FISH 458 Lab 4: Introduction to R

Aims: Using Rstudio, variables, vectors, matrices, scope, loops, functions, commenting, spacing, defensive coding.

Lab notes: those familiar with R can skip over the first few sections and

What is R and why use it?

R is a statistical programming language that can be used to create models, fit models to data, apply statistics, and create beautiful scientific graphics. It operates from a command line. In other words, every command you want executed has to be typed in and then run, one line at a time, in the R console. R has become the most widely used language for statistics, mainly because it is open-source and contains a huge number of packages to run all kinds of analyses. Each package is contributed by individuals belonging to the community. Using R as a standalone programming language can be done in several ways, but most commonly by typing a series of commands (called a "script") into a text file (e.g. in Notepad) and then pasting the commands into the R console.

Getting started with Rstudio

Rstudio (www.Rstudio.org) is an "integrated development environment" (IDE) that provides a much more intuitive and user-friendly way of running R code. Rstudio combines a text editor, R console, graphics outputs, and methods for "inspecting" (looking inside) the values contained in variables within R. Spend some time familiarizing yourself with the Rstudio environment. There are four main windows open. The top left window is for storing R script files (by convention these end in ".r"). The bottom left window is the R console, where you can execute R commands directly by typing them after the ">" and pressing enter to run the command. The top right window (Workspace) will fill up with functions and variables accessible in R, as you run commands. The bottom right window contains either Files (list of files in the working directory), Plots (if you create plots they appear here), Packages (some R commands require loading new packages), and Help.

Typing commands in the console

First, let's experiment with running some commands directly in the console (bottom left). Move your cursor to the ">" (shortcut key ctrl+2). Type the following at the cursor (press <enter> after each line):

```
print("Hello world")
2+3
4*5
4^2
16/2
log(10)
exp(3)
x <- 7*7
print(x)
y <- x-6
print(y)
z <- 10
z <- z-1
print(z)
z <- z-1
print(z)
z <- z+x
```

```
print(z)
```

The `print()` command tells R to send output to the console even if the code is within a function call and is a powerful way of checking that your code is correct (debugging your code). The `<-` command assigns a value to a variable (you can use `=` but it is conventional to use `<-`). The `log()` command is natural logarithm (use `log10()` for the base-10 version). In this code x, y, and z are variables that store values. The shorthand `z <- z-1` means subtract one from the value contained in z, and store the result back in z. Similarly `z <- z+x` means add the values inside z and x together and store the value in z.

Creating a workspace and starting a new R script file

Your life in Rstudio will be immeasurably simplified if you create a directory containing your input data files, output files, and R script files, and tell Rstudio to create a workspace in that directory. Then everything you run will use inputs and outputs in that directory by default, and you can easily switch between different R script files that make up a bigger project.

To do this go to Project-Create Project-Existing Directory and select the directory where you will conduct the work. Once the project is created (look for a new file ending with .Rproj), start a new R script file using File-New-R Script File (or ctrl+shift+N). Save the file something helpful like "Lab 4.r" (the ".r" is conventional for an R script file). Next time you want to open your project, just double-click on the .Rproj file and it will load all the R files you are working on, plus the history of what you did last time.

Running code from an R script file

Now let's type some code into the R script file you just saved:

```
for (i in 1:10) {
    print(i^3)
}
```

Select all the lines containing this code, and press ctrl+enter (quickest method), or click the Run button just above the script window, or go to Code-Run Line(s). If the code won't stop running (infinite loop), **press <escape> to halt code execution**. You should see in the Console window, after the code itself, the following:

```
[1] 1
[1] 8
[1] 27
[1] 64
[1] 125
[1] 216
[1] 343
[1] 512
[1] 729
[1] 1000
```

What this did was loop through all the values from 1 to 10 and print out their cubes. The advantage of using a script file is that you can save, modify, debug, and rerun your code.

If you want to run all the code in a script file, click on the Source button, or go to Code-Source, or go to the Console window (ctrl+2) and type source("Lab 4.r"). The last option can be useful if you store all your functions in one script file, and run your code in another script file: you can "source" your functions and then use them in the second script file.

One more useful feature of Rstudio is builtin help files for the functions. So if you start typing in a (built-in) function and press <tab> it will complete the function and bring up help. If you include the starting parenthesis and then press tab it will tell you what parameters that function expects.

