ECL/PBG 233: Computational methods in population biology (Marissa Baskett and Sebastian Schreiber)
Programming info sheet
December 29, 2016

Rules of coding

1. *Any time you write a line of code more than once, program it as a function* so you only have one place where you need to look for any corrections or changes.

2. *Break your code down to individual functions that each preform an individual task* so you can debug and test piece by piece ("functional decomposition").

3. *Plan out your code beforehand*, writing a phrase or sentence for each general step you plan to take (these might be your functions), then breaking those into smaller steps, until each is something you can turn into a line of code ("pseudocoding"); this is analogous to outlining a paper before writing full sentences to make sure you have logical flow and all of the pieces fit together.

4. *Comment while you code* so it's easier to remember what the code means when revisiting it.

5. *Define parameters up front rather than using numbers within coded calculations* so it's easy to find and change them.

6. *Write your code as a script* instead of at the prompt so it's easier to edit, save what worked, and run it again another day.

7. Specific to R (and Matlab): *any time you can use vectors or matrices instead of for loops, try it*; it's usually much faster.

8. *Debugging:*

   (a) Test your functions for parameters/cases where you know the answer before running it for a more complicated case so you can make sure they work ("testing").

   (b) When you can't figure out a bug, go line-by-line through the function, checking that you're getting what you expect from each command ("desk-checking"; R functions: `debug, browser`); you can also comment out lines to help you isolate a bug (in R, text after a `#`).

   (c) Especially for code where others might be using your functions, build in warning and error messages for inappropriate values that might accidentally be passed (R functions: `warning, stop`; e.g., cases where zero or negative parameter values will give invalid results).

For more on best practices of coding, see:

- http://swcarpentry.github.io/slideshows/best-practices/index.html#slide-0
- http://journals.plos.org/plosbiology/article?id=10.1371/journal.pbio.1001745

Commands in R

Basics	
help(functionName)	Get quick help on function *functionName*; you can also use the Help menu
+ - * / ^	Simple addition, subtraction, multiplication, division, and power; element-by-element if you're using vectors or matrices
x = 4	Assign the value 4 to x
sin(x), **cos**(x), **tan**(x)	sine, cosine, and tangent
exp(x), **log**(x), **log10**(x)	exponential, natural log, base-10 log
abs(x), **sqrt**(x)	Absolute value, square root
Re(x), **Im**(x)	Real and imaginary parts of x
round(x), **floor**(x), **ceiling**(x)	Rounded value, floor (drop everything after the decimal), or ceiling (opposite of floor, round up anything with a decimal to the next largest integer) of x
rm(x)	clear the value stored in x; **rm**(list=ls()) clears all values

Plotting	
plot(x, y, type="l", xlab="X label", ylab="Y label", col="color", main="Title")	Plot y vs. x with a line (type l, could also be p for points, b for both, etc.) in color *color* (e.g., red, blue, etc.), with x-label *X label*, y-label *Y label*, and title *Title*; all specifications but x and y are optional; if you want to put parameter values into any labeling, use the **paste** command
lines(x, y, ...), **points**(x, y, ...)	Plot lines or points on an existing plot; note that you have to start the plot with **plot** (can be a blank line with type="n") and then use these; use the **legend** command to add a legend if desired
matplot(X, Y, ...)	Plot the columns of matrix X against the columns of matrix Y
barplot(vals, beside=TRUE, names.arg=labs, ...)	Bar plot of *vals* where bars are next to each other (beside=TRUE, instead of stacked) with labels *labs* for the bars
hist(x,breaks=20, ...)	Histogram plot of **x** with data broken in the specified number of equally space intervals (e.g. 20)
contour(x,y,z,nlevels = 10)	Contour plot with nlevels contours of the matrix z where x and y are the locations the grid lines where the z values were computed. To get filled contours, use **filled.contour**. Alternative contour plot commands are available in the **lattice** package.
image(z)	Color map plot based on values of z.
pdf(file="fileName.pdf")	Create file *fileName.pdf* to save plot in. Use this command before creating the plot.
dev.off()	Shut down current plot. For creating pdfs, you need to shut down the current plot before it saves the image as a pdf file.
par(mfrow=c(m,n), cex.axis=q, ...)	This command can be used to control the way things are being formatted in plots, e.g. the mfrow option creates a layoff of m by n subplots that get filled as further plot commands are executed, the cex.axis magnifies the axes by a factor of q, etc.

