# https://www.r-project.org/Rlogo.pngIntroduction to R

R is a computing environment and programming language for statistical analysis and graphics that is available free. All things R, including the files needed install it on your own computer, are available here:

<https://www.r-project.org/> Note especially the links to FAQs and Manuals.

One benefit of R is the wealth of references and resources that are available online.

# Install R

Open a web browser and navigate to <https://www.r-project.org/>. On the left-hand side of the page, click on “[CRAN](https://mirror.rcg.sfu.ca/mirror/CRAN/)” – short for the Comprehensive R Archive Network – that will take you to a list of archives or “mirrors” where you can download the installation files for your operating system (Windows, Mac, etc.). Any archive should work, but you might choose one near you (scroll to the bottom of the page to see those in the USA).

Once you have clicked on the link for an archive, you are greeted with a page listing "**Pre-compiled Binary Distributions**" from which you can choose the file appropriate for you operating system. These are the files you need to download to your computer to install the base R package.

Once downloaded, install R in the usual way for your operating system.

# Install RStudio



RStudio (<https://rstudio.com/>) is an integrated environment for working in R that has a number of nice features and is free to individuals. Click the ‘[download](https://rstudio.com/products/rstudio/download/)’ button at the top of the page to access the appropriate installation files for your operating system.

# Basics

To start R or RStudio, Windows users should double click the appropriate icon on the desktop or within the Start Menu; Mac users should double click the icon under the Applications menu. When you open R or RStudio, you will see the symbol “>” in the console window; this is the **command prompt**, which indicates R is waiting for a command. To quit R, type “q()” or “quit()” at the command prompt; to quit RStudio type “Ctrl-Q”; R will ask you if you want to save the **workspace** before quitting.

The workspace is where R stores **objects** you create during your current session (vectors, matrices, data frames, lists, functions, all described below). If you choose to save the workspace, you can reload it the next time you start R and it will pick up where you left off.

R works in a straightforward way: you type a command and it executes that command. For example, if you type 2+2 at the command prompt followed by the Enter key, R acts like a calculator:

> 2+2

[1] 4

The [1] in front of the 4 is an R convention that counts the number of items produced by a function.

Typing individual commands at the command prompt works for simple analyses that you plan to run once, but more often, you will want to run a series of commands. So instead of entering these commands into the console directly, you should instead enter them into a text file that you can save, edit, and use later to rerun the same command sequence easily. Files that hold command sequences are called **scripts** and usually have ".R" as the file extension (e.g., R\_script\_example.R).

# Scripts

We can use any plain-text editor to create script files to **store sequences of R commands**, but those built into R and RStudio are especially helpful (the “Source” window). From the menu in R Studio, choose “File, New File, R Script” to create a new file or “File, Open File” to open an existing script file.

When you use the built-in editor in R Studio, you can execute any line in a script by placing the cursor in that line and typing **Ctrl-Enter** simultaneously. Similarly, you can execute multiple lines in a script at once by first selecting them with the mouse or usual keystrokes, then typing **Ctrl-Enter**. Alternatively, you can click on the  icon at the top of the editor window to run the current line of selected lines. In R, the direct execute command is **Ctrl-R**.

# Objects

R is designed around **objects**, so the data you enter or the output produced by R functions can be saved and used in later analyses. For example, entering this line of code into the console (followed by the Enter key)

rnorm(10,50,5)

will execute a function that creates a set of *n* = 10 observations drawn at random from a normal distribution (*r* for random, *norm* for normal) with mean = 50 and standard deviation = 5. Output from functions is sent to the console and **is not saved** unless you **assign** it to an object with the assignment operator “<-” (note “=” works for this too):

x <- rnorm(10,50,5)

Storing output as an object allows you to access the information it contains by referring to the object’s name. For example, typing x at the command prompt will display the contents of the object x:

> x # write contents of object x to the console

[1] 56.02193 52.37202 39.90401 54.59511 52.73739 54.16517 48.72249 49.48370

[9] 51.24416 56.72495

As illustrated above, R ignores whatever follows the pound sign (**#**), which is a useful way to add **comments** to your scripts.