Essentials of R

The R code examples are contained in the file "**4 Code examples.r**" but you will only learn how to code (and how to debug your mistakes) if you retype the commands into your own text file and try running them yourself. This will teach you what are common coding errors and how to find and fix them. It is commonly stated that 90% of your time spent programming is in fixing bugs, not writing computer code. So almost any investment of time, formatting, or checking that will reduce the number of bugs in your code, will pay off by reducing the total time to get the program to work.

Variables

Upper and lower case matter in R, thus `NUM`, `Num`, and `num` are three distinct variables. Variables can hold boolean values (true/false), integers, real numbers, and text strings; there is no need to specify which type of data they are holding before they are used. Variables can also hold vectors, matrices, lists, and a number of other more complex data structures. Usage: `x <- 3` or `x <- 3.45566` or `x <- "Hello"`. The `<-` operator indicates you are assigning a value to a variable. The special character `NA` indicates a missing value.

Vectors

A variable can be a vector, but must be defined first: `x.vec <- vector(length=10)` creates a vector called `x.vec` which will store 10 values (indexed from 1 to 10). To allocate the value 3.4 to element 3 use this: `x.vec[3] <- 3.4`. To set all values to NA use this: `x.vec[] <- NA`. To store the value 10.3 in elements 1 through 4, use this: `x.vec[1:4] <- 10.3`. To change elements 3, 5, and 9 to store 11.6: `x.vec[c(3,5,9)] <- 11.6`. Note that `c(3,5,9)` is a vector containing three values: 3, 5, and 9. Finally, one counterintuitive but very useful shorthand is that if you want to do something to every element except element 1, use a negative sign: `x.vec[-1] <- 6.7`, which will assign the value 6.7 to all elements except the first element.

To inspect the values inside a vector, either use a `print(x.vec)` statement or go to the top right window in Rstudio under Workspace-Values and double click on the name of the variable (not available if inside a function).

Matrices

A variable can store a matrix (two dimensions) or array (two or more dimensions). Here we will just deal with matrices. These need to be defined before use, and require the programmer to specify the number of rows and the number of columns: `x.mat <- matrix(nrow=3, ncol=4)`. To set all cells to a particular value: `x.mat[] <- 2.89`. To set a single value in row 2 and column 4: `x.mat[2,4] <- 0`. To access the entire first row: `x.mat[1,] <- NA`. To access the fourth column: `x.mat[,4] <- -1`. To access subsets of the matrix: `x.mat[1:2,3:4] <- "help"`. Two notes about text values: first, the nicely shaped "" in Word is not recognized in R as the same character as "" and will cause an error. Second, setting any cell to a text value will make R assume that all values in a matrix are text values, which can hinder simple mathematical operations. As with vectors, negative indices indicate cells to be left out of an operation. For example to set all values except the cell in row 1, column 1: `x.mat[-1,-1] <- 0`

If-then-else statements

In R we can make decisions using if-then-else statements (we won't cover the `switch()` function or the alternative `ifelse()` form). The key is remembering to use curly brackets to make it clear which statements belong to the then part and which belong to the else part. For example:

```
x <- 1
if (x < 3) {
  y <- x+2
} else {
  y <- x+4
}
```

This will result in `y` being assigned the value 3. If the first statement was `x <- 7`, then `y` would be assigned the value 11. The decision is made within the `(x < 3)` section, which either evaluates as TRUE or as FALSE. Any number of statements can be included inside the curly brackets. Note that it is considered good **defensive programming** to always include the curly brackets, even though they can be omitted if there is only a single statement to run.

Loops

The most common type of loop used in R is a `for()` loop, whereas `while()` loops and `repeat()` loops are much less common. For this course we will only need `for()` loops. The standard way we will use `for()` loops is to loop through a sequence of values and do some kind of action. By programming convention the looping variables are `i`, `j`, `k` for standard loops, dating back to the idiosyncrasies of the programming language Fortran.

```
for (i in 1:10) {
    print(i^3)
}
```

Here the `1:10` is a vector of 10 numbers going from 1 to 10 (similarly `2:5`, or for a decreasing series `5:2`), and the function will be therefore be evaluated 10 times. You can have any sequence in a `for()` loop, by using the `c()` command that takes values and concatenates them into a vector: `c(3,9,5,7)`, or by using the `seq()` command to create sequences: `seq(from=1,to=11,by=2)`. To take this idea to the extreme, here is a crazy example:

```
for (crazyval in c(100,"help", "August")) {
   print(crazyval)
}
```

