EEB 201 – Introduction to R for Ecology and Evolutionary Biology

… aka 'Bootcamp'

Course description

Lecture, six hours; discussion, six hours. 1 unit

Requisites: Enrollment in EEB doctoral program or permission of instructors.

Introduction to basic concepts and practice of scientific programming, using the language R. Topics include working at the command line, writing scripts and functions, flow control, graphics, and conducting basic simulations in discrete and continuous time.

Offered as intensive two-day course at beginning of quarter. S/U grading.

Learning objectives for my section

By the end of this section you will:

- Have a working knowledge of control statements in R
- Be able to write your own computer programs to simulate and plot population dynamics (in discrete time).
- Convert your simulation to a function, and write a program to perform sensitivity analysis.

- Have seen how to write your own computer programs to simulate and plot population dynamics in continuous time (ODE models)

Lightning review of day 1…

Programming in R

- **Scripts** are simply a collection of commands saved in a text file.
  - Make sure you use a non-formatted text editor, not a word processor!
  - Save the script in your working directory
  - Run the script using the `source()` command

- **Functions** are objects in R's workspace (just like variables)
  - We can define functions to do specific tasks.
  - Note: we need to get functions into R's workspace, by `source`-ing them once (or entering at the command line).

Flow control: conditional execution

- Sometimes we only want to execute part of our program under certain conditions.

- Use an **if statement**
  - express your condition as a logical expression

```
xx > 10                    xx == 2
xx >= yy                   length(xx) != 0
xx < 2 | xx > 20
```

  - then you can set a block of code to be executed only **if** that condition is fulfilled

Flow control: conditional execution

Syntax in R

```
if (logical_expression) {
      expression_1
      ...
}
```

or for two options

```
if (logical_expression) {
      expression_1
      ...
} else {
      expression_2
      ...
}
```

Flow control: looping

- Often there will be tasks that you want to repeat many times.
- Can accomplish this using a **for loop** or a **while loop**.

- **for loops**
  - Use to repeat a fixed number of times.
  - Define a vector with a set of values, and go through the loop once for each value.
  - Often: use index variable, e.g. (ii in 1:10)

- **while loops**
  - Use to repeat until some condition isn't met anymore.
  - Define the logical condition that needs to be satisfied.
  - Repeat the loop until the condition is false.

Flow control: looping

Syntax in R – for loops

```
for (xx in 1:100) {
      expression_1
      ...
}
```

or more generally

```
for (xx in someVector) {
      expression_1
      ...
}
```

Syntax in R – while loops

```
while (logical_expression) {
      expression_1
      ...
}
```

Flow control: looping

Example

- Two ways of doing an operation 10 times

With a **for loop**

```
nn <- 10
RR <- 1.1

for (tt in 1:10) {
      nn[tt+1] <- RR*nn[tt]
}
```

With a **while loop**

```
nn <- 10
RR <- 1.1

tt <- 1

while (tt <= 10) {
      nn[tt+1] <- RR*nn[tt]
      tt <- tt+1
}
```

Programming style

- Be clear rather than clever.
  - You'll be glad when you have to come back to a program you wrote two years ago…. to analyze new data or modify your analysis.

- Use meaningful variable names
  - birthRate and probSurvival instead of b and p

- When using simple variable names (x, y, i, j, etc), it's better to use double letters (xx, yy) since they're easier to find (and replace) using automated text search functions.

Programming style

- Comments! `# you can never have too many`

- Put a line or two of comments at the top of your program to say what it does.
  - Use 'sub-headings' to mark sections of code as well

- White space
  - leave blanks lines to separate blocks of code
  - use indentation consistently to indicate lines of code within loops or if statements
  - R doesn't insist on this like some languages, but it will help you keep your thinking straight, keep your parentheses matched up, and avoid bugs.

Pseudo-code

- **Pseudo-code** is an informal way to plan out the structure and logic of your programs.

- When pseudo-coding, DON'T worry about the syntax of the programming language you're using, but DO pay attention to the variables and flow control structures you will use.

- Because the basic structures of many high-level programming languages are the same, your pseudo-code will translate easily R, Matlab, python, or many other languages.

