Bio720 - Simulating Data

Ian Dworkin

11/19/2018

Contents

Introduction

One of the most important skills in your bag as new computational biologists is the ability to perform simulations. In particular, to simulate data or evaluate models numerically (or both).

Introduction

In biology mathematical models are the basis for much of the theoretical and conceptual background in disciplines like ecology, evolution, population genetics, molecular evolution & biochemistry.

Introduction

Many of the models that are developed are not *analytically* tractable. That is, without making very strong biological assumptions there are no *closed form solutions*.

That is the models can not be solved for the general case.

Using computers to find numerical solutions

- For such models, "solutions" can still be found.
- For some models, stability analysis can be performed (among other techniques).
- However, most often, scientists resort to computers to identify *numerical solutions*

Deterministic VS. Stochastic

- Within *dynamical* models, that include the majority of models in biology there are two broad categories.
- **Deterministic** - where the outcome of the model entirely depends on the model itself and the starting conditions.
- **Stochastic** - where random events influence the outcomes of the model, and only the probability of events can be predicted, not exact outcomes.

A simple deterministic model one-locus model of natural selection.

- In population genetics we often start with a simple deterministic model of how selection on an allele influences its frequency in the population over time.
- In a simple haploid model we can envision two alternative alleles, A and a.
- a is initially fixed in the population, but the new mutation A arrives and it is beneficial. What will happen to this allele?

Haploid selection model

- Frequency of A at time t is $p(t)$
- Fitness of A is W_A, and for a is W_a

$p(t+1) = \frac{p(t)W_A}{p(t)W_A + W_a(1-p(t))}$

$p(t+1) = \frac{p(t)W_A}{\overline{W}}$

- Where $\overline{W}$ is mean fitness for the population
- How would you convert this into R code for one generation to the next?
- Start with what variables you need.

Converting this into R code - What variables

- We need variables for the fitness values for each allele, W_A, W_a and for allele frequency of A $p(t+1)$ at time t and t+1.
- With this we can write the equation for allele frequency change from one generation to the next.

Converting this into R code - What variables

```
p_t1 <- function(w_A, w_a, p_t0) {
        w_bar <- (p_t0*w_A) + ((1-p_t0)*w_a) # mean pop fitness
        p_t1 <- (w_A*p_t0)/w_bar
        return(p_t1)}

p_t1(w_A = 1.1, w_a = 1.0, p_t0 = 0.5)
```

```
## [1] 0.5238095
```

Is this deterministic or stochastic, what would be a quick way to check?

Is this deterministic or stochastic, what would be a quick way to check?

```
replicate(n = 100, p_t1(1.1, 1.0, 0.5))
```

```
##   [1] 0.5238095 0.5238095 0.5238095 0.5238095 0.5238095 0.5238095 0.5238095
##   [8] 0.5238095 0.5238095 0.5238095 0.5238095 0.5238095 0.5238095 0.5238095
##  [15] 0.5238095 0.5238095 0.5238095 0.5238095 0.5238095 0.5238095 0.5238095
##  [22] 0.5238095 0.5238095 0.5238095 0.5238095 0.5238095 0.5238095 0.5238095
##  [29] 0.5238095 0.5238095 0.5238095 0.5238095 0.5238095 0.5238095 0.5238095
```

```
##  [36] 0.5238095 0.5238095 0.5238095 0.5238095 0.5238095 0.5238095 0.5238095
##  [43] 0.5238095 0.5238095 0.5238095 0.5238095 0.5238095 0.5238095 0.5238095
##  [50] 0.5238095 0.5238095 0.5238095 0.5238095 0.5238095 0.5238095 0.5238095
##  [57] 0.5238095 0.5238095 0.5238095 0.5238095 0.5238095 0.5238095 0.5238095
##  [64] 0.5238095 0.5238095 0.5238095 0.5238095 0.5238095 0.5238095 0.5238095
##  [71] 0.5238095 0.5238095 0.5238095 0.5238095 0.5238095 0.5238095 0.5238095
##  [78] 0.5238095 0.5238095 0.5238095 0.5238095 0.5238095 0.5238095 0.5238095
##  [85] 0.5238095 0.5238095 0.5238095 0.5238095 0.5238095 0.5238095 0.5238095
##  [92] 0.5238095 0.5238095 0.5238095 0.5238095 0.5238095 0.5238095 0.5238095
##  [99] 0.5238095 0.5238095
```

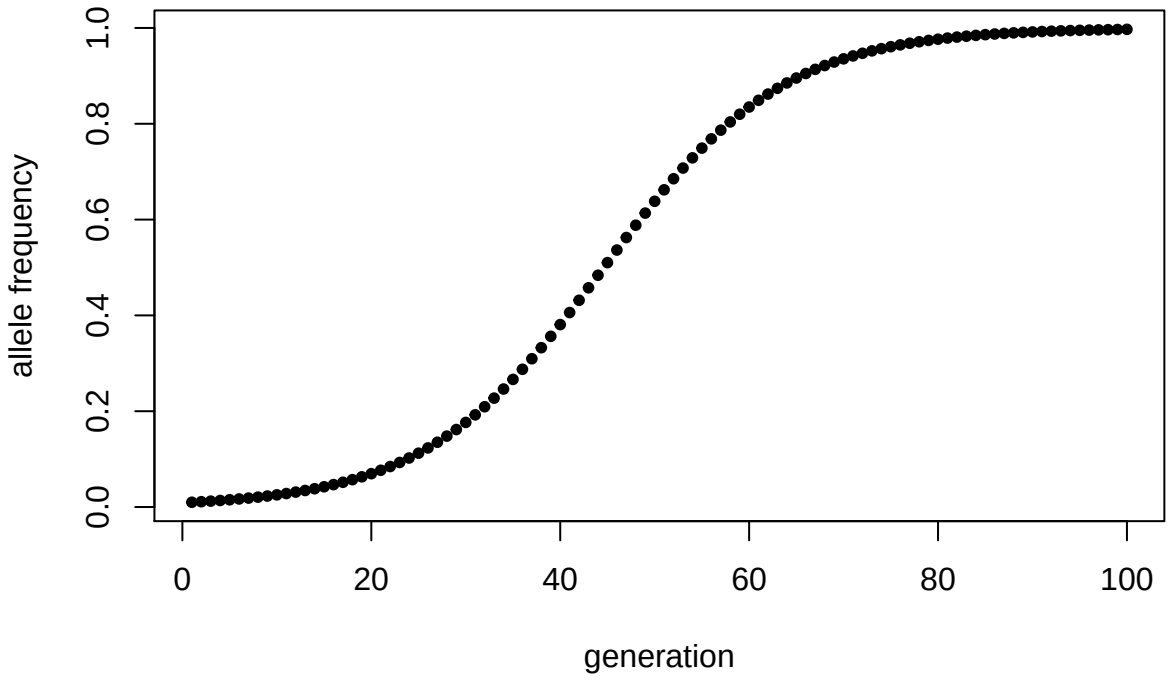
Allele frequency dynamics.

