R tutorial

Arithmetic and logical syntax in R

This is an R tutorial, which should help you get familiar with the basics of R, including math functions, manipulating vectors, reading in data, writing loops and functions, and simulating and displaying phylogenetic trees.

Similar to MATLAB, R can be used at the prompt like a caluclator

```
> 7 + 5
[1] 12
> 7/5*5
[1] 7
> 7/(5*5)
[1] 0.28
> 7^5
[1] 16807
> (7 + 5) %% 4
[1] 0
```

The last operation `%%` used the modulus operator, which returns the remainder when the argument on the left is divided by the argument on the right. Ignore the `[1]` for now. This will be explained shortly.

Values can be assigned to variables as well. This can be done either using `=` or `<-`. When a value is assigned. The result is stored but not displayed unless this is asked for.

```
> x <- 5^7
> y <- 7^5
> y
[1] 16807
```

Comparisons can be made to check which value is larger.

```
> x > y
[1] TRUE
> w <- x > y
> w
> y + w
```

In the above example, the variable `w` gets the value `TRUE`. `TRUE` evaluates to 1 in arithmetic statements and `FALSE` evaluates to 0. Logical comparisons can also be made by surrounding a statement such as an equality or inequality with parentheses:

```
> x <- 5^7
> y <- 7^5
> w <- (x>y) + (y> x/10)
> w
[1] 2
```

Equality is written as `==` and lack of equality is written as `!=`, which is similar syntax to many other languages.

```
> x == 5^7
[1] TRUE
> y != 7^5
[1] FALSE
```

In addition to being combined in arithmetic statements, logical operators can be used on statements. Here we use `&` for AND and `|` for OR, instead of `&&` and `||` that you might see in other languages.

```
> x>y & x<y
[1] FALSE
> x>y | x<y
[1] TRUE
```

Vectors in R

R is a vector-oriented language and operates on vectors similarly to MATLAB. A variable can store a vector of values. This is often accomplished using the `c` operator, which stands for concatenate. Here is an example:

```
> First3primes <- c(2,3,5)
> Second3primes <- c(7,11,13)
> First6Primes <- c(First3primes,Second3primes)
> First3primes
[1] 2 3 5
> First6Primes
[1]  2  3  5  7 11 13
```

The fact that vectors can be appended to one another to create new vectors explains the "concatenation" idea.

If a range of integers is wanted, a quick way to create a vector is the `:` operator.

```
> 3:7
[1] 3 4 5 6 7
> 10:2
[1] 10 9 8 7 6 5 4 3 2
```

Arithmetic operations work piecewise on vectors.

```
> x <- 1:10
> x/2
[1] 0.5 1 1.5 2 2.5 3 3.5 4 4.5 5
> y <- x^2
> y
[1] 1 4 9 16 25 36 49 64 81 100
> x*y
 [1]    1    8   27   64  125  216  343  512  729 1000
```

```
> sqrt(x)
 [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751 2.828427
 [9] 3.000000 3.162278
```

This is very useful for defining new variables from old variables, for example to create a plot. Note that `x*y` gives the cubes from 1 to 10. When the square root function was called on my system, we finally see a line of output starting with something other than `[1]`. Here the `[9]` means that the row is starting with the 9th element of the output vector. `[1]` means the row starts with the first element of the output vector, even if the vector only has length 1.

Note that R does not (by default) work with complex numbers, so these can lead to errors or to values called `NaN` for "Not a Number". With some cleverness, you can sometimes get around this using `TRUE` and `FALSE` values:

```
> x <- (-5:10)
> x
 [1] -5 -4 -3 -2 -1  0  1  2  3  4  5  6  7  8  9 10
> sqrt(x)
 [1]      NaN      NaN      NaN      NaN      NaN 0.000000 1.000000 1.414214
 [9] 1.732051 2.000000 2.236068 2.449490 2.645751 2.828427 3.000000 3.162278
Warning message:
In sqrt(x) : NaNs produced
> (x>0)*sqrt(x*(x>0)) \# clever solution to make the argument of the square root equal 0 if x is negati
 [1] 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 1.000000 1.414214
 [9] 1.732051 2.000000 2.236068 2.449490 2.645751 2.828427 3.000000 3.162278
```