Pseudo-code example

Goal: Determine how many students pass the bootcamp.

```
# set the number of passes to zero
# set the number of fails to zero

# loop over the number of students in the class
  # if the student completed their exercises from day 1
     # add one to the number of passes
  # or else if they didn't complete their exercises
     # add one to the number of fails
     # print "Come on, this is the first thing we've asked you do in grad
                    school!"

# show the result
```

Exercise break: Pseudo-code

The **geometric growth model** is the simplest model for population growth in discrete time. It assumes that every year the size of the population changes by the same factor, R.

$$N(t+1) = R \times N(t)$$

Exercise: write pseudo-code for a program that simulates the growth of a population for 10 years, starting with N=100 animals and assuming R = 1.05, and prints the final population size.

Bonus: modify the pseudo-code so the program will make a plot of N versus t.

General layout of modeling scripts

1. Setup statements, if needed (e.g. loading packages)

2. Input data, set parameter values, and/or set initial conditions

3. Perform the calculations

4. Display the results by plotting, saving, or showing on-screen.

One useful habit is to lay out the program in pseudocode, then use the pseudocode as commented headings in your real code.

Exercise break: your first model

1. Modify the pseudocode for your geometric growth model, using this general layout as a template.

    1. Setup statements, if needed (e.g. loading packages)
    2. Input data, set parameter values, and/or set initial conditions
    3. Perform the calculations
    4. Display the results by plotting, saving, or showing on-screen.

2. Fill this in with actual R code, so you have a program that simulates the geometric growth model for given initial value (N0) and R, and plots the result.

A program for the geometric model

```
# geometricGrowthScript.R
# a script to simulate and plot the discrete logistic model

# Setup


# Set initial conditions and parameter values




# initialize variable to a vector of NA values


# use a loop to iterate the model the desired number of times


# plot the results
```

Exercise break: your first model

2. Fill this in with actual R code, so you have a program that simulates the geometric growth model for given initial value (N0) and R, and plots the result.

    - Save this script, with an appropriate name. Your first modeling script!

3. Modify this program to simulate the discrete logistic growth model, which is defined by this equation:

$$N(t+1) = N(t)\left(1 + r_d\left(1 - \frac{N(t)}{K}\right)\right)$$

   What do you need to change in the code? (how many lines will change?)

4. Play around with your model. Use different parameter values of N0 and R, to figure out the conditions where your population will approach carrying capacity, shrink to zero, or do something else.

So that was fun.

But also kind of annoying.

Wouldn’t it be nice if there was an efficient way to re-run your program with different parameter values?

Programming your own functions

Writing your own functions is the most powerful way to make R work flexibly and efficiently for you.

→ User-defined functions enable you to reduce long stretches of complicated programming to a single command.

A function takes in arguments, performs some operations, and returns a value.

To define a function, you need to

- choose a name for the function (avoid reserved words)
- define the arguments that need to be passed to the function
- define the operations
- set what value the function returns

Programming your own functions

R syntax

```
name <- function(argument_1, argument_2, ...) {
      expression_1
      expression_2
      ...
      return(output)
}
```

Example: program to build a three-digit number

```
> numberify <- function(xx, yy, zz) {
+     myNumber <- xx * 100 + yy * 10 + zz
+     return(myNumber)
+ }
```

Inputs: default values for arguments

You can assign default values to function arguments by giving them a value in the function definition.

```
> numberify <- function(xx=1, yy=1, zz=1) {
+     myNumber <- xx * 100 + yy * 10 + zz
+     return(myNumber)
+ }
```

Any argument with a default value is optional when you call the function. If you pass fewer values than the function has arguments, then R will assign them from left to right, unless you name the arguments as you pass values.

```
> numberify()
[1] 111
> numberify(3,2)
[1] 321
```

```
> numberify(3)
[1] 311
> numberify(yy=3)
[1] 131
```

The function workspace and variable scope

- When a function is executed, R sets aside memory for a separate workspace for the function to operate.

- The function can define new variables and conduct all its operations in complete isolation from the main workspace.

- When the operations are complete, the function returns the appropriate quantity and all other information in the function workspace is **lost**.

  → If you want to use something outside the function, you need to `return` it!

Stopping and outputs

The function will continue evaluating until:

- it hits the first `return()` statement, after which it returns the requested value and quits.
  - note that there can be multiple `return()` statements in a function, e.g. in a branching program structure

- it runs out of commands to run, without hitting a `return()` statement.
  - sometimes the returned value isn't the point of the function, e.g. in a function whose purpose is to make a plot.
  - In this case, if you try to assign the function output to a variable, it returns the value of the last expression it evaluated (even if NULL).