- Now let's extend this across many generations. We need to rewrite the function as we expect the allele frequencies to change each generation. Also, mean population fitness will change each generation (as allele frequency changes)
- write this simulation (a *for loop* would be sensible) and go for 200 generations, and look at the dynamics (starting allele frequency is p = 0.01). Wrap it all as a function called `haploid_selection`

```
haploid_selection <- function(p0 = 0.01, w1 = 1, w2 = 0.9, n = 100) {

    # Initialize vectors to store allele frequencies and mean pop fitness
    p <- rep(NA,n)  # a vector to store allele frequencies

    w_bar <- rep(NA, n)

    # starting conditions
    p[1] <- p0 # starting allele frequencies

    w_bar[1] <- (p[1]*w1) + ((1-p[1])*w2)

    # now we need to loop from generation to generation
    for ( i in 2:n) {
        w_bar[i - 1] <- (p[i - 1]*w1) + ((1-p[i - 1])*w2) # mean population fitness
        p[i] <- (w1*p[i - 1])/w_bar[i - 1]
    }

    return(p)
}
```

Test the model and plot it.

```
p <- haploid_selection()
generations <- 1:length(p)
plot(p ~ generations, pch = 20,
     ylab = "allele frequency",
     xlab = "generation")
```



Using simple numerical simulation to gain intuition for the system.

- Try altering fitness advantage of A a little bit. Or reducing initial allele frequency
- Is this deterministic or stochastic?

Making a more general function

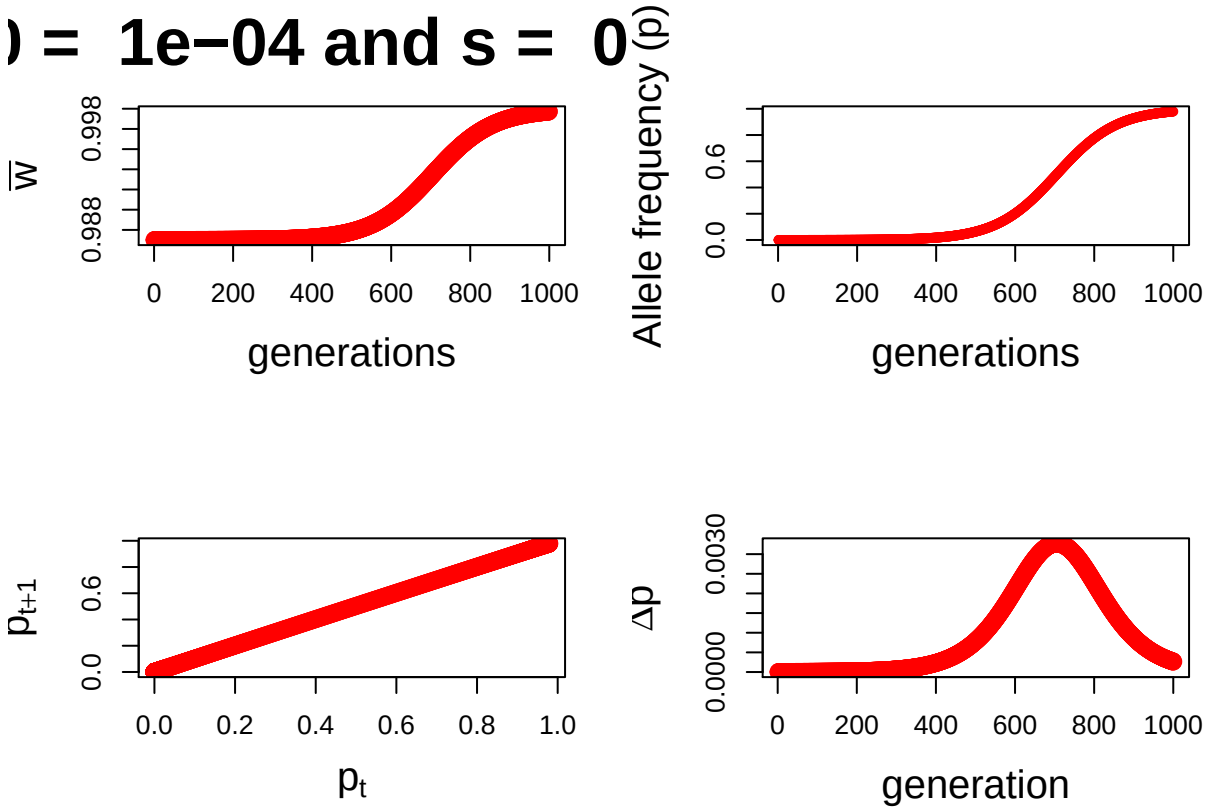
```
haploid.selection <- function(p0 = 0.01, w1 = 1, w2 = 0.9, n = 100) {

# Initialize vectors to store p, delta p and mean pop fitness

p <- rep(NA,n)

delta_p <- rep(NA, n)

w_bar <- rep(NA, n)

# starting conditions
p[1] <- p0 # starting allele frequencies

delta_p[1] <- 0 #change in allele frequency

w_bar[1] <- (p[1]*w1) + ((1-p[1])*w2)

# now we need to loop from generation to generation
for ( i in 2:n) {
    w_bar[i - 1] <- (p[i - 1]*w1) + ((1-p[i - 1])*w2)
        p[i] <- (w1*p[i - 1])/w_bar[i - 1]
        delta_p[i] <- p[i] - p[i-1]
```

```
    }

    if (any(p > 0.9999)) {
    fixation <- min(which.max(p > 0.9999))
    cat("fixation for A1 occurs approximately at generation:", fixation )
    } else {
        maxAlleleFreq <- max(p)
        cat("fixation of A1 does not occur, max. allele frequency is:", print(maxAlleleFreq, digits = 2)
    }

# Let's make some plots
par(mfrow=c(2,2))

# 1. mean population fitness over time
plot(x = 1:n, y = w_bar,
     xlab = "generations",
     ylab = expression(bar(w)),
     pch=20, col="red", cex = 2, cex.lab = 1.5, cex.main = 2.5,
     main = paste("p0 = ", p0, "and s = ", (1 - (w2/w1))))

    # 2. change in allele frequency over time
plot(x = 1:n, y = p,
     xlab="generations",
     ylab="Allele frequency (p)",
     pch = 20, col = "red", cex.lab = 1.5)

    # 3. plot of p[t+1] vs p[t]
p.1 <- p[-n]
p.2 <- p[-1]

plot(p.2 ~ p.1,
     xlab = expression(p[t]),
     ylab = expression(p[t+1]),
     pch = 20, col = "red", cex = 2, cex.lab = 1.5)

# 4. plot of allele frequency change
plot(x = 2:n, y = delta_p[-1],
        xlab = "generation",
        ylab= expression(paste(Delta,"p")),
        pch = 20, col = "red", cex = 2, cex.lab = 1.5)
}
```

Let's see what this does.

```
haploid.selection(p0 = 0.0001, w1 = 1, w2 = 0.987, n = 1000)
```

```
## [1] 0.98
## fixation of A1 does not occur, max. allele frequency is: 0.9794053
```



Stochastic simulations

- The real power for your computational skills comes from performing stochastic simulations in a computer.
- That is adding some sources of random variation at various places in your model, and allowing it to *propogate* and influence your outcomes.

