> print(gg)
```



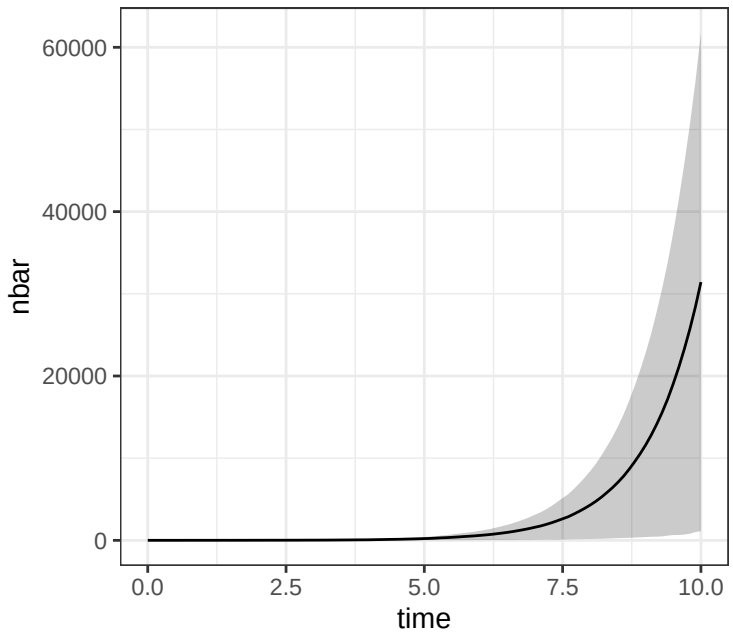
As you can see from the plot above, the trajectories resulting from this stochastic population growth model tend to either go extinct or get to a level at which they start to grow in a way that looks very much like deteministic exponential growth (note the log-scaled y-axis!). However, even those trajectories which do ultimately experience runaway exponetial growth show a lot of variability in how long it takes them to get there, with the result that at any given time t, some of these trajectories have much larger $n(t)$ values than others!

Let's try to characterize how this variability in $n(t)$ for surviving trajectories is related to the overall average values of $n(t)$ for the same set of trajectories. We'll use another R package, `dplyr`, to efficiently compute means and standard deviations of the $n(t)$ values for different trajectories at the same time t from the data as formatted in `plotData` with the aid of the functions `group_by` and `summarize`:

```
> ## install.packages("dplyr")  ## uncomment and run if necessary
> library(dplyr)
>  ## determine which trajectories end up extinct by checking to see
>  ## which ones have n value of 0 somewhere
>  ## (if they ever get to 0, they will stay there!):
> extinct = unique(plotData[plotData$n == 0, ]$rep)
>  ## here only compute means and sds for surviving trajectories:
> stats = plotData[!(plotData$rep %in% extinct), ] %>%
       ## use group_by to indicate that mean and sd should be applied
       ## to data with the same values of the column time in plotData:
      group_by(time) %>%
       ## now use summarize to calculate mean and sd values for
       ## either column n or function log(n+1) of column n like so:
      summarize(
           ## compute statistics on untransformed n values:
          nbar=mean(n), sigma=sd(n),
           ## and also on log(n+1) values:
          lognbar=mean(log(n+1)), sigmalogn=sd(log(n+1))
      )
`summarise()` ungrouping output (override with `.groups` argument)
```

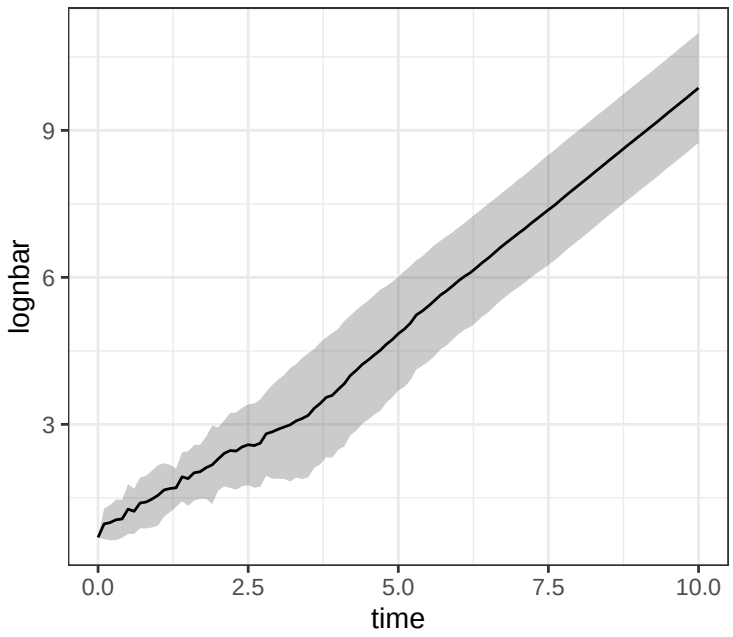
First let's try plotting the means and sds of the untransformed variables (i.e., `stats$nbar` and `stats$sigma`):

```
>  ## first plot sd-vs-mean for untransformed n:
> gg = ggplot(stats, aes(x=time, y=nbar, ymin=nbar-sigma, ymax=nbar+sigma))
> gg = gg + geom_ribbon(alpha=0.25)
> gg = gg + geom_line()
> print(gg)
```



You can see that as the mean population at time t grows, so does the variance. This sort of behavior is an example of what statisticians call *heteroskedasticity*, and it means that untransformed population levels resulting from such stochastic population growth processes violate the equal variance, or *homoskedasticity*, assumption underlying many classical statistical procedures. As you may recall from biostats class, one potential remedy for such heteroskedasticity is to try applying a transformation, such as a log transformation, to the offending variable(s); here we'll use $\log(n+1)$ instead of just $\log(n)$, so as to avoid any problems with resulting from the fact that $\log(0) = -\infty$:

```
> gg = ggplot(stats, aes(x=time, y=lognbar,
                         ymin=lognbar-sigmalogn, ymax=lognbar+sigmalogn))
> gg = gg + geom_ribbon(alpha=0.25)
> gg = gg + geom_line()
> print(gg)
```



Once the initial transient noise resulting from stochasticity in the small starting population sizes settles down, you can see that the variance in log-transformed population sizes across the different trajectories does appear to be pretty constant regardless of the mean log-transformed population size. This means that the log transformation functions as a *variance-stabilizing* transformation for data distributed as in this stochastic population growth model.

2.3.3 Modeling biochemical reaction networks

While I've presented applications of differential equation and stochastic process modeling techniques predominantly in the context of modeling populations of organisms, all of these methods are at least as applicable to modeling the dynamics of biochemical reaction networks. These sorts of models typically have time-dependent variables each representing how many molecules of a given biochemical species—or type of molecule—are present at time t together with a set of reactions whereby the numbers of molecules of some species are reduced and those of others are increased. The phrase "reaction network" indicates that, as both the propensities of reactions to occur and the changes in system state that reactions produce generally involve multiple species simultaneously, the reactions in such systems serve to connect the various different biochemical species together—or, if you prefer, you might regard the biochemical species as connections between the different types of reactions!

Both deterministic and stochastic models are employed to study such systems, and the characteristics of such models that we have touched on in these lectures—such as pronounced heteroskedasticity—are exactly the same regardless of the nature of the system being modeled. There is, however, often a difference in the *complexity* of the models used to model reaction networks, which by definition involve many distinct species, and the 1- or 2-variable models we've studied here!

Chapter 3

Biological Sequence Analysis

3.1 Strings in R

The data type known as "strings" in most programming languages are generally dealt with as character vectors in R, as we have seen so far. Strings are fundamental data structures in pretty much any context in computer science, but they are of special importance in computational biology since they provide a natural representation for biological sequences, be they DNA, RNA, or protein.

A few of the basic operations you can perform with character vectors in R include:

3.1.1 Querying string length

Unlike most other languages, the standard R function for finding out how long a string—or, more generally, vector of strings, since R sees a single string as just a character vector of length 1—is not named length or any variant thereof, but instead `nchar`:

```
> length("This is a string of 58 characters but a vector of length 1")
[1] 1
> nchar("This is a string of 58 characters but a vector of length 1")
[1] 58
> stringVec = c("string 1", "string 2 is longer")
> nchar(stringVec)  ## returns integer vector
[1]  8 18
```

3.1.2 Concatenating strings

Using `paste` or `paste0`:

```
> s = paste("long", "horn")
> s
[1] "long horn"
```

```
> s = paste0("long", "horn")
> s
[1] "longhorn"
```

paste0 is really just paste with the separator set to the empty string instead of a single space:

```
> paste("long", "horn", sep="")   ## same as paste0
[1] "longhorn"
> paste("long", "horn", sep="_SEPARATOR_")
[1] "long_SEPARATOR_horn"
```

You can paste together as many strings as you want:

```
> paste("abc", "def", "gh", "iklm", sep="_:_")
[1] "abc_:_def_:_gh_:_iklm"
```

paste and paste0 can both be used with character vectors of length > 1 as well; the best way to understand what they do in such cases is to study some examples:

```
>  ## can use the collapse argument to paste a single vector together:
> s = paste(c("long", "horn"), collapse="")
> s
[1] "longhorn"
>  ## compare to default (no collapse argument provided) behavior:
> paste(c("long", "horn"))
[1] "long" "horn"
>  ## or
> paste(c("long", "horn"), "suffix")
[1] "long suffix" "horn suffix"
>  ## or even:
> paste(c("long", "horn"),
        c("suffix1", "suffix2"))
[1] "long suffix1" "horn suffix2"
```

3.1.3 Taking substrings

Using substr:

```
> s = "longhorn"
> substr(s, 1, 4)         ## returns first 4 characters of s
[1] "long"
> substr(s, 2, 3)         ## returns substring with only 2 characters
[1] "on"
> substr(s, 5, nchar(s))  ## returns suffix of s starting at 5th character
[1] "horn"
> substr(stringVec, 3, nchar(stringVec))
[1] "ring 1"           "ring 2 is longer"
```

3.1.4 String splitting

String splitting in R can be done using `strsplit`. Because strings are intrinsically embedded in character vectors in R, the output of `strsplit` is provided as a nested `list` of `vectors`, with one element of the outer `list` for each element of the input character `vector`:

```
> strsplit(stringVec, split="ng")
[[1]]
[1] "stri" " 1"

[[2]]
[1] "stri"     " 2 is lo" "er"
> strsplit(stringVec, split="ng")[[2]]
[1] "stri"     " 2 is lo" "er"
> strsplit(s, split="ng") ## still a list of vectors!
[[1]]
[1] "lo"   "horn"
> strsplit(s, split="ng")[[1]]
[1] "lo"   "horn"
> strsplit(s, split="ng")[[1]][[2]]
[1] "horn"
```

3.1.5 Pattern matching

Using `grepl`:

```
> grepl("string", stringVec)
[1] TRUE TRUE
> grepl("longer", stringVec)
[1] FALSE  TRUE
> grepl(s, stringVec)     ## "longhorn" not in either element of stringVec
[1] FALSE FALSE
```

3.1.6 Pattern replacement

Using `gsub` (for **g**lobal **sub**stitute):

```
> gsub("string", "longhorn", stringVec)
[1] "longhorn 1"           "longhorn 2 is longer"
> gsub("long", "stringi", stringVec)
[1] "string 1"              "string 2 is stringier"
```

3.2 Modeling biological sequences

We've been modeling systems which are described by one or more numbers, often packed together into a vector, which evolve in time. But many biological questions, especially at the molecular level, concern sequences more naturally represented as character strings than as numeric vectors changing in time.

What does it mean to "model" biological sequences? It is difficult to cover all of the different usages of this term with a single definition, but two main goals stand out:

1. identify patterns and structure, and
2. predict functions of sequence, via answers to questions such as:
    - where might it interact with other biomolecules?
    - does it contain subsequences which might be targeted for biochemical modification?

The models we posed for modeling the trajectories of numeric systems were all generative models which offered the ability to simulate artificial trajectories which could be compared with the real thing. This is still true for many—though not all—of the methods applied to biological sequence data, but such simulation plays a much smaller role in sequence analysis.

Still, the idea of generating a simulated sequence to compare with real biological sequences gives us a starting point for thinking about how to model biological sequences. How might one generate "simulated DNA", for instance? Likely the first approach that came to your mind was to start at the first base and randomly pick an A, C, G, or T, then move on to the second and do it again, and so forth. If we take the probability of picking each base to the same every time we do this and pick independently at each position, we will generate a sequence of **i**ndependent, **i**dentically **d**istributed (iid) characters. Here's some R code taking this approach:

```
> nBases = 25
>  ## set random number generator seed for reproducibility:
> set.seed(123)
>  ## sample from A, C, G, T *with replacement* nBases times:
> seqAsVector = sample(c("A", "C", "G", "T"), 25, replace=TRUE)
> seqAsVector
 [1] "G" "G" "G" "C" "G" "C" "C" "C" "G" "A" "T" "C" "C" "A" "C" "G" "T" "A" "G"
[20] "G" "A" "T" "A" "A" "A"
> seqAsString = paste(seqAsVector, collapse="")
> seqAsString
[1] "GGGCGCCCGATCCACGTAGGATAAA"
```

The first refinement we might make, even while keeping the iid assumption, is to consider the possibility that the different bases do not occur with the same frequency in a stretch of real DNA. Let's read in the sequences of some real human DNA and check whether the observed proportions are consistent with the 25%-for-each base hypothesis (we'll use the Chi-Square Goodness of Fit test you might recall from biostats):

```
> library(seqinr)
> humanDna = read.fasta("two_human_dna_sequences.fa")
> class(humanDna)
[1] "list"
> names(humanDna)
[1] "A" "B"
> head(humanDna$A)
[1] "c" "a" "g" "c" "t" "g"
> table(humanDna$A)
  a   c   g   t
104 162  91 143
>  ## use chi-square goodness of fit test to check proportions in
>  ## each sequence:
> chisq.test(table(humanDna$A), p=rep(0.25, 4))
        Chi-squared test for given probabilities

data:  table(humanDna$A)
X-squared = 26.32, df = 3, p-value = 8.174e-06
>  ## what about sequence B?
> table(humanDna$B)
  a   c   g   t
757 797 717 729
> chisq.test(table(humanDna$B), p=rep(0.25, 4))
        Chi-squared test for given probabilities

data:  table(humanDna$B)
X-squared = 5.0507, df = 3, p-value = 0.1681
```

So for sequence A we can resoundingly reject the hypothesis of equal base probabilities, but for sequence B the null hypothesis seems plausible. This points out an important feature of biological sequences (DNA or otherwise): They tend to have heterogeneous composition, with different regions from the same genome having different statistical properties!

3.3 Markov models

When we studied random walks, we learned about the Markov property, which describes any system for which the state at time t is a random variable whose probability distribution depends only on the state at time $t-1$.

This idea can be applied to modeling biological sequences as well if we replace the word "time" with the word "position". The simplest version of this idea replaces the iid assumption by allowing for the probabilities of each base to occur at position i to depend on what the base observed at position $i-1$ is (I will generally use i in place of t when indexing position of character within sequence instead of indexing discrete time points). That means that instead of specifying just 4 probability values (1 for each possible base), we now have to specify 4*4=16. These will usually be arranged in a 4x4 matrix often called a *transition matrix*, though different authors disagree on whether the rows should represent the observed base at position $i-1$ and the columns the potential base at position i or vice versa. Here we'll take the rows-represent-last-observed-base and columns-represent-potential-next-base convention to be consistent with the R package `HMM` (which we'll get to a little later on).

Just as we used the probabilistic random variable notation to simplify our discussion of stochastic models for numeric variables, we'll apply it again here in the context of stochastic models of categorical random variables B_i representing what specific base b_i might observed at each position i in a DNA sequence. You can think of B_i as a placeholder that allows us to talk about what base might be in position i in a mysterious random biological sequence that is not actually known to us, while b_i is a specific base—one of A, C, G, or T—discovered to be at position i after we've done the necessary sequencing.

If we want to test the naive null hypothesis that all "transition probabilities" starting from a given base b_i—which are really just the conditional probabilities $\mathbb{P}(B_{i+1} = b_{i+1} \mid B_i = b_i)$ that base $i+1$ is observed to be b_{i+1} **given that** base i is observed to be b_i—are equal to the same value (which must then each be $\frac{1}{4}$), we can use the R function `chisq.test` again, but we need to convert a sequence vector into a "dinucleotide vector", the i^{th} element of which is a string of two characters b_i followed by b_{i+1}:

```
> seqVecB = humanDna$B
> length(seqVecB)
[1] 3000
> head(seqVecB, 10)
 [1] "t" "c" "c" "a" "g" "a" "a" "g" "c" "c"
> dinucVecB = paste0(
      seqVecB[1:(length(seqVecB)-1)],
      seqVecB[2:length(seqVecB)]
  )
> length(dinucVecB)
[1] 2999
> head(dinucVecB, 10)
 [1] "tc" "cc" "ca" "ag" "ga" "aa" "ag" "gc" "cc" "ct"
```

Let's now count the occurrences of the various dinucleotide combinations:

```
> dinucCountsB = table(dinucVecB)
> dinucCountsB
dinucVecB
 aa  ac  ag  at  ca  cc  cg  ct  ga  gc  gg  gt  ta  tc  tg  tt
223 150 235 149 214 237  96 250 195 180 212 130 125 230 174 199
```

We can then test the null hypothesis that if base b_i is, say, C, the probability of base b_{i+1} is 25% for each of the four possibilities A, C, G, or T:

```
> dinucCountsStartingWithC = dinucCountsB[c("ca", "cc", "cg", "ct")]
> dinucCountsStartingWithC
dinucVecB
 ca  cc  cg  ct
214 237  96 250
> chisq.test(dinucCountsStartingWithC, p=rep(1/4, 4))
        Chi-squared test for given probabilities

data:  dinucCountsStartingWithC
X-squared = 74.674, df = 3, p-value = 4.256e-16
```

So with the aid of this simple Markov model approach, we find that despite the relative ratios of the four different bases being similar overall in sequence B, we find very strong evidence that there are some *di*nucleotide combinations that are more common than others in this sequence!

To get a better understanding of what is going on here, let's actually make estimates of the joint $\mathbb{P}(B_i = b_i,\, B_{i+1} = b_{i+1})$ and conditional $\mathbb{P}(B_{i+1} = b_{i+1} \mid B_i = b_i)$ probability distributions. We'll start by rearranging `dinucCountsB` into a `matrix` structure:

```
>  ## look at dinucCountsB again:
> dinucCountsB
dinucVecB
 aa  ac  ag  at  ca  cc  cg  ct  ga  gc  gg  gt  ta  tc  tg  tt
223 150 235 149 214 237  96 250 195 180 212 130 125 230 174 199
>  ## arrange dinucCountsB into matrix;
>  ## - use byrow=TRUE since we want
>  ##   aa, ac, ag, and at---first 4 elements of dinucCountsB,
>  ##   representing all transitions from a to a, c, g, or t---
>  ##   to be packed into first row of matrix:
> dinucCountsB = matrix(dinucCountsB, nrow=4, ncol=4, byrow=TRUE)
>  ## assign names to rows (which represent base i)
>  ##          and to columns (which represent base i+1)
>  ## using dimnames: rows are 1st dimension, columns 2nd:
> dimnames(dinucCountsB) = list("i" = c("a", "c", "g", "t"),
                                "i+1" = c("a", "c", "g", "t"))
> dinucCountsB
   i+1
i     a   c   g   t
  a 223 150 235 149
  c 214 237  96 250
  g 195 180 212 130
  t 125 230 174 199
```

The joint probabilities are in fact proportional to the raw counts, and can be obtained by simply scaling the matrix of counts down by a factor equal to the overall sum of all counts:

```
> dinucJointFracsB = dinucCountsB / sum(dinucCountsB)
> dinucJointFracsB
   i+1
i           a          c          g          t
  a 0.07435812 0.05001667 0.07835945 0.04968323
  c 0.07135712 0.07902634 0.03201067 0.08336112
  g 0.06502167 0.06002001 0.07069023 0.04334778
  t 0.04168056 0.07669223 0.05801934 0.06635545
```

The conditional probabilities

$$\mathbb{P}\left(B_{i+1} = b_{i+1} \mid B_i = b_i\right) \;=\; \frac{\mathbb{P}\left(B_i = b_i,\, B_{i+1} = b_{i+1}\right)}{\mathbb{P}\left(B_i = b_i\right)} \tag{3.1}$$

are a bit more complex to calculate, but R makes this pretty convenient

```
> iNucCountsB = rowSums(dinucCountsB)
> sum(iNucCountsB)  ## lose 1 count here since last base in sequence B
[1] 2999
>                   ## has no downstream base i+1 to be position i for!
>  ## we can divide the 4x4 matrix dinucCountsB
>  ##            by the 4-vector iNucCountsB;
>  ## in R, this will produce a matrix with i, j entry equal to
>  ##            dinucCountsB[i, j] / iNucCountsB[i]
>  ## which is what we want for conditional prob estimates here:
> dinucCondFracsB = dinucCountsB / iNucCountsB
> dinucCondFracsB
   i+1
i          a         c         g         t
  a 0.2945839 0.1981506 0.3104359 0.1968296
  c 0.2685069 0.2973651 0.1204517 0.3136763
  g 0.2719665 0.2510460 0.2956764 0.1813110
  t 0.1717033 0.3159341 0.2390110 0.2733516
```

So, we can see that if there is a C at position i, we see that the chance of the next base (at $i+1$) being a G is only 12%, while the chance that the there is a T at $i+1$ instead is 31.4%. This depletion of CG dinucleotides is common across mammalian genomes, occurring because:

1. cytosine nucleotides are often *methylated* when followed by gaunine nucleotides, and
2. methylated cytosine (C) residues are spontaneously deaminated, thereby being converted into thymine (T) residues, at a relatively rapid rate

In fact, the frequency of CG dinucleotides in the sequence `humanDna$B` is relatively *high* at 12%! Consider sequence `humanDna$A`:

```
>  ## extract sequence vector for humanDna$A:
> seqVecA = humanDna$A
>  ## construct vector of dinucleotides:
> dinucVecA = paste0(
      seqVecA[1:(length(seqVecA)-1)],
      seqVecA[2:length(seqVecA)]
  )
>  ## use table function to count how many times each dinucleotide occurs
>  ## (won't bother to rearrange into matrix this time):
> dinucCountsA = table(dinucVecA)
>  ## scale by total number of dinucleotides to estimate
>  ## joint probabilities for sequence A:
> dinucCountsA / sum(dinucCountsA)
dinucVecA
        aa         ac         ag         at         ca         cc         cg
0.03206413 0.07214429 0.06412826 0.04008016 0.09819639 0.10020040 0.01402806
        ct         ga         gc         gg         gt         ta         tc
0.11222445 0.04008016 0.05210421 0.03607214 0.05410822 0.03807615 0.09819639
        tg         tt
0.06813627 0.08016032
```

For sequence `humanDna$A`, the CG dinucleotide frequency is only 1.4%! This is in fact pretty close to the overall frequency of just about 1% across the entire human genome.

Why the difference between the two sequences A and B considered here? Sequence A was selected randomly from the entire human genome, while sequence B is the region immediately upstream of the gene TP53. The cytosine residues in CG nucleotides are much more likely to be left unmethylated in such regions, as methylation would tend to repress transcription of the downstream gene. Thus, the rate of conversion of such C residues to T residues is much less than in most of the genome.

In fact, identification of so-called "CpG islands" (CGIs), or regions in which CG dinucleotides occur much more frequently than in the rest of the genome, has played a key part in methods for identifying where genes might be found in mammalian genomes. Markov models can be used to find CGIs, but the standard way in which this is done requires introducing a so-called "hidden" (or *latent*) state variable specifying whether each position i is within a CGI or not. We'll talk about such hidden Markov models (HMMs) a bit later, but let's first consider extending the basic dinucleotide approach we've been using to take longer nucleotide runs into consideration.

3.4 Higher-order Markov models and k-mers

We've seen that biological sequences involve interdependence between the identities of consecutive monomer units, and that we can build capture some of these relationships by modeling the likelihood of seeing a particular character b_i at the i^{th} position in a sequence as a random variable whose distribution depends on the identity b_{i-1} of the character at the $(i-1)^{\text{th}}$ position. But why stop there? Perhaps we could do better with a model which allows b_i to depend on several characters—let's say, k characters—preceding it, so that instead of estimating joint probabilities for just two-character combinations like "AA", "AC", ..., "TT", we would characterize sequences by the joint probabilities for k-character substrings of the full sequence.

A run of k consecutive individual units—such as nucleotides or amino acid residues—of a polymer is referred to in the literature as a k-mer. Higher-order Markov models (of order $k > 1$) are characterized by the relative frequencies of different k-mers and $(k+1)$-mers. To see this, recall from the definition of conditional probability that

$$\mathbb{P}(B_i = b_i \mid B_{i-k} = b_{i-k}, \ldots, B_{i-1} = b_{i-1}) = \frac{\mathbb{P}(B_{i-k} = b_{i-k}, \ldots, B_{i-1} = b_{i-1}, B_i = b_i)}{\mathbb{P}(B_{i-k} = b_{i-k}, \ldots, B_{i-1} = b_{i-1})} \tag{3.2}$$

which I'll write here more compactly, by omitting the capital letter random variable names (B_i, etc.) when they are obvious from the names of the values they take (b_i, etc.), as:

$$\mathbb{P}(b_i \mid b_{i-k}, \ldots, b_{i-1}) = \frac{\mathbb{P}(b_{i-k}, \ldots, b_{i-1}, b_i)}{\mathbb{P}(b_{i-k}, \ldots, b_{i-1})} \tag{3.3}$$

Given the number of occurrences of each distinct k-mer—along with the number of occurrences of each distinct $(k+1)$-mer—in our sequence of interest, the conditional probabilities of Eq (3.3) can be easily derived: Note that the right-hand side of Eq (3.3) is the ratio of two joint probabilities, for which the standard statistical estimators would be simply:

$$\mathbb{P}(b_{i-k}, \ldots, b_{i-1}, b_i) = \frac{\text{number of occurrences of } (k+1)\text{-mer } b_{i-k} \cdots b_{i-1} b_i \text{ in sequence}}{\text{length of sequence} - k} \tag{3.4}$$

(where the denominator counts the number of distinct positions in the sequence where the $(k+1)$-mer $b_{i-k} \cdots b_{i-1} b_i$ could potentially occur) and, similarly,

$$\mathbb{P}(b_{i-k}, \ldots, b_{i-1}) = \frac{\text{number of occurrences of } k\text{-mer } b_{i-k} \cdots b_{i-1} \text{ in sequence}}{\text{length of sequence} - (k-1)} \tag{3.5}$$

However, as the order k of the Markov model increases, the number a^k of distinct k-mers which can be formed from an alphabet of a different characters increases exponentially, leading to very imprecise estimates with undesirable statistical properties when the standard estimators of Eq (3.4) and Eq (3.5) are employed. There are ways of improving these estimates for larger k values—some as simple as adding a so-called "pseudocount" to both the numerator and denominator of Eq (3.4) and Eq (3.5)—but, because time is limited, I'm not going to pursue higher-order Markov models any further. Instead, I'm going to shift focus temporarily from *modeling* sequences to simply *characterizing* them by the occurrence and frequencies various k-mers that may be found in them.

3.5 Biological sequence motifs

Many interactions between biopolymers like DNA and proteins are mediated by short localized segments whose physicochemical properties are just right for one molecule to fit together with and bind to another. These properties are largely a function of the local sequence, and if the requirements for the interaction to occur are stringent enough, there may be a single specific k-mer which must be present in one of the molecules for it to interact with its potential partner. However, there is often some tolerance for a bit of variation at some positions, so that there may be multiple very similar k-mers which would work just fine.

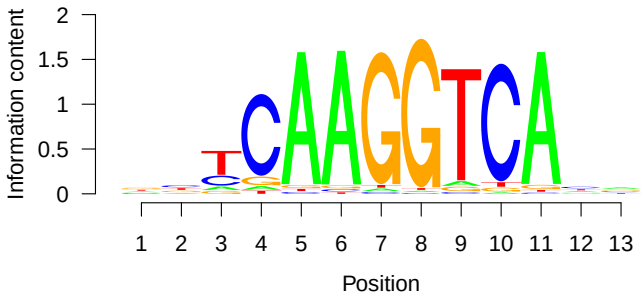
The term *sequence motif* is used in biology to describe any sequentially-localized short (at least, relative to the full sequence) pattern found in many different places which is thought to have some biological significance. Many of these motifs are thought to mediate biochemical interactions as described above, but it is rare that they may be described by a single k-mer.

Instead there are many different ways of representing sequence motifs, with some representations more appropriate than others depending on the details of the scenario being modeled. There is one way of describing motifs that is simultaneously complex enough to adequately describe a wide range of biological sequence motifs while being simple enough to handle without overly onerous algorithms and which has the hard-to-overestimate value of being easily visualizable: the *position weight matrix* (PWM).

Before trying to give any sort of definition, I'll show you an example:

	1	2	3	4	5	6	7	8	9	10	11	12	13
A (%)	20.1	13.6	8.9	3.7	94.9	95.2	1.8	0.9	3.9	1.1	94.8	21.3	35.9
C (%)	19.5	33.8	24.8	84.1	1.4	1.6	1.3	0.6	1.5	92.3	0.9	35.2	21.5
G (%)	34.6	24.3	7.1	8.9	2.1	2.2	94.9	97.4	3.7	3.0	3.0	17.1	22.7
T (%)	25.7	28.3	59.2	3.3	1.6	1.0	1.9	1.1	91.0	3.6	1.3	26.4	20.0

And a so-called *sequence logo* visualization of the motif described by the PWM above:



From a **g**rammar of **g**raphics perspective, the sequence logo maps:

x the within-motif position to the x-axis position;

y the *information content* of each position—which measures how selective the motif is regarding the character occurring at the position—to the height of the letter stack;

size the size of each letter within a column is proportional to the relative frequency of the character at that position;

shape the identity of each letter (=shape of the plot symbol) is that of the encoded character; and

color rendundant with shape, encodes which character from the sequence alphabet is being visualized.

The PWM shown above represents the DNA binding motif for the transcription factor ESRRA (estrogen related receptor alpha). The ESRRA protein shows a clear preference for the 7-mer AAGGTCA making up positions 5-11 of this motif—though even within these positions you can see that there is a bit of room for variation—with some less pronounced preferences for the bases immediately surrounding this core.

Incidentally, this motif is of pretty typical length for eukaryotic transcription factor binding site (TFBS) motifs—prokaryotic TFBS motifs tend to be a bit longer! It's thought that this reflects a greater tendency for the control of eukaryotic transcription—which is a much more daunting task, given that unlike prokaryotes, eukaryotic cells vary greatly in what portions of the genome they make use of—to be handled through the integration of many distinct but interdependent transcription factor binding events (Stewart *et al.* (2012)). You might think of prokaryotic transcription as being controlled by invocation of specific complex code words, while eukaryotic transcription requires instead fully formed sentences composed of smaller words but with complex grammatical structure.

Getting back to PWMs: As you can see from the ESRRA example, a PWM is a matrix whose (i, j)-th entry gives the relative frequency with which the character b_i occurs in position j within the motif. Highly specific positions in the motif correspond to columns in the PWM for which one row receives nearly 100% of the weight, leaving very little to disperse amongst the other possible characters at that position.

Note that PWMs do *not* offer any way to indicate the character present at one position might be in any dependent on the characters present at other positions in the matrix, nor do they allow for motifs to incorporate internal insertions or deletions! These are definite limitations of PWMs in terms of their ability to fully describe biological sequence motifs, though they can be partially mitigated by providing multiple PWMs representing variations on the same underlying biological motif.

I pulled this PWM from the JASPAR database (Fornes *et al.* (2019)) by reading from the file at `http://jaspar.genereg.net/download/CORE/JASPAR2020_CORE_vertebrates_non-redundant_pfms_meme.txt`:

```
> jaspURL = paste0("http://jaspar.genereg.net/download/CORE/",
                   "JASPAR2020_CORE_vertebrates_non-redundant_pfms_meme.txt")
>  ## read lines from file at jaspURL into character vector using readLines:
> jaspLines = readLines(jaspURL)
>  ## examine by printing first 40 lines from jaspLines to screen,
>  ## use cat function to print without quotation marks, set sep arg to "\n"
>  ## so that lines are separated by standard newline character:
> cat(head(jaspLines, n=40), sep="\n")
MEME version 4

ALPHABET= ACGT

strands: + -

Background letter frequencies
A 0.25 C 0.25 G 0.25 T 0.25

MOTIF MA0006.1 Ahr::Arnt
letter-probability matrix: alength= 4 w= 6 nsites= 24 E= 0
 0.125000  0.333333  0.083333  0.458333
 0.000000  0.000000  0.958333  0.041667
 0.000000  0.958333  0.000000  0.041667
 0.000000  0.000000  0.958333  0.041667
 0.000000  0.000000  0.000000  1.000000
 0.000000  0.000000  1.000000  0.000000
URL http://jaspar.genereg.net/matrix/MA0006.1

MOTIF MA0854.1 Alx1
letter-probability matrix: alength= 4 w= 17 nsites= 100 E= 0
 0.190000  0.440000  0.170000  0.200000
 0.070000  0.210000  0.650000  0.070000
 0.380000  0.330000  0.140000  0.150000
 0.430000  0.090000  0.310000  0.170000
 0.050000  0.400000  0.020000  0.530000
 0.010000  0.070000  0.000000  0.920000
 0.989899  0.000000  0.010101  0.000000
 0.980000  0.000000  0.010000  0.010000
 0.010000  0.010000  0.000000  0.980000
 0.000000  0.010101  0.000000  0.989899
 0.920000  0.000000  0.070000  0.010000
 0.530000  0.020000  0.400000  0.050000
 0.150000  0.210000  0.110000  0.530000
 0.230000  0.320000  0.160000  0.290000
 0.393939  0.313131  0.121212  0.171717
 0.240000  0.290000  0.230000  0.240000
 0.150000  0.400000  0.140000  0.310000
URL http://jaspar.genereg.net/matrix/MA0854.1
```

The file at `jaspURL` is in "MEME format" where MEME (Bailey *et al.* (2006)) is one of many bioinformatics programs for identifying and characterizing sequence motifs given a set of biological sequences of interest. Each individual motif is specified by a block of non-empty lines

- beginning with a line of the form "MOTIF <*motif id*> <*motif name*>",
- followed by a line "letter-probability matrix: <*more info*>",
- and then a series of lines, one per position within the motif, containing the entries in the PWM
  - note these are transposed relative to the PWM definition, with rows representing position and columns representing which base;
- finally concluding with either an empty line (not shown in example) or a line "URL <*JASPAR URL for the individual motif*>".

This gives me a good excuse to spend a minute or two discussing a nuisance task that is unfortunately unavoidable for anyone pursuing computational biology or bioinformatics: writing code to parse custom text file formats. For relatively small files like this one, it is often easiest to to simply "slurp" all lines of the file into a character vector using `readLines` (or the equivalent in whatever programming language you're using) and then process them in a `for` loop as done in the example `parseMeme` function provided below, but do be aware that **file slurping is not a good idea with very large files**.

That said, the motif files from JASPAR are quite small and easily slurp-able. Thus `parseMeme` will be defined here using `readLines` to slurp in the contents of the JASPAR motif file—whether it be a local file on your computer or a remote file described by its URL—before looping through the slurped lines. In each pass of the `for` loop, `parseMeme` looks for one of the specific patterns described above indicating that either

1. a motif block is beginning (in which case it records the id+name of the motif in the variable `activeMotif`);
2. we are about to enter the letter-probability matrix section itself for whatever the `activeMotif` is, in which case it flips logical flag `inMatrix` to `TRUE`; and finally
3. if `inMatrix` it checks whether or not the line looks like numbers which need to be read into the PWM or
4. like an empty string or URL line indicating that the PWM has been fully read.

These 4 possibilities correspond to the 4 blocks in the `if`...`else if`...`else if`...`else if` section of the code below:

```
> parseMeme = function(f) {
       ## read all lines of file f into character vector, add empy line at end:
      lines = c(readLines(f), "")
      pwms = list()       ## output list will be populated with matrices below
      activeMotif = NULL  ## will track which motif we're currently parsing
      inMatrix = FALSE    ## are we currently parsing text for a matrix or not?
      for (line in lines) {
          if (substr(line, 1, 6) == "MOTIF ") {
               ## about to start reading in a PWM; read name of motif;
               ## use everything from character 7 on to remove "MOTIF ":
              activeMotif = substr(line, 7, nchar(line))
              pwms[[activeMotif]] = list()  ## initialize empty list to read
                                            ## matrix lines into as vectors
          } else if (substr(line, 1, 26) == "letter-probability matrix:") {
               ## this line immediately precedes numeric PWM info, so set:
              inMatrix = TRUE
          } else if (inMatrix && line != "" && substr(line, 1, 4) != "URL ") {
               ## parse line into numeric vector; first remove
               ## any space characters ("\\s+") following line start ("^"):
              line = gsub("^\\s+", "", line)
               ## now strsplit on any white space ("\\s+") and extract first
               ## (and only) element [[1]] of list returned by strsplit,
               ## convert resulting character vector to numeric, and finally
               ## add numeric vector to growing list pwms[[activeMotif]]:
              pwms[[activeMotif]][[length(pwms[[activeMotif]])+1]] =
                      as.numeric(strsplit(line, "\\s+")[[1]])
          } else if (inMatrix && (line == "" || substr(line, 1, 4) == "URL ")) {
              if (length(activeMotif) > 0) {
                   ## get here when just finished parsing a PWM;
                   ## combine list of vectors pwms[[activeMotif]] together
                   ## into matrix using do.call with cbind so that each vector
                   ## becomes one column in matrix version of pwms[[activeMotif]]:
                  pwms[[activeMotif]] = do.call(cbind, pwms[[activeMotif]])
                  rownames(pwms[[activeMotif]]) = c("A", "C", "G", "T")
              }
               ## switch off activeMotif, inMatrix since just finished a PWM:
              activeMotif = NULL
              inMatrix = FALSE
          }
      }
      return(pwms)
  }
```

Now let's use `parseMeme` to parse the full file at `jaspURL`:

```
> jaspVert = parseMeme(jaspURL)
```

So: we've seen what a PWM is and how to extract them from the files provided by JASPAR, but how do we find actual real-world occurrences of the motifs they represent in DNA sequences? The first thing to note is that the one obvious thing we can do with a PWM is to "score" any k-mer with k equal to the number of columns in the PWM (that is, the number of positions or sites associated with the motif): If we make the traditional PWM assumption that the each site in the motif is selected independently of all of the others, then the probability score associated with a k-mer is simply the product of the entries `pwm[`b_i`, i]` of each of the characters $b_1, \ldots, b_k$ making up the k-mer:

```
>  ## arguments to pwmProbScore:
>  ## - pwm is position weight matrix;
>  ##   rows should be named by characters from sequence alphabet
>  ## - kmer can be either single string or vector of k single characters
> pwmProbScore = function(pwm, kmer) {
      if (is.character(kmer) && length(kmer) == 1) {
           ## split kmer up into vector of individual bases:
          kmer = strsplit(kmer, "")[[1]]
      }
       ## use lapply to select the prob value from the i-th column of pwm
       ## for the character kmer[[i]] as the i-th element of ouput vector p:
      p = lapply(1:ncol(pwm), function(i) {pwm[kmer[[i]], i]})
       ## would rather have p as vector (for prob function below) than list;
       ## can use R's unlist function to accomplish this:
      p = unlist(p)
       ## use prod function to multiply p[[1]] * p[[2]] ... * p[[k]],
       ## where k is the number of characters in kmer:
      return(prod(p))
  }
```

Thus, if we wanted to see how good of a match is the the 13-mer AAAAAAAAAAAAA to the ESRRA motif introduced above, we would run:

```
> pwmProbScore(jaspVert[["MA0592.3 ESRRA"]], "AAAAAAAAAAAAA")
[1] 4.103863e-13
```

OK, but how does that compare to the score associated with a 13-mer which might actually look like a match to the ESRRA TFBS? In order to answer this question, it is useful to have a function `bestKmer` for identifying the best matching k-mer for a PWM `pwm`:

```
> bestKmer = function(pwm) {
       ## pick out maximum probability entry from each column:
      kmer = apply(pwm, 2, which.max)
       ## convert kmer from integer representation to character vector:
      kmer = rownames(pwm)[kmer]
       ## paste together with arg collapse="" into single string and return:
      return(paste(kmer, collapse=""))
  }
```

Now we can use `bestKmer` to find a good binding site for ESRRA and assess the `pwmProbScore` for it:

```
> esrraBestKmer = bestKmer(jaspVert[["MA0592.3 ESRRA"]])
> esrraBestKmer
[1] "GCTCAAGGTCACA"
> pwmProbScore(jaspVert[["MA0592.3 ESRRA"]], esrraBestKmer)
[1] 0.004898776
```

The 13-mer GCTCAAGGTCACA matches the ESRRA binding motif 11936989043.25 times better than does AAAAAAAAAAAAA.

Is this enough to rule out the possibility that any given occurrence of AAAAAAAAAAAAA in a genome could be a binding site for ESRRA? Probably, but how do we know? This is actually a difficult problem which no one has really presented a particularly satisfactory answer for as yet. So, while decidedly *not* a universally acceptable rule for doing so, I'm just going to adopt a simple convention of considering all k-mers scoring within a factor of 10 of `bestKmer(pwm)` as good matches to `pwm` for the purposes of illustration in this section.

This brings up a couple of points:

- given a threshold score for identifying matches to a PWM, the PWM becomes essentially a simple list of those k-mers whose scores exceed the threshold,
- so we can find matches to the PWM by first enumerating all good k-mer matches and then simply searching for those k-mers in whatever biological sequence we want to identify motifs in.

In fact this is the generally the fastest way to identify **short** motifs in **long** sequences (Ambrosini *et al.* (2018)).

(For identification of longer motifs in not-so-long sequences, it may be faster to, e.g., slide the PWM window along the sequence and evaluate each potential match directly, since when k is large, the number a^k—a being the size of the sequence alphabet—of possible k-mers may exceed the length of the sequence in which we're searching for motifs.)

3.5.1 Backtracking to enumerate k-mers matching a PWM

The problem of identifying all `kmer`s which match a `pwm` given a score `threshold` is in principle quite a simple one: enumerate all a^k distinct `kmer`s in the sequence alphabet, run `pwmProbScore(pwm, kmer)` on each of them, and return a data structure indicating which ones had scores $\geq$ `threshold`.

Even if both a and k are relatively small—say, 4 and 13, respectively—this may not be a very efficient approach, however: 4^{13} is more than 67 million distinct `kmer`s to consider!

We can do better by noting that: if we split `kmer` into two parts—a prefix consisting of the first $p < k$ characters and a suffix consisting of the last $s = k - p$ characters—the score of `kmer` is the product of the score of the prefix using the first p columns of `pwm` and the score of the suffix using the last s columns of `pwm`. Since both the prefix and suffix scores must be ≤ 1, if the prefix score is already below `threshold`, it doesn't matter what the suffix is: `kmer` cannot be a good match!

For problems with this "partial candidate solution" property, in which it is possible to easily verify that many partial candidates cannot possibly be completed to a full solution, we can use an approach known as *backtracking* to greatly reduce the number of candidates which must be explicitly considered. Instead of only considering full candidate solutions, we try out partial candidates, steadily extending them until we either (1) have a good full solution or (2) find that there cannot be any solutions extending the partial candidate. At that point we backtrack to the next partial candidate, thus ruling out a^s (the number of suffixes which could have extended our bad prefix) possible solutions at once!

The key implementation detail in such a backtracking approach is how to find the next partial candidate solution prefix once we're done with a given prefix. For the k-mer enumeration problem, we'll rely on lexicographic (alphabetical) ordering to do this in the `nextKmerPrefix` function defined here:

```
> ## arguments to nextKmerPrefix:
> ## - kmer is represented as *integer* vector for efficiency; for example,
> ##   "TCGAGG" would be represented by c(4, 2, 3, 1, 3, 3)
> ## - activeIndex is single integer indicating that kmer[1:activeIndex]
> ##   is the actual prefix under current consideration in enumerateKmers;
> ##   (the suffix kmer[(activeIndex+1):length(kmer)] is ignored)
> nextKmerPrefix = function(kmer, activeIndex) {
      while (activeIndex > 0 && kmer[[activeIndex]] == 4) {
           ## there are only 4 possible nucleotides, so if
           ## kmer[[activeIndex]] is 4, need to reset kmer[[activeIndex]]
           ## to 1 (encoding lexicographically first base A)...
          kmer[[activeIndex]] = 1
           ## and then shift activeIndex back to the left one position:
          activeIndex = activeIndex - 1
      }
      if (activeIndex == 0) {
           ## in this case we've exhausted all possible kmers and are done
          return(NULL)
      }
       ## now advance the base at position activeIndex by 1
       ## (from A to C, C to G, or G to T, depending on current value):
      kmer[[activeIndex]] = kmer[[activeIndex]] + 1
       ## need to return both the updated kmer and activeIndex values:
      return(list(kmer=kmer, activeIndex=activeIndex))
  }
```

With the aid of `nextKmerPrefix`, it's not too difficult to put together an `enumerateKmers` implementation of the backtracking method. We just need to keep track in each iteration of whether we've completed consideration of the current prefix—either by finding a good full solution or by ruling out the possibility of any good full solutions—or if it still appears possible to extend the prefix to a good solution. The function below uses a logical variable `callNextKmerPrefix`, which will be set to `TRUE` if we've completed consideration of the current prefix and `FALSE` otherwise, to track this:

```
> enumerateKmers = function(pwm, threshold) {
       ## represent kmer being considered as *integer* vector:
      kmer = rep(1, ncol(pwm))
      nucs = c("A", "C", "G", "T")  ## so nucs[kmer] is actual base sequence
      activeIndex = 1  ## index position in kmer currently under consideration
       ## p[[i]] tracks probability from PWM for kmer base at index i;
       ## initialize for kmer consisting of ncol(pwm) consecutive As:
      p = pwm[1, ]     ## (since A is nucs[[1]])
      out = character(0)   ## good kmer matches will be added to out
      done = FALSE     ## will be reset to TRUE inside while loop when done
      while (!done) {
           ## some passes through loop tell us there cannot be any more kmers
           ## starting with kmer[1:activeIndex] which could be a good match:
           ## in these cases we'll call nextKmerPrefix function to figure out
           ## next kmer prefix to consider, for now assume we will do so:
          callNextKmerPrefix = TRUE  ## but reset to FALSE below if necessary
          if (prod(p[1:activeIndex]) >= threshold) {
               ## then kmer[1:activeIndex] could be prefix of good match(es):
              if (activeIndex == ncol(pwm)) {  ## we've confirmed right-most
                   ## position, thus kmer is a good match to pwm, so paste
                   ## nucs[kmer] together with collapse="" and add to out:
                  out[[length(out)+1]] = paste(nucs[kmer], collapse="")
              } else {  ## haven't checked full kmer yet; check next position:
                  activeIndex = activeIndex + 1
                   ## and make sure p[[activeIndex]] correctly set according
                   ## to whatever base is encoded by kmer[[activeIndex]]:
                  p[[activeIndex]] = pwm[kmer[[activeIndex]], activeIndex]
                   ## still working on extending this kmer, so:
                  callNextKmerPrefix = FALSE
              }
          }
          if (callNextKmerPrefix) {
              nextInfo = nextKmerPrefix(kmer, activeIndex)
              if (length(nextInfo) == 0) {  ## signals we're done:
                  done = TRUE
              } else {
                  kmer = nextInfo$kmer
                  activeIndex = nextInfo$activeIndex
                   ## make sure p[[activeIndex]] correctly set according
                   ## to whatever base is encoded by kmer[[activeIndex]]:
                  p[[activeIndex]] = pwm[kmer[[activeIndex]], activeIndex]
              }
          }
      }
      return(out)
  }
```

Here then are the 13-mers which match the ESRRA binding motif with probability scores at least 10% that of the best matching 13-mer GCTCAAGGTCACA:

```
> esrraPwm = jaspVert[["MA0592.3 ESRRA"]]
> esrraKmers = enumerateKmers(
      esrraPwm,
      threshold = pwmProbScore(esrraPwm, esrraBestKmer) / 10
  )
> length(esrraKmers)
[1] 370
> head(esrraKmers)
[1] "AATCAAGGTCAAA" "AATCAAGGTCACA" "AATCAAGGTCACC" "AATCAAGGTCACG"
[5] "AATCAAGGTCACT" "AATCAAGGTCAGA"
```

Note that in finding the 370 matching `esrraKmers`, `enumerateKmers` passes through the while loop a total 7,856 times, whereas a naive consider-all-4^{13}-13-mers approach would require consideration of 67,108,864 possible matches. The advantage of backtracking over brute force enumeration of all possible solutions—when partial solutions may be easily evaluated—can be very large!

Still, in order to find motif occurrences in large biological sequences, we need an efficient algorithm for locating arbitrary k-mers in a big reference string. This is actually one of the more fundamental problems in bioinformatics (not to mention many other areas of computer science), so we're going to take a fairly deep dive into a pretty complicated—but ultimately extremely useful—set of data structures which facilitate the efficient implementation of a wide range of critical string algorithms: the *suffix array*, the *Burrows-Wheeler transform* and the *FM index*.

3.6 Suffix arrays and the Burrows-Wheeler transform

The computational problem of locating k-mers in a large reference string comes up in many different contexts and has consequently been attacked using numerous different algorithmic approaches. One of the most efficient approaches to tackling this problem—at least, when the reference string is assumed to be fixed and known in advance while the k-mers might be of any number and length—involves the use of a data structure known as a *suffix array* (Manber & Myers (1993); Kärkkäinen & Sanders (2003)) combined with a special transformed version (the *Burrows-Wheeler transform* (Burrows & Wheeler (1994))) of the reference string to *index* (Ferragina & Manzini (2000)) the reference string for rapid searching.

Let's start by defining the *suffix*es of a string: For a string `seqAsString` defined as we did previously:

```
>  ## set random number generator seed for reproducibility:
> set.seed(123)
>  ## sample from A, C, G, T *with replacement* nBases times:
> seqAsVector = sample(c("A", "C", "G", "T"), 25, replace=TRUE)
> seqAsString = paste(seqAsVector, collapse="")
```

the i^{th} `suffix` of `seqAsString` is the substring formed by removing the first $i-1$ characters from `seqAsString`:

```
>  ## add special terminator character "$" to end of string;
>  ## will be useful when we discuss Burrows-Wheeler transform later:
> seqAsString = paste0(seqAsString, "$")
> n = nchar(seqAsString)
>  ## we'll use lapply to loop through i in 1:n,
>  ## removing the first i-1 characters in iteration i:
> suffixes = lapply(1:n, function(i) {substr(seqAsString, i, n)})
>  ## would rather have suffixes as vector than list:
> suffixes = unlist(suffixes)
> cat(head(suffixes), sep="\n")
GGGCGCCCGATCCACGTAGGATAAA$
GGCGCCCGATCCACGTAGGATAAA$
GCGCCCGATCCACGTAGGATAAA$
CGCCCGATCCACGTAGGATAAA$
GCCCGATCCACGTAGGATAAA$
CCCGATCCACGTAGGATAAA$
```

The idea behind the suffix array is that when we lexicographically (alphabetically) sort the suffixes of `seqAsString` we put those suffixes which start with a common k-mer together:

```
> sortSuf = sort(suffixes)
> cat(sortSuf, sep="\n")
```

```
$
A$
AA$
AAA$
ACGTAGGATAAA$
AGGATAAA$
ATAAA$
ATCCACGTAGGATAAA$
CACGTAGGATAAA$
CCACGTAGGATAAA$
CCCGATCCACGTAGGATAAA$
CCGATCCACGTAGGATAAA$
CGATCCACGTAGGATAAA$
CGCCCGATCCACGTAGGATAAA$
CGTAGGATAAA$
GATAAA$
GATCCACGTAGGATAAA$
GCCCGATCCACGTAGGATAAA$
GCGCCCGATCCACGTAGGATAAA$
GGATAAA$
GGCGCCCGATCCACGTAGGATAAA$
GGGCGCCCGATCCACGTAGGATAAA$
GTAGGATAAA$
TAAA$
TAGGATAAA$
TCCACGTAGGATAAA$
>  ## sorted suffixes 16 and 17 correspond to the two suffixes of seqAsString
>  ## which start with the 3-mer GAT:
> sortSuf[16:17]
[1] "GATAAA$"            "GATCCACGTAGGATAAA$"
```

Where do these two suffixes start in `seqAsString`? Recall that (unsorted) `suffixes` vector element i is defined as the substring of `seqAsString` starting at (spatial) position i within the string:

```
>  ## use which to find which element of suffixes is 16th when sorted:
> which(suffixes == sortSuf[[16]])
[1] 20
>  ## store this value in spatialPosOfSortSuf16:
> spatialPosOfSortSuf16 = which(suffixes == sortSuf[[16]])
>  ## double-check:
> substr(seqAsString, spatialPosOfSortSuf16, n)
[1] "GATAAA$"
>  ## more to the point:
> substr(seqAsString, spatialPosOfSortSuf16, spatialPosOfSortSuf16+2)
[1] "GAT"
```

```
>  ## try same thing for suffix which sorts to position 17:
> spatialPosOfSortSuf17 = which(suffixes == sortSuf[[17]])
> spatialPosOfSortSuf17
[1] 9
> substr(seqAsString, spatialPosOfSortSuf17, spatialPosOfSortSuf17+2)
[1] "GAT"
```

We can figure out what the spatial positions corresponding to all of the sorted suffixes at once using the `order` function:

```
> suffixArray = order(suffixes)
> suffixArray
 [1] 26 25 24 23 14 18 21 10 13 12  6  7  8  4 15 20  9  5  3 19  2  1 16 22 17
[26] 11
>  ## double-check specifically the values we already calculated above:
> suffixArray[16:17]
[1] 20  9
```

Note: Enumerating all suffixes and then using `order` to obtain the suffix array is horribly inefficient—in both time and memory—for large strings! The fundamental importance of suffix arrays (and the close related suffix trees) to modern string algorithms is crucially reliant on the existence of better algorithms than this for their construction. Unfortunately, those better algorithms are also much more complicated, so in this class we're only going to see how to use suffix arrays, not the best ways to build them!

As you have just seen, the `suffixArray` of the string `seqAsString` is defined as the array of spatial positions of all of the `suffixes` in `seqAsString` after they have been lexicographically sorted!

Any k-mer that shows up in `seqAsString` must correspond to the first k characters of the set of suffixes starting at the positions where that k-mer is to be found in `seqAsString`. Since all of the suffixes starting with the k-mer in question must form a consecutive block in `sort(suffixes)`, the positions at which the k-mer is found in `seqAsString` must form a consecutive block in `suffixArray`.

Now we just need an algorithm to find the starting and ending indices of that block...

3.6.1 Burrows-Wheeler transform (BWT) and FM index

Luckily, there is just such an algorithm! But we will need one more piece: the *Burrows-Wheeler transform* (BWT) of `seqAsString`. Here is a function to construct it in R—note that it requires both the string to be transformed and the suffix array of that string as arguments:

```
>  ## arguments to burrowsWheelerTransform:
>  ## - seq: the string whose BWT we want
>  ## - sa: the suffix array of the string seq
> burrowsWheelerTransform = function(seq, sa) {
       ## convert seq to vector of single characters:
      seqAsVector = unlist(strsplit(seq, ""))
       ## pre-allocate character vector which will contain BWT of seq:
      transformed = character(length(seqAsVector))
       ## iterate through suffix array sa:
       ## transformed[[i]] will be the character found spatially *preceding*
       ## (immediately before) the suffix specified by sa[i]
      for (i in 1:length(sa)) {
          if (sa[[i]] == 1) {
               ## have to deal with suffix 1 (corresponding to whole string seq!)
               ## specially, since there is no character preceding it:
              transformed[[i]] = "$"
          } else {
              transformed[[i]] = seqAsVector[[sa[[i]]-1]]
          }
      }
      return(transformed)
  }
```

Let's try it out:

```
> burrowsWheelered = burrowsWheelerTransform(seqAsString, suffixArray)
> burrowsWheelered
 [1] "A" "A" "A" "T" "C" "T" "G" "G" "C" "T" "G" "C" "C" "G" "A" "G" "C" "C" "G"
[20] "A" "G" "$" "C" "A" "G" "A"
```

In case you're wondering *why* anyone would define such a thing, imagine that you wanted to locate the block in `sortSuf` starting with "AT", and consider:

1. suffixes starting with "T" form block `sortSuf[24:26]` in the sorted vector of suffixes;
   - there are: 1 "$", 7 "A"s, 7 "C"s, and 8 "G"s, for a total of 23 characters in `seqAsString` which are lexicographically lower than "T"
   - since each character in `seqAsString` is the starting point of exactly one suffix, this tells us that there are 23 suffixes before we get to the block of suffixes starting with "T" in `sortSuf`—so the "T" block must start at 24.
2. The fact that there are `sum(burrowsWheelered[1:23]=="A")`=**5** "A"s each immediately preceding one of the 23 suffixes lexicographically<"T" tells us something useful:
   - there are exactly **5** suffixes of `seqAsString`:
     - A$
     - AA$
     - AAA$
     - ACGTAGGATAAA$
     - AGGATAAA$

     starting with "A?", where the character "?" can be anything <"T"—that is, "$", "A", "C", or "G". The suffixes
     - $
     - A$
     - AA$
     - CGTAGGATAAA$
     - GGATAAA$

     corresponding to the 1st, 2nd, 3rd, 15th, and 20th elements of `sortSuf`—thus all in the first 23—are the **5** satisfying:

     ```
     > burrowsWheelered[c(1, 2, 3, 15, 20)] == "A"
     [1] TRUE TRUE TRUE TRUE TRUE
     ```
3. Recalling that we are looking for all occurrences of "AT" in `seqAsString`:
   - there is 1 "$" < "A" in `seqAsString`, so block in `sortSuf` corresponding to suffixes starting with "A" starts at position 2...
   - from step 2 we know that there are `sum(burrowsWheelered[1:23]=="A")`=**5** suffixes starting with "A?", where "?" < "T".
   - then the block of suffixes in `sortSuf` starting with "AT" must start after both
     - the 1 suffix starting with "$" and then
     - the 5 suffixes starting with "A?" with "?"<"T"

Does the block of suffixes starting with AT start at the first index after 1+5=6?

```
> sortSuf[6:7]
[1] "AGGATAAA$" "ATAAA$"
```

It does indeed: `sortSuf[[6]]` is the lexicographically highest suffix less than "AT", while `sortSuf[[7]]` is the lexicographically lowest suffix starting with "AT"!

The logic laid out in the 3 steps above can be extended to find the positions of both the start and end of the blocks in `sortSuf` corresponding not only to 2-mers like "AT", but to *any* k-mer. We just need to supplement the BWT with two extra data structures:

C named vector `C` with `C[[char]]` counting how many suffixes of `seqAsString` are lexicographically<`char`

- this is just number of characters in `seqAsString` which are <`char`;
- can also be calculated as number of characters in Burrows-Wheeler transform of `seqAsString` which are <`char`
  - since Burrows-Wheeler just shuffles characters' positions.
- for our example `seqAsString`, we have seen that `C[["A"]]` takes value 1, while `C[["T"]]` takes value 23.

O named list(-of-vectors) `O` with `C[[char]][i]` tallying up how many times `char` shows up in in the first i positions of Burrows-Wheeler transform of `seqAsString`

- for our example `seqAsString`, we have seen that that `O[["A"]][[23]]` takes value **5**.

Here's a function in R for constructing and collecting the Burrows-Wheeler transform, the vector `C`, and the list-of-vectors `O` into a single list for ease of use in the `findKmer` function to be defined below:

```
> buildFMindex = function(seq, sa) {
      bwt = burrowsWheelerTransform(seq, sa)
       ## extract the alphabet of unique characters which make up seq
       ## (and, hence, which make up the vector bwt):
      alphabet = sort(unique(bwt))
       ## if we set the names of the vector alphabet, lapply
       ## loops below will retain the names for output lists:
      names(alphabet) = alphabet
       ## C[[char]] is how many suffixes lexicographically < char
       ## (note suffix is < char if and only if first char of suffix < char):
      C = lapply(alphabet, function(char) {sum(bwt < char)})
       ## O[[char]][[i]] is how many times char shows up in bwt[1:i]
       ## (first i characters of Burrows-Wheeler transform of seq):
      O = lapply(alphabet, function(char) {cumsum(bwt == char)})
       ## assemble everything into named list and return it:
      return(list(
          bwt = bwt,
          C = unlist(C),
          O = O
      ))
  }
```

As indicated by the name `buildFMindex`, the aggregate data structure consisting of the `bwt`, `C`, and `O` is often referred as an "FM-index", which stands for **F**ull-text index in **M**inute space—or possibly the names **F**erragina and **M**anzini, the authors to first propose it (Ferragina & Manzini (2000)).

Either way, here is the function `findKmer` generalizing the logic laid out in steps 1-3 above:

```
> findKmer = function(fm, sa, kmer) {
      k = nchar(kmer)
       ## start will represent first position of block
       ## corresponding to kmer in vector of sorted suffixes;
       ## initialize start to lowest possible value (1):
      start = 1
       ## end will represent first position *after* block
       ## corresponding to kmer in vector of sorted suffixes;
       ## initialize end to highest possible value:
      end = length(fm$bwt) + 1
       ## now iterate *backwards* through characters in kmer:
       ## after iteration i, start and end will be set to
       ## correct values for substr(kmer, i, k)
       ## (which are needed in order calculate start, end
       ##  values for substr(kmer, i-1, k):
      for (i in k:1) {
           ## extract character at i-th position in kmer to ichar:
          ichar = substr(kmer, i, i)
           ## block for substr(kmer, i, k) must start somewhere
           ## past blocks for suffixes starting with characters < ichar:
          nextStart = fm$C[[ichar]] + 1
          if (start > 1) {
               ## if this is not first pass through loop, we need
               ## to also move start value past all suffixes starting with
               ## "<ichar><?>",
               ## where "<?>" is any suffix less than substr(kmer, i+1, k),
               ## to get to start for "<ichar><substr(kmer, i+1, k)>"
               ## ( "<ichar><substr(kmer, i+1, k)>" is substr(kmer, i, k) )
              nextStart = nextStart + fm$O[[ichar]][[start-1]]
          }
           ## now set start to nextStart:
          start = nextStart
           ## logic for end is essentially same as for start:
          nextEnd = fm$C[[ichar]] + 1
          if (end > 1) {
              nextEnd = nextEnd + fm$O[[ichar]][[end-1]]
          }
          end = nextEnd
      }
      if (end <= start) {
          return(integer(0))  ## happens when kmer not found in string
      }
       ## use suffix array sa to map block of positions in vector
       ## of sorted suffixes back to spatial positions in original string:
      return(sa[start:(end-1)])
  }
```

Let's try it out:

```
> fm = buildFMindex(seqAsString, suffixArray)
> kmerLocations = findKmer(fm, suffixArray, "GAT")
> kmerLocations
[1] 20  9
> substr(seqAsString, kmerLocations[[1]], kmerLocations[[1]]+2)
[1] "GAT"
> substr(seqAsString, kmerLocations[[2]], kmerLocations[[2]]+2)
[1] "GAT"
> findKmer(fm, suffixArray, "AAAAAAAA")
integer(0)
```

It works!

3.6.2 Back to motif finding

Putting the backtracking algorithm implemented in `enumerateKmers` together with the suffix array/BWT/FM index method for locating k-mers implemented in `findKmer`, we are finally in a position to locate PWM motif occurrences within a large reference sequence.

(There is one major caveat to this: we haven't actually discussed how to construct the suffix array of a truly long sequence efficiently. If you'd like to see how this can be done, you might check out Kärkkäinen & Sanders (2003); unfortunately this is an algorithm which isn't all that well-suited to implementation in R!)

Let's pick out another motif from JASPAR:

```
> pitx1Pwm = jaspVert[["MA0682.2 PITX1"]]
> pitx1Kmers = enumerateKmers(
      pitx1Pwm,
      threshold = pwmProbScore(pitx1Pwm, bestKmer(pitx1Pwm)) / 10
  )
> pitx1Kmers
 [1] "ATAATCCA" "ATAATCCC" "ATAATCCG" "ATAATCCT" "CTAATCCA" "CTAATCCC"
 [7] "CTAATCCG" "CTAATCCT" "GTAATCCA" "GTAATCCC" "GTAATCCG" "GTAATCCT"
[13] "TTAATCCA" "TTAATCCC" "TTAATCCG" "TTAATCCT"
```

PITX1, or Paired-like homeodomain 1, is gene encoding a transcription factor thought to be involved in development. The binding motif for PITX1 is shorter than that for ESRRA (8 nucleotides as opposed to 13), which partly explains why we see so many fewer distinct k-mers matching the PWM with scores within 10% of the maximum score.

We'll search for `pitx1Kmers` in the `humanDna$B` sequence we loaded from the file

`two_human_dna_sequences.fa`,

following all of the steps laid out above:

```
>  ## paste individual characters in humanDna$B together into 1 string:
> stringB = paste(humanDna$B, collapse="")
>  ## convert to upper-case and add "$" to end of stringB:
> stringB = paste0(toupper(stringB), "$")
>  ## build suffix array for stringB (using naive brute force method!):
> suffixArrayB = order(unlist(lapply(
      1:nchar(stringB),
      function(i) {substr(stringB, i, nchar(stringB))}
  )))
> fmB = buildFMindex(stringB, suffixArrayB)
>  ## name the pitx1Kmers so that lapply will produced named output:
> names(pitx1Kmers) = pitx1Kmers
>  ## run findKmer on each kmer km in pitx1Kmers:
> hitsForward = lapply(
      pitx1Kmers,
      function(km) {findKmer(fmB, suffixArrayB, km)}
  )
>  ## hitsForward is a list, each element of which is vector
>  ## of locations of hits for corresponding kmer; many of these
>  ## vectors may be empty, so let's filter them out:
> hitsForward = hitsForward[unlist(lapply(hitsForward, length)) > 0]
> hitsForward
named list()
```

While we see no hits on the forward DNA strand, transcription factors like that encoded by PITX1 can generally bind to either strand of DNA, so we typically want to search the reverse complement of our reference sequence for k-mers matching the binding motif as well. This gives us an excuse to go through the evergreen excercise of writing a function to reverse-complement a DNA sequence:

```
> dnaRevComp = function(s) {
      if (length(s) > 1) {
           ## if s is vector of strings, run dnaRevComp separately
           ## on each one and then package output as vector:
          return(unlist(lapply(s, dnaRevComp)))
      }
       ## set up named character vector to function as dictionary
       ## for translating characters into their complements
       ## (preserving casing):
      dnaComplements = c(A="T", C="G", G="C", N="N", T="A",
                         a="t", c="g", g="c", n="n", t="a")
       ## strsplit s into vector sVec of individual characters:
      sVec = strsplit(s, split="")[[1]]
       ## convert each character to its complement:
      sVec = dnaComplements[sVec]
       ## now reverse the order using rev function:
      sVec = rev(sVec)
       ## paste characters back together and return:
      return(paste(sVec, collapse=""))
  }
```

With `dnaRevComp` in hand, it's just a matter of repeating the steps applied above to the forward orientation of `humanDna$B` to the reverse complement:

```
> revCompB = dnaRevComp(paste(humanDna$B, collapse=""))
> revCompB = paste0(toupper(revCompB), "$")
> saRevCompB = order(unlist(lapply(
      1:nchar(revCompB),
      function(i) {substr(revCompB, i, nchar(revCompB))}
  )))
> fmRevCompB = buildFMindex(revCompB, saRevCompB)
> hitsReverse = lapply(
      pitx1Kmers,
      function(km) {findKmer(fmRevCompB, saRevCompB, km)}
  )
> hitsReverse = hitsReverse[unlist(lapply(hitsReverse, length)) > 0]
> hitsReverse
$GTAATCCG
[1] 58
```

Thus we see that, while there were no hits for PITX1 binding motifs on the forward strand, there *is* a hit on the reverse strand!

PITX1 does indeed appear to stimulate the expression of TP53 (Liu & Lobie (2007)), the gene downstream of the sequence extracted in `humanDna$B`. Interestingly, there is another PITX1 binding site located *within* the TP53 gene itself—the first exon, to be more specific—which appears to be even more important in this regulatory relationship than the binding site we see upstream of the TP53 gene.

3.7 Hidden Markov Models (HMMs)

Let's return to Markov models and to the CpG island (CGI) detection problem posed in section 3.3. Recall that a Markov model for a discrete-(time or position) system assumes that the probabilistic distribution for the state of the system at $i+1$ depends only the state of the system at i (not on the history of the system prior to i).

For the Markov model we considered above, we equated "state of the system at position i" directly with the observed base b_i. Hidden Markov models assume that there is more to the "state" of the sequence being modeled than we can see; for instance, for CGI detection, we might supplement the observed base (referred to in general HMM terminology as the *emitted symbol*) b_i with a hidden state variable indicating whether or not position i is part of a CGI.

To keep things as simple as possible, I'm going to build an HMM for CGI detection which considers only whether each observed dinucleotide is a CG or not, and which does not even look at the distribution of the other 15 possible dinucleotides:

```
>  ## allocate vector dinucBIsCG of same length as dinucVecB
>  ## with all values set initially set to "-" (not "cg"):
> dinucBIsCG = rep("-", length(dinucVecB))
>  ## now set the appropriate values to "cg" instead of "-" using
>  ## logical indexing:
> dinucBIsCG[dinucVecB == "cg"] = "cg"
>  ## double-check that values look like we expect using table function:
> table(dinucBIsCG)
dinucBIsCG
   -   cg
2903   96
```

This allows us to use an HMM with only two states (CGI or not-CGI)—a better HMM for CGI detection might include 8 separate possible states at each position i:

- in CGI and emitting an A
- in CGI and emitting a C
- in CGI and emitting a G
- in CGI and emitting a T
- not in CGI and emitting an A
- not in CGI and emitting a C
- not in CGI and emitting a G
- not in CGI and emitting a T

You can see how quickly such models become quite complex!

Before applying an HMM to `dinucBIsCG`, I do want to point out one reason why the more complex model I just laid out would be a better approach to the CGI problem: Let's say the base b_i at position i is a T. Then the chance that the dinucleotide consisting of bases b_i followed by b_{i+1} is a CG must be exactly 0, since a CG must have $b_i = \text{C}$! Using the `dinucBIsCG` approach, our model doesn't have any way of knowing when the previous dinucleotide ended with a C—in which case CG is a possibility for the next dinucleotide—or when it ended up with an A, G, or T, meaning that CG is not a possible value for the next dinucleotide.

Despite this rather problematic obscured interdependence of the successive values of `dinucBIsCG`, let's go ahead and build the model using this vector as input anyway to get an introduction to how to use HMMs in R. We'll use the package `HMM` to

- **fit** the HMM using the *Baum-Welch algorithm* and then
- **determine most likely state trajectory**—that is, decide which positions we think are in CGI vs. not in CGI—using the *Viterbi algorithm*.

We're not going to go into the details of the Baum-Welch algorithm, which is a type of algorithm known as an "Expectation-Maximization" (EM) aglorithm, here, but you will get a chance to dig into the basic idea underlying the Viterbi algorithm in the homework. For now, let's see how you can apply these using `HMM` in R. Fitting an HMM is done using an iterative process whereby an initial guess at what the parameters—which, for an HMM, consist primarily of:

- the (# states)x(# states) matrix of **transition probabilities** for going from state x_i ("CGI" or "not-CGI") at i to state x_{i+1} at position $i+1$, and
- the (# states)x(# symbols) matrix of **emission probabilties** containing the chances of emitting symbol y_i ("-" or "cg") given the value of the state x_i (in CGI or not).

We'll use the function `initHMM` to set things up:

```
>  ## install.packages("HMM")  ## uncomment and run if necessary
> library(HMM)
>  ## first we'll need to set up an initial guess at the parameters
>  ## of the model, which will then be tuned by the Baum-Welch fitting
>  ## procedure.
> cgiHmmGuess = initHMM(
      States = c("not-CGI", "CGI"),
      Symbols = c("-", "cg"),
       ## initialize probability of transitioning from not-CGI to CGI
       ## (or vice versa) at 2.5% at each step from i to i+1
       ## (meaning 97.5% chance of staying in current state):
      transProbs = matrix(c(0.975, 0.025,
                            0.025, 0.975),
                          nrow=2, ncol=2, byrow=TRUE),
       ## initialize probability of emitting a "cg" at 2% (so 98% chance of "-")
       ## if in not-CGI state; make it higher at 10% for "cg" (so 90% for "-")
       ## if in CGI state at position i:
      emissionProbs = matrix(c(0.98, 0.02,
                               0.90, 0.10),
                             nrow=2, ncol=2, byrow=TRUE)
  )
```

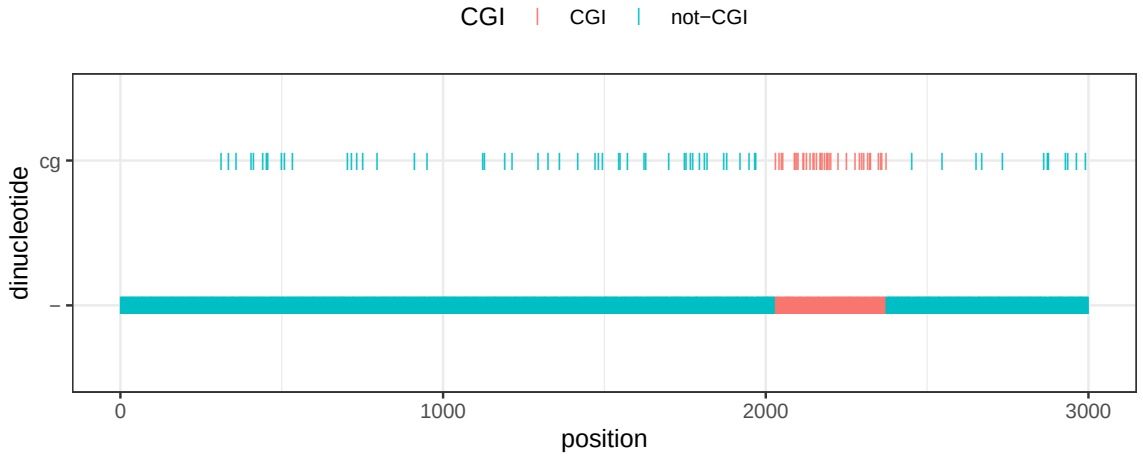
Once we've initialized our HMM, we can fit the parameters using `baumWelch` and then examine the results:

```
>  ## use Baum-Welch algorithm to better fit HMM parameters
>  ## (transProbs and emissionProbs) to sequence data:
> cgiHmmBaumWelchFit = baumWelch(hmm = cgiHmmGuess,
                                 observation = dinucBIsCG)
>  ## let's look at fit parameters
>  ## (they live inside of $hmm slot of cgiHmmBaumWelchFit object):
> cgiHmmBaumWelchFit$hmm$transProbs
         to
from           not-CGI          CGI
  not-CGI 0.999463157 0.0005368431
  CGI     0.003645376 0.9963546236
> cgiHmmBaumWelchFit$hmm$emissionProbs
         symbols
states            -         cg
  not-CGI 0.9772431 0.02275686
  CGI     0.9052820 0.09471796
```

Finally, we can now use the trained `cgiHmmBaumWelchFit$hmm` algorithm to classify each position i as being within a CGI or not using `viterbi`:

```
> mostLikelyStatePath = viterbi(hmm = cgiHmmBaumWelchFit$hmm,
                                observation = dinucBIsCG)
>  ## how many positions assigned to CGI vs. not-CGI?
> table(mostLikelyStatePath)
```

```
mostLikelyStatePath
    CGI not-CGI
    344    2655
>  ## let's plot the CG dinucleotides and color them by their
>  ## CGI-or-not assignments to visualize these results:
> library(ggplot2)
> theme_set(theme_bw())
> ggdata = data.frame(
      position = 1:length(mostLikelyStatePath),
      dinucleotide = factor(dinucBIsCG),
      CGI = factor(mostLikelyStatePath)
  )
> gg = ggplot(ggdata, aes(x=position, y=dinucleotide, color=CGI))
> gg = gg + geom_point(shape="|", size=2.5)
> gg = gg + theme(legend.position="top")
> print(gg)
```



From the plot you can see that there is only one predicted CGI in `humanDna$B`...let's get the coordinates of this CGI:

```
> min(which(mostLikelyStatePath == "CGI"))
[1] 2030
> max(which(mostLikelyStatePath == "CGI"))
[1] 2373
```

Remember that these are coordinates for *di*nucleotides: thus, the detected CGI starts with the dinucleotide corresponding to nucleotides 2030 and 2031 and ends with the dinucleotide spanning positions 2373 and 2374, so in terms of nucleotide coordinates, it runs from nucleotide 2030 through nucleotide 2374.

We can compare this (over)simplified-HMM-predicted CGI with the results of more sophisticated HMM methods applied to the whole human genome. This requires first figuring out what the coordinates we calculated relative to sequence humanDna$B read in from two_human_dna_sequences.fa map to in the current human genome reference hg38. I mentioned earlier that humanDna$B corresponds to the 3 kilobases immediately upstream of the the gene TP53; if we consult the Ensembl database entry for TP53 (which has **Ens**embl **g**ene id **ENSG**00000141510) at:

https://m.ensembl.org/Homo_sapiens/Gene/Summary?db=core;g=ENSG00000141510

we find that TP53 is on chromosome 17 on the reverse strand—so larger coordinates with respect to location in humanDna$B correspond to smaller coordinates in hg38—running from position 7,661,779 to 7,687,550. Since TP53 is in reverse orientation—relative to the arbitrary choice made in establishing a coordinate system for chromosome 17—for humanDna$B to be "upstream" of TP53 means that it corresponds to positions 7,687,551 through 7,690,550.

Thus the left boundary of our CGI—position 2,030 in humanDna$B—corresponds to position (7,690,550+1-2,030)=**7,688,521** in hg38, while the right boundary at 2,374 of our CGI in humanDna$B corresponds to position (7,690,550+1-2,374)=**7,688,177**.

Loading up a table of annotated CGIs for the human genome extracted from the built-in genomic annotations in the R package annotatr (Cavalcante & Sartor (2017)) from a data file annotatr_hg38_cpgs.tsv.gz:

```
> hg38cpgs = read.table("annotatr_hg38_cpgs.tsv.gz",
                        sep="\t", header=TRUE, row.names=NULL)
```

and looking at rows 20,420-20,424 of the file (columns 1-4):

```
> hg38cpgs[20420:20424, 1:4]
      seqnames   start     end width
20420    chr17 7650821 7652020  1200
20421    chr17 7685972 7686185   214
20422    chr17 7688176 7688521   346
20423    chr17 7704857 7705638   782
20424    chr17 7716934 7717801   868
```

we see (row 20,422, third from top shown) that there is a CGI annotated on chromosome 17 running from base 7,688,176 to 7,688,521—almost exactly the interval we detected using this stripped-down HMM!—and no other CGIs in the range from 7,687,551 through 7,690,550.

3.7.1 Viterbi algorithm

I named the variable in which I collected the viterbi output mostLikelyStatePath because the Viterbi algorithm is designed to find the trajectory of states x_i, for i running from 1 to the final position n in the sequence, such that the conditional probability

$$\mathbb{P}\left(X_1 = x_1, \ldots, X_n = x_n \mid Y_1 = y_1, \ldots, Y_n = y_n\right) \tag{3.6}$$

where y_i is the observed symbol at position i, is higher than it would be for any other choices of all of the $\{x_i\}$ values.

(An aside on notation: the "oberved symbol" y_i plays a similar role to what I had previously called b_i, for i^{th} base, in section 3.3 above. I'm switching notation here both (1) because I've swiched symbols from bases to `"-"` vs. `"cg"` and (2) to emphasize that, unlike the simpler versions of Markov models without hidden state, HMMs must keep separate track of states x_i and symbols y_i.)

Given an HMM, we can calculate the likelihood of any particular path in R as follows:

```
> pathLikelihood = function(hmm, stateTraj, symTraj) {
      n = length(stateTraj)
       ## initialize likelihood (lhd) to appropriate startProbs value:
      lhd = hmm$startProbs[[ stateTraj[[1]] ]]
       ## now loop through trajectory, multiplying by...
      for (i in 1:(n-1)) {
           ## ...appropriate emissionProbs value for emitting
           ## - symbol symTraj[[i]] given system in
           ## - state stateTraj[[i]]...
          lhd = lhd * hmm$emissionProbs[ stateTraj[[i]], symTraj[[i]] ]
           ## ...and then by appropriate transProbs value for changing
           ## - from the state stateTraj[[i]]
           ## - to the state stateTraj[[i+1]]:
          lhd = lhd * hmm$transProbs[ stateTraj[[i]], stateTraj[[i+1]] ]
      }
       ## Finally multiply by probability of emitting symbol symTraj[[n]]
       ## - given ending in state stateTraj[[n]]:
      lhd = lhd * hmm$emissionProbs[ stateTraj[[n]], symTraj[[n]] ]
      return(lhd)
  }
```

Let's try it out on `mostLikelyStatePath`:

```
> pathLikelihood(cgiHmmBaumWelchFit$hmm, mostLikelyStatePath, dinucBIsCG)
[1] 1.420387e-182
```

Yikes, that's a very small number! Still, if we randomly flip any particular state like so:

```
> notAsLikelyStatePath = mostLikelyStatePath
> notAsLikelyStatePath[[100]] = "CGI"
> pathLikelihood(cgiHmmBaumWelchFit$hmm, notAsLikelyStatePath, dinucBIsCG)
[1] 2.577769e-188
```

We get an even smaller number. Still, because R—like most computer languages—does not allocate all that many bits to keeping track of the exponent of floating point numbers, it's better to work with logarithms of real numbers which are expected to be of very large or very small magnitude. Noting that the trajectory which maximizes the likelihood will also necessarily be the trajectory which maximizes the log-likelihood (since log is a strictly increasing function), let's work instead with:

```
> pathLogLikelihood = function(hmm, stateTraj, symTraj) {
      n = length(stateTraj)
      logLhd = log(hmm$startProbs[[ stateTraj[[1]] ]])
      for (i in 1:(n-1)) {
           ## note: log(x * y) = log(x) + log(y), so multiplications done
           ## in pathLikelihood become additions here:
          logLhd = logLhd +
                   log(hmm$emissionProbs[ stateTraj[[i]], symTraj[[i]] ])
          logLhd = logLhd +
                   log(hmm$transProbs[ stateTraj[[i]], stateTraj[[i+1]] ])
      }
      logLhd = logLhd +
               log(hmm$emissionProbs[ stateTraj[[n]], symTraj[[n]] ])
      return(logLhd)
  }
```

Let's just verify that it works:

```
> pathLogLikelihood(cgiHmmBaumWelchFit$hmm, mostLikelyStatePath, dinucBIsCG)
[1] -418.7196
> exp(pathLogLikelihood(cgiHmmBaumWelchFit$hmm, mostLikelyStatePath, dinucBIsCG))
[1] 1.420387e-182
```

which is the same value we saw above using `pathLikelihood`.

So...given that there are (for this example) 2^n possible state trajectories to consider, how does the Viterbi algorithm figure out what one trajectory yields the maximum (log-)likelihood? The crucial idea it exploits is that the most likely trajectory must start with either

- the most likely path through the first $n-1$ states with $x_{n-1} =$ not-CGI, *or*
- the most likely path through the first $n-1$ states with $x_{n-1} =$ CGI,

followed by one of $x_n =$ not-CGI or $x_n =$ CGI. This is true because the `pathLogLikelihood` of the full trajectory consists of:

1. the `pathLogLikelihood` for the first $n-1$ steps plus
2. `log(hmm$transProbs[ stateTraj[[n-1]], stateTraj[[n]] ])` plus
3. `log(hmm$emissionProbs[ stateTraj[[n]], symTraj[[n]] ])`,

where terms 2 and 3 do not depend on any of the earlier states `stateTraj[[i]]` for $i < (n-1)$. Hence a higher (log-)likelihood trajectory for the first $n-1$ steps ending at `stateTraj[[n-1]]` would have to yield a higher (log-)likelihood trajectory for the full n steps when the final state `stateTraj[[n]]` was added to the end.

To verify the decomposition into terms 1-3 above:

```
> n = length(mostLikelyStatePath)
> hmm = cgiHmmBaumWelchFit$hmm
> stateTraj = mostLikelyStatePath
> symTraj = dinucBIsCG
> pathLogLikelihood(hmm, stateTraj[1:(n-1)], symTraj[1:(n-1)]) +
      log(hmm$transProbs[ stateTraj[[n-1]], stateTraj[[n]] ]) +
      log(hmm$emissionProbs[ stateTraj[[n]], symTraj[[n]] ])
[1] -418.7196
```

As advertised, this is the same as `pathLogLikelihood(hmm, stateTraj, symTraj)`.

The Viterbi algorithm then only needs to consider the best (highest likelihood) trajectories ending in each possible state at each of the n positions i—this is many fewer than all 2^n total state trajectories! Specifically, the algorithm consists of two phases:

forward pass iterating through i from 2 up to n, keeping track of the (log-)likelihood `v[state, i]` (`v` is the name of this array in the function `viterbi` from `HMM`) of the best path from 1 up to i ending in each possible state `state`, followed by

reverse pass iterating from $i = n$ down to 1, starting from `mostLikelyStatePath[[n]]` set to the end state producing the most likely overall trajectory, then successively setting `mostLikelyStatePath[[i]]` to whichever value `state` minimizes `v[state, i] + log(hmm$transProbs[ state, mostLikelyStatePath[[i+1]] ])`

3.7.2 Other uses of HMMs

Hidden Markov models are routinely used in many areas of computational biology (Yoon (2009)), as well as many other fields. They can be applied to modeling sequence motifs (Wu & Xie (2010)), for which they are capable of somewhat more flexibility than the simple PWM approach we introduced above—at the expense of additional complexity, of course! They can also be used in constructing *multiple sequence alignments* (Durbin *et al.* (1998)), which we unfortunately will not have time to study in any depth in this course, but which are extremely useful in many areas of computational biology and bioinformatics. The software package `HMMER` (`http://hmmer.org/`) includes implementations of algorithms for many different biosequence analyses one might be interested in performing, and is highly recommended!

3.8 Pairwise alignment

3.8.1 Edit distance

One way to think about sequence alignment is as an editing problem: How do we turn one sequence v into another w by a sequence of edit substitution, insertion, or deletion operations? More specifically,

- how do we do this with the *fewest possible* such operations,
- how many such operations is the fewest number possible,
- and how can we unambiguously specify what those operations are.

To make this a bit more concrete let's take as example $v = \text{TACGATT}$ and $w = \text{TACATTA}$. If we just stack these two sequences next to each other:

```
TACGATT
|||||||
TACATTA
```

We can immediately see that one possible sequence of edits is to make the three suggested substitutions G $\to$ A, A $\to$ T, and T $\to$ A. But we can do better, with one deletion combined with one insertion:

```
TACGATT-
|||||||
TAC-ATTA
```

The light gray lines connecting the matched letters suggest that we might consider enlarging the set of operations we use to describe the transformation of v into w to include *matching* two characters as well. One way of describing this is to count a match as a type of substitution which "changes" a character into the same character, but to say that unlike a non-identical substitution, an insertion, or a deletion, such a no-change "substitution" does not increase the count of edits made.

The advantage of this accounting trick is that it gives us a way to exactly and uniquely describe a sequence alignment using a compact notation known as a "cigar string." Given any alignment depicted in a diagram like the two above, label:

M each line with a sequence character both above and below—that is, corresponding to either a **M**atch (gray vertical line) or a **M**ismatch (red vertical line)—with an M;

D each line with a valid sequence character above the vertical line and no sequence character, i.e., a `"-"`, below the vertical line—a **D**eletion of the indicated character—with a D; and

I each line with a `"-"` above the vertical line and a valid sequence character below the vertical line—representing an **I**nsertion of that sequence character—with an I.

Then the first alignment depicted above, with edit distance 3, is assigned a cigar string of MMMMMMM, or, in even more compact notation, **7M**. Note that the cigar string, while uniquely specifying the alignment itself—as long as you know what the two sequences v and w being aligned are!—does *not* give you a way to immediately quantify the edit distance, since it does not differentiate between matches and mismatches!

The second alignment above, with edit distance 2, is assigned a cigar string MMMDMMMI, or in the standard compact version, **3M1D3M1I**.

The classic algorithm for finding the optimal alignment between two strings—in the sense of minimal edit distance—is a type of *dynamic programming* algorithm which works by building up a matrix of solutions to related sub-problems:

- Let's represent the strings v and w each as a vector of individual characters (e.g., `v=c("T","A","C","G","A","T","T")`), and
- define `i=length(v)+1` and `j=length(w)+1`.
- We want to build up a matrix `cost` containing the minimal edit distances `cost[m,n]` ($m \leq i$ and $n \leq j$) for converting `v[1:(m-1)]` into `w[1:(n-1)]`.
  - in particular, `cost[i-1,j-1]` is minimal edit distance of best alignment converting all-but-last-character of v into all-but-last-character of w, while
  - `cost[i-1,j]` is minimal edit distance for converting all-but-last-character of v into whole string w, and
  - `cost[i,j-1]` is minimal edit distance for conversion of whole string v into all-but-last-character of w.

The key insight of the algorithm is to partition the possible alignments converting v to w according to *final* step, which can then be computed via:

```
>  ## arguments to bestFinalStep:
>  ## - v is a vector of the single characters to align
>  ## - w is a vector of the single characters to be aligned to
>  ## - cost is matrix of minimal edit distances; must *already* include:
>  ##   - cost[length(v), length(w)]
>  ##   - cost[length(v), length(w)+1]
>  ##   - cost[length(v)+1, length(w)]
>  ## output from bestFinalStep will be used to define
>  ## cost[length(v)+1, length(w)+1]
> bestFinalStep = function(v, w, cost) {
       ## is the last character of v different from that of w?
      finalPositionMismatched = (v[[length(v)]] != w[[length(w)]])
       ## need to offset i and j to deal with boundary conditions
       ## for no aligned characters: e.g. cost[1, 2] is the edit
       ## distance for converting empty string into w[[1]]
      i = length(v) + 1
      j = length(w) + 1
      costOptions = c(
           ## convert v[1:(length(v)-1)] into w[1:(length(w)-1)]
           ## with edit distance cost[i-1, j-1] and then either
           ## match or mismatch last character according to
           ## finalPositionMismatched:
          M = cost[i-1, j-1] + finalPositionMismatched,
           ## convert v[1:(length(v)-1)] into w and then
           ## delete last character of v:
          D = cost[i-1, j] + 1,   ## deletion always adds to cost
           ## convert v into w[1:(length(w)-1)] and then
           ## insert final character of w:
          I = cost[i, j-1] + 1    ## insertion always adds to cost
      )
       ## which of M, D, or I is the best final step/edit?
      bestEdit = names(which.min(costOptions))
      return(list(            ## return both:
          edit = bestEdit,    ## and:
          cost = costOptions[[bestEdit]]
      ))
  }
```

As alluded to in the comments preceding the definition of `bestFinalStep`, the dynamic programming algorithm for computing the optimal alignment of v and w proceeds by building the matrix `cost` of minimal edit distances for prefixes of v and w:

- using `bestFinalStep` to compute `cost[i, j]`
- based on having previously filled in:
  - `cost[i-1, j-1]`,
  - `cost[i-1, j]`, and
  - `cost[i, j-1]`.

Here is a function to do just this:

```
> buildCostMatrix = function(v, w) {
       ## pre-allocate matrix to hold cost matrix, initialize with 0 values:
      cost = matrix(0, nrow=length(v)+1, ncol=length(w)+1)
       ## pre-allocate character matrix to keep track of
       ## best-prior-edit priorEdit[m+1, n+1] (will be "M", "D" or "I")
       ## for converting v[1:m] into w[1:n]:
      priorEdit = matrix("", nrow=length(v)+1, ncol=length(w)+1)
       ## first column of priorEdit corresponds to successively deleting
       ## the beginning characters of v:
      priorEdit[2:nrow(priorEdit), 1] = "D"
      cost[ , 1] = 0:length(v)  ## column 1: successive deletions
       ## first row of priorEdit corresopnds to successively inserting
       ## the beginning characters of w:
      priorEdit[1, 2:ncol(priorEdit)] = "I"
      cost[1, ] = 0:length(w)   ## row 1: successive insertions
       ## now ready to use bestFinalStep to fill in the rest of
       ## priorEdit and cost:
      for (i in 2:nrow(cost)) {
          for (j in 2:ncol(cost)) {
              priorStep = bestFinalStep(v[1:(i-1)], w[1:(j-1)], cost)
              cost[i, j] = priorStep[["cost"]]
              priorEdit[i, j] = priorStep[["edit"]]
          }
      }
      return(list(
          cost = cost,
          priorEdit = priorEdit
      ))
  }
```

Let's try it out:

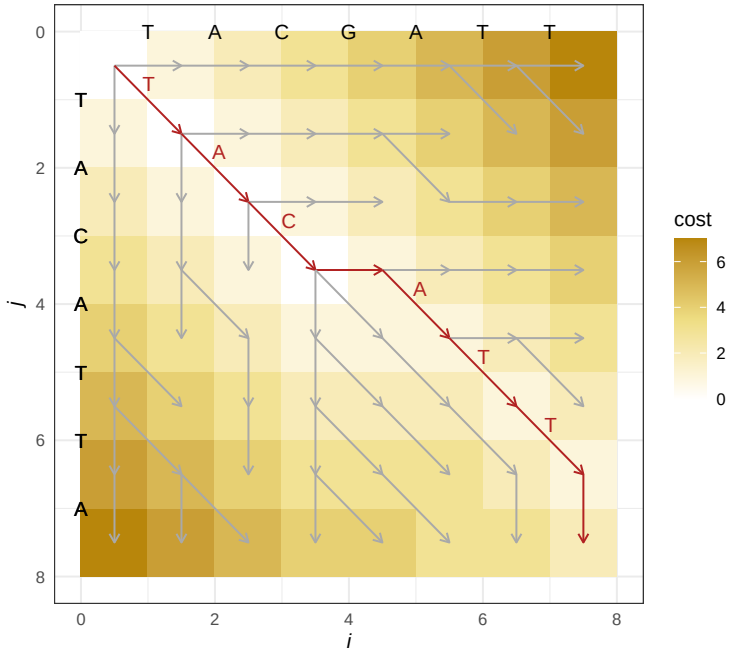
```
> vString = "TACGATT"
> wString = "TACATTA"
>  ## split vString into vector of single characters:
> vVector = strsplit(vString, split="")[[1]]
> vVector
[1] "T" "A" "C" "G" "A" "T" "T"
>  ## and now same for w:
> wVector = strsplit(wString, split="")[[1]]
> wVector
[1] "T" "A" "C" "A" "T" "T" "A"
> alignment = buildCostMatrix(vVector, wVector)
> alignment
$cost
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
[1,]    0    1    2    3    4    5    6    7
[2,]    1    0    1    2    3    4    5    6
[3,]    2    1    0    1    2    3    4    5
[4,]    3    2    1    0    1    2    3    4
[5,]    4    3    2    1    1    2    3    4
[6,]    5    4    3    2    1    2    3    3
[7,]    6    5    4    3    2    1    2    3
[8,]    7    6    5    4    3    2    1    2

$priorEdit
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
[1,] ""   "I"  "I"  "I"  "I"  "I"  "I"  "I"
[2,] "D"  "M"  "I"  "I"  "I"  "M"  "M"  "I"
[3,] "D"  "D"  "M"  "I"  "M"  "I"  "I"  "M"
[4,] "D"  "D"  "D"  "M"  "I"  "I"  "I"  "I"
[5,] "D"  "D"  "D"  "D"  "M"  "M"  "M"  "M"
[6,] "D"  "D"  "M"  "D"  "M"  "M"  "M"  "M"
[7,] "D"  "M"  "D"  "D"  "D"  "M"  "M"  "I"
[8,] "D"  "M"  "D"  "D"  "D"  "M"  "M"  "I"
```

While this captures all of the information necessary to specify the optimal alignment, we'd like to distill this into the nice compact cigar string representation we introduced above. This is conceptually straightforward—perhaps I should say "straightbackward," since it proceeds in reverse, first consulting `priorEdit[nrow(priorEdit), ncol(priorEdit)]` to identify the final step of the alignment and then tracing backwards through the matrix as indicated:

```
>  ## arguments to cigar both obtained from output of buildCostMatrix:
>  ## - cost is list element "cost" from buildCostMatrix output
>  ## - priorEdit is list element "priorEdit" from buildCostMatrix output
> cigar = function(cost, priorEdit) {
       ## start at the bottom right corner of cost and priorEdit matrices:
      i = nrow(cost)
      j = ncol(cost)
       ## build up vector of M, D, or I steps in reverse order:
      bestPathBack = character(0)
      while (i > 1 && j > 1) {
          if (priorEdit[i, j] == "M") {
              i = i-1
              j = j-1
              bestPathBack = c(bestPathBack, "M")
          } else if (priorEdit[i, j] == "D") {
              i = i-1
              bestPathBack = c(bestPathBack, "D")
          } else if (priorEdit[i, j] == "I") {
              j = j-1
              bestPathBack = c(bestPathBack, "I")
          }
      }
       ## use rev to reverse steps into forward order:
      forwardSteps = rev(bestPathBack)
      cigar = ""
       ## compactify notation:
      activeStep = forwardSteps[[1]]
      nConsecutiveSame = 1
      for (i in 2:length(forwardSteps)) {
          if (forwardSteps[[i]] != activeStep) {
               ## add representation of last nConsecutiveSame steps:
              cigar = paste0(cigar, nConsecutiveSame, activeStep)
               ## reset activeStep and nConsecutiveSame:
              activeStep = forwardSteps[[i]]
              nConsecutiveSame = 1
          } else {
               ## anoter consecutive step in run of type activeStep:
              nConsecutiveSame = nConsecutiveSame + 1
          }
      }
       ## need to add the final run of consecutive steps now:
      cigar = paste0(cigar, nConsecutiveSame, activeStep)
      return(cigar)
  }
> cigar(alignment$cost, alignment$priorEdit)
[1] "3M1D3M1I"
```

This confirms that the second of the two alignments we considered at the beginning of this chapter is indeed the optimal alignment—in the sense of having the minimal edit distance—between the sequences v = TACGATT and w = TACATTA.

It's always nice to have a visual representation of how a complicated algorithm—like the dynamic programming one we're studying here—works:



The sequence of red arrows in the plot visualizes the chosen alignment 3M1D3M1I, with:

- each diagonal arrow representing a **M**(is)?match operation,
- the horizonal arrow representing a **D**eletion, and
- the vertical arrow representing an **I**nsertion.

I've labeled the arrows for the matching operations with the corresponding matched character and I've also colored each cell `m, n` of the grid according to the minimal edit distance `alignment$cost[m, n]` required to get there from `0, 0`—that is, the minimal edit distance to convert `v[1:(m-1)]` into `w[1:(n-1)]`.

The gray arrows show the optimal alignments for various substrings of v and w which do not appear as part of the optimal edit sequence for the full conversion of v into w.

3.8.2 Local alignment

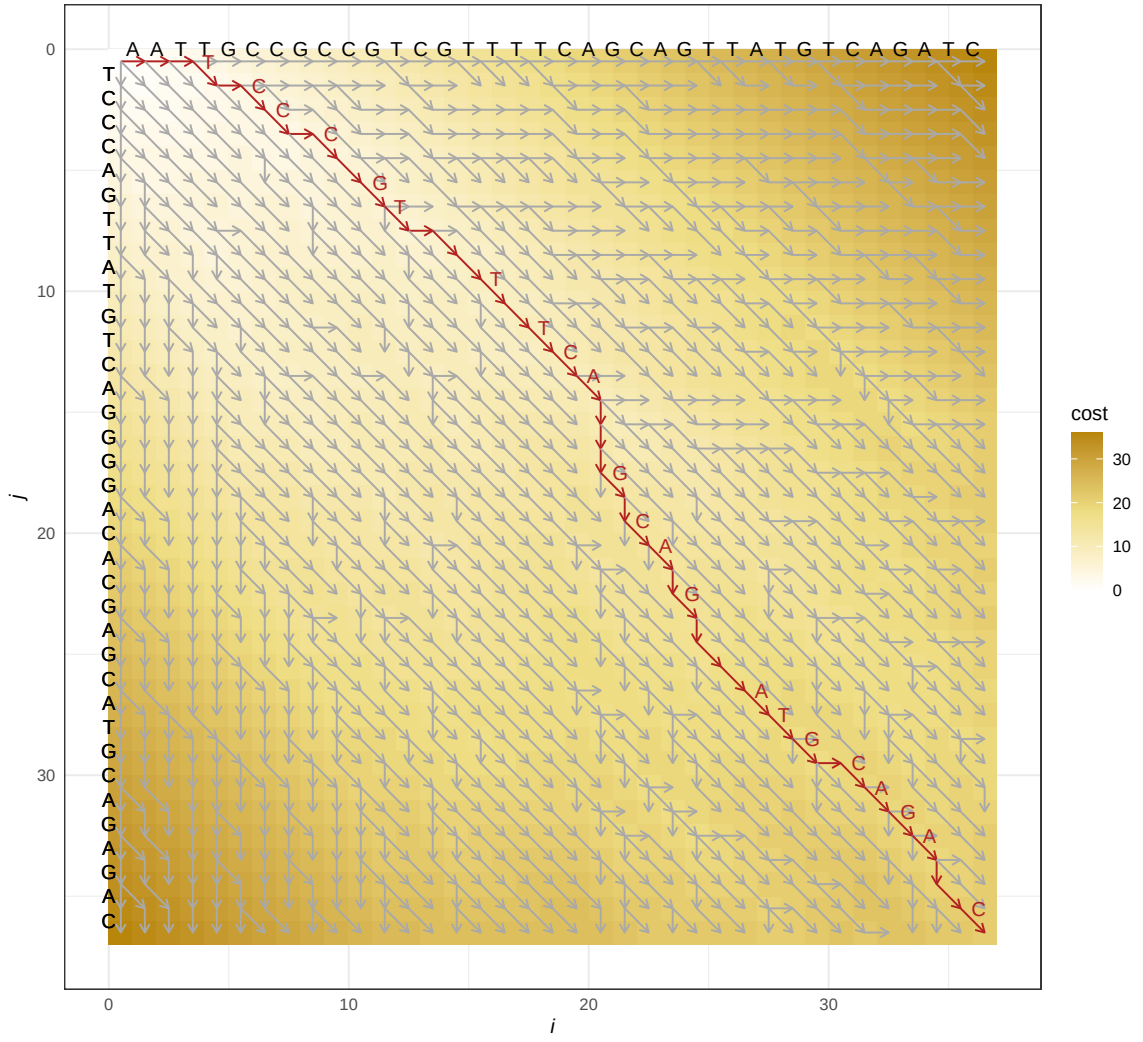
The dynamic programming edit distance algorithm presented above assumes that you want to find the optimal *global* alignment converting all of v into all of w. In many cases we are more interested in *local* alignments, in which some substring of v is converted into some substring of w. Consider the following example: Align

$$v = \text{AATTGCCGCCGTCGTTTTCAG}\mathbf{CAGTTATGTCAG}\text{ATC}$$

to

$$w = \text{TCC}\mathbf{CAGTTATGTCAG}\text{GGGACACGAGCATGCAGAGAC}$$

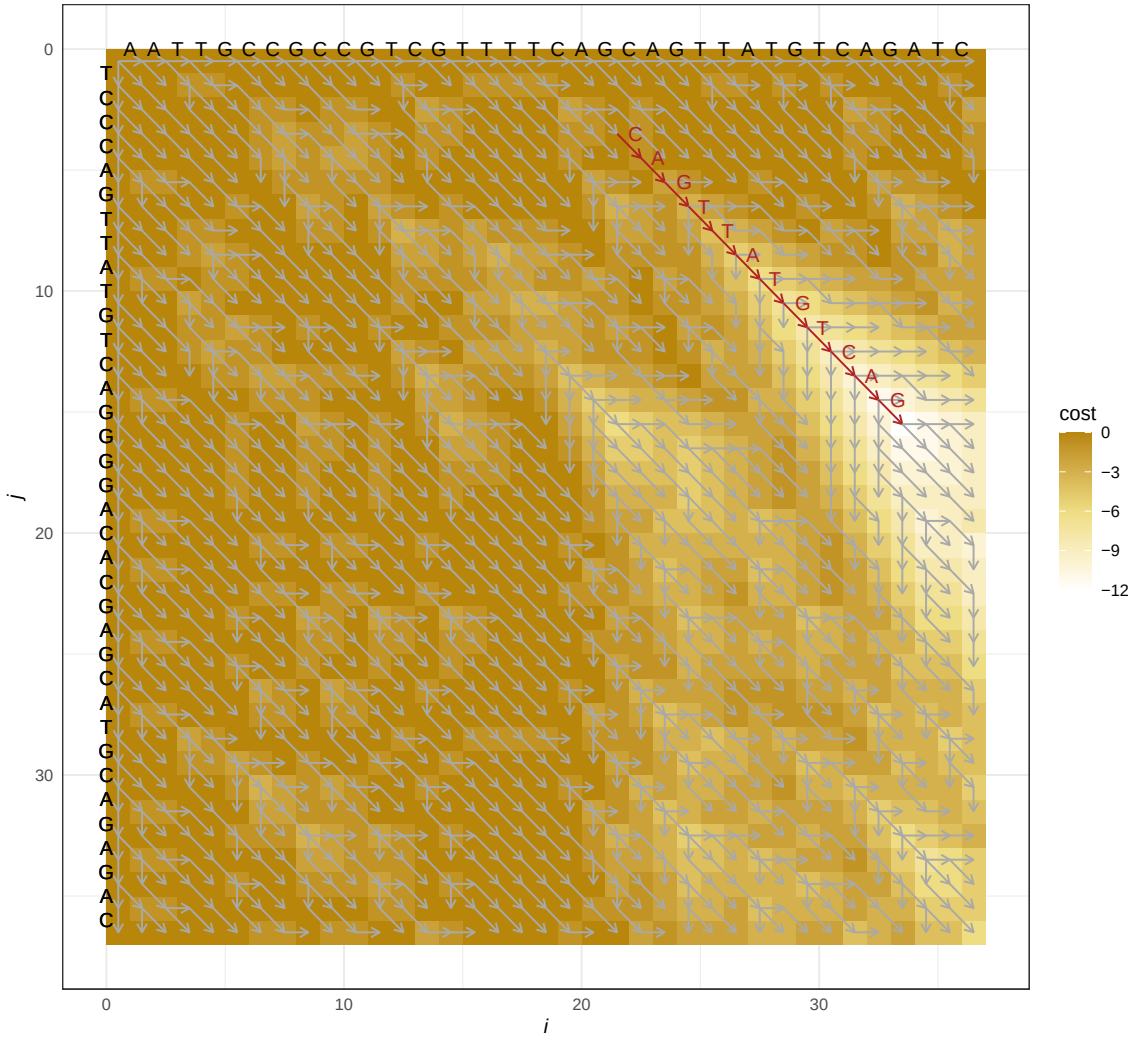
Note that both v and w as defined above contain the common substring CAGTTATGTCAG, though otherwise they don't look very similar. In this case, however, applying the global edit distance alignment algorithm to this problem results in the alignment 3D1M1D2M1D4M1D7M3I1M1I2M1I1M1I5M1D4M1I2M:



With a few slight modifications to the algorithm described above it can be used to construct a local alignment instead of a global alignment. I won't go through the full details here, but the essential changes include:

- not only penalizing mismatches, but actively rewarding matches
  - say, giving them "cost" of -1 as opposed to +1 for mismatch,
  - though usually the signs are flipped score is maximized instead of minimizing cost in this case;
- removing cost associated with deletions or insertions at ends of v or w;
- setting any costs > 0 (or scores < 0 if signs flipped) to 0; and
- score tracking starting from lowest cost (or highest score if signs flipped) cell instead of from the bottom right-most cell.

With these modifications (but keeping the language of minimizing costs instead of maximizing scores), we instead find a local alignment described by cigar string 21D3I12M3D21I:



It's worth reiterating that the local alignment 21D3I12M3D21I does still have a much higher edit distance (21+3+3+21=48) than does the global alignment (edit distance of 21)! It's only a better solution if you pose a different (and more complicated) problem.

There are many other variations which may be applied to dynamic programming algorithms:

- Some mismatches may be penalized more than others
  - more commonly applied in aligning peptides or proteins: some amino acid residue substitutions retain important chemical properties (acidity, etc.)—and thus shouldn't "cost" very much—while other change them greatly, and thus should be "expensive."
- More useful alignments may be obtained by applying a larger penalty for the first insertion or deletion ("gap opening") than the following ones in a consecutive run ("gap extension").

3.8.3 Name dropping

Local alignment via dynamic programming (including the two modifications described immediately above) is often referred to as the *Smith-Waterman* algorithm after the authors of the paper which introduced it (Smith & Waterman (1981)).

Similarly, the global version of the dynamic programming algorithm for pairwise alignment is often referred to as the *Needleman-Wunsch* algorithm (Needleman & Wunsch (1970)). Note that it only computes edit distance when the specific scoring system employed above in section 3.8.2 is used. In particular, when the modifications for non-uniform mismatch penalties or added gap opening penalty are incorporated, Needleman-Wunsch alignments will not minimize the edit distance between the two strings.

Edit distance itself is also referred to as *Levenshtein distance*, after an earlier article (Levenshtein (1966)) making use of the concept.

3.8.4 Basic Local Alignment Search Tool: BLAST

Dynamic programming algorithms have the desirable property that they are guaranteed to find an optimal solution to specific formulations of the alignment problem, but even though they are surprisingly efficient, they are still too slow for large-scale alignment problems.

Thus, when faced with a problem like figuring out what organism some sequence detected in an experiment (we'll call it a "query" sequence) represents when we don't know what the source was, we don't turn to dynamic programming. Instead we use so-called *heuristic* methods, which do not necessarily find the optimal solution but often do provide reasonably good alignments much more quickly.

The most famous such heuristic alignment method is BLAST (Altschul *et al.* (1990)), which stands for "Basic Linear Alignment Search Tool" (though it isn't really "basic" in any meaningful sense). BLAST is really a collection of related heuristic methods for doing various different types of biological sequence alignments. The BLAST family of methods generally start by identification of short subsequences ("words") of the query sequence: Occurrences of these words within the reference sequence database (which may contain many sequences from many different species) then nucleate potential local alignments. The algorithms then attempt to extend the nucleated seed matches in both directions, throwing away those which cannot be satisfactorily extended according to a specific scoring scheme and finally reporting the remaining maximally extended alignments.

Chapter 4

High-Dimensional Data Analysis

4.1 Neves data set

In order to have specific RNA sequencing (RNA-seq) data to consider in our discussion of differential expression analysis, we'll use the data from a study by Neves et al. (publicly available at Gene Expression Omnibus, accession GSE120430), which aimed to analyze transcriptional targets of core promoter factors in Drosophila neural stem cells (Neves & Eisenman (2019)).

This study contains 12 samples divided into 4 distinct groups labelled by what RNA interference (RNAi) transgene—or gene artificially introduced by the investigators encoding an RNA molecule capable of neutralizing the mRNA for the targeted gene—they expressed:

mCherry the control group;

TAF9 experimental group with RNAi transgene reducing expression of the gene TAF9;

TBP experimental group in which RNAi transgene targets TBP; and

TRF2 experimental group in which RNAi targets TRF2.

The three genes TAF9, TBP and TRF2 targeted for reduced expression in the three experimental groups here were found in this study to be necessary for maintaining neural stem cell identity: The RNA sequencing data we will be examining from GSE120430 is useful for understanding how reducing the expression of these core promoter factors impacts the expression of other genes composing the full transcriptome.

4.1.1 Short read sequence files: fastq

The sequences of the short reads as called by the sequencer will be found in so-called "fastq" files (usually given extension ".fastq" or ".fq", but often compressed by gzip so that ".fastq.gz" or ".fq.gz" are most common). When not compressed, these files are plain text files which generally describe each sequencing read using a four-line block; here's an example:

```
@SRR7900135.1236837 1236837 length=50
CTCCACATCAGTAAATTGTGATATATAAAAATAATCAAACTACATCGACA
+
G.GA..G<AGIIIAAGGA<<GAAAAGGAA.GGGGGI.G.<<GAG..<<AA
```

Let's take this line by line:

1. Must start with an `@` sign immediately followed by the read id (here, `SRR7900135.1236837`).
    - Optionally, additional information (such as `1236837 length=50`) can then be provided after a space, but this is not necessary.
2. The sequence.
3. Must start with a `+` sign (sometimes the read identifier is repeated after).
4. The quality scores q_i for the base calls b_i in the sequence:
    - will be same number of characters as sequence in line 2;
    - quality score q_i: estimated probability base call b_i is incorrect is $p = 10^{-0.1q_i}$
        - e.g., $q_i = 20 \implies p = 10^{-2} = 0.01$, so estimated 1% chance base i is not really b_i but some other nucleotide.
    - Quality scores encoded using single character by using ASCII table to encode integers as characters
        - unfortunately different conventions have been used here,
        - though *usually* ASCII character corresponding to decimal number $q_i + 33$ in ASCII table is used to encode quality score q_i.

4.1.2 Short read alignment files: SAM and BAM

The alignments of the short reads can be found in "BAM" files (which generally are given a ".bam" extension): these are **b**inary versions of SAM files, where SAM stands for **s**equence **a**lignment/**m**ap format.

SAM formatted files are plain text files consisting of:

- an optional *header* section, in which lines must start with an @ sign, followed by
- the main *alignment* section, in which lines are tab-delimited with 11 mandatory fields (with a specified order) which may be followed by additional optional fields giving more information.
  - each line of the alignment section corresponds to one *alignment*
    - can be multiple lines for same short read if it aligns well to multiple locations in reference sequence
  - **1**st field contains **short read id**
  - **3**rd field contains **reference sequence name** (chromosome in our example) short read aligned well to
  - **4**th field contains (1-based) left-most **position** of alignment in reference sequence
  - **6**th field contains **cigar** string describing pairwise alignment

The list above obviously does not tell you what every one of the 11 mandatory fields represent, for that you should consult:

`https://samtools.github.io/hts-specs/SAMv1.pdf`

Here is an example of *single* SAM alignment line—it's long, so it wraps when displayed in a fixed-width environment—for the short read described the fastq lines shown in section 4.1.1 above:

```
SRR7900135.1236837    83    mitochondrion_genome    5470    20    50M
=    5291    -229    TGTCGATGTAGTTTGATTATTTTTATATATCACAATTTACTGATGTGGAG
AA<<..GAG<<.G.IGGGGG.AAGGAAAAG<<AGGAAIIIGA<G..AG.G    HI:i:1    NH:i:1
NM:i:1
```

Picking out the 1st, 3rd, 4th, and 6th fields, we see that this is an alignment for

- short read `SRR7900135.1236837`
- to the `mitochondrion_genome` reference sequence
- starting at the base numbered `5470`, and that
- the alignment is described by the (rather boring) cigar string `50M`, indicating no indels.

If you've been looking at this closely, you may have noticed that fields 10 and 11 look like sequence and quality score fields. And if you've been looking *really* closely, you might have noticed that:

- the sequence in field 10 doesn't agree with the sequence we saw in the fastq description of this read up in section 4.1.1,
- nor do the quality scores in field 11 seem to line up with the fastq quality scores for read `SRR7900135.1236837`!

But: try reading the quality scores in field 11:

```
AA<<..GAG<<.G.IGGGGG.AAGGAAAAG<<AGGAAIIIGA<G..AG.G
```

backward and comparing with the sequence from the fastq file:

```
G.GA..G<AGIIIAAGGA<<GAAAAGGAA.GGGGGI.G.<<GAG..<<AA
```

to get a sense of what's going on here. Read `SRR7900135.1236837` has been aligned in *reverse-complement* orientation to the `mitochondrion_genome` reference sequence, so field 10 of the bam record for this alignment shows the reverse-complement of short read sequence documented in the fastq file, while field 11 displays the quality scores in reversed order.

The fact this read has been aligned in reverse-complemented orientation (or, "aligned to the reverse (-) strand of the reference sequence," as might be more commonly indicated), along with several other useful bits of information about the alignment, are encoded in the number 83 appearing in the **2**nd field of the alignment record. This value in this field is known as the "SAM flag" for the alignment: Each bit in the binary expansion of the number tells you something specific about the alignment. This is convenient for a computer to read but not so much for a human; probably the easiest way for you to decode it is to navigate to

`https://broadinstitute.github.io/picard/explain-flags.html`

in your browser and type "83" into the "SAM Flag" box before clicking on the "Explain" button (or hitting enter). You should see that this `83` means that:

- the read is paired,
- is mapped (or aligned) in proper pair
  - that is: both reads in pair mapped to opposing strands of the same reference sequence within believable distance of each other,
- the read is mapped to the reverse strand,
- and is the first in the pair.

4.2 Differential expression analysis

4.2.1 Short-read alignment/mapping

Now that I've shown you how the alignments of short read sequences are stored in bam files, let's consider how they are determined in the first place. The most popular short read alignment algorithms currently in use—**bwa-mem2**, **bowtie2** (and its splice-aware extension **hisat2**), and **star**)—generally rely on suffix arrays and/or the Burrows-Wheeler transform (where the "b" and "w" in **bw**a and **bo**w**t**ie come from) to find an alignment "seed" match for part of the read, then switch over to various other algorithms—often some sort of dynamic programming algorithm—to either extend or stitch together seeds into full aligments. The aligned short read counts hosted by Gene Expression Omnibus for the Neves data set were generated based on alignments made using **tophat2**, an older extension of **bowtie2** predating hisat2.

Unfortunately for the purposes of this class, none of these algorithms are easy to set up and run within R. However, there is an alternative short read aligner which is easily available within R via the package `Rsubread`. Thus, to save time and energy on the extensive set-up which would be required using one of the above choices of aligner, we'll go ahead and use this package to get a sense of what's involved in constructing and interpreting short read alignments.

Just as we found in sections 3.6-3.6.1 that we can efficiently match any k-mer against a reference sequence if we first build an appropriate *index* data structure for the reference. While `Rsubread` relies on a different index structure than the FM index approach employed by many short read aligners, it still involves a two-step approach in which first a function `buildindex` generates the required index files:

```
> ## BiocManager::install("Rsubread")  ## uncomment and run if necessary
> library(Rsubread)
>  ## provide path to fasta file containing reference sequences:
> referenceFasta = "drosophila_melanogaster_genome.fa"
>  ## provide name to be used as basename for (multiple) files
>  ## which will be generated by buildindex:
> indexBasename = "drosophila_melanogaster_index"
> buildindex(indexBasename, referenceFasta)
```

and then a second function `align` aligns the short reads in your fastq files against the reference sequences:

```
> alignStats = align(
      index = indexBasename,
      readfile1 = "SRR7900135_downsampled_r1.fastq.gz",
      readfile2 = "SRR7900135_downsampled_r2.fastq.gz",
      output_file = "SRR7900135_downsampled.bam",
       ## if you have multiple CPUs available, you can tell align
       ## to use multiple threads to in order to align reads
       ## in parallel; uncomment line below and set to desired number:
      ## nthreads = 4,
      type = "rna"         ## we are aligning RNA-seq data
  )
```

```
        ==========     _____ _    _ ____  _____  ______          _____
        =====         / ____| |  | |  _ \|  __ \|  ____|   /\   |  __ \
          =====      | (___ | |  | | |_) | |__) | |__     /  \  | |  | |
            ====      \___ \| |  | |  _ <|  _  /|  __|   / /\ \ | |  | |
              ====    ____) | |__| | |_) | | \ \| |____ / ____ \| |__| |
        ==========   |_____/ \____/|____/|_|  \_\______/_/    \_\_____/
       Rsubread 2.2.6

//================================== setting ==================================\\
||                                                                            ||
|| Function      : Read alignment (RNA-Seq)                                   ||
|| Input file 1  : SRR7900135_downsampled_r1.fastq.gz                         ||
|| Input file 2  : SRR7900135_downsampled_r2.fastq.gz                         ||
|| Output file   : SRR7900135_downsampled.bam (BAM)                           ||
|| Index name    : drosophila_melanogaster_index                              ||
||                                                                            ||
||                    ----------------------------------                      ||
||                                                                            ||
||                                Threads : 1                                 ||
||                           Phred offset : 33                                ||
||                # of extracted subreads : 10                                ||
||                         Min read1 vote : 3                                 ||
||                         Min read2 vote : 1                                 ||
||                      Max fragment size : 600                               ||
||                      Min fragment size : 50                                ||
||                         Max mismatches : 3                                 ||
||                       Max indel length : 5                                 ||
||             Report multi-mapping reads : yes                               ||
|| Max alignments per multi-mapping read : 1                                  ||
||                                                                            ||
\\============================================================================//

//================ Running (31-Aug-2020 20:41:51, pid=12882) =================\\
||                                                                            ||
|| Check the input reads.                                                     ||
|| The input file contains base space reads.                                  ||
|| Initialise the memory objects.                                             ||
|| Estimate the mean read length.                                             ||
|| The range of Phred scores observed in the data is [2,40]                   ||
|| Create the output BAM file.                                                ||
|| Check the index.                                                           ||
|| Init the voting space.                                                     ||
|| Global environment is initialised.                                         ||
|| Load the 1-th index block...                                               ||
|| The index block has been loaded.                                           ||
|| Start read mapping in chunk.                                               ||
||   Estimated fragment length : 295 bp                                       ||
||   84% completed, 0.3 mins elapsed, rate=0.5k fragments per second          ||
||                                                                            ||
||                          Completed successfully.                           ||
||                                                                            ||
\\====================================    ====================================//
```

4.2.2 Counting reads aligning to each gene

For differential expression analysis of RNA sequencing data, we then need to figure out which alignments correspond to expressed RNA molecules for which genes. Thus we will need to import a gene annotation file telling us which regions of the genome correspond to pieces of which genes:

```
>  ## need to load data.frame in (s)imple (a)nnotation (f)ormat:
>  ## - require columns named GeneID, Chr, Start, End, Strand
>  ## in order to figure out which alignments correspond to which genes
>  ## when running featureCounts function below
> safAnnot = read.table("d_melanogaster_gene_annotations.saf.gz",
                        sep="\t", row.names=NULL, header=TRUE, quote="")
>  ## let's take a look at this data.frame:
> head(safAnnot)
       GeneID Chr Start   End Strand GeneName
1 FBgn0002121  2L  9839 21376      -   l(2)gl
2 FBgn0031209  2L 21823 25155      -    Ir21a
3 FBgn0051973  2L 25402 65404      -     Cda5
4 FBgn0067779  2L 66482 71390      +      dbr
5 FBgn0031213  2L 71757 76211      + galectin
6 FBgn0031214  2L 76348 77783      +  CG11374
>  ## usually would have more than one bam file for which we
>  ## wanted to find gene alignment counts for:
> bamFiles = c("SRR7900135_downsampled.bam")
> fcOut = featureCounts(
      files = bamFiles,
      annot.ext = safAnnot,
       ## if you have multiple CPUs available, you can tell featureCounts
       ## to use multiple threads to in order to count gene alignments
       ## in parallel; uncomment line below and set to desired number:
      ## nthreads = 6,
      isPairedEnd = TRUE
  )
```

```
        ==========     _____ _    _ ____  _____  ______          _____
        =====         / ____| |  | |  _ \|  __ \|  ____|   /\   |  __ \
          =====      | (___ | |  | | |_) | |__) | |__     /  \  | |  | |
            ====      \___ \| |  | |  _ <|  _  /|  __|   / /\ \ | |  | |
              ====    ____) | |__| | |_) | | \ \| |____ / ____ \| |__| |
        ==========   |_____/ \____/|____/|_|  \_\______/_/    \_\_____/
       Rsubread 2.2.6

//========================== featureCounts setting ===========================\\
||                                                                            ||
||             Input files : 1 BAM file                                       ||
||                           o SRR7900135_downsampled.bam                     ||
||                                                                            ||
||              Annotation : R data.frame                                     ||
||      Dir for temp files : .                                                ||
||                 Threads : 1                                                ||
||                   Level : meta-feature level                               ||
||               Paired-end : yes                                             ||
||      Multimapping reads : counted                                          ||
|| Multi-overlapping reads : not counted                                      ||
||   Min overlapping bases : 1                                                ||
||                                                                            ||
||          Chimeric reads : counted                                          ||
||        Both ends mapped : not required                                     ||
||                                                                            ||
\\============================================================================//

//================================= Running ==================================\\
||                                                                            ||
|| Load annotation file .Rsubread_UserProvidedAnnotation_pid12882 ...         ||
||    Features : 13942                                                        ||
||    Meta-features : 13942                                                   ||
||    Chromosomes/contigs : 8                                                 ||
||                                                                            ||
|| Process BAM file SRR7900135_downsampled.bam...                             ||
||    Paired-end reads are included.                                          ||
||    Total alignments : 10000                                                ||
||    Successfully assigned alignments : 7882 (78.8%)                         ||
||    Running time : 0.00 minutes                                             ||
||                                                                            ||
|| Write the final count table.                                               ||
|| Write the read assignment summary.                                         ||
||                                                                            ||
\\============================================================================//
```

Here, since we already have counts of numbers of reads aligned to the various genes provided by Gene Expression Omnibus, we've only aligned a small number (10,000) of reads from a single sample in the Neves data set using `Rsubread` as an example of how to do it.

4.2.3 Assembling counts table from individual counts files

The counts resulting from the tophat2 alignments performed by the original authors can be downloaded from:

`https://www.ncbi.nlm.nih.gov/geo/download/?acc=GSE120430&format=file`

After extracting the files from the .tar archive, you should have 12 files all of which have names starting with "GSM34007" and ending with the extension ".hts.txt.gz":

hts indicates that the counts were generated with the program HTSeq

txt they are plain text files (I would prefer to use the extension tsv for these, since they are tab-separated and that is a more informative extension in this case, but you'll often find tab-delimited files listed as ".txt" files)

gz they have been compressed with gzip (no need for you to uncompress them!)

Once you've downloaded and extracted the files, I'd suggest making a subdirectory "neves_counts" in your working directory and moving them into that subdirectory. Once this is done, create a named vector mapping brief sample descriptions to the file paths like so:

```
> countsDir = "neves_counts/"
> countsFiles = list.files(countsDir,
                           ## only want files ending in .hts.txt.gz:
                          pattern = "\\.hts\\.txt\\.gz")
>  ## include directory in file paths:
> countsFiles = paste0(countsDir, countsFiles)
>  ## assign names to countsFiles vector so that lapply below will
>  ## produce named list;
>  ## use everything (.*?) in between underscore _ and period \\.,
>  ## as name to associate with sample counts vector:
> names(countsFiles) = gsub(".*_(.*?)\\..*", "\\1", countsFiles)
> head(countsFiles)
                                      mCherryrep1
"neves_counts/GSM3400712_mCherryrep1.hts.txt.gz"
                                      mCherryrep2
"neves_counts/GSM3400713_mCherryrep2.hts.txt.gz"
                                      mCherryrep3
"neves_counts/GSM3400714_mCherryrep3.hts.txt.gz"
                                         TAF9rep1
   "neves_counts/GSM3400715_TAF9rep1.hts.txt.gz"
                                         TAF9rep2
   "neves_counts/GSM3400716_TAF9rep2.hts.txt.gz"
                                         TAF9rep3
   "neves_counts/GSM3400717_TAF9rep3.hts.txt.gz"
```

```
>  ## get rid of "rep" in names(countsFiles):
> names(countsFiles) = gsub("rep", "_", names(countsFiles))
> head(countsFiles)
                                        mCherry_1
"neves_counts/GSM3400712_mCherryrep1.hts.txt.gz"
                                        mCherry_2
"neves_counts/GSM3400713_mCherryrep2.hts.txt.gz"
                                        mCherry_3
"neves_counts/GSM3400714_mCherryrep3.hts.txt.gz"
                                           TAF9_1
   "neves_counts/GSM3400715_TAF9rep1.hts.txt.gz"
                                           TAF9_2
   "neves_counts/GSM3400716_TAF9rep2.hts.txt.gz"
                                           TAF9_3
   "neves_counts/GSM3400717_TAF9rep3.hts.txt.gz"
```

Now we can load all of the counts files by `lapply`ing an anonymous function to the paths in the character vector `countsFiles` and then convert the `lapply` output `list` to a `data.frame`:

```
>  ## use lapply to iterate through countsFiles, applying function
>  ## of file path which...
> counts = lapply(countsFiles, function(path) {
       ## reads in file as data.frame named out:
      out = read.table(path, sep="\t", header=FALSE, row.names=1, quote="")
       ## saves rownames of data.frame read in to vector geneNames:
      geneNames = rownames(out)
       ## replaces data.frame out with counts vector version of out:
      out = out[ , 1]
       ## adds geneNames back as vector names of out:
      names(out) = geneNames
       ## and returns vector
      return(out)
  })
>  ## convert counts from list of vectors to data.frame;
>  ## note we are implicitly assuming all counts files have same
>  ## genes present in same order---this should be checked!
> counts = data.frame(counts)
>  ## remove counts for non-gene rows using logical indexing:
> counts = counts[!(rownames(counts) %in% c("__no_feature",
                                            "__ambiguous",
                                            "__alignment_not_unique",
                                            "__too_low_aQual",
                                            "__not_aligned")), ]
```

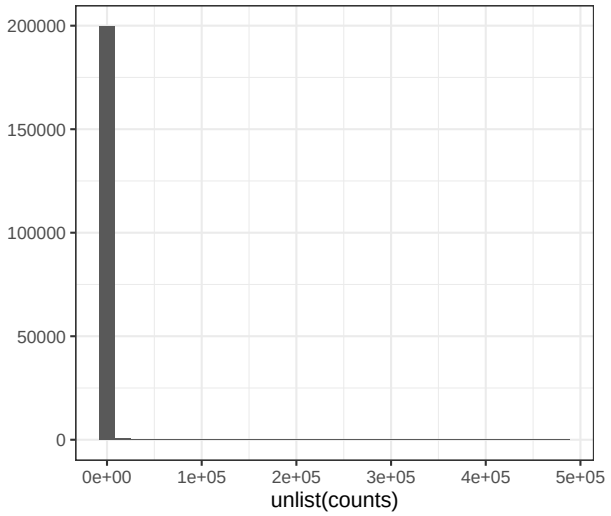
Let's save this `data.frame` to a .tsv file using `write.table` so that we don't have to repeat the whole process we just went through every time we want to load this data in:

```
>  ## save counts data.frame to tsv (tab-separated values) file:
> write.table(
       ## make new data.frame with first *column* (named "gene")
       ## containing gene names so that tsv file has column name
       ## for gene names column; other columns are same as in counts:
      data.frame(gene=rownames(counts), counts),
       ## 2nd arg to write.table should be name of file we're creating
       ## (be careful, will overwrite any existing file of same name!):
      "neves_counts.tsv",
       ## use tab ("\t") separators/delimiters:
      sep = "\t",
       ## don't want to include row.names b/c we added gene column with
       ## same info:
      row.names = FALSE,
       ## no need to put quotation marks around strings in output:
      quote = FALSE
  )
>  ## now compress output counts table file with gzip:
> R.utils::gzip("neves_counts.tsv", overwrite=TRUE)
```

4.2.4 Exploratory analysis

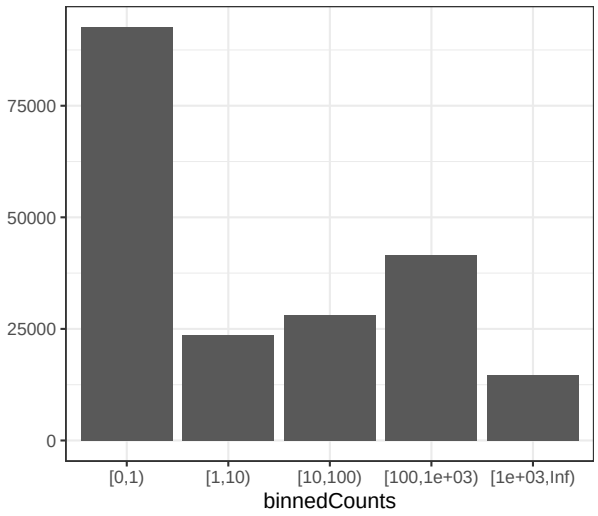
Before turning to more involved statistical modeling of the patterns of gene expression, let's take a moment to visualize and explore the some basic descriptive stats for the `counts` data. Let's start by just visualizing the overall distribution with a histogram:

```
>  ## load ggplot2 and set desired theme:
> library(ggplot2); theme_set(theme_bw())
>  ## recall that a data.frame is a type of list,
>  ## so, if the columns are all of numeric type,
>  ## we can use unlist to convert it to a long numeric vector:
> qplot(unlist(counts), geom="histogram")
```



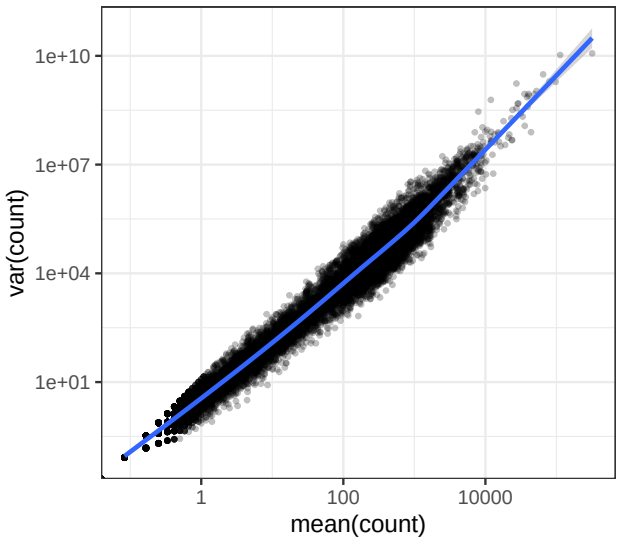
While this plot isn't going to win any prizes, it *does* tell us something very useful: there are many very small values in `counts` but a few much larger ones. Let's use the R function `cut` to break the numeric counts values up into a few (logarithmically spaced) discrete categories and make a bar chart of those:

```
> binnedCounts = cut(
      unlist(counts),
       ## what count values should form discretized bin boundaries:
      breaks = c(0, 1, 10, 100, 1000, Inf),
       ## tell cut to include left (lower) value in each bin
       ## (and exclude right (higher) value) instead of reverse
       ## by setting argument right=FALSE:
      right = FALSE
  )
> qplot(binnedCounts, geom="bar")
```



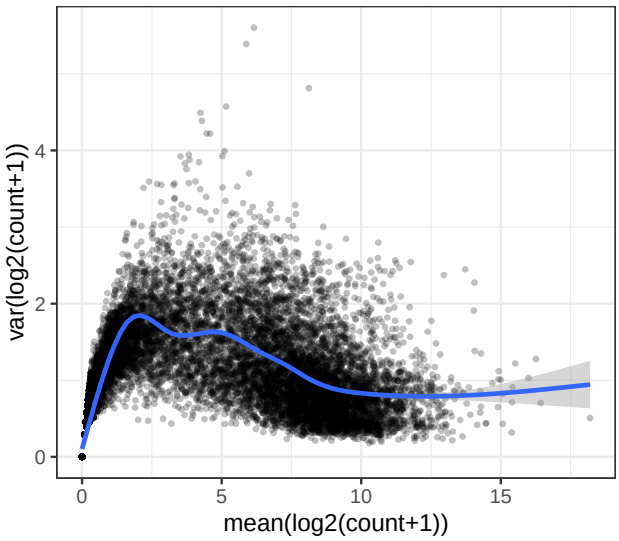
From this plot, you can see that there are many 0s (>75000 of them, out of 200724 total entries in `counts`), and then roughly similar numbers of counts in the 1-9, 10-99, 100-999, and 1000+ ranges (with the most nonzero counts falling in the 100-999 range). From this you may already expect that log-transformed counts may be more amenable to classical analysis methods than untransformed values, but let's also consider some standard gene-wise count summary statistics:

```
> ## install.packages("matrixStats")  ## uncomment and run if necessary
> library(matrixStats)
>  ## remember that counts data.frame is set up with
>  ## rows corresponding to genes and columns to samples:
> genewise = data.frame(
      mean = rowMeans(counts),
       ## rowVars function from matrixStats only accepts matrix arguments
       ## (will not implicitly convert data.frame, unlike rowMeans):
      variance = rowVars(as.matrix(counts))
  )
> gg = ggplot(genewise, aes(x=mean, y=variance))
>  ## make points small (size arg) and transparent (alpha arg):
> gg = gg + geom_point(size=0.75, alpha=0.25)
>  ## avoid plot that shoves most of data into tiny corner of plot
>  ## (like our first histogram) by using log-scaled axes:
> gg = gg + scale_x_log10() + scale_y_log10()
>  ## add smoothed trendline:
> gg = gg + stat_smooth()
>  ## more descriptive axis labels:
> gg = gg + xlab("mean(count)") + ylab("var(count)")
> print(gg)
```



Not too surprisingly, given the skewed nature of the data apparent from this histogram and bar chart above, the variance in the number of counts across samples for gene g shows a strong trend of increasing with the mean for gene g. What if we try the standard classical remedy of log-transformation? Actually, we'll use $\log_2(x+1)$; the "+1" fixes the singular behavior of logarithms at 0—which is a real problem here, since we have many values of exactly 0—while the use of $\log_2$ (instead of, say, natural log or $\log_{10}$) doesn't make any real difference but has become conventional in analysis of gene expression data:

```
> genewiseLogged = data.frame(
      mean_logged = rowMeans(log2(counts+1)),
      sd_logged = rowSds(as.matrix(log2(counts+1)))
  )
> gg = ggplot(genewiseLogged, aes(x=mean_logged, y=sd_logged))
> gg = gg + geom_point(size=0.75, alpha=0.25)
> gg = gg + stat_smooth()
> gg = gg + xlab("mean(log2(count+1))") + ylab("var(log2(count+1))")
> print(gg)
```



While log-transformation does get rid of the positive mean-variance relationship for more highly expressed genes, it appears to create an opposite *negative* relationship between mean and variance of counts for more lowly expressed genes—you might say it "overcorrects" the heteroskedasticity (just a fancy word for a mean-variance trend) at the lowe end. Still, the general consensus is that among the two options of no-transformation vs. log-transformation, log-transformation is definitely better for transcriptomic data analysis. Luckily, there are other analysis options as well!

4.2.5 Normalization and differential expression analysis with `DESeq2`

One such option is `DESeq` from the `DESeq2` package. Before getting into the details of exactly what it does, I'll demonstrate the basic workflow for the simplest usage of it. First, let's reload the data from file (as we would usually be doing prior to `DESeq`ing):

```
>  ## re-load counts data.frame from tsv file made above:
> counts = read.table("neves_counts.tsv.gz",
                      sep="\t", row.names=1, header=TRUE, quote="")
>  ## assemble data.frame annotating the group information for the
>  ## samples represented by the columns of counts:
> sampleAnnotation = data.frame(
       ## set row.names of sampleAnnotation to match col.names of counts:
      row.names = colnames(counts),
       ## use gsub to remove single digit \\d right before end $
       ## of sample names to get group name for each sample:
      group = gsub("_\\d$", "", colnames(counts))
  )
> head(sampleAnnotation)
            group
mCherry_1 mCherry
mCherry_2 mCherry
mCherry_3 mCherry
TAF9_1       TAF9
TAF9_2       TAF9
TAF9_3       TAF9
```

In this case the grouping information for the different samples represented in `counts` can be easily extracted from `colnames(counts)`, but in more complex analyses it is more likely that you would have a separate tabular data file containing whatever information you wanted to load into `sampleAnnotation`. Exither way, you will generally want to get such info into an R `data.frame` object which you can provide to the function `DESeqDataSetFromMatrix` (or possibly one of the other `DESeqDataSet...` functions depending on what format your data is available in):

```
> ## BiocManager::install("BiocParallel")  ## uncomment and run if necessary
> ## BiocManager::install("DESeq2")        ## uncomment and run if necessary
> ## library(BiocParallel)         ## uncomment if you have multiple cores
> ## register(MulticoreParam(4))   ## available and want to parallelize DESeq
> library(DESeq2)
>  ## first initialize DESeq object:
> dds = DESeqDataSetFromMatrix(
       ## countData argument should be numeric matrix (or data.frame with
       ## only numeric columns which can be coerced to numeric matrix):
      countData = counts,
       ## colData argument should be data.frame with grouping information
       ## (and any other non-gene-expression covariates that might be
       ##  useful in modeling the counts) whose rows correspond to the
       ## columns (hence name colData) of countData:
      colData = sampleAnnotation,
       ## note tilde preceding the name group of the relevant column in
       ## the data.frame sampleAnnotation provided as the colData argument
       ## in the design argument---design argument must be formula object,
       ## similar to the right-hand side of the formulas used for linear
       ## modeling with lm:
      design = ~group
  )
>  ## now run DESeq analysis pipeline on object dds:
> dds = DESeq(dds, parallel=FALSE) ## set parallel=TRUE if using BiocParallel
```

That's basically all there is to *running* the `DESeq` pipeline in most cases, though obviously you will need to perform various other computational steps to extract useful results from the resulting `dds` object; we will see some examples of how to do that in the sections yet to come.

But first, let's pause to consider what is really going on when you call the function `DESeq` on your just-initialized `dds` object: The operation of `DESeq` can be decomposed into three separate steps, each of which can be called indvidually as its own function in R:

1. `estimateSizeFactors` calculates a "size factor" (a number) for each sample;
    - used for *normalizing* expression levels so that samples with larger total numbers of reads are not treated as having higher expression of all or most genes
    - RNA-seq is generally considered a method for estimation of *relative* RNA abundance, not *absolute* RNA abundance!
2. `estimateDispersions` models the relationship between mean expression and a measure of expression dispersion (similar to variance or standard deviation, but more suitable for modeling of count data).
3. `nbinomWaldTest` implements the negative-binomial generalized linear model (GLM) significance testing to get p-values for each gene using the model parameters fit in the previous two steps.

It is sometimes more useful to break them out separately like so:

```
>  ## running these three steps is equivalent to single DESeq call above:
> dds = estimateSizeFactors(dds)
> dds = estimateDispersions(dds)
> dds = nbinomWaldTest(dds)
```

though again there is no need to do so in most cases. For the purposes of understanding DESeq, however, we will now consider each step in turn.

DESeq2 normalization

DESeq produces several useful outputs; the first one we'll consider are the normalized expression values (often referred to as "normalized counts"). We can extract them (as a matrix) from the object dds (of class DESeqDataSet) using the counts *function* from DESeq2 with the argument normalized=TRUE.

```
>  ## use counts function with normalized arg set to TRUE to extract
>  ## "normalized counts" (which are not integers) from dds
>  ## (note: counts *function* from DESeq2 package is different object
>  ##  in R than counts *data.frame* we loaded from tsv file;
>  ##  R can tell based on context which one we mean):
> normed = counts(dds, normalized=TRUE)
>  ## save the normalized counts matrix to a tsv file:
> write.table(data.frame(gene=rownames(normed), normed),
              "neves_deseq_normalized.tsv",
              sep="\t", row.names=FALSE, quote=FALSE)
>  ## and compress it with gzip:
> R.utils::gzip("neves_deseq_normalized.tsv", overwrite=TRUE)
```

Recalling that the estimateSizeFactors step performed as part of DESeq is involved in the normalization, let's make a quick comparison:

```
>  ## use sizeFactors function to see what the estimated size factor
>  ## for each sample is:
> sizeFactors(dds)
mCherry_1 mCherry_2 mCherry_3    TAF9_1    TAF9_2    TAF9_3     TBP_1     TBP_2
1.1369027 0.9903182 0.8065829 0.9050339 1.1965980 1.0210430 1.0412352 0.8723615
    TBP_3    TRF2_1    TRF2_2    TRF2_3
0.7967642 1.2136036 1.3415207 1.1834430
>  ## try out the simplest way one might try to use these to
>  ## scale "raw" counts to get normalized values:
> head(counts[ , "mCherry_1"] / sizeFactors(dds)[["mCherry_1"]])
[1]  1313.217094 45777.885891  7155.405932   558.535067     9.675411
[6]   290.262318
>  ## compare to normed[ , "mCherry_1"]:
> head(normed[ , "mCherry_1"])
       128up 14-3-3epsilon    14-3-3zeta         140up           18w
  1313.217094  45777.885891   7155.405932    558.535067      9.675411
      26-29-p
   290.262318
```

So DESeq normalization just consists of estimating sizeFactors and then dividing the input counts for each sample by the size factor value estimated for that sample. The formula for estimating the size factors for each sample isn't all that complicated, but it is definitely not as simple as the "old-school" reads-per-million approach to RNA-seq normalization, which was shown to have some major problems!

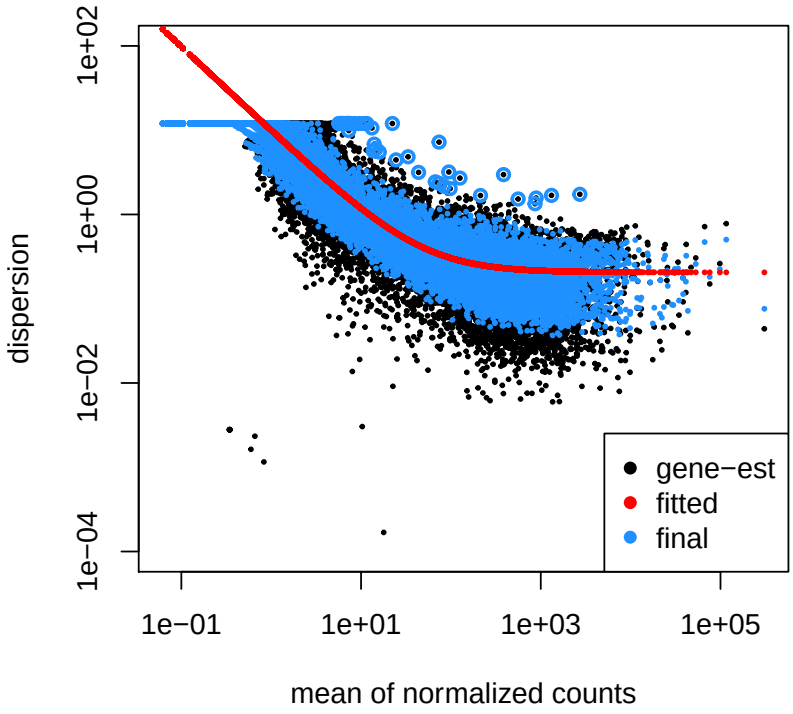
In case you're interested in exactly what formula is used by estimateSizeFactors—at least, the simplest version used by default; some more complicated versions are available if you play with the optional arguments—it boils down to:

```
> defaultDESeqNormalize = function(x) {
       ## convert argument x to matrix if data.frame
       ## (only changes x within body of function!):
      x = as.matrix(counts)
       ## remove all rows (genes) with a 0 count in any sample:
      x = x[rowMins(x) > 0, ]
       ## construct "pseudo-sample" with "count" for gene g
       ## equal to geometric mean of counts for g across all samples
       ## (geometric mean can be calculated by logging,
       ##  taking average of logged values, and then un-logging with exp):
      pseudoSampleProfile = exp(rowMeans(log(x)))
       ## calculate ratio for gene g on sample i by dividing
       ## x[g, i] / pseudoSampleProfile[g]:
      ratios = x / pseudoSampleProfile  ## dividing matrix by vector!
       ## (note: this is how dividing m-by-n matrix by m-vector works in R;
       ##  will *not* work for dividing m-by-n matrix by n-vector!)
       ## ---
       ## size factor for sample i is median of ratios[ , i]:
      medianRatios = colMedians(ratios)
       ## set names on output vector to sample names:
      names(medianRatios) = colnames(x)
      return(medianRatios)
  }
>  ## compare:
> head(sizeFactors(dds))
mCherry_1 mCherry_2 mCherry_3    TAF9_1    TAF9_2    TAF9_3
1.1369027 0.9903182 0.8065829 0.9050339 1.1965980 1.0210430
>  ## to:
> head(defaultDESeqNormalize(counts))
mCherry_1 mCherry_2 mCherry_3    TAF9_1    TAF9_2    TAF9_3
1.1369027 0.9903182 0.8065830 0.9050339 1.1965980 1.0210430
```

DESeq2 mean-dispersion relationship estimation

The mean-dispersion relationship estimated by estimateDispersions can be visualized using the plotDispEsts function from DESeq2:

```
> plotDispEsts(dds, ylim=c(1e-4, 1e2))
```



This plot illustrates the method used by `estimateDispersions`, with:

1. the **gene-est** points in the plot correspond to initial gene-wise dispersion estimates made by modeling each gene g separately;
2. the **fitted** line is the mean-dispersion trend fit (by "robust gamma-family GLM") to the gene-est points; and finally
3. the **final** points represent the final individual gene dispersion values assigned to each gene by "shrinking" the distance between the original gene-est point and the fitted trend line value vertically above or below it.

Thus the final dispersion value for gene g represents a compromise between the initial value based solely on the data for g and the global mean-dispersion relationship determined based on the data for all genes. This is a common strategy employed in high-dimensional data sets when the variables—genes in this case—are all of a similar nature (same units, method of measurement, etc.), so that information about one may be "borrowed" from what has been seen with others.

Looking at the plot you may be surprised to see that the estimated dispersion *decreases* for genes with increasing mean expression values. You may also notice that the plot is shown with both axes displayed in log-scale. Both of these facts are related to the log-transform built into the particular statistical model employed by `DESeq2`, as we'll see in the next section.

`DESeq2` statistical modeling and significance testing

Note: This section begins by describing the statistical model employed by `DESeq2`, the details of which are unfortunately quite complicated. While I think you can learn quite a bit by studying this description, I don't expect you to understand all of it—I don't think most users of `DESeq2` do—and you're more than welcome to skip on down to the code for extracting the p-values and examining the expression levels of the significant genes to get an intuitive sense of what they tell if you like!

For gene g with (final) dispersion parameter α_g, `DESeq2` models the count for g in sample i as a negative binomially-distributed random variable X_{gi} (Love *et al.* (2014)):

$$X_{gi} \sim \text{NBin}\left(\mu_{gi}, \alpha_g\right) \tag{4.1}$$

The *negative binomial* distribution is a probability distribution applicable to integer-valued random variables, such as the number of reads aligned to a gene g in sample i. In case you are familiar with the more well-known Poisson distribution, you can think of the negative binomial distribution as a more flexible generalization of that distribution which adds an additional parameter to allow for the variance of the counts to be larger than the mean μ_{gi}. The dispersion α_g controls how much larger, with the variance $\mathbb{V}[X_{gi}]$ given by:

$$\mathbb{V}[X_{gi}] = \mu_{gi} + \alpha_g \mu_{gi}^2 \tag{4.2}$$

Note that for fixed dispersion α_g, the variance $\mathbb{V}[X_{gi}]$ of the negative binomial distribution in Eq (4.2) tends to grow with the square of its mean μ_{gi}: Unlike modeling based on the normal distribution, negative binomial models do *not* assume that variance is independent of mean! In fact, they assume that variance increases with mean in a way quite similar to that which results when we apply classical normal-distribution based methods to log-transformed data. This is why we see the dispersion parameter α_g actually shrinking in the `plotDispEsts` output in section 4.2.5 above: While log transformation tends to work well to remove heteroskedasticity for highly expressed genes, it can overcorrect at lower expression levels leading to increased dispersion at the low end as observed here.

While we've already discussed where the dispersion parameter α_g comes from in section 4.2.5 above, the mean parameter μ_{gi} is new: It is itself modeled using a logarithmic *link function*:

$$\log_2(\mu_{gi}) = \log_2(s_i) + \beta_{g,\rho(i)} \tag{4.3}$$

where

- s_i = `sizeFactors(dd)[[i]]` is the normalization size factor for sample i,
- $\beta_{g,\rho(i)}$ is a parameter of the model for each gene g and sample **group** $\rho(i)$, and
- ρ is a function of sample index such that $\rho(i)$ is the group to which sample i belongs (e.g., $\rho(i)$ could be mCherry, TAF9, TBP, or TRF2 for the Neves data set).

The model encoded by Eqs (4.1) and (4.3) is fit to the `counts` data by estimating the best values of all of the parameters $\beta_{g,\rho}$ according to same rather involved statistical criteria detailed in Love *et al.* (2014).

The estimated "log-fold-change" (LFC) for gene g between two different groups ρ and τ is then, according to this model:

$$\mathrm{LFC}(g;\, \rho, \tau) = \beta_{g,\rho} - \beta_{g,\tau} \tag{4.4}$$

and the p-values estimated by `nbinomWaldTest` apply to the null hypothesis that this difference $\beta_{g,\rho} - \beta_{g,\tau} = 0$.

Because we have built our `DESeq2` object `dds` for an experiment with more than 2 groups (4 to be exact: mCherry, TAF9, TBP, and TRF2), there are multiple LFC comparisons which we could consider for each gene g. By default, `DESeq2` will treat the first group—in this case, the lexicographically lowest group name is mCherry—as a reference group, and will compute LFC values for all comparisons of one of the other groups against this group. You can see what comparisons are available using the `resultsNames` function:

```
> resultsNames(dds)
[1] "Intercept"              "group_TAF9_vs_mCherry" "group_TBP_vs_mCherry"
[4] "group_TRF2_vs_mCherry"
```

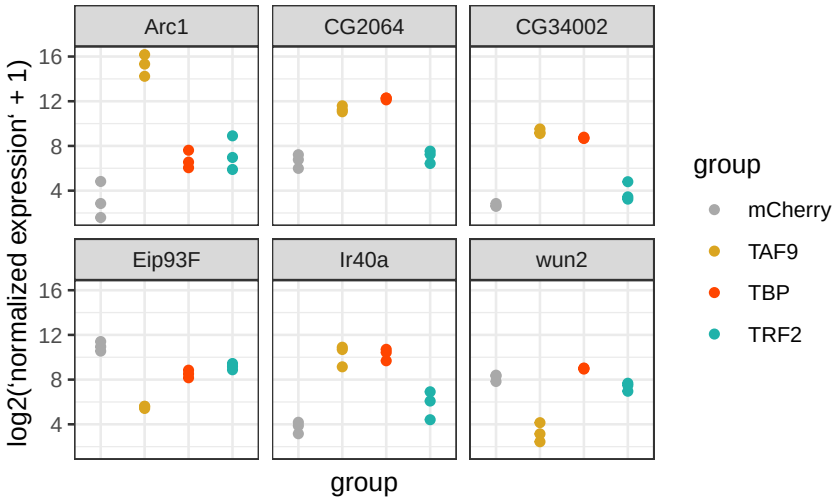
The "Intercept" comparisons are generally of little interest, as is the case with most of R's statistical modeling functions. So let's look instead at the second element of `resultsNames(dds)`:

```
> taf9results = results(dds, name="group_TAF9_vs_mCherry")
>  ## sort taf9 results data.frame by p-value:
> taf9results = taf9results[order(taf9results$pvalue), ]
>  ## look at most signficant results:
> head(taf9results)
```

```
log2 fold change (MLE): group TAF9 vs mCherry
Wald test p-value: group TAF9 vs mCherry
DataFrame with 6 rows and 6 columns
         baseMean log2FoldChange     lfcSE      stat      pvalue        padj
        <numeric>      <numeric> <numeric> <numeric>   <numeric>   <numeric>
Eip93F    764.955       -5.49622  0.365716  -15.0287 4.76610e-51 4.27662e-47
Arc1    11276.079       11.86634  0.866427   13.6957 1.07686e-42 4.83134e-39
CG34002   266.307         6.83435  0.551318   12.3964 2.73331e-35 8.17534e-32
CG2064   1897.912         4.58475  0.406261   11.2852 1.55208e-29 3.48171e-26
wun2      245.086        -4.88517  0.470483  -10.3833 2.95400e-25 5.30125e-22
Ir40a     687.855         6.74286  0.664450   10.1480 3.38135e-24 5.05681e-21
```

Let's visualize the normalized expression levels of the most significantly differentially expressed genes:

```
> top6 = rownames(taf9results)[1:6]
> plotData = data.frame(gene=top6, normed[top6, ])
> library(tidyr)  ## for pivot_longer
> plotData = plotData %>%
      pivot_longer(-gene, names_to="sample", values_to="normalized expression")
>  ## add group information to plotData:
> plotData$group = sampleAnnotation[plotData$sample, "group"]
>  ## need backticks around `normalized expression` in ggplot2::aes
>  ## function call below b/c of space in column name:
> gg = ggplot(plotData,
             aes(x=group, y=log2(`normalized expression`+1), color=group))
>  ## use facet_wrap to split out data for each gene into separate
>  ## panel (or facet); note that tilde required prior to column name gene:
> gg = gg + facet_wrap(~ gene)
> gg = gg + geom_point()
> gg = gg + scale_color_manual(
      values = c("darkgray", "goldenrod", "orangered", "lightseagreen")
  )
>  ## remove x-axis labels, since legend for color indicates same info:
> gg = gg + theme(axis.text.x=element_blank(), axis.ticks.x=element_blank())
> print(gg)
```



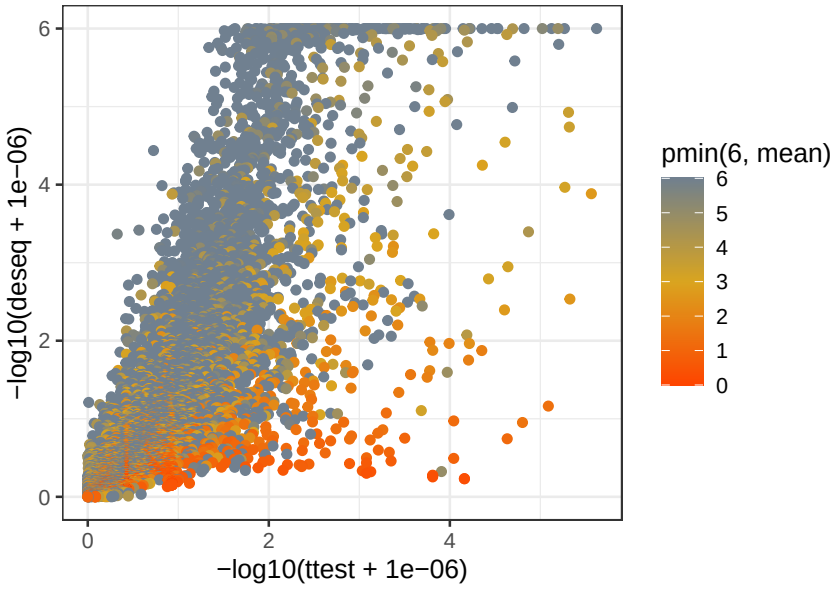
4.2.6 DESeq2 vs. plain old *t*-tests

A much simpler approach to finding genes with evidence of differential expression in group ρ compared to group τ would be to apply a t-test to each gene g. We could still use the (log-transformed) normalization procedure from DESeq2, which has generally held up well in studies comparing different RNA-seq normalization procedures (Dillies *et al.* (2013)) in order to remove as much "technical noise" as possible. The Bioconductor package genefilter even makes application of t-tests to every row or column of a data.frame or matrix quite straightforward (and efficient) in R:

```
> ## BiocManager::install("genefilter")  ## uncomment and run if necessary
> library(genefilter)
>  ## use rowttests (since genes in rows)
>  ## to do pairwise comparison of mCherry and TAF9 samples
>  ## (first 6 samples in Neves data, hence 1:6 showing up below);
>  ## log-transform normed data to deal with heteroskedasticity:
> taf9t = rowttests(x = log2(normed[ , 1:6]+1),
                    fac = factor(sampleAnnotation[1:6, "group"]))
>  ## sort t-test results to same order as taf9results from DESeq2:
> taf9t = taf9t[rownames(taf9results), ]
> head(taf9t)
         statistic         dm      p.value
Eip93F   21.263995   5.424453 2.891981e-05
Arc1    -11.107259 -12.156461 3.737770e-04
CG34002 -45.540536  -6.580583 1.390477e-06
CG2064  -11.866226  -4.635396 2.888108e-04
wun2      9.387566   4.939495 7.174296e-04
Ir40a   -10.300664  -6.508050 5.010535e-04
```

The p-values produced for these genes by these t-tests are clearly much higher (less significant) than those produced by DESeq, though this may be somewhat biased by the fact that we're considering only the genes declared most signficant by DESeq. (A more subtle source of bias here is that the model built by DESeq considered all 12 samples and thus has more degrees of freedom associated with it's error estimates than the t-test approach, which considers only the 6 samples in the two groups mCherry and TAF9 being directly compared.) Let's check whether the p-values produced by these two different methods are at least correlated even if very different in magnitude:

```
> plotData = data.frame(
      ttest = taf9t$p.value,
      deseq = taf9results$pvalue,
       ## keep track of DESeq2's base mean (log-transformed) for each gene:
      mean = log2(taf9results$baseMean+1),
       ## as well as DESeq2-estimated log fold change:
      lfc = taf9results$log2FoldChange
  )
> gg = ggplot(plotData,
               ## add offset of 1e-6 to p-values so that plot axes
               ## are not distorted by a few very low p-values:
              aes(x = -log10(ttest+1e-6),
                  y = -log10(deseq+1e-6),
                   ## pmin is parallel mean function; here using
                   ## to set all values greater than 6 down to 6:
                  color = pmin(6, mean)))
> gg = gg + geom_point(size=1.5)
> gg = gg + scale_color_gradientn(
       ## use colorRampPalette to generate list of smoothly varying
       ## colors ranging through those in supplied vector argument:
      colors = colorRampPalette(rev(c(
          "slategray", "goldenrod", "orangered"
      )))(100)  ## generate 100 different gradations of these colors
  )
> print(gg)
```



While overall those genes with very low p-values—and thus high $-\log_{10}(p+10^{-6})$ values—by one method tend to also have low p-values by the other method, those genes with (log-transformed) base mean expression values much less than 6 stand out as much more likely to get relatively low p-values by t-test than they are by `DESeq`. This is largely due to the unsatisfactory results achieved by log-transformation for lowly expressed genes which we discussed earlier in section 4.2.4: There is no transformation capable of converting data composed of very small integer counts into something which looks normally distributed!

`DESeq` resolves this problem by abandoning the normal distribution and replacing it with the (much less friendly) negative binomial distribution. While this produces more satisfactory results than the naive t-test approach considered here, do note that the main effect is to declare pretty much all lowly expressed genes insignificant. A cynic might suggest that this could be more easily achieved by simply filtering out all genes with very low average expression prior to testing! This approach, combined with various refinements of the classical (normal distribution-based) approach to statistical modeling can also produce pretty good results with RNA-seq data, as has been demonstrated by authors making use of the Bioconductor package `limma` (originally named as shorthand for **li**near **m**odels for **m**icro**a**rrays, as it predates the widespread use of RNA-seq in transcriptome analysis).

4.2.7 False discovery rates and adjusted p-values

The `DESeq` results `taf9results` included a column named `padj` for which there was no analogue in the `rowttests` results `taf9t`. This column includes false discovery rate-adjusted p-values.

You may recall from biostats the statistical problem associated with multiple hypothesis testing: While using a set significance threshold of θ for p-value testing limits the chance of any one test yielding a false positive result to θ, the chance of getting at least one false positive significance finding (the *family-wise error rate*, or FWER) will generally be $> \theta$ if $m > 1$ separate tests are run and each assessed for signficance at threshold θ.

The simplest approach to dealing with the multiple hypothesis testing problem—aside from just ignoring it, which, while popular, is not a good idea—is the Bonferonni correction technique: Use a threshold of $\frac{\theta}{m}$ to ensure that the FWER is $\leq \theta$. This may be equivalently formulated as comparing Bonferonni-adjusted p-values—which are just the actual p-values multipled by the number of tests m—with the original threshold θ in order to determine significance.

However, when the number of tests m becomes large—as it usually is in differential expression testing, where we want to test thousands to tens of thousands of genes at once—the Bonferonni technique is usually considered to be too *conservative*, for two reasons:

1. In many cases the actual FWER using a Bonferonni correction will be *much* less than θ, meaning that a higher threshold—which would have allowed the detection of more truly significant results—could have been used.
2. It's not really clear that maintaining FWER less than θ is a good goal to begin with: Must we really through out hundreds of likely positive results in order to ensure that not a single one is a false positive?

Benjamini and Hochberg proposed a method for instead controlling the *false discovery rate*, or FDR: what *fraction* of positive findings are expected to be false (Benjamini & Hochberg (1995)). This approach embodies the philosophy that it's OK to allow a few reporting of a few false positives as long as it is expected that a suitably large fraction of the reported findings are likely to be true positive results.

The false discovery rate associated with declaring all genes with p-values below a threshold θ is expected to be

$$\mathrm{FDR}(\theta) = \frac{m\,\theta}{s_\theta} \tag{4.5}$$

where s_θ is the number of genes with p-value $p_g \leq \theta$ which would be declared significant using the threshold θ. The numerator of Eq (4.5) is just the number of tests m times the threshold θ, since by definition of a p-value, the chance of a false positive with threshold θ should be θ.

We define the adjusted p-value p_g^{adj} to be the smallest false discovery rate which can be achieved using a threshold $\theta \geq p_g$ high enough to declare g significant:

$$p_g^{\mathrm{adj}} = \min_{\theta \geq p_g} \mathrm{FDR}(\theta) \tag{4.6}$$

If it were true that $\mathrm{FDR}(\theta)$ was always a decreasing function of the threshold θ—that is, that using a more stringent threshold θ always resulted in a lower FDR value—then we could simplify the right-hand side of Eq (4.6) to just

$$\mathrm{FDR}(p_g) = \frac{m\,p_g}{s_{p_g}} \tag{4.7}$$

However, it may be the case that using a higher threshold θ will increase the denominator s_θ of Eq (4.5) by a larger factor than it does the numerator $m\,\theta$!

However, note:

- if θ is not equal to one of the p_g values for some gene g,
- then the FDR must shrink upon reducing θ down to whichever $p_g < \theta$ value is the highest while still being less than θ
- since this shrinks the numerator $m\,\theta$ of Eq (4.5)
- but does not change the denominator s_θ.

This shows that the only thresholds we need to consider in trying to find the minimum on the right-hand side of Eq (4.6) are of the form $\theta = p_h$ for genes h with $p_h \geq p_g$:

$$p_g^{\text{adj}} = \min_{p_h \geq p_g} \text{FDR}(p_h) \tag{4.8}$$

$$= \min_{p_h \geq p_g} \frac{m\,p_h}{s_{p_h}} \tag{4.9}$$

If we sort the genes py p-value, so that $p_h \geq p_g$ if and only if $h > g$, we can rewrite Eq (4.9) as:

$$p_g^{\text{adj}} = \min_{h \geq g} \frac{m\,p_h}{s_{p_h}} \tag{4.10}$$

$$= \min_{h \geq g} \frac{m\,p_h}{h} \tag{4.11}$$

where we know the denominator $s_{p_h} = h$ in moving from Eq (4.10) to Eq (4.11) because the p-values are sorted, so that the p-values $\leq$ the threshold $\theta = p_h$ must be exactly the set $\{p_1, p_2, \ldots, p_h\}$.

Here's a (rather verbose) implementation of this algorithm in R:

```
> bhfdr = function(p) {
      m = length(p)
      pOrder = order(p)
       ## set up inverse permutation pOrderBack to get original order back:
      pOrderBack = order(pOrder)
       ## now sort p:
      pSorted = p[pOrder]
       ## multiply first element of pSorted by m,
       ## second by (m/2), third by (m/3), etc.:
      adjSorted = pSorted * m / (1:m)
       ## set i-th element of adjSorted to minimum of all values
       ## from adjSorted[i:m]:
       ## use cumulative min function cummin,
       ## defined by cummin(v)[[i]] = min(v[i:length(v)]),
       ## applied to reverse-ordered adjSorted to do this efficiently:
      minAdjValAfterI = cummin(adjSorted[m:1])
       ## now need to reverse the reverse-ordering back:
      adjSorted = minAdjValAfterI[m:1]
       ## maximum sensical value of adjusted p-value is 1,
       ## so reset anything above this to 1:
      adjSorted[adjSorted > 1] = 1
       ## re-order adjSorted to be consistent with order of input p
       ## and return:
      return(adjSorted[pOrderBack])
  }
```

But do note that R already has a built-in function `p.adjust` which can be used to calculate false discovery rate adjusted p-values via the same basic algorithm:

```
>  ## DESeq removes genes with very low expression values from
>  ## consideration before adjusting p-values for significance
>  ## testing; these get assigned NA padj values, so it's
>  ## easy to figure out which genes DESeq did include in testing:
> testedGenes = which(!is.na(taf9results$padj))
> head(taf9results[testedGenes, "padj"])
[1] 4.276624e-47 4.831345e-39 8.175336e-32 3.481712e-26 5.301252e-22
[6] 5.056805e-21
>  ## compare to values calculated using bhfdr as defined above;
> head(bhfdr(taf9results[testedGenes, "pvalue"]))
[1] 4.276624e-47 4.831345e-39 8.175336e-32 3.481712e-26 5.301252e-22
[6] 5.056805e-21
>  ## compare to built-in p.adjust function:
> head(p.adjust(taf9results[testedGenes, "pvalue"], method="fdr"))
[1] 4.276624e-47 4.831345e-39 8.175336e-32 3.481712e-26 5.301252e-22
[6] 5.056805e-21
```

Let's now add the adjusted p-values into `taf9t`:

```
>  ## only adjust the p-values for the testedGenes:
>  ## - first set all padj to NA:
> taf9t$padj = NA
>  ## - then fill in padj for testedGenes using p.adjust:
> taf9t[testedGenes, "padj"] = p.adjust(taf9t[testedGenes, "p.value"])
> head(taf9t)
         statistic         dm      p.value       padj
Eip93F    21.263995   5.424453 2.891981e-05 0.25787797
Arc1     -11.107259 -12.156461 3.737770e-04 1.00000000
CG34002  -45.540536  -6.580583 1.390477e-06 0.01242809
CG2064   -11.866226  -4.635396 2.888108e-04 1.00000000
wun2       9.387566   4.939495 7.174296e-04 1.00000000
Ir40a    -10.300664  -6.508050 5.010535e-04 1.00000000
```

The naive t-test approach finds insufficient evidence to declare several of the top DESeq-identified genes significant. While this suggests—correctly, in my view!—that DESeq is a superior method to a totally naive t-testing approach for analyzing differential expression patterns in RNA-seq data, do keep in mind that just because method A finds more significant results than method B, that does *not* necessarily mean that method A is better. (If so, method C consisting of just declaring all tests significant would always be the best!)

4.3 Principal component analysis (PCA)

Gene expression data, such as obtained via RNA-sequencing experiments, is massively multivariate, typically yielding simultaneous quantitative estimates of the expression levels of tens of thousands of distinct genes.

```
> normed = read.table("neves_deseq_normalized.tsv.gz",
                      sep="\t", header=TRUE, row.names=1, quote="")
>  ## simplify data by removing genes with no data:
> normed = normed[rowSums(normed) > 0, ]  ## logical indexing...
>  ## log-transform data to reduce heteroskedasticity:
> lgNorm = log2(normed + 1)   ## log2 works element-wise on data.frame
>  ## what are the dimensions of this data set?
> dim(lgNorm)
[1] 12599    12
> sampleAnnotation = data.frame(
       ## set row.names of sampleAnnotation to match col.names of normed:
      row.names = colnames(normed),
       ## use gsub to remove single digit \\d right before end $
       ## of sample names to get group name for each sample:
      group = gsub("_\\d$", "", colnames(normed))
  )
```

So we have data on 12599 genes from each of 12 samples. How can we meaningfully summarize all of this information? Principal component analysis, or PCA, offers one way to do this.

There are many different ways to describe the underyling idea of PCA (Roweis & Ghahramani (1999); Izenman (2008));, and the best way to really get to understand it is to read through several of them. Here's one: PCA fits a series of *principal components* to model the expression levels z_{ig} of all genes g across all samples i. We'll start with a single principal component (PC1) model:

$$z_{ig} = \mu_g + x_{i1} r_{g1} + e_{ig}^{(1)} \tag{4.12}$$

where:

- $\mu_g = \frac{1}{n} \sum_i z_{ig}$ is the mean expression level of gene g,
- x_{i1} is the "score" of sample i on PC1,
- r_{g1} is the "loading" of gene g on PC1, and
- $e_{ig}^{(1)}$ is the error residual for gene g on sample i using PC1 model.

Fitting PC1 thus requires estimating x_{i1} for all samples i and r_{g1} for all genes g. This is generally done so as to minimize the sum of squared residuals $\sum_{i,g} \left(e_{ig}^{(1)}\right)^2$ (PCA is another least-squares algorithm). It so happens that there is a beautifully elegant and efficient algorithm solving just this problem via something known in linear algebra as singular value decomposition (SVD) implemented in the R function `prcomp`:

```
>  ## use prcomp function for PCA in R
>  ## on transposed version of lgNorm, t(lgNorm),
>  ## since prcomp assumes variables (genes) are in columns, not rows
>  ## * z_ig = lgNorm[g, i] = t(lgNorm)[i, g]
> pca = prcomp(t(lgNorm))
> class(pca)
[1] "prcomp"
> is.list(pca)
[1] TRUE
> names(pca)
[1] "sdev"     "rotation" "center"   "scale"    "x"
>  ## the PCA scores x_i1 for samples are found in pca$x:
> head(pca$x[ , 1, drop=FALSE])  ## drop=FALSE: keep as matrix w/ one column
                PC1
mCherry_1 -56.979532
mCherry_2 -28.236714
mCherry_3  -2.009504
TAF9_1    -13.152929
TAF9_2    -19.260352
TAF9_3    -32.580905
>  ## the PCA loadings r_g1 for genes are found in pca$rotation:
> head(pca$rotation[ , 1, drop=FALSE])
```

```
                      PC1
128up          0.003794324
14-3-3epsilon 0.003303006
14-3-3zeta     0.003839142
140up          0.003059941
18w            0.002916722
26-29-p        0.008712380
```

You might notice that I extracted only the first column from both `pca$x` and `pca$rotation`. This is because these matrices contain PCs beyond PC1. For instance, the second column of each of these matrices corresponds to PC2, which is defined by

$$z_{ig} = \mu_g + x_{i1} r_{g1} + x_{i2} r_{g2} + e_{ig}^{(2)} \tag{4.13}$$

where PC1 is obtained from the single-PC model and PC2 is then again fit to minimize the remaining $\sum_{i,g} \left(e_{ig}^{(2)} \right)^2$.

This process can be repeated to obtain higher order PCs as well; in general, the k^{th} PC has

- scores x_{ik} which can be found in the R object `pca$x[i, k]` and
- loadings r_{gk} stored in `pca$rotation[g, k]`

and minimizes the squared sum of the error residuals

$$e_{ig}^{(k)} = z_{ig} - \mu_g - \sum_{j=1}^{k} x_{ik} r_{gk} \tag{4.14}$$

It is worth noting that after fitting n PCs (recall n is the number of samples, 12 here), there is no error left—that is, $e_{ig}^{(n)} = 0$—so that we will have a perfect fit for z_{ig}:

$$z_{ig} = \mu_g + \sum_{j=1}^{n} x_{ik} r_{gk} \tag{4.15}$$

$$= \mu_g + \left[X R^{\mathrm{T}} \right]_{ig} \tag{4.16}$$

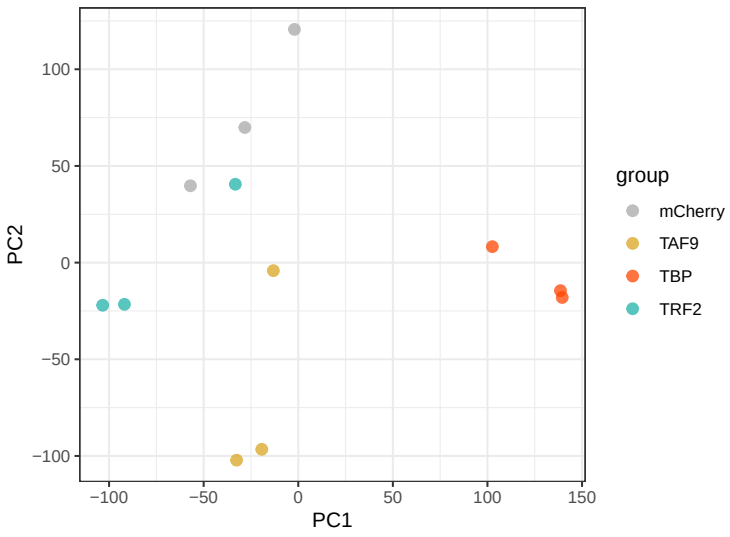
where in Eq (4.16) we have switched over to matrix notation. This can be confirmed in R via:

```
> pcaFit = rowMeans(lgNorm) + t( pca$x %*% t(pca$rotation) )
>  ## have to transpose pca$x %*% t(pca$rotation) above b/c lgNorm is t(z)
> lgNorm[1:3, 1:3]
              mCherry_1 mCherry_2 mCherry_3
128up          10.35999  10.64793  10.84714
14-3-3epsilon  15.48239  15.52924  15.86000
14-3-3zeta     12.80502  13.16790  13.55206
> pcaFit[1:3, 1:3]  ## same thing!
```

```
               mCherry_1 mCherry_2 mCherry_3
128up           10.35999  10.64793  10.84714
14-3-3epsilon   15.48239  15.52924  15.86000
14-3-3zeta      12.80502  13.16790  13.55206
```

By now you may be wondering what we do with all of these sample scores x_{ik} and gene loadings r_{gk}. Well, the first thing we usually do is make a PCA plot by plotting x_{i2} (which we usually label as simply "PC2") on the vertical axis against x_{i1} ("PC1") on the horizontal axis:

```
>  ## set up data.frame pcaData for ggplot...
> pcaData = data.frame(pca$x[ , 1:2])
>  ## add in sample annotation info
> pcaData$group = sampleAnnotation[rownames(pcaData), "group"]
>  ## and sample names
> pcaData$sample = rownames(pcaData)
>  ## make sure ggplot2 library is loaded
> library(ggplot2)
>  ## bw theme...
> theme_set(theme_bw())
> gg = ggplot(pcaData, aes(x=PC1, y=PC2, color=group, label=sample))
> gg = gg + geom_point(size=2.5, alpha=0.75)
> colVals = c("darkgray", "goldenrod", "orangered", "lightseagreen")
> gg = gg + scale_color_manual(values=colVals)
> print(gg)
```



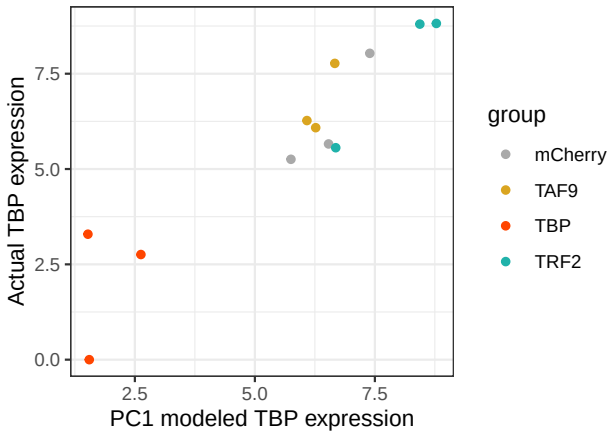
This shows us something interesting: despite PCA knowing nothing about the sample groupings, it has fit PC1 so as to split the TBP experimental group apart from all others (in the sense that the TBP group samples have large positive scores x_{i1} while all other samples have negative PC1 scores). This tells us that, in some sense, TBP is the most different sample group relative to all of the others.

4.3.1 Modeling expression levels with first PC (or two)

I introduced PCA as a way to model expression levels using multiplicative factors of sample scores and gene loadings. You might well wonder how good it is as such a model, so let's go ahead and look at the expression patterns (sometimes called "profiles") of a gene or two.

Let's start with the gene TBP:

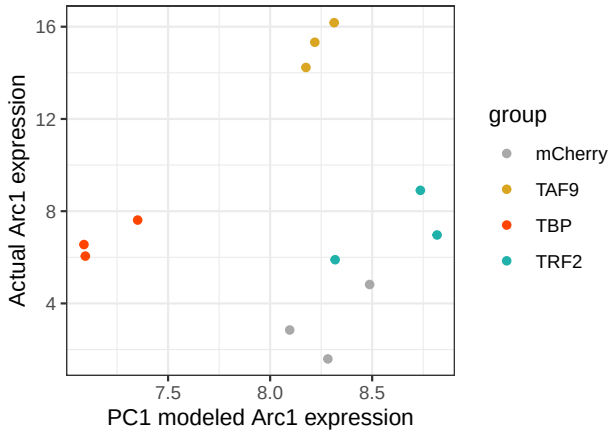
```
> z = t(lgNorm)
> tbpData = data.frame(
      pc1model = mean(z[ , "Tbp"]) +
                 pca$x[ , 1] * pca$rotation["Tbp", 1],
      actual = z[ , "Tbp"],
      group = sampleAnnotation$group
  )
> tbpPlot = ggplot(tbpData, aes(pc1model, actual, color=group))
> tbpPlot = tbpPlot + geom_point()
> tbpPlot = tbpPlot + scale_color_manual(values=colVals)
> tbpPlot = tbpPlot + xlab("PC1 modeled TBP expression")
> tbpPlot = tbpPlot + ylab("Actual TBP expression")
> print(tbpPlot)
```



The PC1 (only) model (calculated based on Eq (4.12) above) for TBP expression appears to do a pretty job! But perhaps we should be suspicious that this performance may not be totally representative, given that we noted that PC1 split the "TBP" sample group out from the other samples. Indeed, recall that this sample group is defined by the presence of an experimental RNAi transgene for TBP, and indeed we see that expression of the TBP gene itself is quite significantly reduced in this sample group relative to all others.

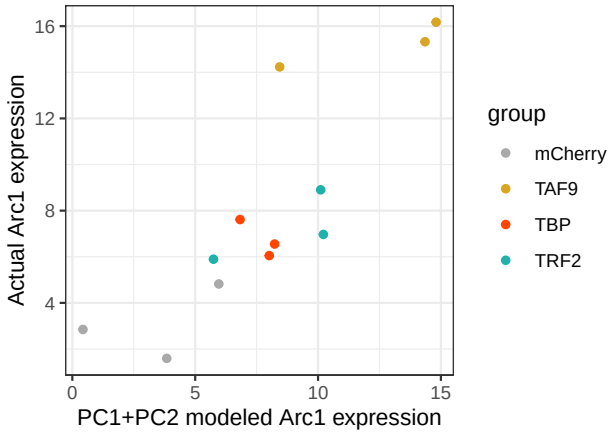
So let's consider a different gene, Arc1:

```
> arc1Data = data.frame(
      pc1model = mean(z[ , "Arc1"]) +
                 pca$x[ , 1] * pca$rotation["Arc1", 1],
      actual = z[ , "Arc1"],
      group = sampleAnnotation$group
  )
> arc1Plot = ggplot(arc1Data, aes(pc1model, actual, color=group))
> arc1Plot = arc1Plot + geom_point()
> arc1Plot = arc1Plot + scale_color_manual(values=colVals)
> arc1Plot = arc1Plot + xlab("PC1 modeled Arc1 expression")
> arc1Plot = arc1Plot + ylab("Actual Arc1 expression")
> print(arc1Plot)
```



So...PC1 model does not do so well this time! It can't, because `pca$x[ , 1]` assigns the most extreme values to samples from the TBP group, while the actual expression levels of Arc1 in the TBP sample group are quite middle-of-the-road. But don't despair, we can always try PC1+PC2 model as defined by Eq (4.13):

```
> arc1Data$pc1and2model = mean(z[ , "Arc1"]) +
                          pca$x[ , 1] * pca$rotation["Arc1", 1] +
                          pca$x[ , 2] * pca$rotation["Arc1", 2]
> arc1Plot = ggplot(arc1Data, aes(pc1and2model, actual, color=group))
> arc1Plot = arc1Plot + geom_point()
> arc1Plot = arc1Plot + scale_color_manual(values=colVals)
> arc1Plot = arc1Plot + xlab("PC1+PC2 modeled Arc1 expression")
> arc1Plot = arc1Plot + ylab("Actual Arc1 expression")
> print(arc1Plot)
```



Maybe not perfect, but definitely much better! In case you're curious as to why I picked Arc1 in particular, it is the gene with the largest magnitude loading on PC2 (as you can confirm by running `which.max(abs(pca$rotation[ , 2]))` if you're so inclined). Thus we might have expected it to be a gene for which the PC1+PC2 model would be notably better than a PC1-only model.

4.3.2 Percent variation explained by PC k

This leads us to a generally ill-understood point in PCA: quantification of how much each PC contributes to modeling the gene expression values. Perhaps you recall the use of R^2 to characterize the quality of fit for linear models (regression, ANOVA, etc.) in statistics; we can use this idea in PCA as well, but we have to decide what to do about the fact that it is a truly *multivariate* model: We are simultaneously modeling thousands of genes!

The general approach taken is more or less the simplest generalization of the R^2 idea available. Recalling that R^2 gives the "percent of variation explained," quantified as the percentage reduction in sum of squared residuals, we first define

$$\sigma_0^2 = \frac{1}{n-1}\sum_{i,g}(z_{ig}-\mu_g)^2 \tag{4.17}$$

$$\sigma_k^2 = \frac{1}{n-1}\sum_{i,g}\left(e_{ig}^{(k)}\right)^2 \tag{4.18}$$

where the k^{th} matrix of error residuals $e_{ig}^{(k}$ is defined by Eq (4.14). We can regard Eq (4.17) as just Eq (4.18) with $k = 0$ if we accept the natural definition

$$e_{ig}^{(0)} = z_{ig} - \mu_g \tag{4.19}$$

for the error residuals of a "no-PC model." Then we can also rewrite Eq (4.14) for any $k > 0$ as as:

$$e_{ig}^{(k)} = e_{ig}^{(k-1)} - x_{ik}r_{gk} \tag{4.20}$$

Let's take a second to do a few of these calculations in R, using the name `resid0[i, g]` to represent $e_{ig}^{(0)}$, `resid1[i, g]` for $e_{ig}^{(1)}$, etc.:

```
> resid0 = t( lgNorm - rowMeans(lgNorm) )  ## matrix of residuals e^(0)_ig
>  ## now use outer function to construct matrix with i, g entry
>  ##  pca$x[i, 1] * pca$rotation[g, 1]
>  ## and subtract this from resid0 to obtain resid1:
> resid1 = resid0 - outer(pca$x[ , 1], pca$rotation[ , 1])
>  ## resid1 contains error residuals e^(1)_ig after fitting PC1-only model
> resid2 = resid1 - outer(pca$x[ , 2], pca$rotation[ , 2])
>  ## resid2 contains error residuals e^(2)_ig from PC1+PC2 model
```

and, using `sigmaSq0` to represent σ_0^2, `sigmaSq1` for σ_1^2, etc.:

```
> n = ncol(lgNorm)  ## number of samples (12)
> sigmaSq0 = sum(resid0^2) / (n-1)
> sigmaSq1 = sum(resid1^2) / (n-1)
> sigmaSq2 = sum(resid2^2) / (n-1)
```

Now the variance explained by PC k is defined to be the overall reduction in variance associated with adding PC k into our model, and is given by

$$\Delta\sigma_k^2 = \sigma_{k-1}^2 - \sigma_k^2 \tag{4.21}$$

and the "fraction of variation" explained by PC k is finally

$$\frac{\Delta\sigma_k^2}{\sigma_0^2} = \frac{\Delta\sigma_k^2}{\sum_{k=1}^{n}\Delta\sigma_k^2} \tag{4.22}$$

where the right-hand side of Eq (4.22) holds because, as we verified above, the variance σ_n^2 remaining after fitting all n PCs is 0; that is

$$\begin{aligned} 0 &= \sigma_n^2 && (4.23)\\ &= \sigma_0^2 - \left(\sigma_0^2 - \sigma_1^2\right) - \left(\sigma_1^2 - \sigma_2^2\right) - \cdots - \left(\sigma_{n-1}^2 - \sigma_n^2\right) && (4.24)\\ &= \sigma_0^2 - \sum_{k=1}^{n}\left(\sigma_{k-1}^2 - \sigma_k^2\right) && (4.25)\\ &= \sigma_0^2 - \sum_{k=1}^{n}\Delta\sigma_k^2 && (4.26)\\ \sigma_0^2 &= \sum_{k=1}^{n}\Delta\sigma_k^2 && (4.27) \end{aligned}$$

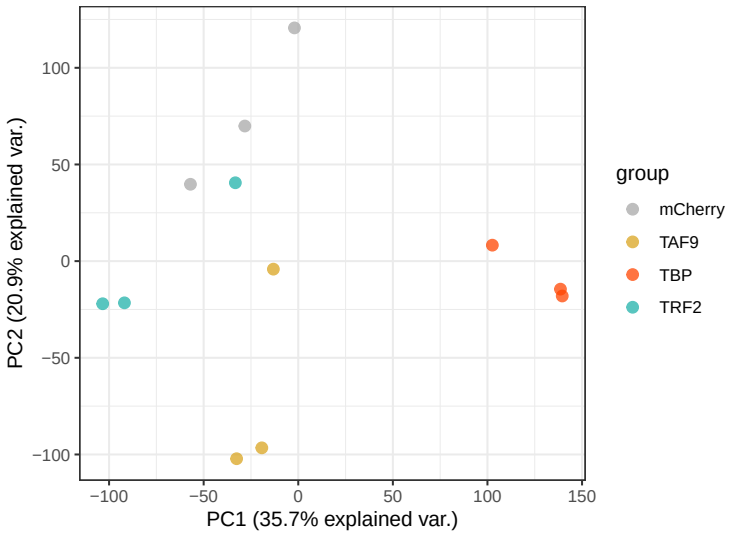
The output `pca` from `prcomp` stores the quantities $\sqrt{\Delta\sigma_k^2}$ in the field `pca$sdev`:

```
>  ## let's compare:
> sqrt(sigmaSq0 - sigmaSq1)
[1] 82.62015
> pca$sdev[[1]]  ## same thing!
[1] 82.62015
>  ## what about the second PC?
> sqrt(sigmaSq1 - sigmaSq2)
```

```
[1] 63.24594
> pca$sdev[[2]]  ## same thing!
[1] 63.24594
>  ## fraction of variation explained by PC1, "from scratch":
> (sigmaSq0 - sigmaSq1) / sigmaSq0
[1] 0.3571321
>  ## fraction of variation explained by PC1, using pca$dev:
> pca$sdev[[1]]^2 / sum(pca$sdev^2)  ## same thing!
[1] 0.3571321
>  ## fraction of variation explained by PC2, from scratch:
> (sigmaSq1 - sigmaSq2) / sigmaSq0
[1] 0.2092774
>  ## fraction of variation explained by PC2, using pca$dev:
> pca$sdev[[2]]^2 / sum(pca$sdev^2)  ## same thing!
[1] 0.2092774
```

So we see that the first PC in the `pca` model explains 35.7% of the overall gene expression variance, while the second PC explains an additional 20.9% of overall gene expression variance. Let's update the axis labels of our PCA plot to include this information:

```
> pctVar = 100 * pca$sdev^2 / sum(pca$sdev^2)
> gg = gg + xlab(
          paste0("PC1 (", round(pctVar[[1]], 1), "% explained var.)"))
> gg = gg + ylab(
          paste0("PC2 (", round(pctVar[[2]], 1), "% explained var.)"))
> print(gg)
```



Keep in mind that these percentages reflect the fraction of variance explained across *all* genes; some genes (e.g., TBP) may be much better explained by PC1 alone than are other genes (e.g., Arc1)!

4.4 Hierarchical clustering

```
> normed = read.table("neves_deseq_normalized.tsv.gz",
                      sep="\t", header=TRUE, row.names=1)
>  ## simplify data by removing genes with no data:
> normed = normed[rowSums(normed) > 0, ]  ## logical indexing...
>  ## log-transform data to reduce heteroskedasticity:
> lgNorm = log2(normed + 1)   ## log2 works element-wise on data.frame
> sampleAnnotation = data.frame(
       ## set row.names of sampleAnnotation to match colnames of normed:
      row.names = colnames(normed),
       ## use gsub to remove single digit \\d right before end $
       ## of sample names to get group name for each sample:
      group = gsub("_\\d$", "", colnames(normed))
  )
```

There are many other "unsupervised" approaches to computational analysis of highly multivariate data sets such as those obtained via RNA-sequencing experiments. Probably the most popular is agglomerative hierarchical clustering (Mary-Huard *et al.* (2006); Hastie *et al.* (2009)), which we will study here.

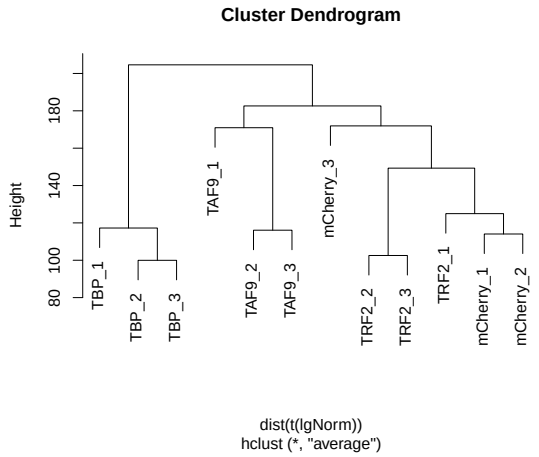
Hierarchical clustering approaches are so named because they seek to generate a hierarchy—generally represented as *dendrogram*, a structure to be discussed shortly—of clusterings of the data.

By "clustering" I mean here a partition of objects (e.g. samples) to discrete clusters, such as may be compactly represented by assigning a single cluster label to each object. Objects which are assigned the same cluster label are then considered to be in the same cluster, while objects with different cluster labels are in different clusters.

A hierarchy of clusterings is then a set of clusterings with, at the lowest level, n distinct clusters—so that no two objects are assigned to the same cluster—followed by a clustering with $n-1$ clusters, in which exactly two objects are assigned to the same cluster, and then a clustering with $n-2$ clusters, and so on, until finally the top level has only one cluster to which all n objects are assigned.

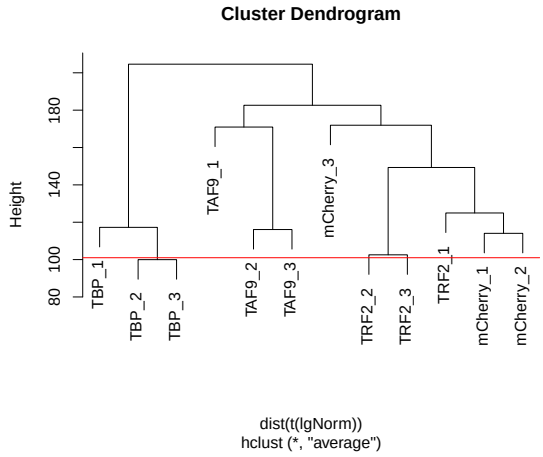
Each level of the hierarchy also must be consistent with the level below it: this means that the clustering with $m < n$ clusters must be the result of taking the clustering with $m+1$ clusters and merging two of those $m+1$ clusters together into one. This constraint is what makes it possible to represent the hierarchy with a dendrogram; let's consider an example:

```
>  ## use hclust function to perform hierarchical clustering in R
> sampleClust = hclust(
      dist(t(lgNorm)),    ## will discuss both of these arguments
      method = "average"  ## below!
  )
>  ## can use generic plot function to generate dendrogram from hclust
> plot(sampleClust)
```



The different clusterings correspond to different vertical levels in this dendrogram. At the very bottom—below all of the lines—each of the samples are assigned to its own cluster. Then, at the level indicated by the red line here:

```
> plot(sampleClust)
> abline(h=101, col="red")  ## draw red horizontal line at y=101
```



we have joined the two samples TBP_2 and TBP_3 together into a single cluster, since the lines connecting these two samples are below the red line, while each of the other 10 samples is still assigned to its own cluster.

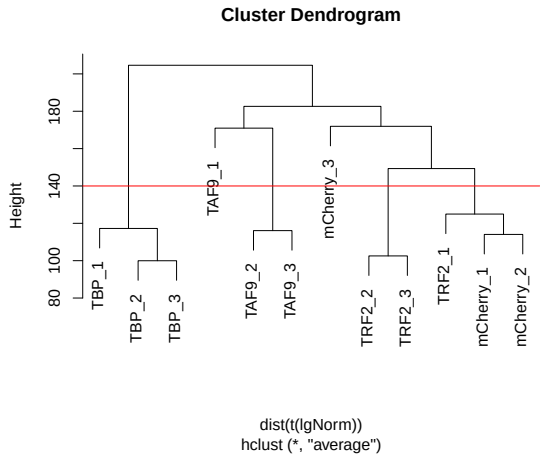
We can also extract the cluster identities directly in R without bothering to look at plots:

```
> cutree(sampleClust, h=101)  ## cut tree at height 101
```

```
mCherry_1 mCherry_2 mCherry_3    TAF9_1    TAF9_2    TAF9_3     TBP_1     TBP_2
        1         2         3         4         5         6         7         8
    TBP_3    TRF2_1    TRF2_2    TRF2_3
        8         9        10        11
>  ## output is vector containing sample cluster labels
>  ## note TBP_2 and TBP_3 are both assigned to cluster 8,
>  ## while all other samples get their own cluster id number
```

Alternatively, if we try a different height cutoff:

```
> cutree(sampleClust, h=140)
mCherry_1 mCherry_2 mCherry_3    TAF9_1    TAF9_2    TAF9_3     TBP_1     TBP_2
        1         1         2         3         4         4         5         5
    TBP_3    TRF2_1    TRF2_2    TRF2_3
        5         1         6         6
> plot(sampleClust)
> abline(h=140, col="red")
```



We find:

- on the far left, cluster **5**, containing TBP_1, TBP_2, and TBP_3, then
- cluster **3** containing only TAF9_1,
- cluster **4** containing TAF9_2 and TAF9_3,
- cluster **2** containing only mCherry_3,
- cluster **6** containing TRF_2 and TRF_3, and, finally,
- on the far right, cluster **1** containing TRF2_1 along with mCherry_1 and mCherry_2.

Often when we want a specific clustering, we want to specify the number of clusters instead of trying to figure out what height to cut at; this can be done with `cutree` using the `k` argument instead of the `h` argument:

```
> cutree(sampleClust, k=6)  ## generates same 6 clusters as h=140 did
```

```
mCherry_1 mCherry_2 mCherry_3    TAF9_1    TAF9_2    TAF9_3     TBP_1     TBP_2
        1         1         2         3         4         4         5         5
    TBP_3    TRF2_1    TRF2_2    TRF2_3
        5         1         6         6
```

4.4.1 Dissimilarity metrics

When we first ran `hclust`, we supplied two arguments; the first of these was `dist(t(lgNorm))`. The `t(lgNorm)` part of this simply takes the transpose of `lgNorm`, but we haven't seen the function `dist` before, so let's take a look:

```
> dist(t(lgNorm))
          mCherry_1 mCherry_2 mCherry_3    TAF9_1    TAF9_2    TAF9_3     TBP_1
mCherry_2 114.05621
mCherry_3 156.40909 138.69538
TAF9_1    177.73120 170.73140 195.12139
TAF9_2    175.29114 184.64995 222.16949 162.48501
TAF9_3    169.31009 189.23000 225.06567 179.41812 116.08768
TBP_1     181.25260 169.87392 185.82754 187.83986 187.79612 191.69510
TBP_2     211.06401 197.27819 212.58982 215.71942 199.15185 206.46457 116.54055
TBP_3     213.06247 199.04232 213.69624 216.25712 197.69429 206.42932 117.96770
TRF2_1    129.74618 120.17321 158.73412 162.88573 167.93500 181.93795 175.26441
TRF2_2    133.37494 151.54456 198.82353 199.10996 166.43727 154.77274 207.15915
TRF2_3    147.60022 161.45168 206.99668 203.50251 172.84402 168.45028 219.21606
              TBP_2     TBP_3    TRF2_1    TRF2_2
mCherry_2
mCherry_3
TAF9_1
TAF9_2
TAF9_3
TBP_1
TBP_2
TBP_3      99.96386
TRF2_1    198.60278 198.59828
TRF2_2    227.44316 228.61060 149.83309
TRF2_3    238.37997 238.75807 152.04711 102.55346
```

What we've done here is to compute the *Euclidean distances* of each of the 12 samples from each of the other 11 samples. The Euclidean distance is defined here as in geometry as the square root of the sum of the squared differences in each coordinate of a vector; since this is more easily comprehended via math or code than English words,

```
> coordinateDifferenceSample1Sample2 = lgNorm[ , 1] - lgNorm[ , 2]
> sqrt( sum( coordinateDifferenceSample1Sample2^2 ) )
[1] 114.0562
>  ## results in same value as
> as.matrix(dist(t(lgNorm)))[1, 2]
[1] 114.0562
```

We want these distances here as a way to measure how dissimilar one sample's expression profile is from another (Euclidean distance is not the only dissimilarity metric which can be used with `hclust`; you can consult the help documentation for `dist` function and its own `method` argument to see what other options are available). The agglomerative hierarchical clustering algorithm implemented by `hclust` uses these quantified dissimarilities between pairs of samples to decide, at each step, which two clusters to join together from the clustering with `m+1` clusters to form the clustering with `m` clusters.

This is easiest to do in the first step, where we start with every sample in its own cluster. In this case, we just pick the two samples with the smallest dissimilarity value (in this case, TBP_2 and TBP_3, with a dissimilarity score of 99.96 between them) and join them together into a cluster.

4.4.2 Agglomeration linkage methods

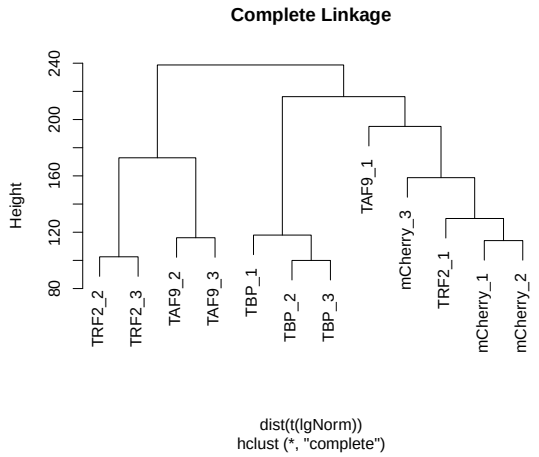
After we've created a cluster containing two separate objects a new problem confronts us: How do we decide in the next round of clustering whether to join together two singleton objects (objects which are in their own cluster not containing any other objects) or instead to join a singleton object into the two-object cluster we created in our first step?

We need a way to assign numeric dissimilarities between *clusters* of objects based on the numeric dissimilarities we've already calculated between individual objects. In the example I've constructed here, I've done this using a very simple approach: The dissimilarity between cluster A and cluster B is defined to be the average of the dissimarilities between all pairs of objects we can form taking one object from cluster A and the other object in the pair from cluster B. This is the meaning of the code `method = "average"` in the `hclust` call above.

(Note: we supplied the second argument to `hclust` using the argument name `method`; keep in mind this is distinct from the `method` argument of the `dist` function referenced above—different functions can use the same argument names to mean different things, since they operate in different scopes.)

This way of defining dissimilarities between clusters based on dissimilarities between objects is known as "average linkage." Many alternatives exist; one particularly common one (the default `method` for `hclust`) is "complete linkage." Complete linkage defines the dissimilarity between cluster A and cluster B as the largest dissimilarity value for all pairs of objects taken one from A and the other from B:

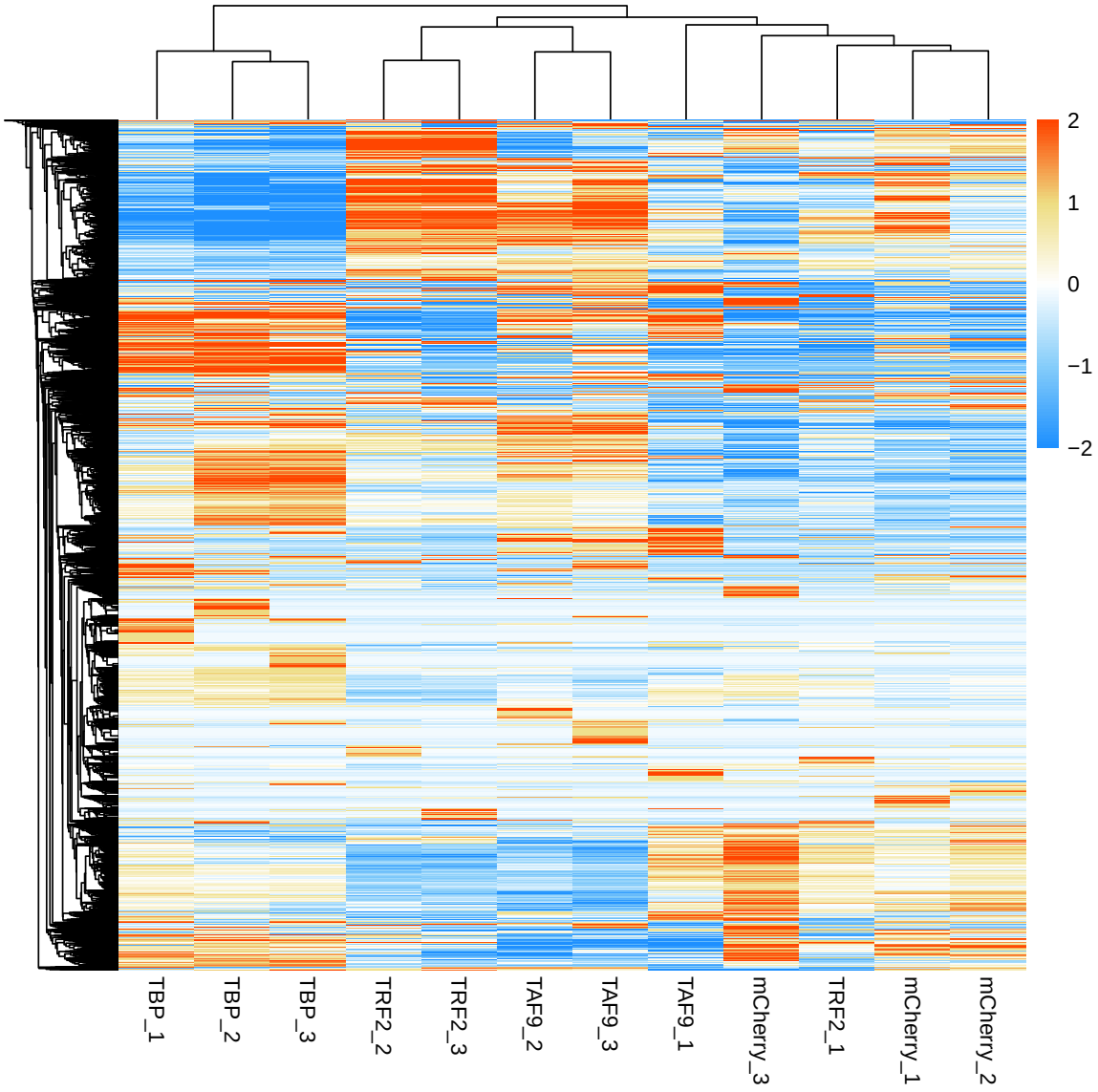
```
> sampleClust2 = hclust(dist(t(lgNorm)), method="complete")
> plot(sampleClust2, main="Complete Linkage")
```



With complete linkage we can see that at the higher levels of the dendrogram we obtain different clusterings than we did with average linkage. In particular, with average linkage the three samples TBP_1, TBP_2, and TBP_3—are the last to be merged together with the remainder of the sample set, while with complete linkage this is not the case.

4.4.3 Clustered heatmaps

```
> ## install.packages("pheatmap")  ## uncomment and run if necessary
> library(pheatmap)
>  ## usually most interested in expression levels relative to mean:
> heatData = lgNorm - rowMeans(lgNorm)
>  ## often want to limit dynamic range heatmap considers so that
>  ## color palette range is not dominated by a few extreme values:
> heatData[heatData > 2] = 2; heatData[heatData < -2] = -2
>  ## pheatmp is not a grammar-of-graphics style plotting function:
>  ## specify all options as arguments to single function call
>  ## intead of building plot up in modular fashion:
> pheatmap(
      heatData,
      color = colorRampPalette(c(
          "dodgerblue", "lightskyblue",
          "white",
          "lightgoldenrod", "orangered"
      ))(100),
      clustering_method = "average",
      show_rownames = FALSE
  )
```


2
1
0
−1
−2
TBP_1
TBP_2
TBP_3
TRF2_2
TRF2_3
TAF9_2
TAF9_3
TAF9_1
mCherry_3
TRF2_1
mCherry_1
mCherry_2

Bibliography

Altschul, Stephen F, Gish, Warren, Miller, Webb, Myers, Eugene W, & Lipman, David J. 1990. Basic Local Alignment Search Tool. *Journal of Molecular Biology*, **215**(3), 403–410.

Ambrosini, Giovanna, Groux, Romain, & Bucher, Philipp. 2018. PWMScan: a fast tool for scanning entire genomes with a position-specific weight matrix. *Bioinformatics*, **34**(14), 2483–2484.

Bailey, Timothy L, Williams, Nadya, Misleh, Chris, & Li, Wilfred W. 2006. MEME: discovering and analyzing DNA and protein sequence motifs. *Nucleic Acids Research*, **34**(suppl_2), W369–W373.

Benjamini, Yoav, & Hochberg, Yosef. 1995. Controlling the false discovery rate: a practical and powerful approach to multiple testing. *Journal of the Royal Statistical Society: Series B (Methodological)*, **57**(1), 289–300.

Berliner, L Mark. 1992. Statistics, probability and chaos. *Statistical Science*, 69–90.

Burrows, Michael, & Wheeler, David J. 1994. A block-sorting lossless data compression algorithm.

Cavalcante, Raymond G, & Sartor, Maureen A. 2017. annotatr: genomic regions in context. *Bioinformatics*, **33**(15), 2381–2383.

Dillies, Marie-Agnès, Rau, Andrea, Aubert, Julie, Hennequet-Antier, Christelle, Jeanmougin, Marine, Servant, Nicolas, Keime, Céline, Marot, Guillemette, Castel, David, Estelle, Jordi, *et al.* . 2013. A comprehensive evaluation of normalization methods for Illumina high-throughput RNA sequencing data analysis. *Briefings in Bioinformatics*, **14**(6), 671–683.

Doob, Joseph L. 1945. Markoff chains–denumerable case. *Transactions of the American Mathematical Society*, **58**(3), 455–473.

Durbin, Richard, Eddy, Sean R, Krogh, Anders, & Mitchison, Graeme. 1998. *Biological sequence analysis: probabilistic models of proteins and nucleic acids*. Cambridge University Press.

Ferragina, Paolo, & Manzini, Giovanni. 2000. Opportunistic data structures with applications. *Pages 390–398 of: Proceedings 41st Annual Symposium on Foundations of Computer Science*. IEEE.

Fornes, Oriol, Castro-Mondragon, Jaime A, Khan, Aziz, vanÂăderÂăLee, Robin, Zhang, Xi, Richmond, Phillip A, Modi, Bhavi P, Correard, Solenne, Gheorghe, Marius, BaranaÅąiÄĞ, Damir, Santana-Garcia, Walter, Tan, Ge, ChÃĺneby, Jeanne, Ballester, Benoit, Parcy, FranÃğois, Sandelin, Albin, Lenhard, Boris, Wasserman, Wyeth W, & Mathelier, Anthony. 2019. JASPAR 2020: update of the open-access database of transcription factor binding profiles. *Nucleic Acids Research*, **48**(D1), D87–D92.

Gause, Georgii Frantsevich. 1932. Experimental studies on the struggle for existence: I. Mixed population of two species of yeast. *Journal of Experimental Biology*, **9**(4), 389–402.

Gillespie, Daniel T. 1977. Exact stochastic simulation of coupled chemical reactions. *The Journal of Physical Chemistry*, **81**(25), 2340–2361.

Hastie, Trevor, Tibshirani, Robert, & Friedman, Jerome. 2009. *The Elements of Statistical Learning*. Springer.

Izenman, Alan Julian. 2008. *Modern Multivariate Statistical Techniques*. Springer.

Kärkkäinen, Juha, & Sanders, Peter. 2003. Simple linear work suffix array construction. *Pages 943–955 of: International Colloquium on Automata, Languages, and Programming*. Springer.

Levenshtein, Vladimir I. 1966. Binary codes capable of correcting deletions, insertions, and reversals. *Pages 707–710 of: Soviet Physics Doklady*, vol. 10.

Liu, DX, & Lobie, PE. 2007. Transcriptional activation of p53 by Pitx1. *Cell Death & Differentiation*, **14**(11), 1893–1907.

Lotka, Alfred J. 1920. Analytical note on certain rhythmic relations in organic systems. *Proceedings of the National Academy of Sciences*, **6**(7), 410–415.

Love, Michael I, Huber, Wolfgang, & Anders, Simon. 2014. Moderated estimation of fold change and dispersion for RNA-seq data with DESeq2. *Genome Biology*, **15**(12), 550.

Manber, Udi, & Myers, Gene. 1993. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, **22**(5), 935–948.

Mary-Huard, Tristan, Picard, Franck, & Robin, Stéphane. 2006. Introduction to statistical methods for microarray data analysis. *Mathematical and Computational Methods in Biology. Paris: Hermann*.

May, Robert M. 1976. Simple mathematical models with very complicated dynamics. *Nature*, **261**(5560), 459–467.

Needleman, Saul B, & Wunsch, Christian D. 1970. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, **48**(3), 443–453.

Neves, Alexandre, & Eisenman, Robert N. 2019. Distinct gene-selective roles for a network of core promoter factors in Drosophila neural stem cell identity. *Biology Open*, **8**(4), bio042168.

Roweis, Sam, & Ghahramani, Zoubin. 1999. A unifying review of linear Gaussian models. *Neural Computation*, **11**(2), 305–345.

Schavemaker, Paul E, Boersma, Arnold J, & Poolman, Bert. 2018. How important is protein diffusion in prokaryotes? *Frontiers in Molecular Biosciences*, **5**, 93.

Smith, Temple F, & Waterman, Michael S. 1981. Comparison of biosequences. *Advances in Applied Mathematics*, **2**(4), 482–489.

Stewart, Alexander J, Hannenhalli, Sridhar, & Plotkin, Joshua B. 2012. Why transcription factor binding sites are ten nucleotides long. *Genetics*, **192**(3), 973–985.

Strogatz, Steven H. 2015. *Nonlinear Dynamics and Chaos: With Applications to Physics, Biology, Chemistry, and Engineering*. CRC press.

Verhulst, Pierre-Francois. 1838. Notice sur la loi que la population suit dans son accroissement. *Correspondance MathÃl'matique et Physique*, **10**, 113–121.

Wu, Jing, & Xie, Jun. 2010. Hidden Markov model and its applications in motif findings. *Pages 405–416 of: Statistical Methods in Molecular Biology*. Springer.

Yoon, Byung-Jun. 2009. Hidden Markov models and their applications in biological sequence analysis. *Current Genomics*, **10**(6), 402–415.