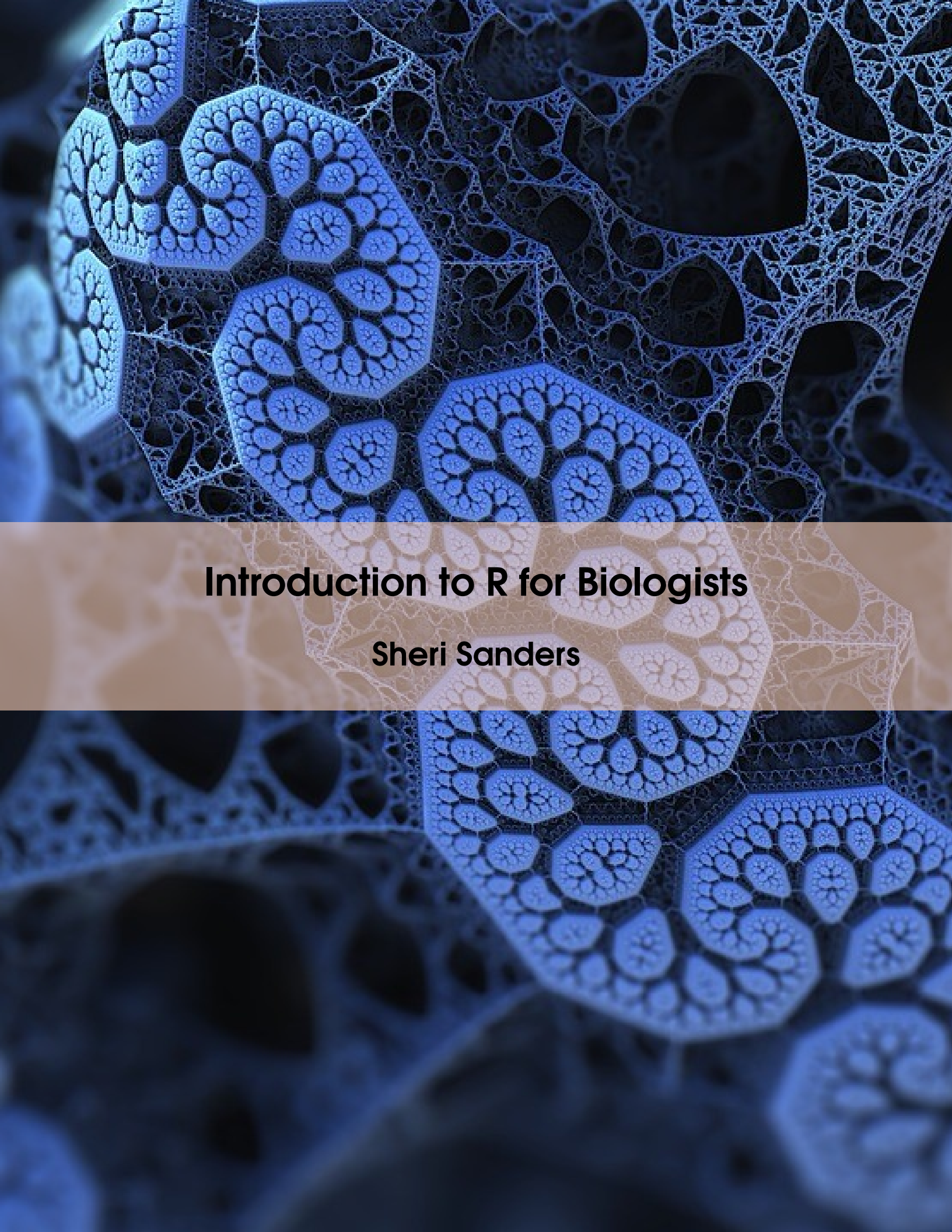




Introduction to R for Biologists

Sheri Sanders

Copyright © 2019 Sheri Sanders

SELF PUBLISHED

NCGAS.ORG/RBOOK.PHP

Licensed under the Creative Commons Attribution-NonCommercial 3.0 Unported License (the “License”). You may not use this file except in compliance with the License. You may obtain a copy of the License at <http://creativecommons.org/licenses/by-nc/3.0>. Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Image sources:

Cover and fill: <https://pixabay.com/en/structure-math-biology-network-1473346/>

Chapter 1: <https://pixabay.com/en/digital-abstract-binary-code-1742687/>

Chapter 2: <https://pixabay.com/en/dna-biology-medicine-gene-163466/>

Chapter 3: Google Watercolor image from IU’s CIB

Chapter 4: [https://en.wikipedia.org/wiki/Genetic_history_of_Europe/media/](https://en.wikipedia.org/wiki/Genetic_history_of_Europe/media/File:PCA_of_the_combined_autosomal_SNP_data_of_Jewish_and_Eurasians.png)