Will the following work? Why or why not?

```
> (x>0)*sqrt(x)
```

In some cases, you want to extract an element from a vector, or a range of elements from a vector. Here we extract the 5th element of a vector, and also a range of elements

```
> x <- c(1,5,4,2,4,6,7,4,5,1,2)
> x[5]
[1] 4
> y <- x[2:4]
> y
[1] 5 4 2
```

Here are some other things you can do with vectors that might be useful. Try the following and write in what the results are:

```
> x <- c(1,5,4,2,4,6,7,4,5,1,2)
> length(x)
[1]
> sort(x,decreasing=T)
 [1]
> y <- sort(x)
> y
 [1]
> x
[1]
> max(x)
```

```
[1]
> min(x)
[1]
> mean(x)
[1]
> median(x)
[1]
> range(x)
[1]
> table(x)



> max(table(x))
[1]
> which(x==4)
[1]
> which(x==max(x))
[1]
> unique(x)
[1]
> length(unique(x))
[1]
> y <- sort(unique(x))
> y
[1]
```

These examples were all instances of using functions in R. They are all built in functions. Functions in R have a keyword or user defined object followed by parentheses, usually with one or more arguments in the parentheses. The function then typically generates output or returns a value that can be stored. The functions used above do the following

1. `length` returns the number of elements in a vector
2. `sort` prints a rearranged version in either ascending (default) or descending order. If the order isn't specified, the order is ascending. Using the optional argument `decreasing = T` or `decreasing = TRUE` (it doesn't matter if you type `T` or `TRUE`) makes it decreasing. Note that this function doesn't change the order of the vector, but only the way it is printed or saved to a new variable.
3. `max` and `min` return the maximum and minimum value in the vector.
4. `mean` returns the arithmetic average of the values.
5. `median` returns the sample median of the values.
6. `table` returns the values in the vector and a count of how often each value arises.
7. `max(table)` returns the maximum frequency in the table, not the maximum value for x
8. `which` in this case returns the index or indices of the vector x for which x is equal to 4
9. `unique` returns a vector which has all of the distinct values of x, each occurring only once

FOR loops

An important part of programming is writing loops. A typical example of using a loop is automate a long summation. For example the sum

$$\sum_{i=1}^{10} i^2$$

can be written as loop in R as follows:

```
> sum <- 0
> for(i in 1:10) {
+ sum <- sum + i^2
+ }
> print(sum)
```

For this code, the loop starts with `for`, and the body of the loop has curly braces. You can continue on more than one line, and R puts the `+` sign to indicate that the `for` statement hasn't finished

As another example, we might wish to evaluate a product of numbers. For example

$$1 \times 2 \times \cdots n$$

which is often written as $n!$ can be evaluated using a loop

```
> factorial <- 1
> for(i in 1:n) {
+ factorial <- factorial * i
+ }
> print(factorial)
```

An example related to probability is the Birthday Problem.

```
> n <- 20
> prob <- 1
> for(i in 1:(n-1)) {
+ prob <- prob*(365-i)/365
+ }
 > print(prob)
```

which in this case returns the probability that 20 independent people with random birthdays (equally likely throughout the year) have 20 distinct birthdays.

Try modifying this code to verify that 23 people is the smallest number for whom the probability is greater than 50% that at least two people have the same birthday.

Functions

It also helpful to write your own functions. The notation for this is to write

```
myfunction <- function() {
...
}
```

where `myfunction` is the name you choose for your function. You can also have one or more parameters for example

```
myfunction2 <- function(n,m) {
...
}
myfunction3 <- function(n,m=10) {
...
}
```

In this example, `myfunction2` has two parameters which must be specified by the user. However, `myfunction3` has two parameters for which the first parameter must be specified by the user, but the second parameter has a default value of 10. If the user specifies a value for `m`, this value is used, if the user doesn't specify it, then 10 is used. Thus

```
myfunction3(5,8)
myfunction3(5)
```

would both be valid function calls.

Exercise. Write a function `bd(n)` that computes the probability that at least two people out of n have the same birthday assuming no one is born on February 29th and assuming all birthdays are equally likely and independent. Plot the probability as a function of n using the following:

```
prob <- 1:30
for(i in 1:30) {
prob[i] <- bd(i)
}
plot(1:30,prob,ylab="Probability",xlab="Number of people")
points(1:30, prob,type="l")
```

`Points` adds a smooth line to the plot.

Exercise. Write a function that approximates Euler's constant. I.e., write a function to compute

$$f(n) = \left(\sum_{k=1}^{n} 1/k\right) - \log n.$$

Euler's constant is $\gamma = 0.57721566\cdots$ and is obtained by taking the limit of $f(n)$ as $n \to \infty$. Find out how good the approximation is as a function of n.