Functions

Functions allow you to contain your code in smaller modules, and to reuse your code. They have a name, input parameters, a sequence of code that does something and a command that tells the function what to return (the answer). Not all functions have all of these components. For example let's create a function called `SSvalue` which takes the difference of two values, squares that and returns the result:

```
SSvalue <- function(x,y) {
   SS <- (x-y)^2
   return(SS)
}
SSvalue(x=10,y=3)
```

Note that `SSvalue` is treated as if it were a variable (it is being assigned a function with the `<-` operator). This means that the `SSvalue` function can be passed to other functions as if it were a variable! This will be useful later when we do function minimization in R using `optim()`. Also, just like in Vegas, anything that happens inside a function stays inside a function except for the value returned in the `return()` statement. So you can write code like this: `answer <- SumSquares(x=10,y=3)`, and then the variable answer will store the result of the calculations within `SumSquares`. Some functions just do things, they don't return values (formally these are called procedures). For example:

```
PlotRandom <- function(x,y) {
   plot(x=x,y=y)
}
PlotRandom(x=1:10,y=rnorm(n=10))
```

Here the function just creates a plot (that appears in the bottom right window), plotting on the x-axis the numbers 1 to 10, and on the y-axis 10 values randomly sampled from a normal distribution with mean 0 and SD 1. The function `rnorm()` does the random sampling.

Functions can also be nested within other functions. For example, say we are interested in obtaining the sum of squares from the difference between two vectors of values xvec and yvec. One way of doing this (not the most efficient or bullet-proof) would be to use the function SSvalue to calculate the sums of squares for each pair, and then add the answers up.

```
SSvecs <- function(xvec,yvec) {
   n <- length(xvec)  #assume two vectors are same length
   SS <- 0
   for (i in 1:n) {
      SS <- SS + SSvalue(xvec[i],yvec[i])
   }
   return(SS)
}
SSvecs(x=c(3,4,5),y=c(2,4,6))
```

Some advanced concepts and new functions are used here. The function `length()` tells us how many values are contained in a vector. Once we know how many values there are, we can loop over every value, send those values from `xvec` and `yvec` to the function `SSvalue()`, which returns the sum of squares, and then add the result to the variable `SS`. Always set your variable storing the sum to zero before the `for()` loop.

Scope

The concept of "scope" refers to where you can use the values stored in a variable. Everything that you run in the console becomes available everywhere, including within a function. Such variables are called "global variables". But since any subsequent operation can then delete or modify their value, it is generally considered poorer programming practice to rely on global variables. To delete all global variables (and functions!) run `rm(ls=list())`. You will see in the upper right hand window all variables are cleared and disappear. The opposite of global variables are "local variables", which can only be accessed from within a function. The following code illustrates the difference, where xx is a global variable and yy is a local variable:

```
xx <- 4
dumb <- function() {
   yy <- xx+4
   print(yy)
   xx <- 3
   print(xx)
}
dumb()  #calls the function
```

```
print(xx)  #value was not changed inside the function, instead new variable xx created inside function
print(yy)  #variable not available outside the function
```

Thus global variables cannot have their values changed inside functions. It is good programming practice to explicitly specify the parameters needed in a function, and return the results you need in a `return()` statement in the last line of the function. This makes it easy to see what variables your function depends on. The function above can be rewritten to bring it into compliance as follows:

```
smarter <- function(x) {  #function has one parameter x
   yy <- x+4
   print(yy)   #does not return the value, just prints it to the console.
}
xx <- 4        #global variable
smarter(x=xx)  #passes global variable to function
```

File input and output

Getting data into R is always a bit of a challenge. My standard method is to open a new clean Excel file, copy the data into a solid block starting from the top left in cell A1, with headings in the first row, and then Save As, selecting Save As Type "CSV (Comma delimited) (*.csv)", and save that .csv file to the directory containing the project and R code that I am working on. This file format adds a comma between each value, and a new line for each row. If the csv file is called `"values.csv"`, in R the most basic data input command would be to read it into a variable `xdata` as follows:

```
xdata <- read.csv(file="values.csv")
```