Is it really random?

- Computers can not do true random numbers.
- Instead they use *pseudo-random* number generation.
- The details of how they do it, don't matter at the moment.
- What does matter is the effect it has.

Is it really random?

- try running this code and compare your answers with the person next to you.

```
x <- rnorm(1000, mean = 5, sd = 1)
mean(x)
```

```
## [1] 4.979784
```

```
x <- rnorm(1000, mean = 5, sd = 1)
mean(x)
```

```
## [1] 4.979419
```

```
x <- rnorm(1000, mean = 5, sd = 1)
mean(x)
```

```
## [1] 4.955695
```

Is it really random?

Now try it again, but let's use the *seed* for the random number generator

```
set.seed(720)
x <- rnorm(1000, mean = 5, sd = 1)
mean(x)
```

```
## [1] 5.01245
```

```
x <- rnorm(1000, mean = 5, sd = 1)
mean(x)
```

```
## [1] 4.99981
```

```
x <- rnorm(1000, mean = 5, sd = 1)
mean(x)
```

```
## [1] 4.941751
```

resetting the seed to the same number each time.

```
set.seed(720)
x <- rnorm(1000, mean = 5, sd = 1)
mean(x)
```

```
## [1] 5.01245
```

```
set.seed(720)
x <- rnorm(1000, mean = 5, sd = 1)
mean(x)
```

```
## [1] 5.01245
```

```
set.seed(720)
x <- rnorm(1000, mean = 5, sd = 1)
mean(x)
```

```
## [1] 5.01245
```

```
rm(x)
```

Can you explain what is happening?

- Why did you get different results the first time, the same sequence of results the second time and the exact same results the third time?

Is it really random?

- No, but as long as we are aware of it, this can be very useful to check our work.

- If you don't specifically set the seed for random number generation, it defaults to the current time and date.

rolling the dice.

- let's say we want to simulate the rolling of a regular 6-sided die. How would we accomplish this?
- *R* has the `sample()` function.

```
sample(x, size, replace = FALSE, prob = NULL)
```

- where `x` is the object that you are sampling from.
- `size` is how many samples you want from the object `x`
- `replace` is whether you want to allow sampling with replacement. i.e. with a jar full of jellybeans, do you take one out, record its colour and then put it back before shaking the jar and selecting the next colour? Or once you take it out, you can not select it again (`replace = FALSE`).
- `prob` will be explained a bit later.

rolling the dice

- let's try this. A 6 sided die can only land on one face, between 1 and 6 (`1:6`).

So one roll of the die

```
sample(1:6, size = 1, replace = TRUE)
```

```
## [1] 3
```

- Run this a few times.

rolling the die many times.

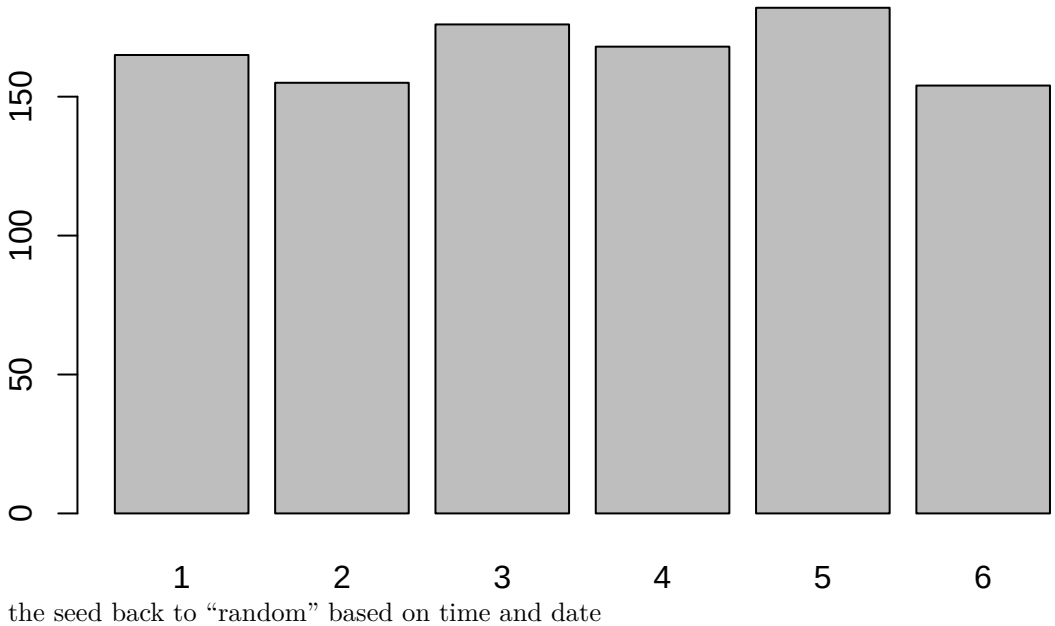
- use the replicate function to repeat this roll of the die 100 times.
- plot the distribution of rolls (how many 1's, 2's etc..)

rolling the die many times.

- use the replicate function to repeat this roll of the die 1000 times.

```
rolls <- replicate(1000, sample(1:6, size = 1, replace = TRUE))
```

```
barplot(table(rolls))
```