Note that **R is case-sensitive**, so x is NOT the same as X. Typing X instead of x into the console yields:

> X

Error: object 'X' not found

Functions – *common functions listed near the end of this document*

R includes numeric, character, statistical, and programming **functions** that are built-in and generally use the same syntax: type a function name, an opening parenthesis, a comma-separated list of arguments that includes the value or object you want the function to work on, and a closing parenthesis.

For example, if you want the square root of 10, you would use the sqrt() function:

> sqrt(10)

[1] 3.162278

You can get information about any built-in function through the **help function**; e.g., help(sd) to learn about the sd function (standard deviation).

If there is no built-in function to do what you want, you can create your own functions that work just like built-in functions. For example, R does not have a built-in function to find the **mode**, so we can create one that takes a collection of numbers called a **vector**, (defined below), and uses the built-in unique(), tabulate(), and match() functions. This version even works when a dataset has more than one mode.

modes <- function(x) {

ux <- unique(x)

tab <- tabulate(match(x, ux))

ux[tab == max(tab)]

}

Once the code is executed in the console, we can refer to the new function just like a built-in function. For example, we might call the function with object z we've defined previously with: modes(z). Or we might send it a collection of numbers that we organized with the combine function: modes(c(1,1,1,2,2)).

# Data modes

R supports different types of data, called **modes**:

* numeric – real numbers or integers
* character – strings or text values
* logical – TRUE or FALSE; R treats these as 1 and 0, respectively
* complex – imaginary numbers

# Data structures or objects

R supports a variety of objects to store data: **scalars**, **vectors**, **matrices**, **data frames**, and **lists**.

A **scalar** is a variable used to store a single value.

a <- 10

A **vector** is a variable used to store a set of values all of the same mode. Note that “c()” is the combine function that joins multiple values into a vector.

a <- c(1, 2, 5.3, 6, -2, 4) # numeric vector  
b <- c("one","two","three") # character vector (single or double quotes work)  
c <- c(TRUE,TRUE,TRUE,FALSE,TRUE,FALSE) # logical vector

You can **identify individual elements of a vector** by using subscripts in brackets [] to indicate the element’s position in a vector:

a[3] # 3rd element of vector a

a[c(2,4)] # 2nd and 4th elements; note the c() command is required to identify multiple elements

A **matrix** is a two-way table of numbers. By default, R fills a matrix by **filling columns first**; all columns must have the same mode and be the same length.

x <- matrix(c(2, 3, 4, 5, 6, 7), nrow=3, ncol=2)

> x

[,1] [,2]

[1,] 2 5

[2,] 3 6

[3,] 4 7

You can refer to rows, columns, or individual elements in a matrix using row and column subscripts:

x[,4] # 4th column of matrix  
x[3,] # 3rd row of matrix  
x[2:4,1:3] # rows 2,3,4 of columns 1,2,3

**Arrays** are similar to matrices but can have more than two dimensions.

**Data frames** are more general than matrices because different columns can be different modes. Data frames are similar to spreadsheets or datasets used by many statistical programs. Below, note use of the **NA** value (= not available), which is used to indicate that a value is missing (more on this below).

d <- c(1, 2, 3, 4)  
e <- c("red", "white", "red", NA)  
f <- c(TRUE, TRUE, TRUE, FALSE)  
mydata <- data.frame(d, e, f)

The *names* function assigns names to variables:

names(mydata) <- c("ID", "Color", "Passed")

There are multiple ways to **identify subsets of elements in a data frame**.

mydata$Passed # variable Passed in the data frame

mydata[ ,2:3] # columns 2 and 3 of data frame  
mydata[ ,c("ID","Color")] # columns ID and Color from data frame (same result as previous)

**Lists** are ordered collection of objects and allow you to assemble several objects into one object.

Example of a list with four components with different modes: a string, a numeric vector, a matrix, and a scalar:

wlist <- list(name="Fred", numbers=d, matrix=mymatrix, age=5.3)

Identify individual elements of a list using double brackets: [[ ]]

wlist [[2]] # 2nd component of the list  
wlist [["numbers"]] # component named numbers in list

**Factors**

*Nominal variables* can be represented as **factors**.