Problems come when some of the data are values and some of the data are text strings or contain NAs. Most commonly I also tell R that there is a header (T is a shortcut for TRUE), and that the separator between values is a comma:

```
xdata <- read.csv(file="values.csv", header=T, sep=",")
```

When some columns contain text strings I usually use this statement, to tell R that the columns with text strings should not be treated as factors but as text (a common use in statistical analysis that results in numbers being assigned to each text string):

```
xdata <- read.csv(file="values.csv", header=T, sep=",", stringsAsFactors=T)
```

One final tip that I use fairly often is to explicitly tell R what type of data are contained in each column using colClasses (note the punctuation), specifying columns as being text (`"character"`) or numbers (`"numeric"`):

```
xdata <- read.csv(file="values.csv", header=T, sep=",", colClasses=c("numeric","numeric","character"))
```

If I run this and then check on the values in the third column of `xdata` using `xdata[,3]` these are converted to text, but `xdata[,2]` is still numeric. To examine the first few rows of the data you just read in, use `head(xdata)`.

To write your data to a file use `write.csv(x=xdata, file="temp.csv")`. To write a file to a subdirectory of your working directory, specify the directory followed by **two** \\: `write.csv(x=xdata, file="tables\\temp.csv")`. Of course the directory must exist before you can save files to it! You could also specify the full directory, provided you replace all the \ with \\ (Note: if you did this you would need to keep updating this every time to move your code to a different directory):

```
write.csv(x=xdata, file="C:\\Users\\Trevor Branch\\Documents\\2013 FISH458 Quantitative\\1 Labs\\Lab 4
Intro to R\\tables")
```

Indentation, tabs and readability

Always line up code at the same level, using tabs or spaces. For example, indent lines within a function by 4 spaces, indent lines within a `for()` loop within a function by 8 spaces, indent lines within an if statement within a for loop within a function by 12 spaces, etc. (You could use 3,6,9 spaces, or tabs.) For more complex programs this makes it easy to read and reduces the number of bugs in your code. For example the following piece of code is perfectly legal but highly obfuscated:

```
for (i in 1:4) {  if(i>3) { print(i) }
for (j in 1:3) { for (k in 1:3) { if (j>k) print(j) }}}
```

It is more readable (admittedly its purpose is still baffling) if written out like this:

```
for (i in 1:4) {
   if(i>3) {
      print(i)
   }
   for (j in 1:3) {
      for (k in 1:3) {
         if (j>k) {
            print(j)
         }
      }
   }
}
```

Now it is more obvious that there are three nested for loops, and that the first if statement happens before the other two for loops.

Commenting

Any text after a # is removed by R before running your code. Thus to add comments, just start them with a #. At a minimum, include a comment at the top of your script to explain what it does, before each function in your program, and wherever your code could be difficult for someone else to understand.

In particular, it is good programming practice to create a block of text at the top of your script which describes what is going to be in the file, includes your name and email, and lists the starting date of the code. When you come back to the code a few days or weeks later (let alone years later) you will be extremely grateful for these comments; they also ensure that your coding work is traceable should your code be handed along to someone else. Finally, when you make revisions you can describe the revisions and the date of revision, which helps when there are multiple revisions of the code. For example:

```
####################################################################
#LAB 4: introduction to R
#Creating building blocks for a fisheries stock assessment model.
#length-weight, von Bertalanffy, logistic selectivity curve,
#Beverton-Holt with steepness
#21 April 2013 Created by Trevor A. Branch tbranch@uw.edu
#22 April 2013 Revised to end global overfishing.
####################################################################
```

Debugging

Your code will have mistakes (bugs) in it. Skilled programmers reduce bugs by employing a variety of tactics including defensive programming, multiple tests, and by debugging. One technique is to program the smallest possible piece of code that you can check, check that it works, then add a few more lines, check that they work, add a few more lines, check that they work, and so on. In this way you can be sure that the error in your code is in the last few added lines. If you sit down and write a few hundred lines of code and then start testing it, it will take so long to track down each bug, and there will be so many bugs, that you may never finish your program. During debugging it is a good idea to include numerous print() statements throughout your code to track down what the values are at each step, and pinpoint the line of code where the program is going awry.

I often implement a simple version of my model in Excel in parallel to the version in R, and use the values in Excel to double check the values in R. Once I am satisfied that they match, I move to R.