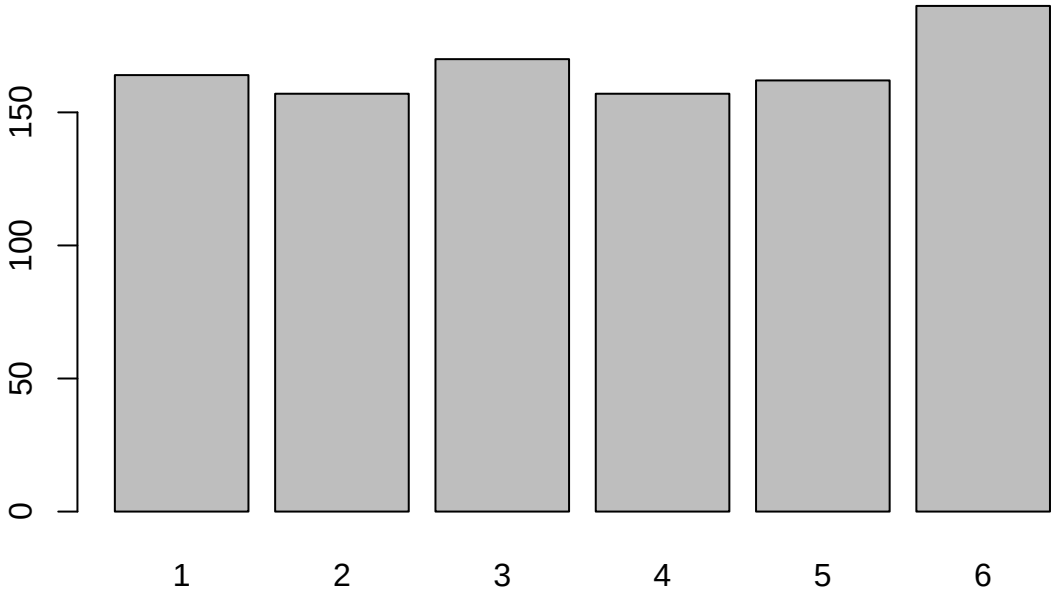
Resetting the seed back to "random" based on time and date

```
set.seed(NULL)
```

A better way to roll the die

- We don't need to invoke the replicate function at all.
- We can simply change the `size` argument in `sample()`.

```
more_rolls <- sample(1:6, size = 1000, replace = TRUE)
barplot(table(more_rolls))
```



what happens with `replace = FALSE`?

```
sample(1:6, size = 1000, replace = FALSE)
```

```
## Error in sample.int(length(x), size, replace, prob): cannot take a sample larger than the population
```

- Why?

what happens with `replace = FALSE`?

```
sample(1:6, size = 1000, replace = FALSE)
```

```
## Error in sample.int(length(x), size, replace, prob): cannot take a sample larger than the population
```

- Why? Because there are only 6 values, so it is sampling them randomly, but removing them after. So we can only sample 6 times for a vector of length 6.

How to use `replace = FALSE`

- This is particularly useful to get a different permutation of the order that you take the jelly beans out, but with the same sample of jelly beans.
- Try it a few times with the dice example.

```
sample(1:6, size = 6, replace = FALSE)
```

```
## [1] 2 5 6 4 3 1
```

```
sample(1:6, size = 6, replace = FALSE)
```

```
## [1] 5 4 1 3 2 6
```

```
sample(1:6, size = 6, replace = FALSE)
```

```
## [1] 2 1 6 4 3 5
```

How is the sample function useful?

- There are many times that sampling with or without replacement is extremely useful, and forms the basis for an important computationally intensive statistical approach, known as *empirical resampling methods* these include:
- randomization/permutation tests (sampling without replacement). Essentially shuffling the order of observations relative to a treatment variable (at its simplest).
- non-parametric bootstrap (sampling with replacement), is a common method to determine standard errors and confidence intervals on estimated quantities based on properties of the observed distribution.
- Bio708 may introduce you to these ideas, but I have a series of youtube screencasts on it if you want to know more.

Using the sample function to model genetic drift

- Genetic drift, a change in allele frequencies not due to natural selection, but to stochastic sampling of which alleles are transmitted from one generation to the next.
- It is particularly important in small populations, and has important implications in species with conservation concerns.

- Imagine we have two alleles at a given genetic locus, "A" and "a", in a population of diploids (each individual has 2 alleles).
- There are a total of 20 individuals, and anyone can mate with anyone else.
- population size stays constant (20 offspring for a total of 40 alleles).
- If the initial allele frequency for A and a are equal ($f(A) = f(a) = 0.5$), can you write a simulation to see how allele frequencies may change from one generation to the next?
- What are the allele frequencies in the next generation?
- Things to consider:
- Obviously using `sample(x, size, replace)`
- What is your vector of observations in this case (what do we want to sample from)?
- How many alleles do we want in the next generation, and which argument allows for this?
- Do we want `replace = TRUE` or `replace = FALSE`?
- Do you expect to get the same result as the person next to you? Why or why not?

Using the sample function to model genetic drift

```
allele_counts <- sample(c("A", "a"),
                        size = 40,
                        replace = TRUE,
                        prob = NULL)

table(allele_counts)/length(allele_counts)
```

```
## allele_counts
##   a   A
## 0.5 0.5
```

How about if we wanted to go one more generation?

- Now our allele frequencies $f(A) \neq f(A)$ so we have to let our function know that the probability of choosing the "a" and "A" allele differs.
- This is where the `prob` argument becomes very important.
- instead of assuming each element of your input vector (in this case alleles "A" and "a") are sampled with equal probability, we can sample them according to their allele frequencies.

```
allele_counts <- sample(c("a", "A"),
                        size = 40,
                        replace = TRUE,
                        prob = c(0.5, 0.5))

allele_freq1 <- table(allele_counts)/length(allele_counts)
allele_freq1
```

```
## allele_counts
##    a    A
## 0.65 0.35
```

```
allele_counts2 <- sample(c("a", "A"),
                         size = 40,
                         replace = TRUE,
                         prob = allele_freq1)

allele_freq2 <- table(allele_counts2)/length(allele_counts2)
allele_freq2
```

```
## allele_counts2
##     a     A
## 0.725 0.275
```

How might you generalize this function to allow you to alter population size, starting allele frequency and arbitrary number of generations?

- This will be one of the questions on the assignment

Numerical stochastic simulations, the basics.

At the heart of stochastic simulations, is needing some sort of probability distribution to simulate from. The easiest one of course is the *uniform distribution*. The idea is to take random draws from this distribution upon repeated sampling. The set of functions we will use all start with lower case *r*, for random, like `rnorm()`, `runif`, `rbinom()`, etc...

runif

- Let's start with generating a single random number from a uniform distribution on [0,1]

```
runif(n = 1, min = 0, max = 1)
```

```
## [1] 0.2494453
```

- the rNameOfDist functions generate random numbers.
- min sets lowest, max sets highest possible value, n is the number of draws from this distribution.
- We can actually use the `runif` to generate random draws from all other distributions if we wanted to, but it is unnecessary here.

runif()

If we wanted 10000 such numbers we could use a for loop, or the replicate() in R. However the easier way would be to ask for 10000 numbers with the n = 10000 argument.

```
ru1 <- runif(n = 10000, min = 0, max = 1)
length(ru1)
```

```
## [1] 10000
```

```
head(ru1)
```