# create variable ‘gender’ with 20 male entries and 30 female entries

# gender is stored internally as 1s and 2s, with 1=female, 2=male associated (alphabetically)

gender <- c(rep("male", 20), rep("female", 30))  
summary(gender)

gender <- factor(gender) # Instruct R to treat gender as a factor

summary(gender)

***Ordinal variables* can be represented as ordered factors**.

# create variable ‘rating’ with values of "large", "medium", "small"

rating <- c("large", "medium", "small")

str(rating)

rating <- ordered(rating) # Instruct R to treat rating as an *ordered* factor

str(rating)

Note that the ordered() function internally recodes values of rating to 1, 2, 3 based on their order.

# Working Directory

R saves scripts, data files, and output in a default location on your computer, which helps organize projects.

getwd() # Displays the current working directory

setwd("C:/myfolder/data") # Sets the location of the working directory

# Import Data

There are several ways to import data into R from other software packages. A good description is available at the Quick-R website: <https://www.statmethods.net/input/importingdata.html>

We will sometimes import data from comma-delimited text files or enter small datasets manually. To import data stored as a comma-delimited text file was can use the read.table() function:

mydata <- read.table("c:/mydata.csv", header=TRUE, sep="," , row.names="id")

# header=TRUE indicates that the first row contains variable names

# sep="," indicates that comma separates fields

# row.names="id" indicates that the variable id should be assigned to row names  
# Once again, note the / instead of \ on windows systems

# Enter Data Manually

x <- c(1,2,3,4) # create a vector object *x* that contains the numbers 1, 2, 3, 4

matrix(data, nrow=, ncol= ) # converts information in object *data* into a new matrix

# with nrow rows and ncol columns

matrix(c(1,2,3,4), nrow=2, ncol=2) # creates a 2 x 2 matrix with 1 and 2 in the first column

# and 3 and 4 in the second column

matrix(c(1,2,3,4), nrow=2, ncol=2, byrow=T) # creates a 2 x 2 matrix with 1 and 2 in the first row

# and 3 and 4 in the second row (T is short for True)

array(data, dim=c(a,b)) # creates an array from object *data* with a rows and b columns

array(c(1,2,3,4), dim=c(2,2)) # creates a 2 x2 array

seq(from, to, by=a) # creates a single row array based on a sequence from *from* to *to* by *by*

seq(0,1,by=0.001) # creates a sequence of numbers from 0 to 1 in increments of 0.001

rep(data, times) # creates a single row array by repeating numbers in object *data*, *times* times

rep(2,3) # create an array by repeating the number 2, 3 times

# Missing data

Missing values are represented by the symbol **NA** (not available). Impossible values (e.g., dividing by zero) are represented by the symbol **NaN** (not a number).

y <- c(1, 2, 3, NA)

> is.na(y) # returns TRUE if value missing

[1] FALSE FALSE FALSE TRUE

# Checking data and objects

It is good practice to inspect objects you’ve created to ensure they are structured appropriately.

If the object is small, you can print its contents to the console by typing the name of the object into the console or wrapping the original assignment command in parentheses:

y <- c(1, 2, 3, 5)

(yy <- y/10) # creates and prints the contents of the object yy

head(x) # prints the first part of x (first 6 elements of a vector or first 6 rows of a matrix or data frame)

tail(x) # prints the last part of x (last 6 elements of a vector or last 6 rows of a matrix or data frame)

summary(x) # returns summary statistics for object x (e.g., minimum, mean, and maximum for

# numeric variables; number of elements in each level for factor variables)

str(x) # provides information about the structure of x (e.g., object type and dimensions)

dim(x) # returns the dimensions of x

levels(x) # lists the levels of a factor variable x

# Packages

Packages are collections of R functions, data, and compiled code. The directory where packages are stored is the R **library**. R comes with a standard set of packages and hundreds of others are available to download and install. Once installed, packages must be loaded to be used in the current session.

To install a new package:

* Use the R function to install packages from the console: install.packages(‘Sleuth2’)
  + Alternatively, choose **Install Packages** from the **Packages** menu (or from the Packages tab in RStudio)
  + Select a CRAN Mirror location (where packages are stored)
  + Select a package (e.g., mosaic or Sleuth2)
* A package only needs to be installed once, but it must be loaded each time you start R. To load a package, use the library(package) function, for example: library(unmarked)

# Cut and paste graphs and text into Word

After you’ve created a graph in R, right mouse-click on in it then choose “Copy as metafile” from the list, which will store the image on the computer’s virtual clipboard; in RStudio, click on the Export icon in the Plot tab, choose ‘Copy to Clipboard,’ then the ‘Copy Plot’ button.