If you want something returned, code it explicitly with `return()`.

Return to logistic model

How can we program the discrete logistic model as a function?

- what do we want it to do?
- what arguments do we want to pass to it?
- what outputs do we want?

(Pseudo)code for a model function is very similar to a script, except many of the parameter values and initial conditions are often passed as arguments to the function, where they were set 'manually' in the script.

Example: converting a model script to a function

By converting a script to a function, you can make it quick and easy to run the model with different parameter values or initial conditions.

The conversion process is easy – for example:

```
# Simple model script
# Define parameter values and ICs
RR <- 1.05
N0 <- 100
ttMax <- 10

# Initialize vector to hold output
NN <- rep(NA,ttMax+1)
NN[1] <- N0

# Use a for loop to step forward
for (tt in 1:ttMax){
    NN[tt+1] <- RR*NN[tt]
}

plot(1:(ttMax+1),NN,lty=2,type='l',...
    xlab='t', ylab='Population size')
```

Basic change: just moved parameters, ICs from body of script to function arguments

```
# Function version

geomFun <- function(RR, N0, ttMax){
    # Initialize vector to hold output
    NN <- rep(NA,ttMax+1)
    NN[1] <- N0

    # Use a for loop to step forward
    for (tt in 1:ttMax){
        NN[tt+1] <- RR*NN[tt]
    }

    plot(1:(ttMax+1),NN,lty=2,type='l',
        ... xlab='t', ylab='Population
        size')
}
```

Example: converting a model script to a function

By converting a script to a function, you can make it quick and easy to run the model with different parameter values or initial conditions.

The conversion process is easy – for example:

```
# Simple model script
# Define parameter values and ICs
RR <- 1.05
N0 <- 100
ttMax <- 10

# Initialize vector to hold output
NN <- rep(NA,ttMax+1)
NN[1] <- N0

# Use a for loop to step forward
for (tt in 1:ttMax){
    NN[tt+1] <- RR*NN[tt]
}

plot(1:(ttMax+1),NN,lty=2,type='l',...
    xlab='t', ylab='Population size')
```

Basic change: just moved parameters, ICs from body of script to function arguments

```
# Function version

geomFun <- function(RR, N0, ttMax){
    # Initialize vector to hold output
    NN <- rep(NA,ttMax+1)
    NN[1] <- N0

    # Use a for loop to step forward
    for (tt in 1:ttMax){
        NN[tt+1] <- RR*NN[tt]
    }

    plot(1:(ttMax+1),NN,lty=2,type='l',
        ... xlab='t', ylab='Population
        size')
    return(NN)
}
```

Example: converting a model script to a function

Now you can run the full model, for any parameter values and initial conditions, using a single statement (by *calling* the function).

```
> geomFun(RR=1.01, N0=200, ttMax=20)

> myOutput <- geomFun(RR=1.01, N0=200, ttMax=20)
```

Exercise break

Convert your discrete logistic model into a function.

Use this function to explore the model's dynamics with ease and grace.

Exploring your model systematically:

sensitivity analysis

Now you can run your whole model in a single line.

You're in a position to do fancier things, such as running the model many times to analyze the sensitivity to different parameter values.

Pseudocode for a sensitivity analysis

Define parameter values; use a vector to hold a range of values for the parameter(s) you wish to vary.

