EEMB247/BMSE247: Computer lab 1: Introduction to R and ODEs

Cherie Briggs

January 19, 2018

Portions of this lesson were adopted from Software Carpentry, and portions were adapted from Introducing R, by German Rodriguez

In this class we will be programming with R.
R is a free, well-documented programming language, with a large user base among scientists. It has a large library of external packages available for performing diverse tasks.

Assignment in R

A **variable** is just a name for a value, such as x, current_temperature, or subject_id. In R, the assignment operator is <-. You can read this as "gets". R now also accepts the equal sign, so x<-2 and x=2 both assign the value 2 to a variable named x. We can create a new variable simply by assigning a value to it.

If we type in the following:

```
x <- 2
```

R creates a new variable and assigns it the value 2. But, it doesn't output anything. You can type in x to see its current value:

```
x
```

```
## [1] 2
```

Vectors

In addition to being a variable, x is also a vector of length 1. A **vector** is a one dimensional array of numbers. The function c, which is short for catenate (or concatenate if you prefer) can be used to create vectors from scalars or other vectors:

```
x <- c(1,3,5,7)
x
```

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

The colon operator : can be used to generate a sequence of numbers:

```
x <- 1:10
x
```

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

You can also use the seq function to create a sequence given the starting and stopping points and an increment. For example here are eleven values between 0 and 1 in steps of 0.1:

```
seq(0, 1, 0.1)
```

```
##  [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

Another function that is useful in creating vectors is rep for repeat or replicate. For example rep(3,4) replicates the number three four times. The first argument can be a vector, so rep(x,3) replicates the entire vector x three times. If both arguments are vectors of the same size, then each element of the first vector is replicated the number of times indicated by the corresponding element in the second vector. Consider this example:

```
rep(1:3, 2)
```

```
## [1] 1 2 3 1 2 3
```

```
rep(1:3, c(2,2,2))
```

```
## [1] 1 1 2 2 3 3
```

The first call repeats the vector 1:3 twice. The second call repeats each element of 1:3 twice, and could have been written rep(1:3, rep(2,3))

R operations are vectorized. If x is a vector, then log(x) is a vector with the logs of the elements of x. Arithmetic and relational operators also work element by element. If x and y are vectors of the same length, then x + y is a vector with elements equal to the sum of the corresponding elements of x and y.

```
x<-c(1:10)
y<-c(1,2,1,8,3,2,4,3,1,2)
x+y
```

```
##  [1]  2  4  4 12  8  8 11 11 10 12
```

If y is a **scalar** it is added to each element of x.

```
x<-c(1:10)
y<-2
x+y
```

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

In R, simple multiplication of two vectors also works on an element by element basis (each term of one vector is multiplied by its corresponding term in the other vector).

```
x<-c(2,4,7,1)
y<-c(1,2,3,4)
x*y
```

```
## [1]  2  8 21  4
```

Note this odd feature of R: If x and y are vectors of different lengths, the shorter one is recycled as needed, perhaps a fractional number of times (in which case you get a warning).

```
x<-c(1:10)
y<-c(1,2)
x
```

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

```
y
```

```
## [1] 1 2
```

```
x+y
```

```
##  [1]  2  4  4  6  6  8  8 10 10 12
```

```
x*y
```

```
##  [1]  1  4  3  8  5 12  7 16  9 20
```

The logical operators | for or and & for and also work element by element. (The double operators || for or and && for and work only on the first element of each vector and use shortcut evaluation; they are used mostly in writing R functions.)

```
a = c(TRUE, TRUE, FALSE, FALSE)
b = c(TRUE, FALSE, TRUE, FALSE)
a & b
```

```
## [1]  TRUE FALSE FALSE FALSE
```

The function length returns the number of elements of a vector.

```
length(a)
```

```
## [1] 4
```

Square brackets are used to identify individual elements of vectors: x[1] is the first element of x, x[2] is the second, and x[length(x)] is the last.

```
x[1]
```

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

```
x[2]
```

```
## [1] 2
```

```
x[length(x)]
```

```
## [1] 10
```

The subscript can be a vector itself, so x[1:3] is a vector consisting of the first three elements of x.

```
x[1:3]
```

```
## [1] 1 2 3
```

A negative subscript excludes the corresponding element, so x[-1] returns a vector with all elements of x except the first one.

```
a <- c(5, 7, 9, 8)
b <- a[-2]
b
```

```
## [1] 5 9 8
```

Interestingly, a subscript can also be a logical expression, in which case you get the elements for which the expression is TRUE. For example to list the elements of x that are less than 5 we use:

```
x[x < 5]
```

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

This expression can be read as: 'x such that x is less than 5'. This works because the subscript $x < 5$ is this vector:

```
x < 5
```

```
##  [1]  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
```

Arrays and Matrices

In R, vectors are dimensionless, whereas arrays and matrices have dimensions associated with them. The function dim retrieve or set the dimension of an object.

```
x<-c(1:10)
x
```

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

```
dim(x)
```

```
## NULL
```

A vector can be used by R as an array only if it is given a dimension:

```
dim(x) <- c(1,10)
x
```

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

This says that x now has 1 row and 10 columns.

```
dim(x)
```

```
## [1]  1 10
```

```
dim(x) <- c(2,5)
x
```

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

```
dim(x) <- c(5,2)
x
```

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

You can also do things like this using the array function:

```
x <- array(1:20, dim=c(4,5))
x
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    5    9   13   17
## [2,]    2    6   10   14   18
## [3,]    3    7   11   15   19
## [4,]    4    8   12   16   20
```

Frequently this is used to initialize arrays to zero:

```
x <- array(0, dim=c(3,10))
x
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,]    0    0    0    0    0    0    0    0    0     0
## [2,]    0    0    0    0    0    0    0    0    0     0
## [3,]    0    0    0    0    0    0    0    0    0     0
```

Matrices are arrays with only 2 dimensions. To create a matrix in R, we may use the matrix function. We need to provide a vector containing the elements of the matrix, and specify either the number of rows or the

number of columns of the matrix. This number should divide evenly into the length of the vector, or we will get a warning. For example, to make a 2 x 3 matrix named M consisting of the integers 1 through 6, we can do this:

```
M <- matrix( 1:6, nrow=2 )
M
```

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

or

```
M <- matrix( 1:6, ncol=3 )
M
```

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

Note that R places the row numbers on the left and the column numbers on top. Also note that R filled in the matrix column-by-column. If we prefer to fill in the matrix row-by-row, we must activate the byrow setting, e.g.:

```
M <- matrix( 1:6, ncol=3, byrow=TRUE )
M
```

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

To obtain the transpose of a matrix, we use the transpose function:

```
t(M)
```

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

To add or subtract two matrices (or vectors) which have the same number of rows and columns, we use the plus and minus symbols.
Create a matrix A:

```
A<-matrix(1:9,ncol=3)
A
```

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

Create another matrix B:

```
B<-matrix(c(5,2,1,4,7,2,8,9,3),ncol=3)
B
```

```
##      [,1] [,2] [,3]
## [1,]    5    4    8
## [2,]    2    7    9
## [3,]    1    2    3
```

Matrix addition:

```
A + B
```

```
##      [,1] [,2] [,3]
## [1,]    6    8   15
## [2,]    4   12   17
## [3,]    4    8   12
```

Matrix subtraction:

```
A - B
```

```
##      [,1] [,2] [,3]
## [1,]   -4    0   -1
## [2,]    0   -2   -1
## [3,]    2    4    6
```

To multiply two matrices with compatible dimensions (i.e., the number of columns of the first matrix equals the number of rows of the second matrix), we use the matrix multiplication operator %*%. For example:

```
A %*% B
```

```
##      [,1] [,2] [,3]
## [1,]   20   46   65
## [2,]   28   59   85
## [3,]   36   72  105
```

If we just use the multiplication operator *, R will multiply the corresponding elements of the two matrices, provided they have the same dimensions. But this is not the way to multiply matrices.