In Word, place the cursor where you’d like to place the image in the document, then choose “Paste” from the menu. Once the image is in Word, you can resize it or change its properties by right mouse-clicking on the image.

To copy text from the console into Word, you can (1) select the target text with the mouse, copy it to the clipboard (Ctrl-c), then paste it into Word (Ctrl-v) or (2) select Edit, Select All from the R menu, then Edit Copy, then in Word, Edit Paste.

# Online Introductions (optional)

You can work through this one-page introduction starting with the section ‘Introduction to R Studio’. Pay special attention to Challenges 1 through 4; no need to do Challenge 5.

<https://swcarpentry.github.io/r-novice-gapminder/01-rstudio-intro/>

Here’s a more advanced introduction oriented to ecologists; menu is across the top of the page:

<https://datacarpentry.org/R-ecology-lesson/index.html>



# Learning materials

A good tutorial: <https://www.tutorialspoint.com/r/index.htm>

Another, but more advanced, with a focus on plotting and analysis: https://www.cyclismo.org/tutorial/R/

A comprehensive introduction: <https://cran.r-project.org/doc/contrib/Lam-IntroductionToR_LHL.pdf>

A sample R session -- a good refresher if you’ve used R before: <https://cran.r-project.org/doc/manuals/R-intro.html#A-sample-session>

Learn R, in R: <https://swirlstats.com/students.html>

A brief introduction to RStudio: <https://dss.princeton.edu/training/RStudio101.pdf>

# Reference materials

A handy list of useful R commands: <http://www.personality-project.org/R/r.commands.html>

Reference card of common R commands: <https://cran.r-project.org/doc/contrib/Short-refcard.pdf>

Quick-R: <https://www.statmethods.net/>

Two-minute videos on many R functions: <http://www.twotorials.com/>

**Arithmetic Operators**

|  |  |
| --- | --- |
| **Operator** | **Description** |
| + | addition |
| - | subtraction |
| \* | multiplication |
| / | division |
| ^ or \*\* | exponentiation |
| x %% y | modulus (x mod y) 5%%2 is 1 |
| x %/% y | integer division 5%/%2 is 2 |

**Logical Operators**

|  |  |
| --- | --- |
| **Operator** | **Description** |
| < | less than |
| <= | less than or equal to |
| > | greater than |
| >= | greater than or equal to |
| == | exactly equal to |
| != | not equal to |
| !x | NOT x |
| x | y | x OR y |
| x & y | x AND y |
| isTRUE(x) | test if X is TRUE |

**General Functions**

|  |  |  |
| --- | --- | --- |
| **Function** | **Description** | |
| q() | quit R; you will be prompted to save the workspace |
| help() or ?*command* | help function; e.g, help(mean) or ?mean supplies info on the *mean* function |
| getwd() | location where R looks for files = current working directory |
| setwd("c:/docs/mydir") | Set current working directory to mydir; note / instead of the usual \ in windows |
| object | typing the object name prints its contents on the console (e.g., x) |
| length(object) | number of elements or components in an object |
| str(object) | structure of an object |
| class(object) | class or type of an object |
| names(object) | get or set names of objects |
| ls() | list the objects in the current workspace |
| c(object,object,...) | combine objects into a vector |
| cbind(object, object, ...) | combine objects column-wise (= column bind) |
| rbind(object, object, ...) | combine objects row-wise (= row bind) |
| apply(x, margin, function) | apply a function to all rows (margin = 1) or columns (margin = 2) of object x (array or matrix or data frame) |
| history() | displays your last 25 commands |
| newobject <- edit(object) | edit copy and save as newobject |
| fix(object) | edit in place |