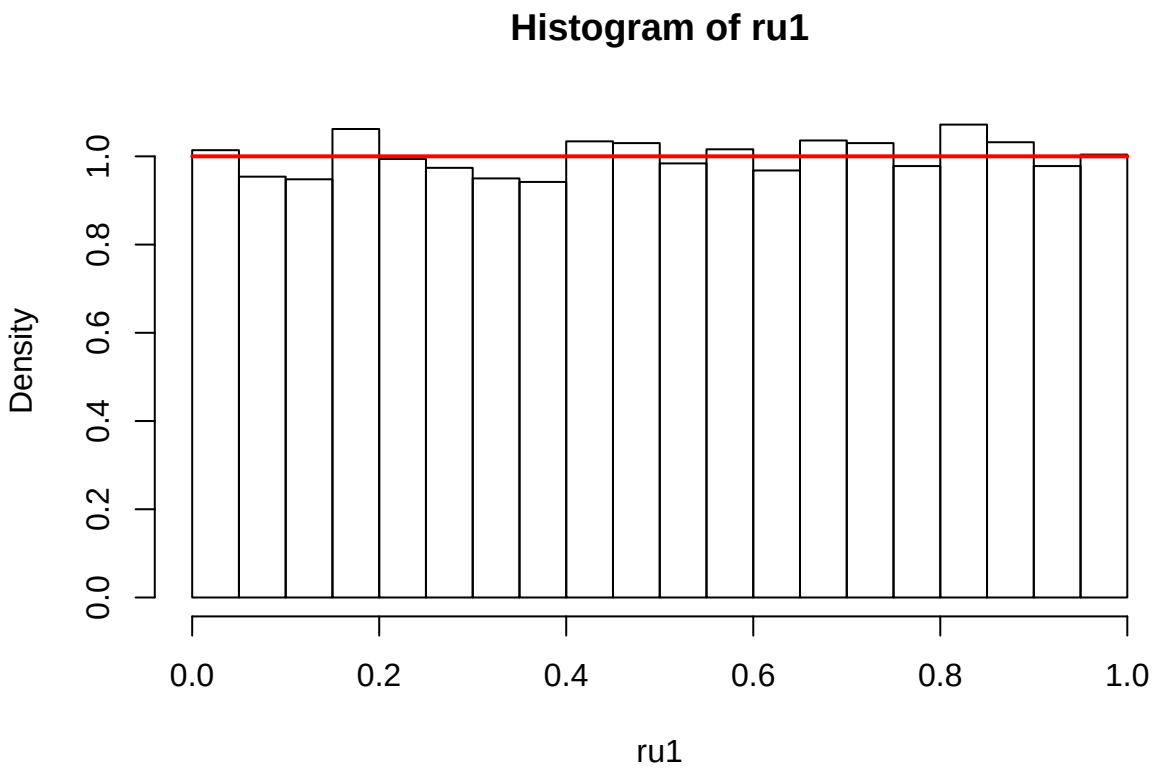
```
## [1] 0.09161287 0.80218684 0.68610930 0.90950508 0.87258869 0.60703398
```

```
tail(ru1)
```

```
## [1] 0.7362784 0.2024902 0.4717204 0.6024064 0.6743810 0.9721301
```

- If I plotted a histogram of this data, what should it look like (at least theoretically)?

```
par(mfrow = c(1,1))
hist(ru1, freq = F)
curve(dunif(x, 0, 1), 0, 1,
      add = T, col = "red", lwd = 2)
```

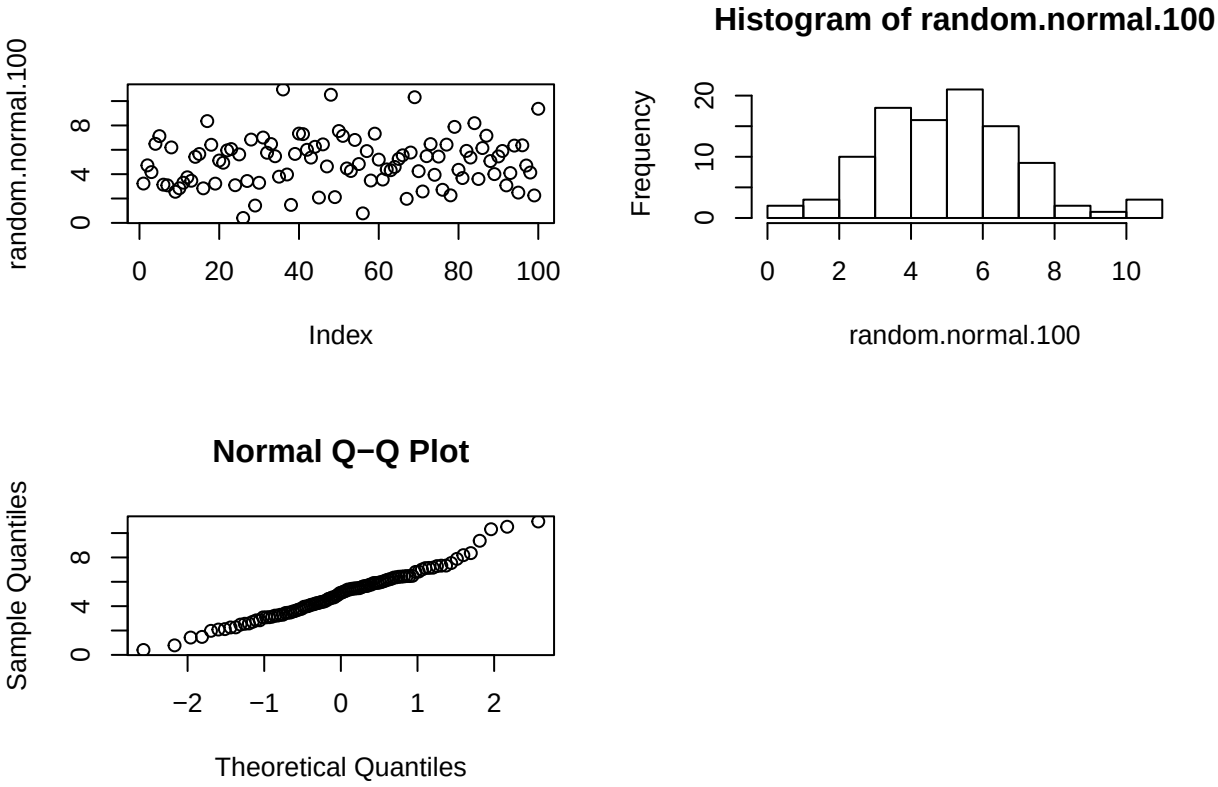


With the theoretical uniform distribution on [1,1] in red

simulating from a normal distribution

- If we wanted to look at 100 draws from a normal distribution with mean =5 and sd=2
- $x \sim N(\mu = 5, \sigma = 2)$

```
random.normal.100 <- rnorm(n = 100, mean = 5,sd = 2)
par(mfrow=c(2,2))
plot(random.normal.100)
hist(random.normal.100)
qqnorm(y = random.normal.100)
```