```
A * B
```

```
##      [,1] [,2] [,3]
## [1,]    5   16   56
## [2,]    4   35   72
## [3,]    3   12   27
```

Likewise, to multiply two vectors to get their scalar (inner) product, we use the same %*% operator.

```
a<-c(1,4,7,2)
b<-c(3,1,2,0)
a %*% b
```

```
##      [,1]
## [1,]   21
```

Technically, we should use the transpose of b. But R will transpose a for us rather than giving us an error message.

To create the identity matrix for a desired dimension, we use the diagonal function:

```
I <- diag(5)
I
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    0    0    0    0
## [2,]    0    1    0    0    0
## [3,]    0    0    1    0    0
## [4,]    0    0    0    1    0
## [5,]    0    0    0    0    1
```

This gives us the 5 x 5 identity matrix.

To find the determinant of a square matrix M, use the determinant function, det(M):

```
M<-c(2,1,1,2)
dim(M)<-c(2,2)
M
```

```
##      [,1] [,2]
## [1,]    2    1
## [2,]    1    2
```

```
det(M)
```

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

To obtain the inverse M^{-1} of an invertible square matrix M, we use the solve function, solve(M).

```
Minv<-solve(M)
Minv
```

```
##            [,1]       [,2]
## [1,]  0.6666667 -0.3333333
## [2,] -0.3333333  0.6666667
```

A matrix multiplied by its inverse should equal the Identity matrix:

```
M %*% Minv
```

```
##      [,1] [,2]
## [1,]    1    0
## [2,]    0    1
```

If the matrix is singular (not invertible), or almost singular, solve gives an error message.

```
S<-c(1,2,2,4)
dim(S)<-c(2,2)
S
```

```
##      [,1] [,2]
## [1,]    1    2
## [2,]    2    4
```

```
det(S)
```

```
## [1] 0
```

```
solve(S)
```

To get the eigenvalues and eigenvectors of a matrix M, use eigen(M).

```
M<-c(2,1,1,2)
dim(M)<-c(2,2)
eigen(M)
```

```
## eigen() decomposition
## $values
## [1] 3 1
##
## $vectors
##           [,1]       [,2]
## [1,] 0.7071068 -0.7071068
## [2,] 0.7071068  0.7071068
```

Functions in R

If you have any experience using R, you have already used many, many built in functions. For example, `mean` is a built in R function that let's you compute the mean of an array of numbers

```
mean(1:8)
```

```
## [1] 4.5
```

`sum` is also a built in function.

```
sum(c(4, 5, 2, 3))
```

```
## [1] 14
```

Notice that to call a function in R we type the name of the function (e.g. `sum`) and two parentheses (`sum()`). Inside these parentheses, we give the function an **argument**. For example, the vector `c(4, 5, 2, 3)` is the argument we give to the function `sum`.

If you think about it, this is just like when you learned about mathematical functions. For example, take the equation

$$f(x_1, x_2, x_3) = x_1 + x_2 + x_3$$

f is a function that takes three arguments (x_1, x_2, x_3) and then computes their sum. The exact same concept applies for functions when programming.

Defining a Function

For more details click here

One really powerful thing about programming languages (like R), is that they allow you to define your own functions. This is something that is incredibly useful when doing any kind of dynamic modeling.

Writing your own functions (aka subroutines)

Let's start simple with the generalized format of a function:

```
myfun <- function(x, y, z) {
      expression 1
      expression 2
      expression n
      Output or return() or list()
}
```

In this generic example, the name of the function is myfun.

x, y, & z are the **arguments** for the function and will have to be provided by the user. They are essentially the raw material the function will need to perform its 'function' and can be vectors, scalars, or even matrices. It is also possible to give the function a default value for an argument by writing function(x, y=5, z) for example. If no value for y is provided, the function will use y=5.

The expressions within the function will describe what myfun will do to x, y, & z in order to produce what you want as an output.

The last line usually ends with what you want the function to output (this must be described within the expressions). If there is more than one object/number you want as output, use list(). The function return() can be especially useful for returning something from the middle of a function (if necessary). An output line is not absolutely necessary, only if you want R to return something.

Let's get some practice by defining a function `fahr_to_kelvin` that converts temperatures from Fahrenheit to Kelvin:

```
fahr_to_kelvin <- function(temp) {
  # Convert Far. to Kelvin
  kelvin <- ((temp - 32) * (5 / 9)) + 273.15
  return(kelvin)
}
```

1. We define `fahr_to_kelvin` by assigning it to the output of `function`.
2. The list of argument names are contained within parentheses. In this case there is one argument, `temp`, but there could be as many as you want, if you chose to define your function that way (i.e. `fahr_to_kelvin(temp1, temp2, temp3)`)
3. Next, the body of the function – the statements that are executed when it runs – is contained within curly braces (`{}`). The statements in the body are indented by two spaces, which makes the code easier to read but does not affect how the code operates. When we call the function, the arguments we pass to it are assigned to those variables so that we can use them inside the function.
4. Inside the function, we use a return statement to send a result back to whoever asked for it.

Using our own function is no different from using any other R function:

```
# Freezing point of water
fahr_to_kelvin(32)
```

```
## [1] 273.15
```

```
# Boiling point of water
fahr_to_kelvin(212)
```

```
## [1] 373.15
```

Notice inside the `fahr_to_kelvin` function there is a variable called `kelvin`. What if I did the following

```
fahr_to_kelvin(212)
kelvin
```

What happened? `kelvin` is only defined *within the scope* of the function. In other words, when the function `fahr_to_kelvin` is called it creates a temporary variable named `kelvin` and then deletes it once it is finished. So the global environment of R does not know what `kelvin` is. Similarly, there is no variable named `temp` in the global environment. R only knows about it inside the function. Check out this excellent tutorial for more information.

Numerical solutions to continuous-time systems, Ordinary Differential Equations (ODEs)

The Euler method, which we will discuss in class, is one of a number of numerical methods for solving ODEs, and it is the simplest method. A number of other methods have been devised to improve the speed and/or accuracy of numerical solultions to ODEs. Many of these methods use "adaptive time steps", that take very small time steps when the state variables are changing rapidly and larger time steps when the state variables are changing slowly. R has a number of built-in functions for solving systems of ODEs using many of these methods. In this exercise, we will use the `lsoda` function from the R package, `deSolve`, to do this (make sure you have the package `deSolve` installed and loaded).

Install the package, if you haven't already done so:

```
# install.packages("deSolve")
```

Load the package into R's working memory:

```
library("deSolve")
```

If you look up `lsoda` in the manual it says that it solves *"initial value problems for stiff or non-stiff systems of first-order ordinary differential equations (ODEs)"*.

What does that mean??

"initial value problem" means that you are supplying the state of the system at some point in time (e.g. the initial conditions at time t=0).

"stiff or non-stiff systems" is apparently harder to define. The Wikipedia definition of a "stiff equation" is: "In mathematics, a stiff equation is a differential equation for which certain numerical methods for solving the equation are numerically unstable, unless the step size is taken to be extremely small. It has proven difficult to formulate a precise definition of stiffness, but the main idea is that the equation includes some terms that can lead to rapid variation in the solution." `lsoda` decides whether or not your system of equations is "stiff", and switches automatically between stiff and non-stiff methods, so you don't have to worry about it.

"systems of first-order ODEs" means that we have any number of equations that include only first derivatives, e.g. $\frac{dS}{dt} = ...$

There are two steps involved in obtaining numerical solutions to differential equations:

- **Step 1.** Write a function `func` (this function can be named anything that you want), that calculates the right hand side of the differential equations. `func` must take as its first three arguments the current time (`t`), the current values of the state variables (`y`), and a vector containing the parameter values. It must also return a list (using `list(item1, item2, item3)`) whose elements the right hand sides of the ODEs.
- **Step 2.** Use `lsoda` to solve the system of ODEs.

Simulating a continuous-time SIR model in R

Let's use `lsoda` to solve the simple continuous-time SIR model:

This model has three **state variables**:

- S: the density of susceptible (uninfected) individuals
- I: the density of infected individuals
- R: the density of recovered individuals, who are now resistant to further infection

And, it has 2 **parameters**:

- β: Transmission parameter (units = infecteds^{-1}time^{-1})
- γ: Recovery rate (units = time^{-1})

The system of first-order ODEs that describes the rate of change of the 3 state variables are:

$$\frac{dS}{dt} = -\beta IS \frac{dI}{dt} = \beta IS - \gamma I \frac{dR}{dt} = \gamma I$$

- **Step 1.** The first step is to write a function that takes as input the current time t, a vector that contains the current values of the 3 state variables x, and a vector that contains the parameter values *params*, and returns a list containing the right hand side of the 3 ordinary differential equations (i.e. it returns the rate of change of the 3 state variables.)

```
SIR.model <- function (t, x, params) {
  S = x[1]
  I = x[2]
  R = x[3]
  beta = params[1]
```

```
  gamma = params[2]

  dSdt = -beta*S*I
  dIdt = beta*S*I - gamma*I
  dRdt = gamma*I

 return(list(c(dSdt,dIdt,dRdt)))
}
```

The first part of this function is really just to make it readable to us. The first 3 lines specify that within this function we are going to define S as the first term in the vector of state variables, I as the second term, and R as the third term. Similarly, the next 2 lines specify that within this function we are going to define β as the first term in the vector of parameters, and γ as the second term. We could have written the ODEs just in terms of $x[1]$, $params[1]$, etc, but that would make the code a lot more difficult to read!

The second part of our `SIR.model` function defines the ordinary differential equations that describe how the state variables change through time, and packs the results into a "list" to export the results.

The `lsoda` function will call our user-defined `SIR.model` function over and over again in order to calculate the numerical solution to this system of ODEs.

Although we don't usually have a reason to do this, we could call the `SIR.model` function by first specifying a value for time, a vector for the current value of the state variables, and a vector with the values of the parameters:

```
current_time=5
current_state=c(1,2,3)
my_parameters=c(0.1,2)

SIR.model(current_time,current_state,my_parameters)
```

```
## [[1]]
## [1] -0.2 -3.8  4.0
```

It should return a list that gives the rate of change of the 3 state variables.
For these made-up conditions, S is decreasing, I is decreasing, and R is increasing.

- **Step 2.** The second step is to use `lsoda` to solve the system of ODEs.

In order to call `lsoda`, we will use the code:

```
lsoda(initial_values, times, function, parameters)
```

The main arguments for `lsoda` are a vector with the starting values of the state variables (`initial_values`), a vector with the times at which you want to compute the values of the variables you are interested in (`times`), the derivative function (`function`), and a vector with the model parameters (`parameters`). (There are a number of additional optional arguments for `lsoda` that do things like set the maximum step size for the numerical algorithm. We won't worry about these for how.)

For the SIR model, `function` will be the `SIR.model` function that we just wrote.

`initial_values` is a vector that gives the starting values of the state variables in the model. They should be in the same order as in the user-defined function (i.e. for our case, in the same order as in the `SIR.model` function). Let's start with 1 infected individual in a population with a total of 100 individuals:

```
S0 = 99 # Initial number of susceptibles
I0 = 1 # Initial number of infecteds
R0 = 0 # Initial number of recovered
```

```
initial_values = c(S0,I0,R0)
```

`parameters` is a vector that gives the values for the model parameters. Again, they should be in the same order as in the user-defined function. Let's use the same parameter values that we used in the last exercise:

```
beta = 0.1 # per host per week
gamma = 1 # per week

parameters = c(beta,gamma)
```

`times` is a vector that lists all of the times for which you want `lsoda` to output the solution to the ODEs. Let's simulate the model for 5 weeks, outputing the data every 0.01 week.

Note: we do not need to specify the time step in `lsoda`. This method uses an adaptive time step, so it is actually using a time step that is changing through time. This part of the code is just specifying at what intervals you would like to see the results.

```
times = seq(0, 5, by=0.01)
```

We now have all of the bits that `lsoda` needs to obtain a numerical solution to the ODEs. We can have it save the results in the matrix `results`:

```
results = lsoda(initial_values, times, SIR.model, parameters)