One more technique is to think hard about how the program would fail if the input values were negative, zero, or `NA`. In each of these cases some functions that appear to be working perfectly fine, will fail. For example, the function `sum()` can be used to obtain the sum of values in a vector `xvec` as follows: `sum(xvec)`. However, if any of the values in `xvec` are `NA`, then the sum will also return `NA`. Usually this indicates a problem in the code used to generate the values in `xvec`, but sometimes they are true `NA` values and we still want to obtain the sum of the non-NA values. In this case `sum(xvec, na.rm=T)` will work, since this removes the `NA` values before calculating the sum.

Defensive programming:

Defensive programming is the concept of being extra careful to avert the possibility of introducing a bug in your code. Usually this involves a bit more coding overhead the first time you write a program, with the payoff being fewer bug to detect and remove, and a higher confidence that your results are correct at the end. Two techniques I always use are to include braces around sections of code to make it clear what belongs with which for-loop and if-then-else statement. For example:

```
if(x>1) {
   print(x)
} else {
   print(x+2)
}
```

I could (and many do) have written this same code in a more compact way with fewer characters:

```
if(x>1) print(x) else
   print(x+2)
```

However, if I later come in and add a new line of code

```
if(x>1) print(x) else
   print(x+2)
   print(x+5)
```

I might think that the last two lines will be executed whenever the else part of the statement is true, whereas the final line will actually execute every time because it is not part of the else statement. Adding brackets make it clear both to the programmer and to R which statements belong where.

Another defensive programming technique particularly useful in R, is to always name the parameters when calling a function in R. For example, the `seq()` command has a number of different ways of calling it. If I try to generate even numbers between 0 and 20, I could call it in two ways:

```
seq(0,20,2)
seq(from=0, to=20, by=2)
```

The latter way takes more time but it is much clearer to understand and less likely to produce a bug. For example if I accidentally switched two values around, then I get a strange error when running it without parameter names, but when I name the parameters the values still go to the right parameter and I get the expected result:

```
seq(20,0,2)
seq(to=20, from=0, by=2)
```

Exercises

The exercises for this lab including creating functions that will act as basic building blocks for a fisheries stock assessment model. I always create a new workspace in the directory I am working in, which simplifies file management (Project-Create Project-Existing Directory). Then open a new text file (File-New-R script) and save it as, for example, “Lab 4.r”.

Exercise 1: length-weight relationship

The length-weight relationship converts length L in cm to weight W in kg, and has two parameters a (usually very small) and b (usually about 3). Write a function LengthWeight that takes three input parameters: length L, a, and b and returns weight using this equation:

(1) $W = aL^b$

To test the function, add another line of code below the function to call it (to run it with specific values) `LengthWeight(L=30, a=0.0002, b=3.05)`. Then select the block of code including the function definition and the line that calls it and press ctrl+enter. You will notice that in the console (bottom left) the code is run and if it works, an answer is printed out: `[1] 6.401029`. In other words, for a = 0.0002, b = 3.05, and L = 30 cm, the resulting weight is W = 6.40 kg.

Exercise 2: von Bertalanffy growth equation

In the von Bertalanffy growth equation (von Bertalanffy 1938), k is a parameter that governs how rapidly a fish increases in length as it ages, Linfinity is the asymptotic maximum length of the fish, and t0 is the (often negative) age at which a fish is zero length. Write a function VonBert that takes four parameters: age (yr), k, Linfinity (cm), and t0 (yr), and returns length (cm) from the equation below:

(2) $L = L_\infty(1-\exp[-k(t-t_0)])$

For t = 5, k = 0.2, L_∞ = 80, t_0 = -0.2, the resulting length should be 51.72 cm. Check with this code:
`VonBertalanffy(age=5, k=0.2, Linfinity=80, t0=-0.2)`

Exercise 3: logistic selectivity

In many fisheries, selectivity of the fishing gear increases with length, from zero at small lengths to one at high lengths. It is often convenient to use a logistic curve with parameters L50 (length at 50% selectivity) and L95% (length at 95% selectivity) as popularized by Punt and Kennedy (1997). Write a function LogisticSel that takes three parameters: length L (cm), L50 (cm), and L95 (cm), and returns the selectivity of the gear at that length (ranging from 0 to 1) using the following equation:

$$(3)\quad S = \frac{1}{1+\exp\left[-\ln(19)\frac{L-L_{50}}{L_{95}-L_{50}}\right]}$$

Note: in R the natural logarithm is `log()`, for base-10 logarithms use `log10()`.
For L = 55, L_{50} = 40, and L_{95} = 60, the selectivity should be 0.901.