Using packages in R

Many different packages exist in R that are uploaded by users, often academics and graduate students. R is being continually added to in this way, much like Wikipedia, and there is no guarantee that things work correctly. Often, however, they do work.

For phylogenetics, a useful package is called `ape` which stands for Analysis of Phylogenetic Evolution. This package does not come with R, but is easy to install. To install it locally, type the following,

```
> install.packages("ape")
> library(ape)
> help(library=ape)
```

After typing `install.packages("ape")`, you'll have to follow instructions on which mirror site to install from. Pick something close like Texas or California for it to install more quickly. After it is done installing, use the `library` command to have access to `ape` functions. The help statement shows all the functions in the `ape` library with other information. For help on an individual function, type for example `help(sort)` or `help(mrca)` for functions both in R and in `ape`.

After a library installed, you still have to type `library(ape)` each new session. I'm not sure if you'll have to reintall the package each day in the lab for the MCTP. If you install this on your laptop, you should only have to use `install.packages` once, but will still need to type `library(ape)` each new session.

As an example of something that `ape` can do, try the following for plotting a phylogeny.

```
> birds <- read.tree("http://math.unm.edu/~james/steac") # or read.tree("steac")
#if steac is in the same directory as you
> plot.phylo(birds)
> plot.phylo(ladderize(birds))
> plot.phylo(birds,type="f",use.edge.length=F)
> plot.phylo(birds,type="fan")
> plot.phylo(birds,type="radial")
> plot.phylo(birds,type="unrooted")
> plot.phylo(birds,use.edge.length=F)
```

Try `help(plot.phylo)` for more options. The data comes from McCormack et al., PLOS ONE, 2013, "A Phylogeny of Birds Based on Over 1,500 Loci Collected by Target Enrichment and High-Throughput Sequencing".

Often you read in more than one tree at a time. For example

```
> library(ape)
> x <- read.tree("http://math.unm.edu/~james/t5all")
> x
105 phylogenetic trees
> x[[1]]

Phylogenetic tree with 5 tips and 4 internal nodes.

Tip labels:
[1] "A" "B" "C" "D" "E"

Rooted; includes branch lengths.
> plot.phylo(x[[2]])
> plot.phylo(x[2])
NULL
Warning message:
In plot.phylo(x[2]) : found less than 2 tips in the tree
```

Note that to access the tree, you need the strange notation of double square brackets. I don't quite understand this, but it works.

The objects `x` and `birds`, having trees, are complex objects with different components to them. To see what is stored with a tree, type

```
> names(birds)
> names(x[[1]])
```

You can access or even modify this information. For example

```
> birds$edge.length
 [1] 0.019034969 0.019035000 0.022326002 0.000412000 0.000311000 0.011407364
 [7] 0.030863606 0.000191000 0.000093000 0.000218000 0.000172000 0.024251703
[13] 0.022416507 0.001327000 0.008643469 0.013302251 0.000021000 0.000041000
[19] 0.000109000 0.000056000 0.000082000 0.000069000 0.000321000 0.010567210
[25] 0.013935630 0.014472210 0.000142000 0.000682000 0.000934000 0.003025000
```

```
[31] 0.016486000 0.022810429 0.029627311 0.061199713 0.024314493 0.034540527
[37] 0.026180195 0.018839491 0.013505608 0.000126000 0.000224000 0.014907216
[43] 0.011785674 0.000392000 0.000095000 0.012297652 0.011220488 0.000302000
[49] 0.001785000 0.010411341 0.008631659 0.015166363 0.000242000 0.022002929
[55] 0.015409571 0.000409000 0.027045184 0.020082836
> birds$edge.length <- sqrt(birds$edge.legnth)
> birds$edge.length
 [1] 0.137967275 0.137967387 0.149418881 0.020297783 0.017635192 0.106805262
 [7] 0.175680409 0.013820275 0.009643651 0.014764823 0.013114877 0.155729583
[13] 0.149721431 0.036428011 0.092970260 0.115335384 0.004582576 0.006403124
[19] 0.010440307 0.007483315 0.009055385 0.008306624 0.017916473 0.102796934
[25] 0.118049271 0.120300499 0.011916375 0.026115130 0.030561414 0.055000000
[31] 0.128397819 0.151031220 0.172125857 0.247385758 0.155931051 0.185850820
[37] 0.161802950 0.137257024 0.116213630 0.011224972 0.014966630 0.122095112
[43] 0.108561843 0.019798990 0.009746794 0.110894777 0.105926807 0.017378147
[49] 0.042249260 0.102035978 0.092906724 0.123151789 0.015556349 0.148333842
[55] 0.124135295 0.020223748 0.164454201 0.141713923
```

This transformation increases the edge lengths but preserves their relative lengths (i.e., if edge i is longer than j, this will be true also after the square root transformation) but makes the nodes look more resolved and easier to see (try it).