Vectors	
c(a, b, c)	A vector with values a, b, and c (can be any number of values)
x = **c**(a=1, b=2, c=3)	A vector with values 1, 2, and 3 labeled as a, b, and c
startVal**:**endVal	A vector from *startVal* to *endVal* in increments of 1
rep(val, rep)	A vector of value *val* repeated *rep* times
seq(startVal, endVal, by=inc)	A vector from *startVal* to *endVal* in increments of *inc*
seq(startVal, endVal, length=nVals)	A vector from *startVal* to *endVal* of length *nVals*
v[n]	n^{th} element of vector v
length(v)	Length of vector v
sum(v)	Sum of all entries in vector v (also works for matrices)
cumsum(v)	Cumulative sum of vector v at each entry (e.g., if $v = (a_1, a_2, a_3)$, cumsum(v)= $(a_1, a_1 + a_2, a_1 + a_2 + a_3)$
min(v), **max**(v), **range**(v)	Minimum value, maximum value, or range of values in vector v (also works for matrices)
mean(v), **var**(v), **sd**(v)	Mean, variance, and (sample) standard deviation of vector v
cor(v,w)	Correlation of vectors v and w
which(v==val)	Which entries of v equal value *val* (also works for matrices and can also use the other logical operators listed in the "Loops" table)
which.max(v)	Which entries of v equal the maximum value
rev(v)	Reverse of vector v
as.data.frame(x)	Turn object x into a data frame

Matrices	
matrix(v, Nrows, Ncols)	Create a matrix filled with entires v (a number, which will be put into all entries, or a vector of values) with *Nrows* rows and *Ncols* columns
diag(v, n)	$n \times n$ matrix with v on the diagonal and zeros everywhere else
cbind(v1, v2, ...)	Combine vectors (or matrices) $v1, v2, ...$ along columns
rbind(v1, v2, ...)	Combine vectors (or matrices) $v1, v2, ...$ along rows
nrow(M), **ncol**(M), **dim**(M)	Dimensions of matrix M (number of rows, number of columns, both dimensions in a vector of [nrow, ncol])
M[m,n], M[m,], M[,n]	For matrix M, entry in row m and column n, m^{th} row, or n^{th} column
t(M)	Transpose of M
det(M)	Determinant of M
ev = **eigen**(M)	Eigenvalues (*ev$values*) and eigenvectors (*ev$vectors*) of M
M%∗%N	Matrix multiplication of M and N
M%**x**%N	Kronecker product of M and N (equivalently, **kronecker**(M, N))
M%**o**%N	Outer product of M and N (equivalently, **outer**(M, N))
rowSums(M), **colSums**(M)	Sum across rows or columns (**sum**(M) sums all entries)

Scripts and functions	
# text	Comment (*text* is not read by R)
source("scriptName.R")	Run *scriptName.R*
fnName = **function**(inputs) {...}	Define function *fnName* with inputs *inputs*
return(x)	Return value x at the end of a function
return(**list**(x,y))	Return multiple values at the end of a function
print(input)	Display *input* (a variable for its value or text in quotes) to the screen
out = **optimize**(fn, c(searchMin, searchMax))	Find the minimum (*out$minimum*) of function *fn* over the range from *searchMin* to *searchMax*
out = **optim**(x0, fn)	Find the minimum (*out$par*) of function *fn* given initial guess $x0$
debug(fn)	Debug function *fn*: lets you step through the function so you can examine it for debugging (hit return to go step by step, c to continue, or Q to quit; **browser** and **traceback** are useful debugging functions as well)
system.time(command)	Returns the amount of time required to execute *command*. Useful for estimating completion times for large simulations.