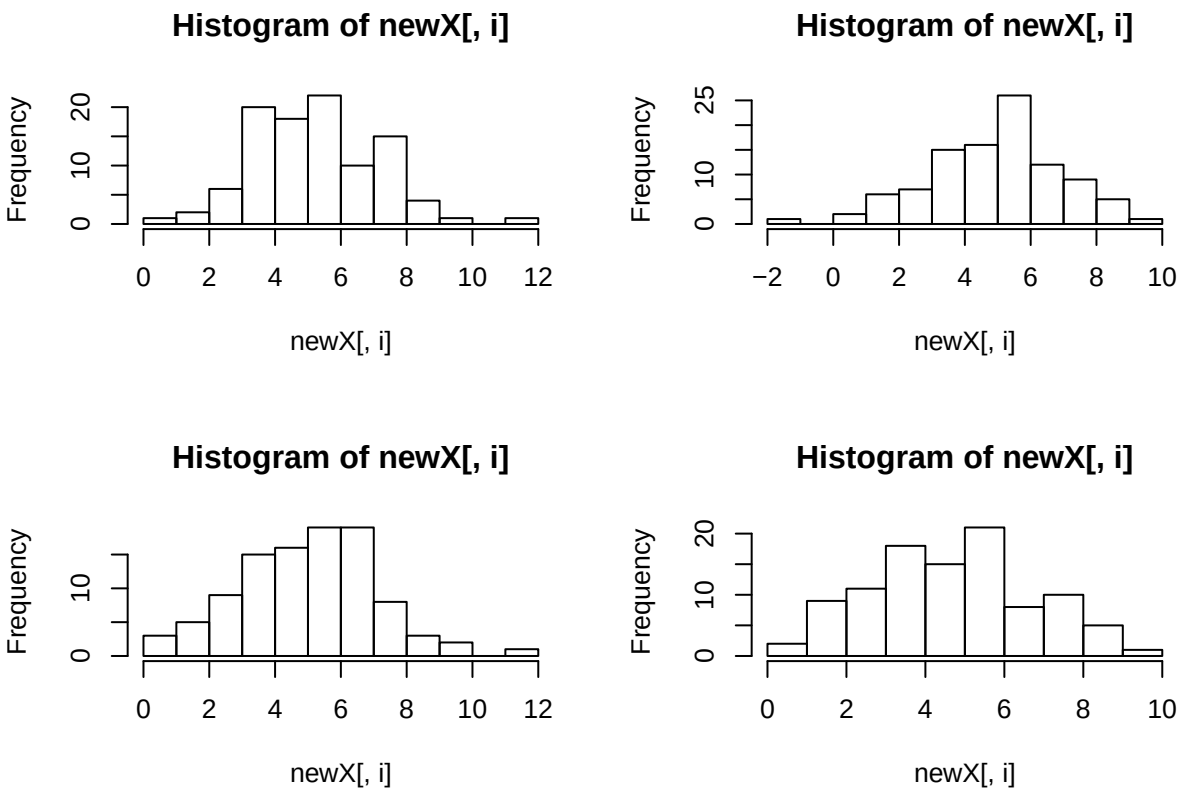
- repeat this simulation a few times to confirm it is changing...

More efficient random sampling using replicate

- Say I was doing this to get a sense of what would happen in a particular experiment with 100 samples. I may want to repeat this a few times. How might I do this efficiently?

More efficient random sampling using replicate

```
random.normal.100.rep <- replicate(n = 4,
                                   rnorm(100, 5, 2))


par(mfrow=c(2,2))
apply(X=random.normal.100.rep,
      MARGIN=2, FUN=hist)
```




```
apply(X=random.normal.100.rep, MARGIN=2, FUN=mean)

apply(random.normal.100.rep, 2, sd)
```

the advantage of this approach is you can also just do

```
summary(random.normal.100.rep)
```

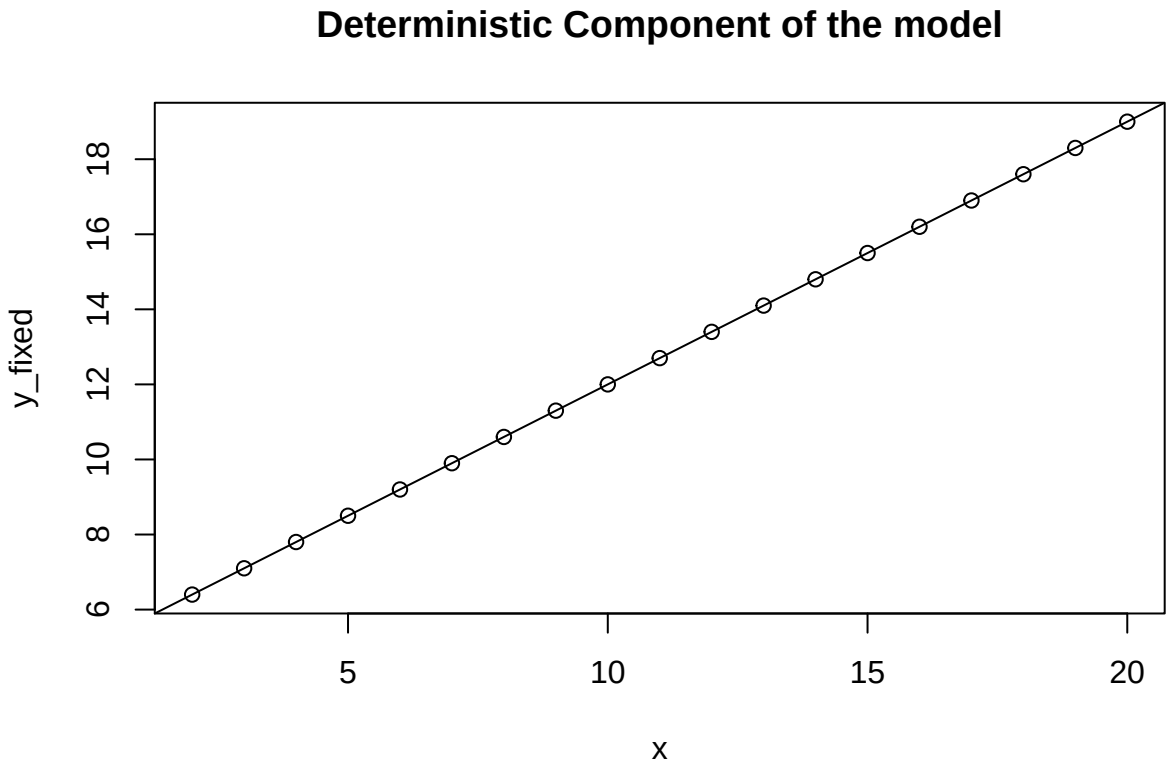
```
##        V1               V2               V3               V4
##  Min.   : 0.8524   Min.   :-1.070   Min.   : 0.3486   Min.   :0.1638
##  1st Qu.: 3.8103   1st Qu.: 3.628   1st Qu.: 3.5363   1st Qu.:3.2549
##  Median : 5.1427   Median : 5.224   Median : 5.1196   Median :4.6700
##  Mean   : 5.2117   Mean   : 4.893   Mean   : 4.9909   Mean   :4.7011
##  3rd Qu.: 6.4915   3rd Qu.: 6.035   3rd Qu.: 6.3486   3rd Qu.:5.9316
##  Max.   :11.4224   Max.   : 9.034   Max.   :11.5092   Max.   :9.4592
```