Day 2

In R, we'll explore generating random variables and doing simulations to estimate probabilities.

A random variable in R can be generated using a built-in function like this

```
> x <- runif(10,0,2)
> x <- rexp(10,5)
> x <- rnorm(5,3,6)
```

Here, `runif` indicates that you are generating a random variable from a uniform distribution. The arguments request a vector of 10 random variables from 0 to 2.
Built in distributions to simulate from include the uniform, exponential, normal, Poisson, gamma, and others. In addition to simulating random variables, there are built in functions for the density or mass function, the quantile of the distribution, and the cumulative.

To simulate fair dice, it easiest to do the following

```
> die <- 1:6
> X1 <- sample(die,2,replace=T)
```

The `sample` command samples from a vector representing a finite population of values. Sampling can be done with replacement or without replacement. The second argument tells it the number of values to simulate.
Exercise. In the game of Dungeons & Dragons, sometimes you roll three six-sided dice and add the dice. This can be used in character generations. For example your character's intelligence in IQ might be the sum of three six-sided dice times 10. Use simulation to find the probability that a character generated this way has an IQ of 60 or less.

Exercise. A more complicated way of generating character values is to roll 4 dice and take the sum of the three highest values (so that your character is "above average", "on average"). Use simulation to

determine the probability that your character has an IQ of 60 or less under this method. (This is possible to do by hand but extremely tedious.)

Exercsie. Try simulating a poker hand with 5 cards from a deck of 52 cards. Try using simulation to determine the probabilities of a two-of-a-kind, a three-of-a-kind, and a four-of-a-kind. (Note, small probabiities are hard to simulate!) For this problem it is hhelpful to use sampling without replacement, e.g., `sample(1:52,5,replace=F)`. It is also helpful to use the modulus operator so that the value of the card is determined by `card %% 4`.

0.1 Visualizing distributions

To visualize a distribution, you can generate a large number of simulated values and plot the values using a histogram, which indicates how often observations fell in a certain range. Try this for the exponential distribution

```
> x <- rexp(1000,1)
> x <- hist(x,nclass=50)
```

The optional argument `nclass` specifies how many bins you want for the histogram, so you can make it more fine-grained than the default.

You can also try this with discrete distributions

```
> x <- rpois(1000,3) # Poisson distribution which generates integers 0, 1, 2, ...
> hist(x)
> plot(table(x),type="h") # I kind of prefer this approach since you see every integer value
```

A tricky mathematical problem can be to determine a formula for functions of random variables. Let X and Y be exponential random variables with mean 1. Then what is the distribution of $Z = X^2 - \sqrt{Y}$. This would be unpleasant to do by hand but is easy to simulate.

```
> x <- rexp(10000,1)
> y <- rexp(10000,1)
> z <- x^2-sqrt(y)
> hist(z,nclass=50)
> length(z[z<1])/10000
```

The last line estimates the probability that $Z < 1$.

In some cases, you can figure out transformations of random variables by hand, and using simulation is a good way to check your work.

In R, type

```
> source("http://math.unm.edu/~james/wf.r")
> WF
> WF()
```

The function **WF()** generates a Wright-Fisher based discrete-time coalescent. The coalescent is obtained from this process as the population goes to infinity. Try different values of `N`, the population size, `samplesize`, which is the number of individuals sampled in the present, `tau`, the number of generations that the function simulates, and `mycex`, the size of the plot cymbals. The value returned at the end is the list of ancestors from the original sample. For example, if you get (results may vary because this is random)

```
> WF(10,5,2,1)
[1] 5 3 2 3 6
```