# Naming columns for easy identification
colnames(results) = c("time", "S", "I","R")
```

R has simulated our model for 5 weeks and stored the values in the matrix `results`.

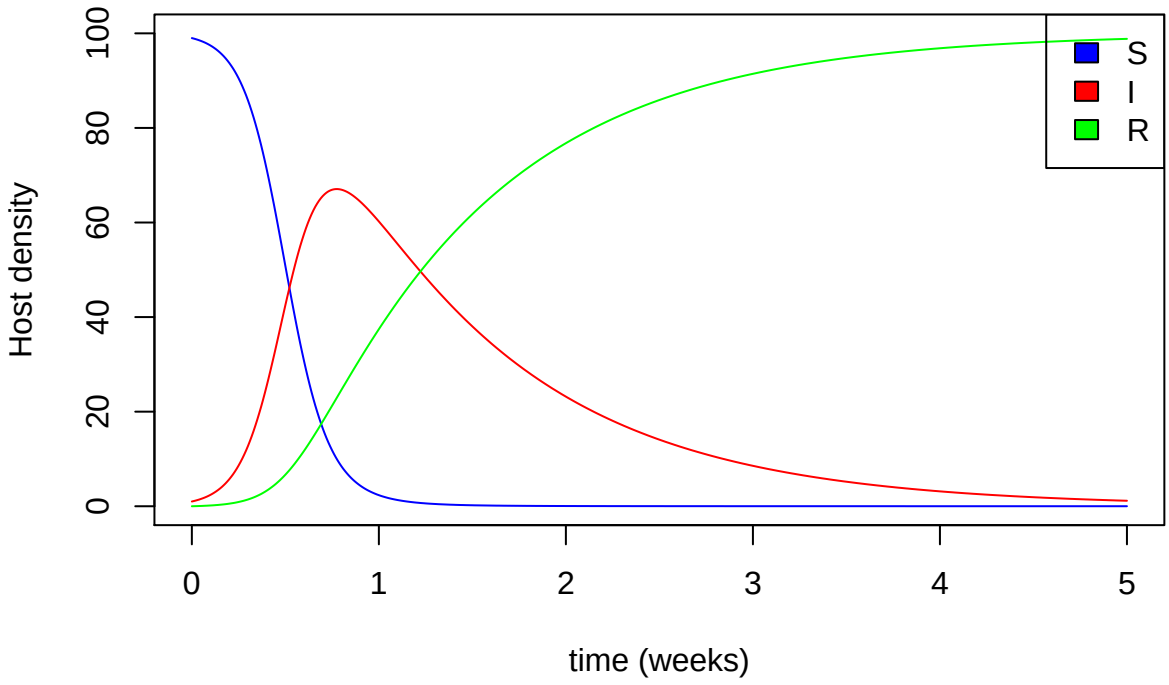
The output looks like this. The column order is: 1) time, 2) state variable 1 (S), 3) state variable 2 (I), and 4) state variable 3 (R).

```
head(results)
```

```
##      time        S        I          R
## [1,] 0.00 99.00000 1.000000 0.00000000
## [2,] 0.01 98.89652 1.093026 0.01045847
## [3,] 0.02 98.78354 1.194574 0.02188893
## [4,] 0.03 98.66021 1.305405 0.03438088
## [5,] 0.04 98.52564 1.426334 0.04803084
## [6,] 0.05 98.37881 1.558246 0.06294416
```

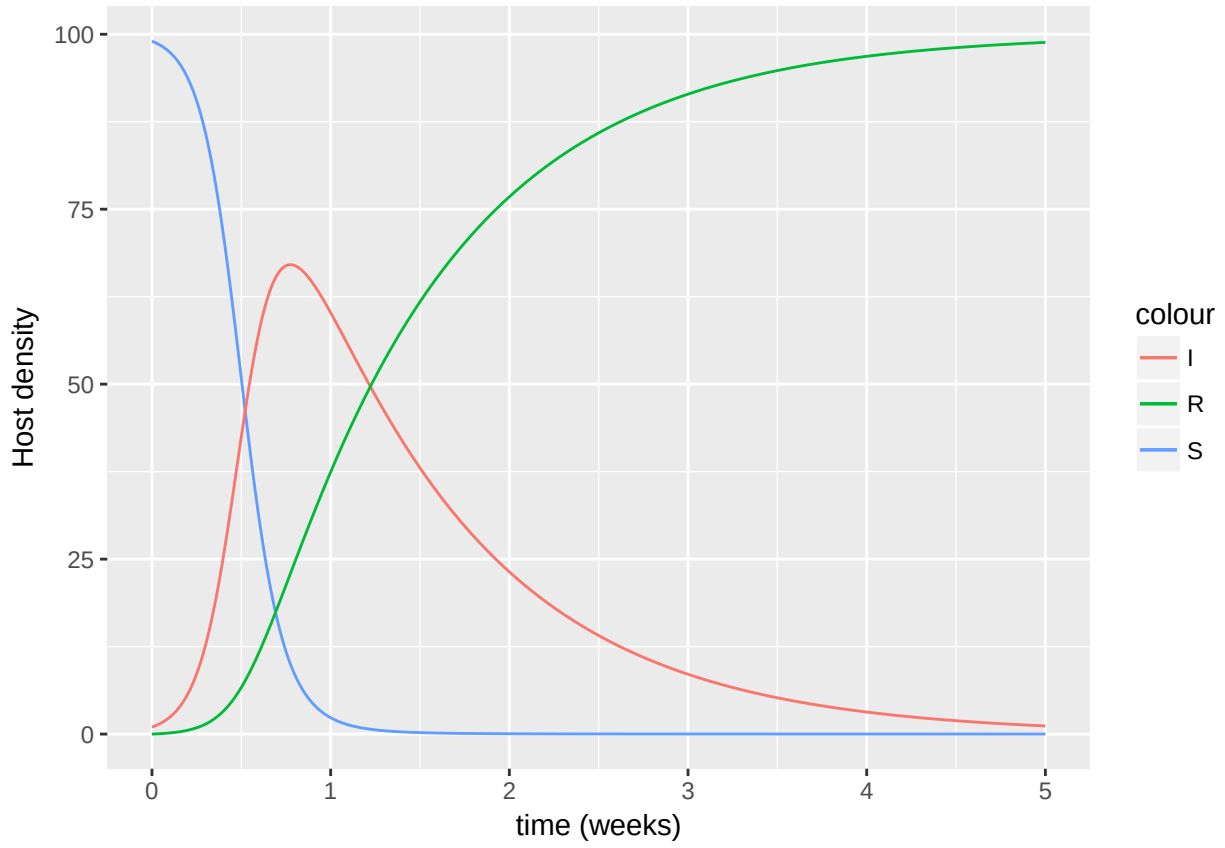
Now you can plot the results, using either R's base plot:

```
# Plotting in base plot
plot(results[, "time"], results[, "S"], type="l", col="blue",
     xlab="time (weeks)", ylab="Host density", ylim=c(0, S0 + 1))
lines(results[, "time"], results[, "I"], type="l", col="red")
lines(results[, "time"], results[, "R"], type="l", col="green")
legend("topright", legend=c("S", "I", "R"), fill=c("blue", "red", "green"))
```



or ggplot2 (you will need to install and load the R package ggplot2 to do this):

```
library(ggplot2)
# Plotting in ggplot
ggplot(data=NULL, aes(x=results[, "time"], y=results[, "S"], color="S")) + geom_line() +
    geom_line(data=NULL, aes(x=results[, "time"], y=results[, "I"], color="I")) +
    geom_line(data=NULL, aes(x=results[, "time"], y=results[, "R"], color="R")) +
  xlab("time (weeks)") + ylab("Host density")
```



It is often easier to use the output from the ODE solver if you convert it to an R dataframe and label the columns. You can do this by changing these two lines of code:

```
# OPTIONAL CODE TO STORE OUTPUT AS A DATAFRAME
initial_values = c(S=S0,I=I0,R=R0)
results = as.data.frame(lsoda(initial_values, times, SIR.model, parameters))
```

Calculating R_0

Consider the initial arrival of the pathogen into an entirely susceptible population (i.e. when all individuals are in the susceptible class, $S(0) = N$). What factors will determine whether an epidemic will occur, or if the pathogen will fail to invade?

The pathogen can invade a fully susceptible population when $\frac{dI}{dt} > 0$.

For the SIR model with density-dependent transmission, the equation for the rate of change of the density of infected can be re-written as: $\frac{dI}{dt} = I(\beta S - \gamma)$

Thus, for the SIR model with density-dependent transmission, $\frac{\gamma}{\beta}$ is the **threshold density for pathogen invasion**. If the initial density of susceptible hosts is greater than $\frac{\gamma}{\beta}$, then the pathogen can invade, and if the initial density of susceptible hosts is less than $\frac{\gamma}{\beta}$ then the infection dies out.

This can be rearranged to show that the disease can invade a fully susceptible population only when (here we're assuming that during the initial stages of the pathogen invasion, all individuals are in the suceptible class, so the total density of hosts (N) is equal to the density of susceptible hosts (S):

$$R_0 = \frac{\beta N}{\gamma} > 1$$

R_0 is called the **basic reproductive ratio**. It is defined as the *average number of secondary cases arising from an average primary case in an entirely susceptible population.* R_0 can be calculated as the rate at which new cases are produced by an infectious individual (βN) multiplied by the average duration of the infectious period ($1/\gamma$).

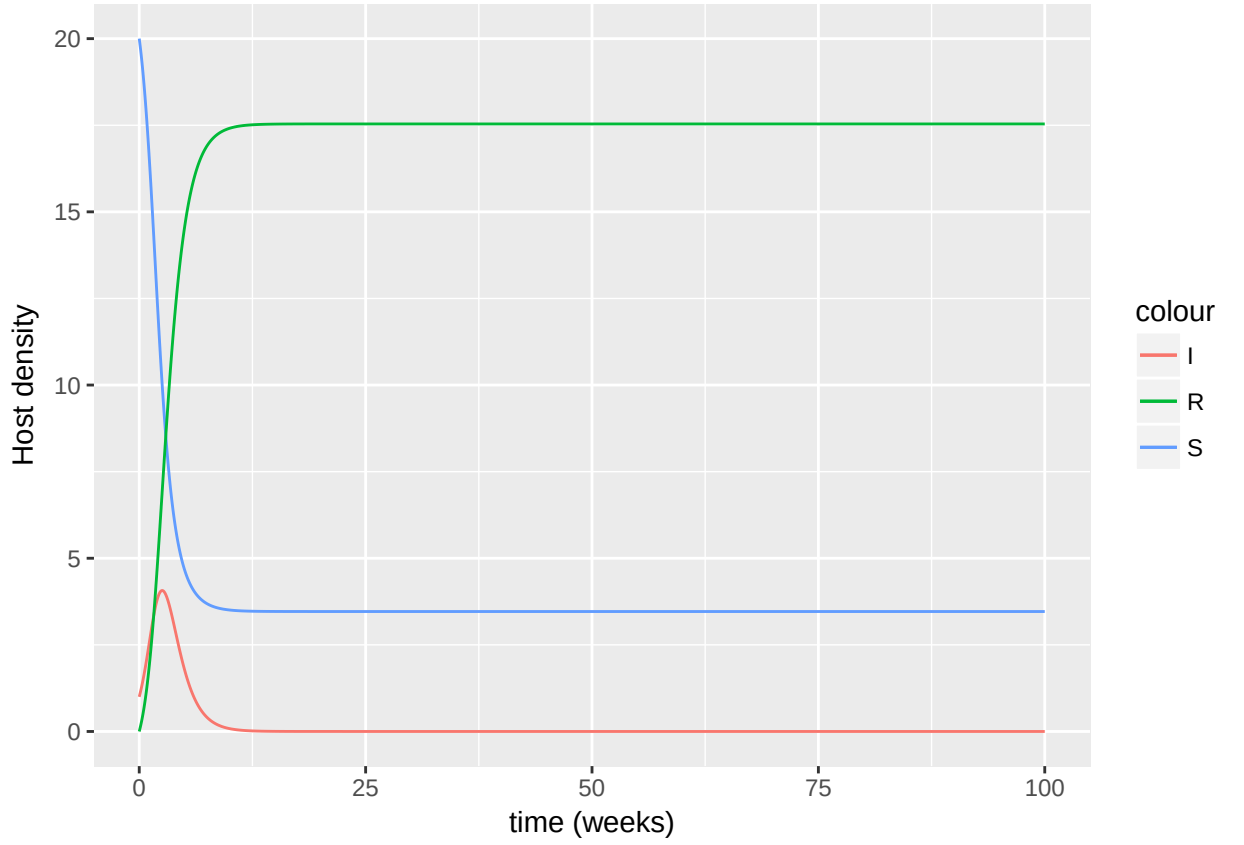
Exercise 1: Explore the effects of changing the initial conditions.

(a) Prove to yourself that $\frac{\gamma}{\beta}$ is the threshold population density for pathogen invasion. Try varying the initial density of the susceptible population. What is the lowest initial density of susceptibles for which the infected population can have a positive growth rate?

(b) In the cases where the pathogen can invade (i.e. for $S(0) > \frac{\gamma}{\beta}$), do all individuals in the population eventually get infected? (You'll need to extend the maximum time for the simulations to see what happens in the long term.)

```
#HIDE ME IF YOU DON'T WANT TO SEE THE ANSWER

# ALL OF THE CODE FOR THE DENSITY-DEPENDENT MODEL

SIR.model <- function (t, x, params) {
  S = x[1]
  I = x[2]
  R = x[3]
  beta = params[1]
  gamma = params[2]

  dSdt = -beta*S*I
  dIdt = beta*S*I - gamma*I
  dRdt = gamma*I

 return(list(c(dSdt,dIdt,dRdt)))
}

S0 = 20 # Initial number of susceptibles
I0 = 1 # Initial number of infecteds
R0 = 0 # Initial number of recovered
initial_values = c(S=S0,I=I0,R=R0)

beta = 0.1 # per host per week
gamma = 1 # per week
parameters = c(beta,gamma)

times = seq(0, 100, by=0.01)

results = lsoda(initial_values, times, SIR.model, parameters)
colnames(results) = c("time", "S", "I","R")

# Plotting in ggplot
ggplot(data=NULL, aes(x=results[, "time"], y=results[, "S"], color="S")) + geom_line() +
    geom_line(data=NULL, aes(x=results[, "time"], y=results[, "I"], color="I")) +
```