Loops	
for(x in 1:xf) {...}	For each value of x from 1 to xf preform set of commands
while(cond){...}	While the conditions *cond* are true, preform set of commands
if(cond){...}**else**{...}	If the conditions *cond* are true, preform set of commands, and if not, preform another set of commands (following *else*, this part is optional); note that if you have a series of if/else statements, **switch** might work better
>, <, >=, <=, ==	Tests for greater/less than, greater/less than or equal to, and equal to (e.g., $x <= y$ returns TRUE if x is less than or equal to y and FALSE if not)
!, &, |	Not, and, or (e.g., $x < y$ & $x < z$ returns TRUE if x is less than both y and z and FALSE otherwise)
lapply(v, fn)	Apply function fn to each element of vector v, returning a list; **sapply** preforms the same operation but returns a vector (or matrix), and both of these are options for avoiding time-consuming for loops

Numerical integration	
library(deSolve)	Load the library for numerical integration, must come before using **lsoda**
lsoda(n0, seq(t0,tf,dt), odeFun, parms)	Numerically integrate the function *odeFun* given parameters *parms* starting with values $n0$ over time vector *seq(t0,tf,dt)*. Final output is a data array. To get final output to be simply a matrix use **ode** instead of **lsoda**.
odeFun = **function**(t, n, parms) {**with**(**as.list**(parms), {dn = ... **return**(**list**(dn))})}	Appropriate structure for a function for use in lsoda: order of inputs is $t, n, parms$, need to extract any input parameter values out of list *parms* using **with**(**as.list**(*parms*),{...}), and need to return *dn* as a list

Random numbers	
rnorm(num, mean=m, sd=s)	*num* random normal variables from a distribution with mean m and standard deviation s. To compute uniform random numbers use **runif**, Poisson distributed numbers **rpoiss**, exponentially distributed numbers **rexp**, binomially distributed numbers **rbinom**, etc.
dnorm(num, mean=m, sd=s)	Computes the density at *num* for a normal distribution with mean m and standard deviation s. To compute densities for uniform random numbers use **dunif**, Poisson distributed numbers **dpoiss**, exponentially distributed numbers **dexp**, binomially distributed numbers **dbinom**, etc.
pnorm(num, mean=m, sd=s)	Computes the distribution function at *num* for a normal distribution with mean m and standard deviation s. To compute densities for uniform random numbers use **punif**, Poisson distributed numbers **ppoiss**, exponentially distributed numbers **pexp**, binomially distributed numbers **pbinom**, etc.
set.seed(seed)	Sets the "seed" of the random number generator to *seed* (a natural number). Allows one to get replicatable results.
sample(v, num, replace=FALSE)	Sample *num* entries from the vector v without replacement (or replace=TRUE for with replacement)

Generalized Linear Models	
glm(formula, family , data)	*formula* is a linear formula of the form *y x1+x2+...+xk* where *y,x1,...,xk* are the names of data in the data frame entered into the *data* field. *family* corresponds to the statistical family being used for the fits. These families include *binomial* for logistic regressions on 0, 1 data (e.g. survival), *gaussian* for normally distributed errors around the linear trend (i.e. classical linear regression), *poisson* for counting data. Associated with each of these families is a "link" function that is used to transform the data e.g. the link function for the Poisson family is the log function.
summary(model)	Provides the summary statistics associated with the output of most statistical packages in R. *model* is the name of the output from a statistical model e.g. **glm**
predict(model,newdata,type)	provides predicted values from the statistical model *model*. The predictor values (e.g. x1,x2,...,xk) need to be supplied as a data frame in *newdata*. *type* determines the format of the output. The default is on the linear scale of the potentially transformed response variable. "response" provides the values on scale of the response variable (usually what one wants).

Data input/output	
save(x, file="data.Rdata")	Save x (can put in multiple objects, e.g., **save**(x,y,...)) as R data in *data.Rdata*
load("data.Rdata")	Load R data in file *data.Rdata*
x=**scan**(file="data.txt")	Input data in *data.txt* file into a vector or list
A=**read.table**("data.txt")	Input data in *data.txt* file into a data frame; apply **as.matrix** to convert to a matrix
write(t(A), file="data.txt", ncolumn=dim(A)[2])	Output matrix A to *data.txt* file; use **write.table** for data frames

Useful packages	
ggplot2	Improved and (more intuitively) flexible plot formatting
deSolve	ODE integration
FME	Includes sensitivity analyses for continuous-time models
mnormt	Multivariate normals
multicore	Parallel processing
knitr	Generating reports that integrate R code with text