Initialize a matrix to collect all outputs

Use a **for loop** to repeatedly run the model and collect output

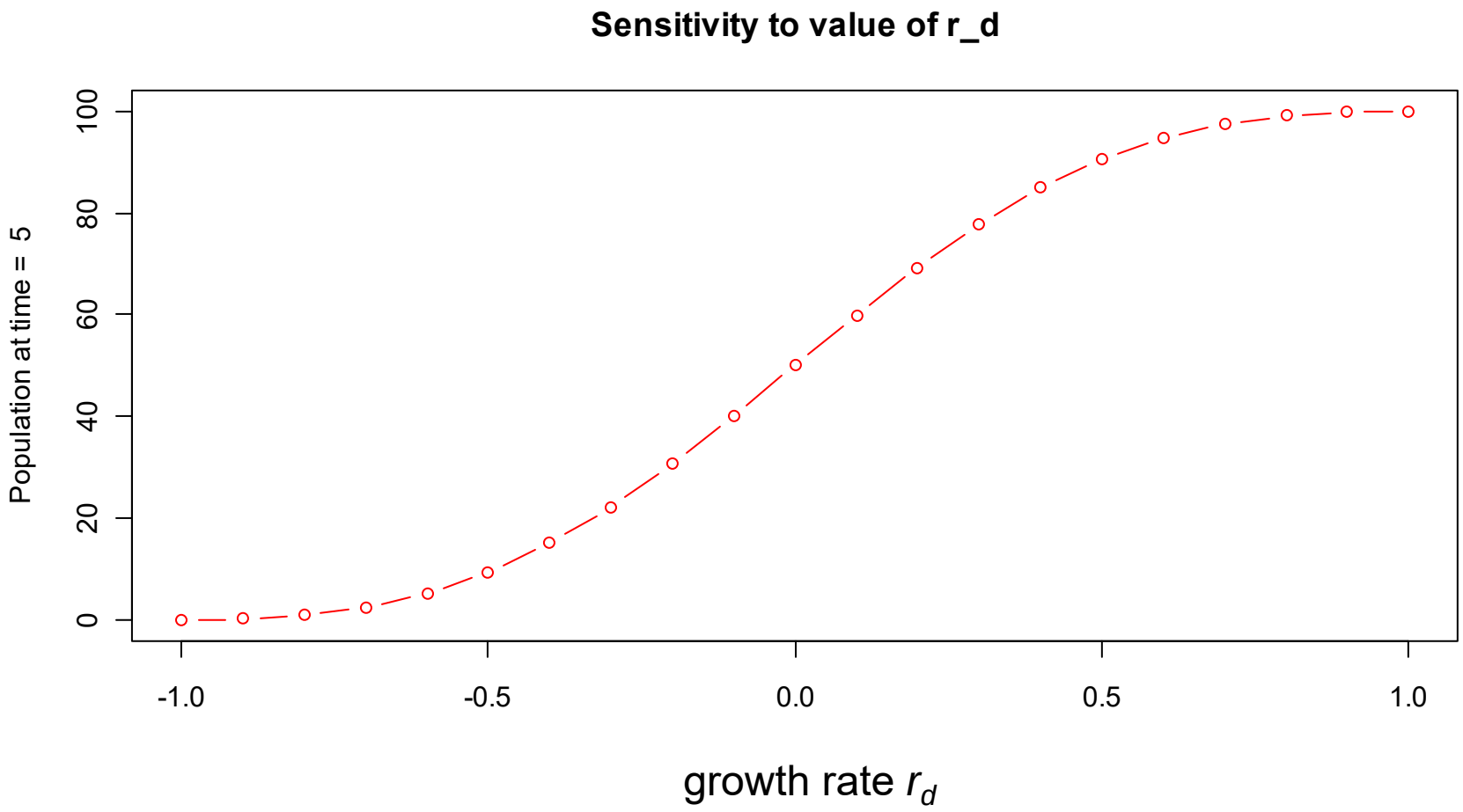
    # NOTE: this for loop is **NOT** over the timesteps of the model, it is over the set
    # of different parameter values for which you want to run the model.
    # Use your function to run the model in a single line within the loop.

analyze results

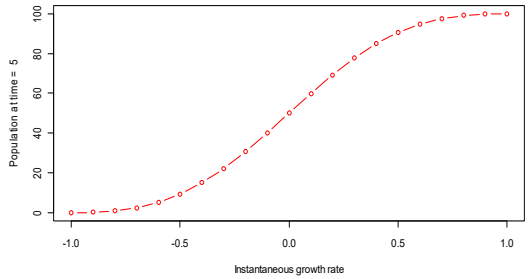
Sensitivity analysis: response to parameter values

How does the population size in 5 years depend on growth rate r_d?

(initial conditions and other parameter values held constant)



To make a plot of model summary output versus parameter value



Define parameter values

for parameter of interest, make a vector of values you want to consider

Initialize vector to hold summary values

Use a **for loop** to repeatedly run the model and plot output

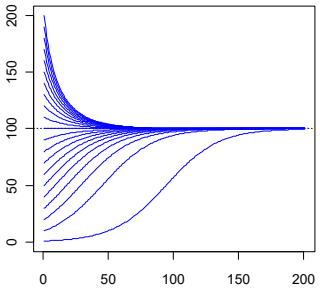
    # You are looping over the list of values you want to use for the
    # parameter you're varying.
    # Each time through loop, run the model with the current parameter
    # values and store the summary values in the i^{th} element of your
    # results vector

Plot the results vector versus the vector of parameter values.