```
  geom_line(data=NULL, aes(x=results[, "time"], y=results[, "R"], color="R")) +
xlab("time (weeks)") + ylab("Host density")
```



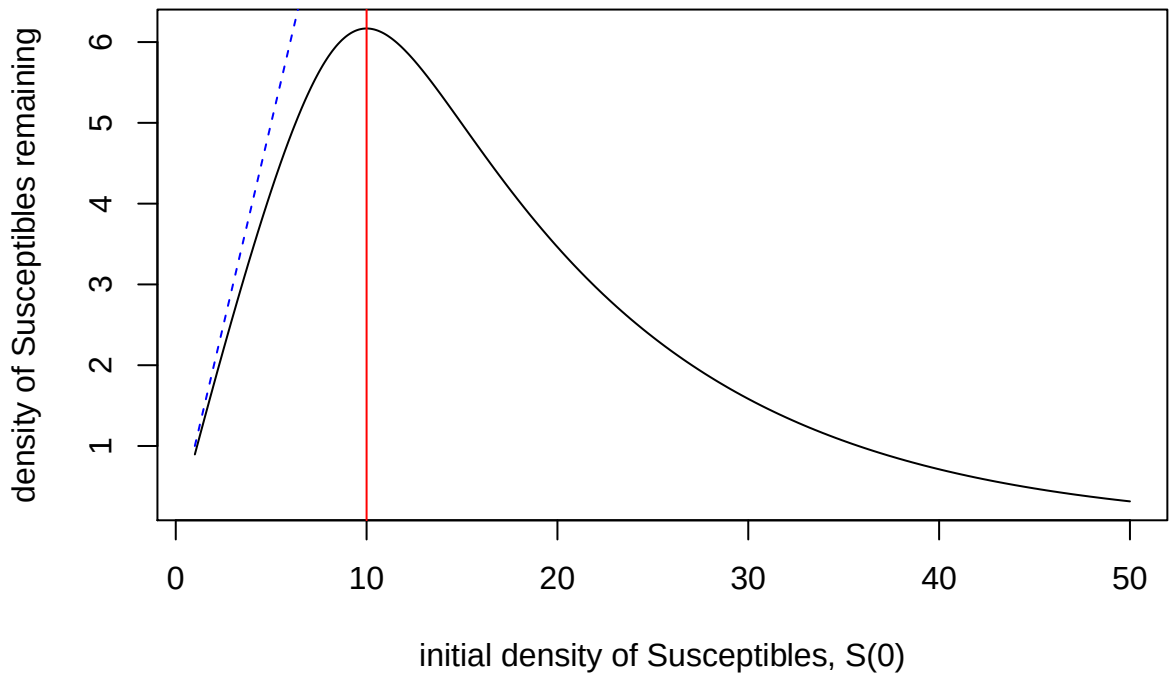
A few things to note are:

- In this **deterministic** version of the model, there are always some susceptible hosts in the population that escape infection (i.e. S is never driven completely to zero, although S can get really, really small).
- The larger the initial density of susceptible hosts, the greater the fraction of hosts that become infected.
- The chain of transmission eventually breaks due to the decline in infectives, not due to the complete lack of susceptibles. **"epidemic burnout"**.

We can illustrate some of these properties through simulation, using a `for` loop.

```
Tend = 100 # end time for simulations
times = seq(0, Tend, by=1)

beta = 0.1 # per host per week
gamma = 1 # per week
parameters = c(beta,gamma)

# start all simulations with 1 infected and 0 recoverds
I0 = 1 # Initial number of infecteds
R0 = 0 # Initial number of recovereds

S0_all<-seq(1,50,0.1)       # use a sequence of values for S0

# set up a vector to store host equilibrium values
```

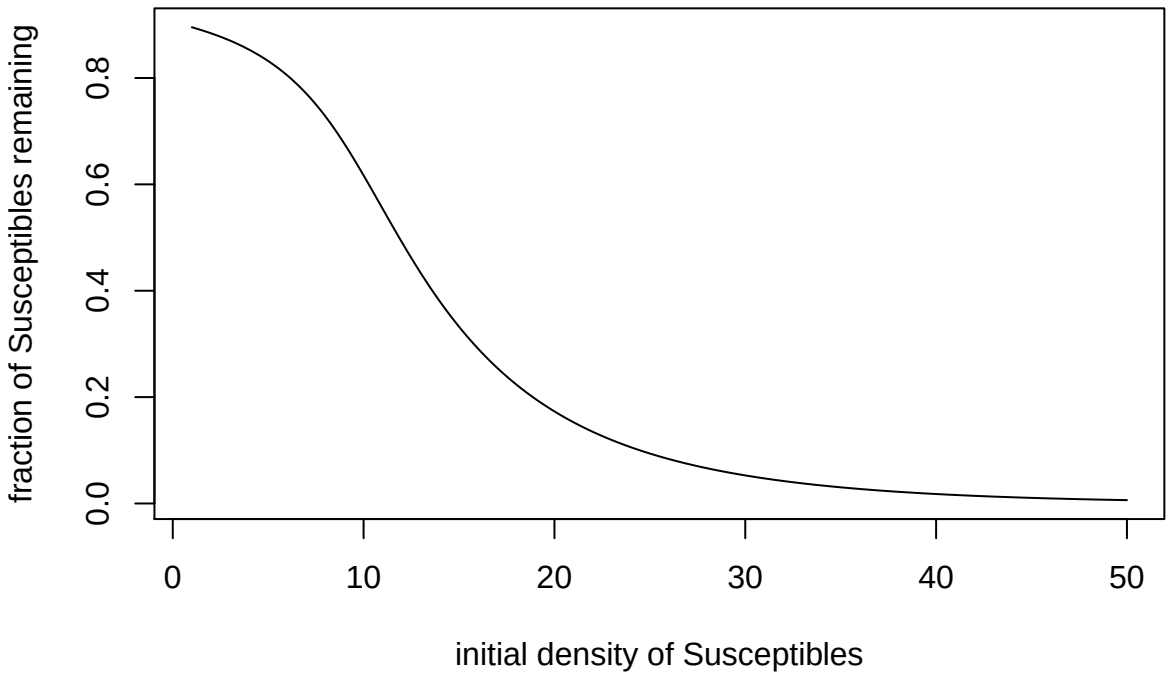
```
S_longterm = rep(0,length(S0_all))
I_longterm = rep(0,length(S0_all))
R_longterm = rep(0,length(S0_all))

for (i in 1:length(S0_all)) {  # loop through all values of S0

    #specify the initial values of state-variables
    initial_values = c(S0_all[i],I0,R0)

    results = lsoda(initial_values, times, SIR.model, parameters)

    #save only the final values of the state variable
    S_longterm[i] = results[Tend+1,2]
    I_longterm[i] = results[Tend+1,3]
    R_longterm[i] = results[Tend+1,4]
}

#plot the final vs. initial number of susceptibles
plot(S0_all, S_longterm, type = "l", xlab="initial density of Susceptibles, S(0)", ylab="density of Sus
lines(S0_all, S0_all, type="l", lty=2, col="blue")
abline(v=gamma/beta,col="red")
```



The blue line has a slope of 1, that is, where the final density of Susceptibles equals the initial density of susceptibles.
The red line is the threshold density of Susceptibles for pathogen invasion.

We could also plot the fraction of Susceptibles that survive as a function of the initial density of Susceptibles:

```
frac_susceptible = S_longterm/S0_all

plot(S0_all, frac_susceptible, type = "l", xlab="initial density of Susceptibles", ylab="fraction of Su
```



Exercise 2: Explore the effects of changing the transmission function

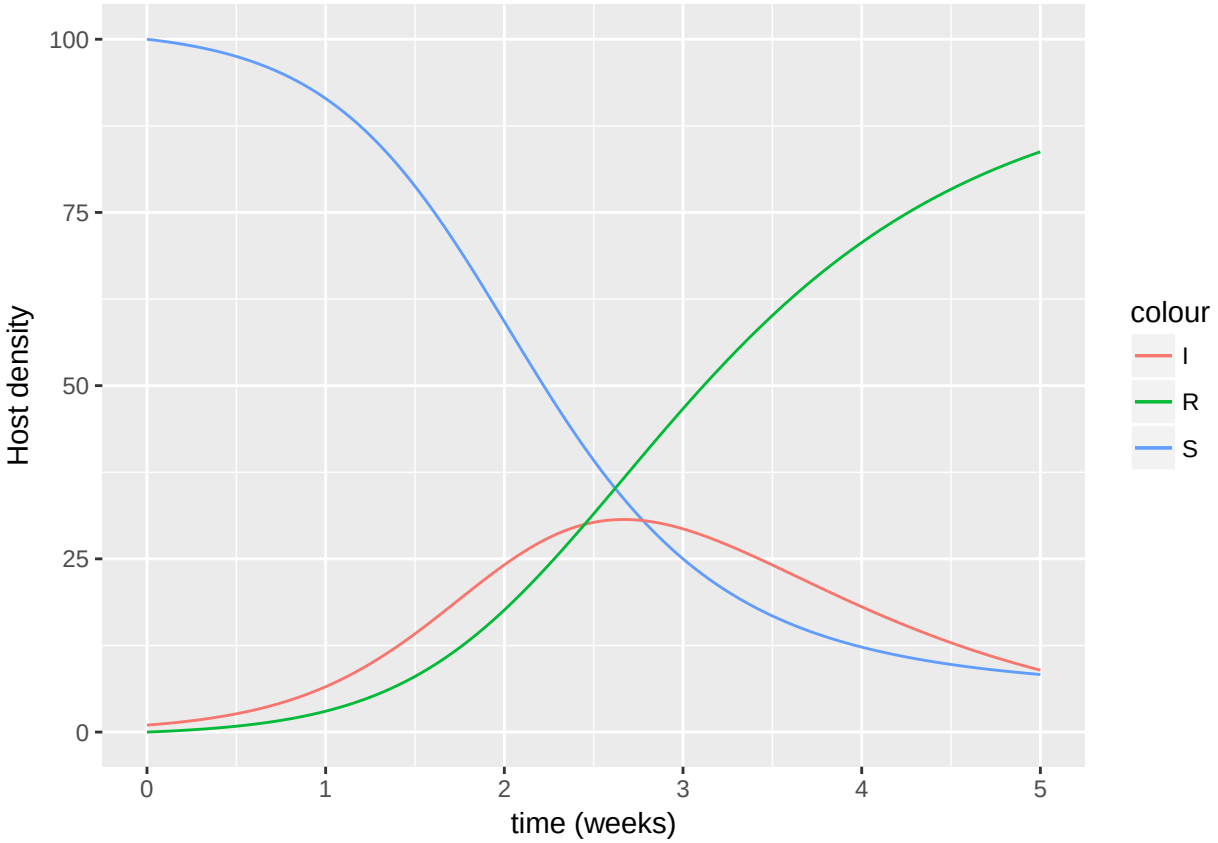
In the version of the model above, we assumed that the transmission rate is density dependent. That is, the force of infection is proportional to the density of infected individuals in the population, βI, and the rate loss of susceptible individuals is βIS.

We could instead assume that the transmission rate is frequency dependent. For frequency dependent transmission, the force of infection is proportional to the fraction of infected individuals, $\frac{\beta I}{N}$, where $N = S + I + R$ is the total population size, and the rate of loss of susceptible individuals is $\frac{\beta IS}{N}$.

Code up the model with frequency-dependent transmission. Now try varying the size of the susceptible population. In this case, is there a threshold population size below which the infected population cannot have a positive growth rate?

```
# HIDE ME IF YOU DON'T WANT TO SEE THE ANSWER

# ALL OF THE CODE FOR THE FREQUENCY-DEPENDENT MODEL

#new function with frequency-dependent transmission rate
SIR.freqdep.model <- function (t, x, params) {
```

```
  S = x[1]
  I = x[2]
  R = x[3]
  beta = params[1]
  gamma = params[2]