Exercise 4: looping over age, length, weight, selectivity

Having built functions that convert age to length, and length to weight and selectivity, now we can loop through vectors of age and predict the length, weight and selectivity for fish at that age. Create a function AgeValues that takes as input a vector of ages age.vec, and the parameters in the above functions, and uses `print()` and the functions you created in exercises 1-3 to output the following results to the console: length at each age, weight at each age, and selectivity at each age.

The function should be written in such a way that calling it with the following values:

```
AgeValues(age.vec=0:5, a=0.0002, b=3.05, k=0.2, Linfinity=80, t0=-0.2, L50=40, L95=60)
```

Should result in the following output in the console:

```
[1] "Lengths"
[1]  3.136845 17.069771 28.477086 37.816606 45.463158 51.723625
[1] "Weights"
[1]  0.006536327  1.146369107  5.460613369 12.970650946 22.745305722 33.711751572
[1] "Selectivities"
[1] 0.004376737 0.033059340 0.154932651 0.420323994 0.690892854 0.848896853
```

Exercise 5: calculating Beverton-Holt steepness curves

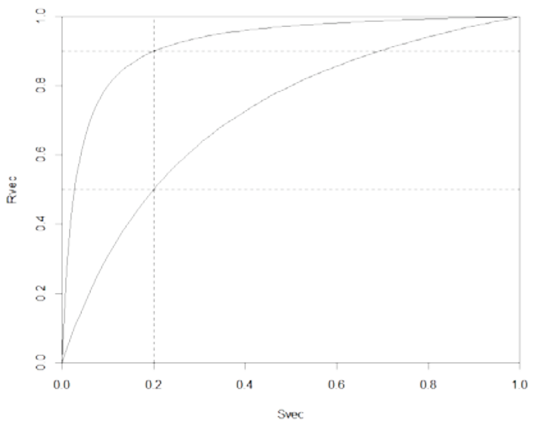
The Beverton-Holt stock-recruitment curve can be parameterized with a steepness parameter h, together with R_0 (unfished recruitment) and S_0 (unfished spawning biomass) using the following set of equations:

$$(4)\quad \begin{aligned} \alpha &= \frac{1-h}{4hR_0}S_0 \\ \beta &= \frac{5h-1}{4hR_0} \\ S_0 &= SPR_0 \cdot R_0 \\ R_{t+1} &= \frac{S_t}{\alpha+\beta S_t} \end{aligned}$$

We will assume that the spawner-per-recruit in the absence of fishing (SPR_0) has been calculated already, and therefore the parameters of the Beverton-Holt equation will be h, R_0 and S_0. Create a function that takes as an input these three parameters, and loops over 100 equally-spaced values of spawning biomass S_t from 0 to S_0, calculating the corresponding recruitment R_{t+1}. Store the values of S_t and R_{t+1} in vectors Svec and Rvec, and create a plot to illustrate the relationship between the two. Vary h to examine how the curve looks at different steepness values. Note: if you set $S_0 = 1$ and $R_0 = 1$ then the plot will be relative to unfished levels. For plotting use this piece of code:

```
plot(x=Svec, y=Rvec, type="l", xaxs="i", yaxs="i")
```

here the “l” (lower case L) tells R to plot lines, and the last two commands tell R to get rid of the space between the axes and zero. With a few more lines of code, specifically abline() and par(new=T), I was able to create this plot to illustrate the difference between `h = 0.5` and `h = 0.9`.



Advanced exercise

If you have completed the exercises for the lab, start planning for Homework 4: estimating MSY, Bmsy, umsy. A full set of equations and instructions for Homework 4 (to be done over two weeks) will be posted by Wednesday night, but I will also post a general overview online during the lab.

References

Punt, A.E., and Kennedy, R.B. 1997. Population modelling of Tasmanian rock lobster, *Jasus edwardsii*, resources. Mar. Freshw. Res. **48**: 967-980.

von Bertalanffy, L. 1938. A quantitative theory of organic growth. Human Biology **10**: 181-213.