A script for sensitivity analysis

```
# Sensitivity analysis of discrete logistic model
# plot population size at t=5 for range of rd values

# set parameter values
ttCollect <- 5
N0 <- 50
KK <- 100
rdVec <- seq(-0.5, 0.5, by=0.1)  # vector of values for key parameter

# initialize vector with null values
nnVec <- rep(NA, 1, length(rdVec))

# loop over values of key parameter, and run model for each value
for (ii in 1:length(rdVec)) {
        logisticOutput <- discreteLogisticFun(N0, rd=rdVec[ii], KK,
                PLOTFLAG=0)
        nnVec[ii] <- logisticOutput[ttCollect]
}

# plot the results
plot(rdVec, nnVec, xlab='Instantaneous growth rate, r_d',
        ylab=paste('Population at time = ',as.character(ttCollect)),
        type='b', col='red', main='Sensitivity to value of r_d')
```

To plot many overlaying curves



Define parameter values

for parameter of interest, make a vector of values you want to consider

Initialize empty plot window to catch all the resulting plots

```
plot(0, type='n',… etc)
```

Use a **for loop** to repeatedly run the model and plot output

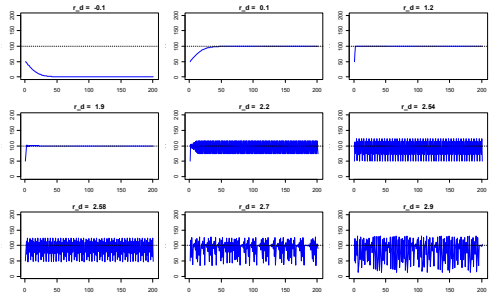
You are looping over the list of values you want to use for the

parameter you're varying.

Each time through loop, run the model with the current parameter

values and plot the results using `lines()`

To make many plots in different sub-windows

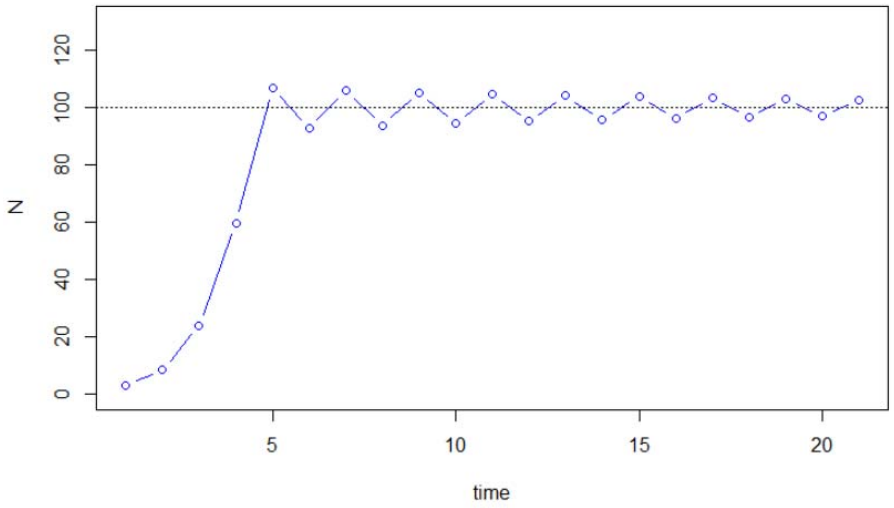


```
# Define parameter values
# for parameter of interest, make a vector of values you want to consider

#  Initialize plot window to have right number of rows and columns
par(mfrow=c(3,3))

# Use a for loop to repeatedly run the model and plot output
    #   You are looping over the list of values you want to use for the
    #      parameter you're varying.
    #   Each time through loop, run the model with the current parameter
    #     values and plot the results using plot()
```