This means that lineages 1 through 5 sampled in the present had ancestors numbered 5, 3, 2, 3, and 6. Since 3 is repeated, this means that individuals 2 and 4 share a common ancestor, so a coalescence occurred in that one generation of time. If the list of 5 ancestors had 5 distinct values, then it would be the case that none of the pairs of lineages experienced a coalescence. On the other hand, the following

```
> WF(10,5,2,1)
[1] 10  6 10 10  8
```

indicates that three individuals all had ancestor 10 in the previous generations. Thus, there was a 3-way merger, which would form a polytomy within the gene tree if this process was extended further back in time so that the lineages formed a tree. The list of ancestors that the program returns is intended to allow you to simulate the probabilities of multiple mergers (more than two lineages coalescing at a time), probabilities of coalescence, and probabilities of simultaneous coalescence events. The latter occurs in the following case,

```
> WF(10,5,2,1)
[1] 5 7 5 8 7
```

where there were two 2-mergers. For simulating this probabilties, it is helpful to surpress the plot, which can be done like this

```
> WF(10,5,2,1,0)
[1] 5 7 5 8 7
> tempAncestors <- WF(10,5,2,1,0)
```

To put this process into a loop (in order to simulate), you'd want to assign the vector of ancestors to a temporary variable. From this you can extract information about the existence of mergers. As an example, no coalescence occurred if the list of ancestors has N distinct values, where N is the population size. Thus, the following code estimates the probability of no coalescence:

```
> nocoalCount <- 0
> iter <- 10000 # number of iterations
> for(i in 1:iter) {
> tempAncestors <- WF(10,5,2,1,0)
> if(length(unique(tempAncestors==5))) nocoalCount <- nocoalCount +1
> print(nocoalCount/iter)
```

The simulation can be varied for different values of `N` and `n`, the population size and sample size. Try determining the probability of no coalescence for `N` = 10, 20, 30, 40, 50 and for `n` = 5, 10.

PHYLIP

PHYLIP stands for PHYLogenetic Inference Package, and consists of a bunch of separate C programs that are bundled together in one folder. It is freely available to download, including source code in C, from the University of Washington. It was developed by Joseph Felsenstein who has been one of the most important researchers in phylogenetics since the 1970s. The program has been used in thousands of research papers. The program is located at

```
C:Users\Public\Documents\phylip-3.695\exe
```

On your system. This is the folder with the executables. The easiest way to use PHYLIP is to copy your data files into the executable directory since PHYLIP assumes that input files are in the same directory as the executable. In practice, I tend to copy the executable into the directory I am using for a project and just use that local copy of the executable. If you want the executables available everywhere on your system, you can copy them into an appropriate bin directory in your path, but this might vary from system to system, and we won't be doing enough to worry about that here.

Here is a description of what some of the most commonly used programs in PHYLIP do:

1. `dnapars` uses parsimony to infer a tree
2. `dnamlk` uses maximum likelihood (the probability of the DNA sequences under some model) to infer a tree when there is a molecular clock (constant mutation rate)
3. `dnaml` uses maximum likelihood to infer a tree without assuming a clock
4. `neighbor` uses distance methods to infer trees using clustering
5. `consense` uses consensus methods to combine multiple trees into an overall tree
6. `dnadist` constructs a distance matrix from sequences that can be used as input for `neighbor`
7. `treedist` uses a couple of metrics to compute distances between pairs of trees

To use some of these functions, download DNA sequence files and a tree into the PHYLIP directory

`C:Users\Public\Documents\phylip-3.695\exe`

from

`http://math.unm.edu/~james/temptrees`
`http://math.unm.edu/~james/align.565`
`http://math.unm.edu/~james/align.566`
`http://math.unm.edu/~james/align.567`

The structure of the alignment file shows you the number of species and the length (number of columns) in each alignment followed by DNA sequences. The file temptrees consists of phylogenetic trees estimated from these species but from different loci. You should be able to look at the trees in R using commands from last time, but to read them into PHYLIP, they should be downloaded locally into your computer.

PHYLIP typically takes an input file, by default called "infile", which might consist of sequences, trees, or a distance matrix, and then creates an output file, typically called "outfile" or "outtrees" or something similar. The `outtrees` file allows you see the tree in Newick format, and can best be visualized using software such as R. So view a tree in R that is saved to your computer, do something like

```
> library(ape)
> x <- read.tree("C:Users/Public/Documents/phylip-3.695/exe/temptrees") # I believe R prefers forward sl
> plot.phylo(x[[1]])
```

Try using `dnapars`, `dnaml`, and `dnamlk` on the alignments in `align.565`, `align.566`, and `align.567` to see if they result in different tree topologies or the same tree topology.