**Common Statistical Functions**

|  |  |
| --- | --- |
| **Function** | **Description** |
| help(package=stats) | list all stats functions |
| abs(*x*) | absolute value of object x |
| sqrt(*x*) | square of object x |
| log(*x*) | natural log of object x |
| exp(*x*) | exponential of object x |
| mean(*x*) | mean of object x |
| median(*x*) | median of object x |
| sd(*x*) | standard deviation of object x |
| var(*x*) | variance of object x |
| mad(*x*) | median absolute deviation of object x |
| quantile(*x*, *probs*) | quantiles; is a numeric vector and probs is a probability |
|  | e.g., 50th and 75th percentile of x: y <- quantile(x, c(0.5, 0.75)) |
| range(*x*) | range of object x |
| min(*x*) | minimum of object x |
| max(*x*) | maximum of object x |
| sum(*x*) | sum of object x |
| scale(*x*, center=TRUE, scale=TRUE) | standardize (or center) object x |
| lm(y~x) | fit a linear model of y as a function of x |
| glm(y~x) | fit a generalized linear model of y as a function of x |

**Common Graphical Functions**

|  |  |
| --- | --- |
| **Function** | **Description** |
| help(package=graphics) | List of all graphical functions |
| plot() | Generic function for plotting objects |
| par() | Set or query graphical parameters |
| points(x,y) | Add another set of points to an existing graph |
| arrows() | Draw arrows or error bars |
| abline() | Adds a straight line to an existing graph |
| lines() | Join specified points with line segments |
| segments() | Draw line segments between pairs of points |
| hist(x) | Create a histogram of x |
| barplot(x) | Creates a bar plot of x with vertical or horizontal bars |
| pairs() | Plot matrix of scatter plots |
| matplot() | Plot columns of matrix x |
|  |  |

**Common Probability Functions**

|  |  |
| --- | --- |
| **Function** | **Description** |
| dnorm(*x*) | ***d****ensity* of the **normal distribution**; defaults are mean=0, sd=1  e.g., plot standard normal curve x <- pretty(c(-3,3), 30) y <- dnorm(x)  plot(x, y, type='l', xlab="Normal Deviate", ylab="Density", yaxs="i") |
| pnorm(*q*) | *cumulative* ***p****robability* -- area under the normal curve to the left of *q*  e.g., pnorm(1.96) is 0.975 |
| qnorm(*p*) | ***q****uantile* -- value at the *p* percentile of normal distribution  e.g., qnorm(0.9) is 1.28 for the 90th percentile |
| rnorm(*n*, m=0,sd=1) | generate *n* ***r****andom* normal deviates with mean=*m* and std. deviation=*sd*  e.g., 50 random normal variates with mean=25, sd=5: x <- rnorm(50, 25, 5) |
| dt(*x*, *df*, *prob*) pt(*q*, *df*, *prob*) qt(*p*, *df*, *prob*) rt(*n*, *df*, *prob*) | **Student’s t** distribution; e.g., 1-tailed critical value (left tail) with *prob*=0.05 and 25 *df*: qt(0.05, 25); right tail: qt(1-0.05, 25)  e.g., two tails: qt(1-(0.05/2), 25) |
| dbinom(*x*, *size*, *prob*) pbinom(*q*, *size*, *prob*) qbinom(*p*, *size*, *prob*) rbinom(*n*, *size*, *prob*) | **binomial** distribution; *size* is sample size and *prob* is probability of a success (pi)   e.g., prob of 0 to 5 heads out of 10 coin flips: dbinom(0:5, 10, 0.5)  e.g., prob of 5 or fewer heads out of 10 coin flips: pbinom(5, 10, 0.5) |
| dpois(*x*, *lambda*) ppois(*q*, *lambda*) qpois(*p*, *lambda*) rpois(*n*, *lambda*) | **Poisson** distribution where mean=std=lambda  e.g., probability of 0, 1, or 2 events with lambda=4: dpois(0:2, 4)  e.g., probability of at least 3 events with lambda=4: 1- ppois(2,4) |
| dunif(*x*, min=0, max=1) punif(*q*, min=0, max=1) qunif(*p*, min=0, max=1) runif(*n*, min=0, max=1) | **uniform** distribution   e.g., 10 uniform random variate: x <- runif(10) |

More distributions in R here: <https://stat.ethz.ch/R-manual/R-devel/library/stats/html/Distributions.html>

Gallery of distributions: <http://atomic.phys.uni-sofia.bg/local/nist-e-handbook/e-handbook/eda/section3/eda366.htm>

Common prob. distributions: <https://medium.com/@srowen/common-probability-distributions-347e6b945ce4>

### Sample R session to estimate single-season occupancy

# Install the packages we’ll need; this step needs to be done only once

install.packages("unmarked", repos="http://R-Forge.R-project.org")