Discrete-time models

$$N(t+1) = R\,N(t)$$

Difference equations, maps



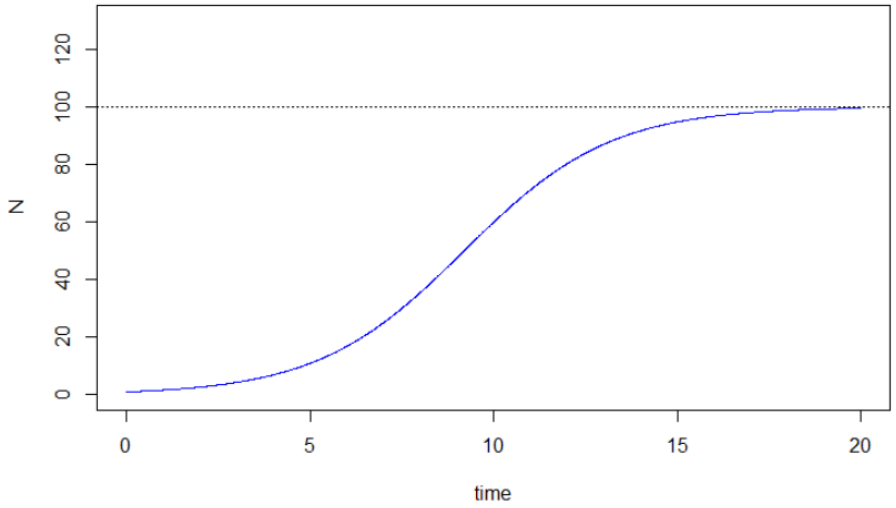
Strengths

- Seasonal systems
- Discretized data
- Easy coding!

Continuous-time models

$$\frac{dN}{dt} = rN\left(1 - \frac{N}{K}\right)$$

ODEs (ordinary differential eqs)



Strengths

- Non-seasonal systems
- Events occur at any time
- Easier math analysis

Simulating an ODE model in R

- install package 'deSolve'
- load the package into memory using `library(deSolve)`

We will use a function called lsoda that is a versatile and robust numerical integrator for ordinary differential equations.

- powerful tool but you need to interact with it in a very specific way – be careful with syntax!

Syntax for `lsoda`

Generic syntax for `lsoda` is:

```
output <- lsoda(init, tseq, ODEfunction, pars)
```

where:

`init` is the initial value of the state variable

`tseq` is a vector of the time points where the model will be evaluated

`ODEfunction` is a place-holder for the name of the function holding the model equations

`pars` is a vector containing any parameters used in the model

`output` is the variable where `lsoda` will return its results.

Syntax for `lsoda`

The function holding the model equations must have syntax:

```
ODEfunction <- function(tt, yy, pars) {
   derivs <- [insert model equations]
   return(list(derivs))
}
```

where:

`tt` is a variable used by R to keep track of the timestep

`yy` is the state variable (or vector of state variables, for multi-variate models)

`pars` is your vector of model parameters

`derivs` is an internal variable that records the time series of results

`lsoda` example: exponential growth

$$\frac{dN}{dt} = rN$$

```
expGrowthODE <- function(tt, NN, pars) {
   derivs <- pars['rr'] * NN
   return(list(derivs))
}
init <- 1
tseq <- seq(0, 20, by=0.01)
pars <- c(rr = 0.1)
```

Note new way of indexing a vector.
Access with `pars['rr']`
Don't need to use this method, but it's useful when you have lots of parameters.

Then call with:

```
expOutput <- lsoda( init, tseq, expGrowthODE, pars)
```

Output from `lsoda`

The output from our command:

```
expOutput <- lsoda( init, tseq, expGrowthODE, pars)
```

will be a matrix made up of two column vectors.

The first column will be the time points where the state of the system is recorded, and the second column will be the corresponding values of the state variable.

```
> expOutput
      time        1
 [1,] 0.00 1.000000
 [2,] 0.01 1.001001
 [3,] 0.02 1.002002
 [4,] 0.03 1.003005
...
```

So we can plot the dynamics with:

```
plot(expOutput[,1], expOutput[,2], col='blue', type='l')
```

Exercise: modify code to model logistic growth

$$\frac{dN}{dt} = rN\left(1 - \frac{N}{K}\right)$$

```
logisticGrowthODE <- function(tt, NN, pars) {
   derivs <- pars['rr'] * NN * (1 - NN/pars['KK'])
   return(list(derivs))
}
init <- 1
tseq <- seq(0, 20, by=0.01)
pars <- c(rr = 0.1, KK = 100)
```

Then call with:

```
logOutput <- lsoda( init, tseq, logisticGrowthODE, pars)
```