R programming with the Wright-Fisher model

To simulate genetic drift under the Wright-Fisher model, try the folloing code. This code is fairly long to type at the R prompt, so you might want to save it in an external file, such as in Notepad, and then either copy and paste into R, or use the source command.

```
p <- .9  #proportion of A allele
N <- 20  #2N is number of allele copies
g <- 100 #number of generations
I <- 10 #number of iterations

plot(c(1,g),c(0,1),type=''n'')

for(i in 1:I) {
```

```
  y1 <- rep(0, round(2*N*p))  # number of copies of allele 0
  y2 <- rep(1, 2*N-round(2*N*p)) # number of copies of allele 1
  x <- c(y1,y2)
  ave <- mean(x)

  for(j in 1:g) {
  x <- sample(x,replace=T)
  ave <- c(ave,mean(x))
  }

  points(ave,type=''l'')
}
```

To use the source command, suppose the code is in a file called `drift.r`, then type

```
> source("drift.r")
```

You can modify this code to simulate the probability that an allele fixates. The above graph simulated the process 10 times. You would want to modify the code to simulate a large number of times (1000 or 10000) and suppress the plot. Here is one way to do it, and the code is at

`http://math.unm.edu/~james/drift.r`

```
p <- .9  #proportion of A allele
N <- 20  #2N is number of allele copies
g <- 100 #number of generations
I <- 10 #number of iterations

drift <- function(p=.9,N=20,g=100,I=10) {
#plot(c(1,g),c(0,1),type=''n'')

  fix0 <- 0
  fix1 <- 0
  for(i in 1:I) {

    y1 <- rep(0, round(2*N*p))  # number of copies of allele 0
    y2 <- rep(1, 2*N-round(2*N*p)) # number of copies of allele 1
    x <- c(y1,y2)
    ave <- mean(x)

    for(j in 1:g) {
      x <- sample(x,replace=T)
      ave <- c(ave,mean(x))
    }
    if(ave[g] == 0) fix0 <- fix0+1
    if(ave[g] == 1) fix1 <- fix1+1
    #points(ave,type=''l'')
  }
  return(c(fix0,fix1))
}
```

1 Coalescent, exponential, and expected value problems

1. What is the average (mean) IQ of a Dungeons & Dragons character whose intelligence is determined by rolling 3 dice and multiplying by 10? What is the average IQ if intelligence is determined by rolling 4 dice and taking the sum of the top 3 and multiplying the result by 10? Use simulation to determine these averages.

2. (a) Verify the lightbulb probability in class: simulate lifetimes of lightbulbs where one lightbulb dies with rate 1, and the other dies with rate 2. Use simulation to verify the probability that the lighbulb with rate 1 has a 2/3 chance of lasting longer. (b) Repeat (a) using lightbulbs with rates 1 and 3. (c) Repeat (a) with rates 3 and 9. Can you guess a general pattern if lighbulb 1 has rate λ and lightbult 2 has rate β?

3. Use simulation to estimate

$$P(X_1 > X_2 > X_3 > X_4) = \int_0^\infty \int_{x_3}^\infty \int_{x_2}^\infty \int_{x_1}^\infty e^{-x_1} \cdot 2e^{-2x_2} \cdot 3e^{-3x} \cdot 4e^{-4x}\, dx_1\, dx_2\, dx_3\, dx_4$$

where X_i is exponential with rate i.

4. Write a simulation to verify that

$$g_{3,1}(t) = 1 - \frac{3}{2}e^{-t} + \frac{1}{2}e^{-3t}$$

To do this recall that this probability is $P(T_2 + T_3 < t)$. Simulate random variables using

```
> T2 <- rexp(10000,1)
> T3 <- rexp(10000,3)
> T <- T2+T3
> length(T[T < 1])/10000
```

This simulates the probability $g_{3,1}(1)$. Check this against the formula to see if they reasonably agree. Of course you can try other values of t against the formula for $g_{3,1}(t)$.

5. Consider two populations, one on the left, and one on the right. The left population has three lineages sampled and the right population has two lineages sampled. If you go back far enough in time, the left population lineages will coalesce as will the right population lineages. (There is never a coalescence between lineages from different populations in this model.)