monte carlo methods

- This is an example of a very simple *monte carlo simulation*
- Let's use this to do something a bit more interesting, simulate from a regression model.
- $Y \sim N(\mu = a + b * x, \sigma)$
- We will make the slope $b = 0.7$, the intercept $a = 5$ and residual variation$\sigma = 2$

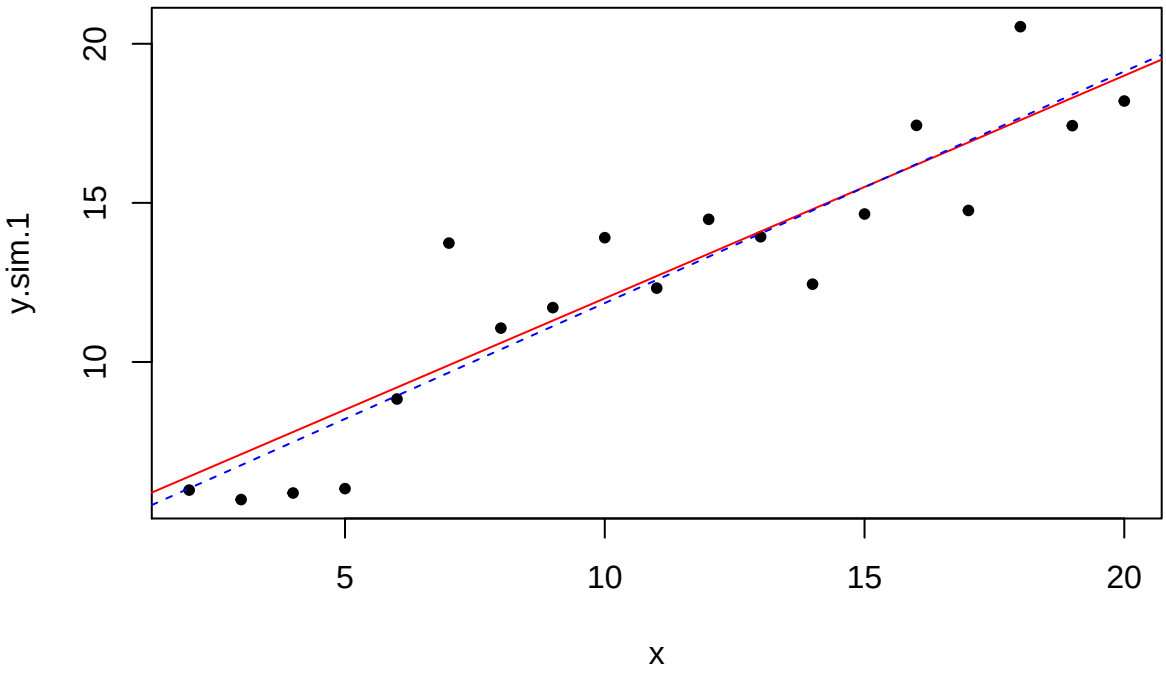
First the deterministic part

```
par(mfrow=c(1,1))
a = 5 # intercept
b = 0.7 # slope
x <- seq(2,20) # values of our predictor "x"

y_fixed <- a + b*x # we are expressing the relationship between y and x as a linear model. In this case

plot(y_fixed ~ x, main= "Deterministic Component of the model") # A linear model
abline(a=5, b=0.7)
```



add in stochastic component now

```
y.sim.1 <- rnorm(length(x), mean = y_fixed, sd = 2)
plot(y.sim.1 ~ x, pch = 20)
abline(a = 5, b = 0.7, col = "red") # Expected relationship based on the parameters we used.

y.sim.1.lm <- lm(y.sim.1 ~ x) # fit a regression with simulated data
abline(reg = y.sim.1.lm, lty = 2, col = "blue") # estimated values based on simulated data.
```



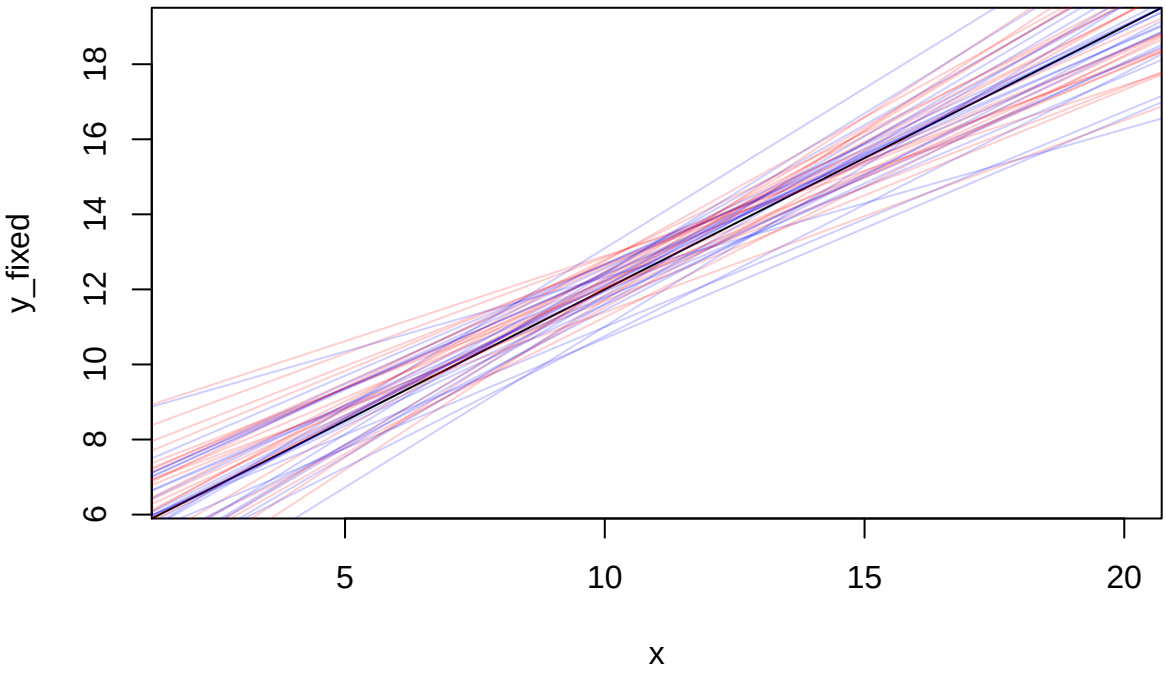
- Re-run this simulation a few times to see how it changes.

how about if you wanted to simulate this relationship 25 times?