# Load unmarked package -- needs to be loaded each session

library(unmarked)

# Assign the working directory so R can find your files – note the backwards slashes

setwd("C:/Users/Bob/Documents/R/Unmarked/")

# Read data file

goag <- read.csv("http://ag.arizona.edu/~steidl/TortoiseOccupancy.csv?attredirects=0")

# Look at the first few observations in data file

head(goag)

# Look at the structure of the R object that holds the data

str(goag)

# Create unmarked frame – the data structure required by the unmarked package

goagUMF <- with(goag, {

unmarkedFrameOccu(

y = cbind(survey.1,survey.2,survey.3,survey.4, survey.5),

siteCovs = data.frame(district, elev, slope, plantCover),

obsCovs = list(obsExp = cbind(obsexp.1,obsexp.2,obsexp.3,obsexp.4,obsexp.5)))

})

head(goagUMF)

# Summary information on elements in the object ElfOwl

summary(goagUMF)

# =========== a few models for detection and occupancy by district ===========

# =========== format is ~ detection ~ occupancy ===========

null <- occu(~1 ~1, data = goagUMF)

# view output of null model

null

district\_null <- occu(~district ~1, data = goagUMF)

null\_district <- occu(~1~district, data = goagUMF)

district\_district <- occu(~district ~district, data = goagUMF)

# Model selection

occMods <- fitList(null=null, district\_null=district\_null, null\_district=null\_district,

district\_district=district\_district)

msDet <- modSel(occMods)

msDet

# =========== Explore results ===========

coef(null) # Logit scale

(ests <- plogis(coef(null))) # Original scale by using inverse logit function

# Another way to backtransform estimates and SEs to original scale

# Note, this function only works when detection or state processes have no covariates

(psi\_SE <- backTransform(null, type="state")) # Occupancy

(p\_SE <- backTransform(null, type="det")) # Det. prob

# Create confidence intervals and tabulate

ci\_Psi <- confint(psi\_SE)

ci\_P <- confint(p\_SE)

resultsTable <- rbind(psi = c(ests[1], ci\_Psi), p = c(ests[2], ci\_P))

colnames(resultsTable) <- c("Estimate", "lowerCI", "upperCI")

resultsTable

# =========== a few more models for detection and occupancy ===========

plantCover\_null <- occu(~plantCover ~1, data = goagUMF)

obsExp\_null <- occu(~obsExp ~1, data = goagUMF)

plantCover\_district <- occu(~plantCover ~district, data = goagUMF)

obsExp\_district <- occu(~obsExp ~district, data = goagUMF)

plantCover.obsExp\_null <- occu(~plantCover + obsExp ~1, data = goagUMF)

plantCover.obsExp\_district <- occu(~plantCover + obsExp ~district, data = goagUMF)

occMods2 <- fitList(null=null, plantCover\_null=plantCover\_null, obsExp\_null=obsExp\_null,

plantCover\_district=plantCover\_district, obsExp\_district=obsExp\_district,

plantCover.obsExp\_null=plantCover.obsExp\_null, plantCover.obsExp\_district=

plantCover.obsExp\_district)

msDet2 <- modSel(occMods2)

msDet2

(ests <- plogis(coef(plantCover\_null)))

#-- Plot relationship between detection probability and plant cover ---#

# Create a variable spanning the observed range of plant coverages

plotData <- data.frame(plantCover = seq(from= min(goag$plantCover),

to= max(goag$plantCover), length=100))

# Compute predicted values of detection probability from the model plantCover\_null

# for different levels of plantCover

coverPred <- predict(plantCover\_null, newdata=plotData, type="det")

# Plot detection probability versus plant cover including upper and lower bounds of 95% CI

plot(plotData$plantCover, coverPred[,"Predicted"], type="l", ylim=0:1,

xlab="Plant Cover", ylab="Detection probability (95% CI)")

lines(plotData$plantCover, coverPred[,"Predicted"]-1.96\* coverPred [,"SE"], lty=2)

lines(plotData$plantCover, coverPred[,"Predicted"]+1.96\* coverPred [,"SE"], lty=2)

Detail: Factors store nominal values as a vector of *k* integers [ 1... *k* ], where *k* is the number of unique values of the nominal variable, and an internal vector of character strings (the original values) that is mapped to these integers.