If you keep track of the sequence of coalescences, there are three possibilities: LLR, RLL, and LRL depending on whether the two left coalescences occur first (most recently), the right coalescence happens most recently, etc. Write a simulation to determine the probabilities of these three events (LLR, RLL, and LRL). You might think of the times of the left coalescences as T_3 and $T_3 + T_2$, while the time of the right coalescence is R_2, where R_2 is exponential with rate 1. Are the three events equally likely?

6. Use simulation to try to verify the the values of $E[T]$ and $E[T_{tot}]$ where T is the height of a coalescent tree and T_{tot} is the sum of the lengths of the branches. Do this for large values of i to see if the values approach the limiting values. You don't need to simulate trees to do this problem. Just simulate $T_2, \ldots, T_i$ for large values of i to determine T and T_{tot}.

7. Distributions of sums. Let $X_1, \ldots, X_n$ be independent in the following

(a) Let X_i be the roll of a die, so that $P(X_i) = 1/6$. Use simulation to plot the distribution of $X = X_1 + X_2$. This can be done with the following code:

```
> die <- 1:6
> x1 <- sample(die,10000,replace=T)
> x2 <- sample(die,10000,replace=T)
> x <- x1+x2
> plot(x,type="h")
```

Repeat this experiment for $X = X_1 + X_2 + X_3$, $X = \sum_{i=1}^{4} X_i$ and $X = \sum_{i=1}^{5}$.

(b) Let X_i be exponential with rate 1. (All the X_is have the same rate). Use simulation to plot the distribution of $X = \sum_{i=1}^{k} X_i$ for $k = 1, 2, 3, 4, 5$.

(c) Let X_i be exponential with rate i. Use simulation to plot the distribution of $X = \sum_{i=1}^{k} X_i$ for $k = 1, 2, 3, 4$.

8. Write a function `g <- function(i,j,t)` that returns $g_{i,j}(t)$. Write another function `g2 <- function(j,t)` that returns $g_{\infty,j}(t)$. Compare the two functions for $i = 10, 20, 30, 40, 50$ for different values of i and j. See how large `i` can be in the `g(i,j,t)` function in R. Recall that

$$gg_{i,j}(t) = \sum_{k=j}^{i} e^{-\binom{k}{2}t} \cdot \frac{(2k-1)(-1)^{k-j} j_{(k-1)} i_{[k]}}{j!(k-j)! i_{(k)}}$$

$$g_{\infty,j}(t) = \sum_{k=j}^{\infty} e^{-\binom{k}{2}t} \cdot \frac{(2k-1)(-1)^{k-j} j_{(k-1)}}{j!(k-j)!}$$

Here is one way to write the function. Note that the function calls other functions, and it is ok for all functions to be in the same file. Functions don't need to be defined in the order they are used. For example, the first function calls the second and third function, and that is fine.

If I have a file called `gij.r` that has all three functions, I can type `> source("gij.r")`, and that gives me access to all three functions (assuming I'm in the same directory as the file — otherwise type the path to the file or the URL).

```
gij <- function(i,j,t) {
  sum <- 0
  for(k in j:i) {
    e <- exp(-choose(k,2)*t)
    num <- (2*k-1)*(-1)^(k-j)*up(j,k-1)*down(i,k)
    den <- factorial(j)*factorial(k-j)*up(i,k)
    sum <- sum + e*num/den
    # doing it all in one step is possible but more likely to have bugs
    # sum <- sum + exp(-choose(k,2)*t)*(2*k-1)*(-1)^(k-j)*up(j,k-1)*down(i,k)/(factorial(j)*factorial(k
    #print(sum) for debugging
  }
 return(sum)
}

down <- function(i,k) {
value <- 1
for(j in i:(i-k+1)) {
value <- value*j
}
return(max(1,value))  # max is used so that the function doesn't return 0.
}

up <- function(i,k) {
value <- 1
for(j in i:(i+k-1)) {
value <- value*j
}
return(max(1,value)) #  I think this max isn't needed, but doesn't seem to hurt!
}
```

Species tree estimation

Copy and paste a subset of trees from `http://math.unm.edu/ james/temptrees.txt` and use as input to STAR or MP-EST and find the resulting species tree estimate. Online versions of STAR and MP-EST exist at `http://bioinformatics.publichealth.uga.edu/SpeciesTreeAnalysis/index.php`

Try to answer the following. How likely are you to get the correct species tree using 2, 4, 6, 8, 10, or 20 gene trees as input? Do STAR and MP-EST always agree?