  N=S+I+R
  dSdt = -beta*S*I/N
  dIdt = beta*S*I/N - gamma*I
  dRdt = gamma*I

 return(list(c(dSdt,dIdt,dRdt)))
}


S0 = 100 # Initial number of susceptibles
I0 = 1 # Initial number of infecteds
R0 = 0 # Initial number of recovered
initial_values = c(S=S0,I=I0,R=R0)
#
beta =  3 # per week
gamma = 1 # per week
parameters = c(beta,gamma)
#
times = seq(0, 5, by=0.01)
#
results.freqdep = as.data.frame(lsoda(initial_values, times, SIR.freqdep.model, parameters))
colnames(results) = c("time", "S", "I","R")
#
# Plotting in ggplot
ggplot(data=NULL, aes(x=results.freqdep[, "time"], y=results.freqdep[, "S"], color="S")) + geom_line() +
    geom_line(data=NULL, aes(x=results.freqdep[, "time"], y=results.freqdep[, "I"], color="I")) +
    geom_line(data=NULL, aes(x=results.freqdep[, "time"], y=results.freqdep[, "R"], color="R")) +
  xlab("time (weeks)") + ylab("Host density")
```



If you kept the values for the parameters the same as in the density-dependent model, you will find that the pathogen always fails to invade. Why is that?

Remember, the pathogen can invade a fully susceptible population when $\frac{dI}{dt} > 0$.

For the SIR model with frequeny-dependent transmission, the equation for the rate of change of the density of infected can be re-written as: $\frac{dI}{dt} = I(\frac{\beta S}{N} - \gamma)$

When the host population is fully susceptible, $S = N$ so the pathogen can invade only when $\beta > \gamma$. For the frequency-dependent model, the criterion for invasion does not depend on the density of susceptible hosts. That is, there is no threshold density for pathogen invasion.

But, in oder for the pathogen to have a positive growth rate, we need to increase the value of the transmission parameter, β. Note, that the **units** of the transmission parameter are different between the density-dependent and frequency-dependent model.

In the density-dependent model, β has units of $\frac{1}{host*time}$

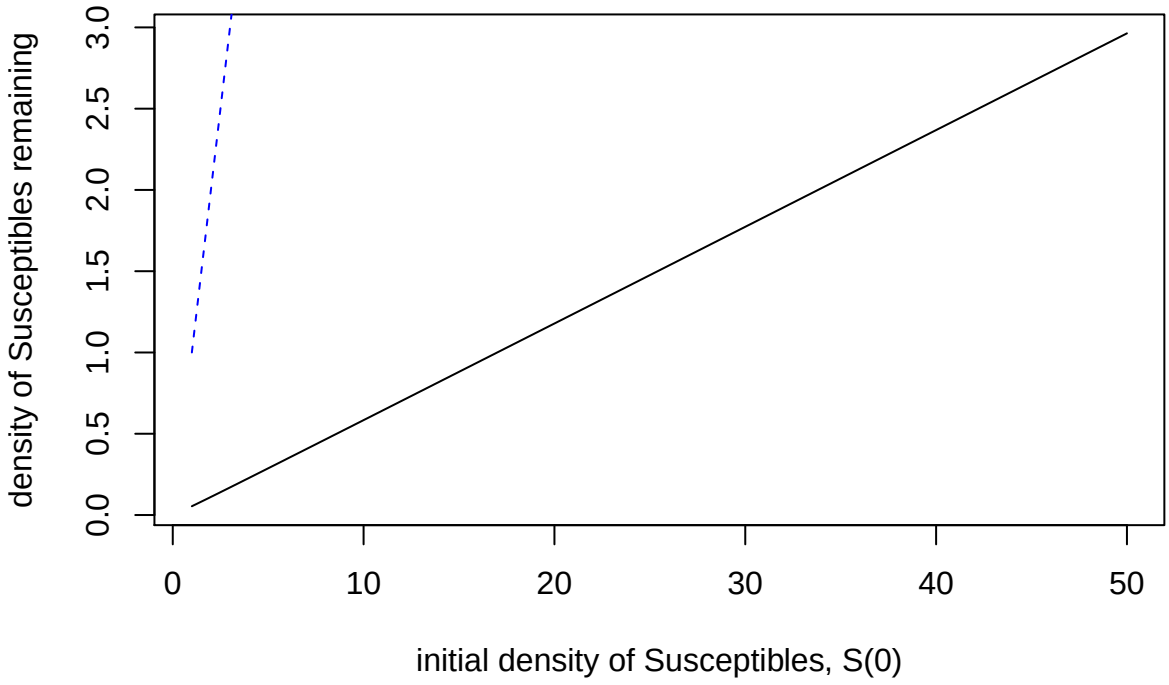
In the frequency-dependent model, β has units of $\frac{1}{time}$

Once again, this can be rearranged to show that the disease can invade a fully susceptible population only when:

$$R_0 = \frac{\beta}{\gamma} > 1$$

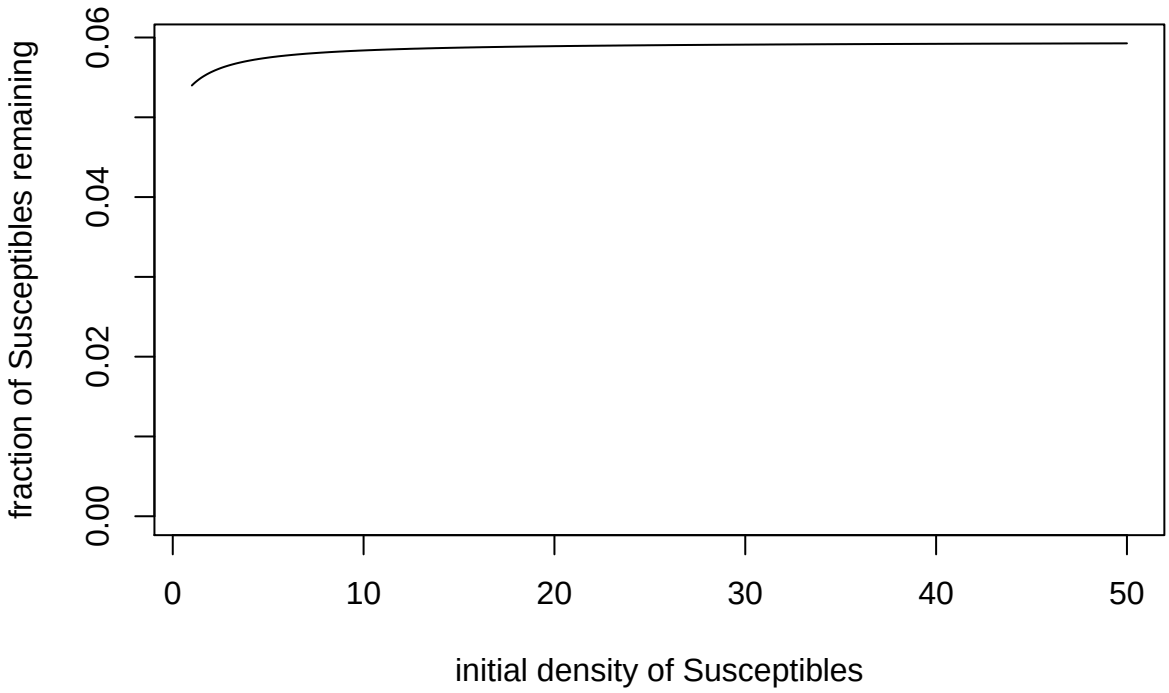
How does changing the initial density of susceptible hosts affect the number or fraction of hosts that become infected during the outbreak?
We can use our code with the `for` loop once again to find this out.

```
Tend = 500 # end time for simulations
times = seq(0, Tend, by=1)

beta = 3 # per week
gamma = 1 # per week
parameters = c(beta,gamma)

# start all simulations with 1 infected and 0 recoverds
I0 = 1 # Initial number of infecteds
R0 = 0 # Initial number of recovereds

S0_all = seq(1,50,0.1)      # use a sequence of values for S0

# set up a vector to store host equilibrium values
S_longterm.freqdep = rep(0,length(S0_all))
I_longterm.freqdep = rep(0,length(S0_all))
R_longterm.freqdep = rep(0,length(S0_all))

for (i in 1:length(S0_all)) {  # loop through all values of S0

    #specify the initial values of state-variables
    initial_values = c(S0_all[i],I0,R0)

    results.freqdep = lsoda(initial_values, times, SIR.freqdep.model, parameters)

    #save only the final values of the state variable
    S_longterm.freqdep[i] = results.freqdep[Tend+1,2]
    I_longterm.freqdep[i] = results.freqdep[Tend+1,3]
    R_longterm.freqdep[i] = results.freqdep[Tend+1,4]
}

#plot the final vs. initial number of susceptibles
plot(S0_all, S_longterm.freqdep, type = "l", xlab="initial density of Susceptibles, S(0)", ylab="densit
lines(S0_all, S0_all, type="l", lty=2, col="blue")
```



We could also plot the fraction of Susceptibles that survive as a function of the initial density of Susceptibles:

```
frac_susceptible = S_longterm.freqdep/S0_all

plot(S0_all, frac_susceptible, type = "l", xlab="initial density of Susceptibles", ylab="fraction of Su
```



Exercise 3: Try it yourself!

Try making any other modification to the SIR model that you're interested in. For example, maybe individuals have to go through an exposed class after they become infected, but before they are infectious. Or, maybe some fraction of the population gets vaccinated and has a reduced susceptibility to the virus. You decide. Explore your modification to the model using numerical simulations.