- Keeping the same deterministic parameters, but change the residual standard error (stochastic variation) to $\sigma \sim 2.5$
- plot the deterministic fit first.
- Simulate the y values 25 times, each time re-fitting the regression and plotting the lines.

how about if you wanted to simulate this relationship 25 times?

```
plot(y_fixed ~ x, col = "black", type = "n")
abline(a=5, b=0.7)

simulated_regression <- function() {
    y.sim.1 <- rnorm(length(x), mean = y_fixed, sd = 2.5)
    y.sim.1.lm <- lm(y.sim.1 ~ x)
    abline(reg = y.sim.1.lm, col = "#FF000032")}

replicate(n =25, simulated_regression())

# or with a for loop instead (in blue)
for (i in 1:25){
    y.sim.1 <- rnorm(length(x), mean = y_fixed, sd = 2.5)
    y.sim.1.lm <- lm(y.sim.1 ~ x)
    abline(reg = y.sim.1.lm, col = "#0000FF32")
}
```



Back to the all important `sample()` function. Simple simulations of sequences.

- Sometimes we don't want to sample from a probability distribution, but for discrete categories (say number of A, C, T and G in a sequence). Like we did with the 6 sided die.
- The all powerful `sample()` is the workhorse function and is incredibly powerful for this!

a random 100000bp sequence with no biases.

- Say you wanted to generate a random DNA sequence with 100000 base pairs, with no GC bias. Do this with the sample function.
- Confirm the length of the sequence.
- What is the distribution of each nucleotide?

a random 100000bp sequence with no biases.

```
seq1_no_bias <- sample(c("A","C","G", "T"),
                       size = 100000, replace = TRUE)

table(seq1_no_bias)/length(seq1_no_bias)
```

```
## seq1_no_bias
##       A       C       G       T
## 0.24919 0.24769 0.25235 0.25077
```

```
length(seq1_no_bias)
```

```
## [1] 100000
```

```
head(seq1_no_bias)
```

```
## [1] "A" "G" "A" "C" "G" "C"
```

- `replace = TRUE` means it can repeatedly draw from the letters ACGT, otherwise it could only use each once (for a maximum of 4 draws)

Convert this vector to a single string, and check how many "letters" are in it

Convert this vector to a single string, and check how many "letters" are in it

- Now we want to convert this to a single sequence.

```
seq1 <- paste(seq1_no_bias, sep = "", collapse = "")
nchar(seq1)
```

```
## [1] 100000
```

Let's say we wanted to identify how often a simple sequence motif occurs.

- How many times does AACTTT occur in this simulated sequence?
- R has many string manipulation functions (see `?grep` and `?regex`).
- We will learn more about them, and the powerful *stringi* and *stringr* packages next week.
- Here we will see just a few.
- use `gregexpr()` to count number of occurrences of your pattern.
- g for global

Let's say we wanted to identify how often a simple sequence motif occurs.

```
x <- gregexpr("AACTTTT", seq1, fixed = T, useBytes = T)
length(unlist(x))
```

```
## [1] 6
```

- `fixed = T`, this means to not treat the pattern as a regular expression, but to interpret as is. This can speed things up at the cost of only allowing fixed expressions.
- useBytes =T, does byte by byte comparisons. Also speeds up computation, BUT at the cost of generality.
- In general use the default settings for fixed and useBytes unless speed is really important.

with some GC bias

- Say our sequence simulation is supposed to be from *Drosophila* with a 60:40 GC:AT bias. We can just add in the prob argument. Generate a sequence of 100000 base pairs.

```
seq2_60GCbias <- sample(c("A","C","G", "T"),
                        size = 100000,
                        prob = c(30,20,20,30),
                        replace = TRUE)

table(seq2_60GCbias)/length(seq2_60GCbias)
```

```
## seq2_60GCbias
##       A       C       G       T
## 0.29981 0.19925 0.19984 0.30110
```

```
seq2_60GCbias <- paste0(seq2_60GCbias, collapse="")

nchar(seq2_60GCbias)
```

```
## [1] 100000
```

```
y <- gregexpr("AACTTTT", seq2_60GCbias, fixed = T, useBytes = T)
length(unlist(y))
```

```
## [1] 18
```

- How would you check?

The code I have written is extremely fragile and error prone.

-Can you think of some reasons?

The code I have written is extremely fragile and error prone.

- What happens if there are no matches? Check.
- `gregexpr()` only includes unique matches, if the pattern is found in overlapping parts of the sequence, it will only count it once.
- What happens if there is whitespace? My code does not account for it.
- What happens if some nucleotides are lower case?
- What happens if ambigous nucleotides are included, or insertions, deletions, Ns?
- DNA is generally double stranded, so we need to deal with the reverse complement as well.

So what should we do.

- We could definitely rewrite this function to handle all such exceptions. Use functions like `toupper()` to make everything upper case, check for other nucleotides remove leading and trailing white space with `trimws()`, which is really just a wrapper around `sub`. From a learning perspective this is a useful exercise, that I leave to you.
- However, if our goal is to manipulate nucleotide sequences, we should be smart about this and use a well developed tool like the Biostrings library. Other than for the purposes of learning the skills to DIY (or when learning to program), or for some specialized purpose that does not exist, using well developed libraries make sense. These have been well vetted by many users and developers and are less buggy and probably more efficient computationally than what most of us would write.

reverse complement with base R

- the `chartr(old, new, x)` function translates one set of characters to another for a string x
- rev, reverses the order of the elements of the vector (but not if it is a single string)

```
seq3_60GCbias <- sample(c("A","C","G", "T"),
                        size = 20,
                        prob = c(30,20,20,30),
                        replace = TRUE)
seq3_60GCbias
```

```
##  [1] "T" "T" "C" "T" "G" "A" "T" "T" "C" "T" "G" "A" "C" "G" "T" "T" "T"
## [18] "A" "T" "T"
```

```
seq3_60GCbias_rev <- rev(seq3_60GCbias)
seq3_60GCbias_revcomp <- chartr("ACTG", "TGAC", seq3_60GCbias_rev)
seq3_60GCbias_revcomp <- paste0(seq3_60GCbias_revcomp, collapse="")
```

Do we care about the reverse complement for this example?

- For this simulation example, why might the reverse complement not matter much?