File:PCA_of_the_combined_autosomal_SNP_data_of_Jewish_and_Eurasians.png

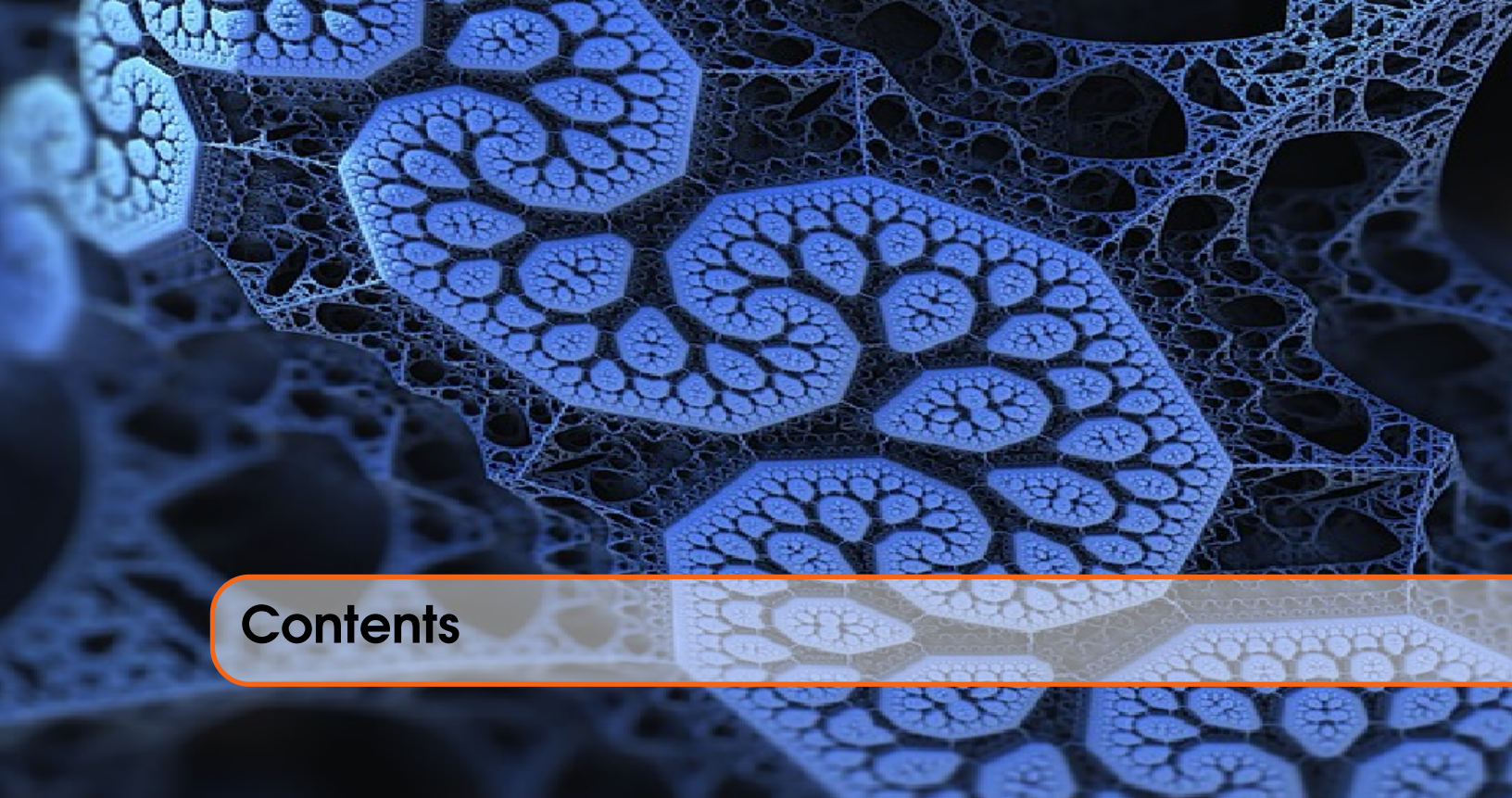
Chapter 5: <https://libreshot.com/binary-code/>

Chapter 6: Data from Lab

Extensive thanks to Bhavya Papudeshi for editing this several times!

Thanks also to Tom Doak, Carrie Ganote, Robert Ping, Julie Wernert, and Winona Snapp-Child for making this course possible.

First printing, February 2019



Contents

1	Using and Manipulating Basic R Data Types	7
1.1	General Pedagogy	7
1.2	Getting Started in R	7
1.3	R is a Language	9
1.4	Working with Scalars	9
1.5	Getting a bit more complicated – Vectors	13
1.6	Vectors of Vectors - Matrices	15
1.6.1	Anatomy of a Matrix	15
1.6.2	Creation	16
1.6.3	Manipulating Matrices	17
1.7	Data Frames	17
1.7.1	Reading a Data Frame from a File	17
1.7.2	Subsetting Data Frames	18
1.8	Quiz! Reading R Syntax	19
1.9	What about REALLY complex data types?	19
1.10	CRAN versus Bioconductor	20
1.11	Getting more help	20
2	R Lab 1: DNA Words	23
2.1	Install your packages	23
2.2	Reading sequence data into R	24

2.3	Length of a DNA sequence	24
2.4	Base composition of a DNA sequence	24
2.5	GC Content of DNA	25
2.6	DNA words	25
2.7	Over-represented and under-represented DNA words	25
3	Graphing and Making Maps with Your Data	29
3.1	Graphing Basics	29
3.2	Mapping	30
3.3	Making a World Map	30
3.4	Mapping Points	31
3.5	Mapping with Objects	32
3.6	Using Real Data	34
3.7	Using Google Satellite Maps	35
3.8	One More Cool Thing	37
4	R Lab 2: Ordination in R	39
4.1	Introduction to PCA	39
4.1.1	Eigenvalues and Eigenvectors	41
4.2	A Simple PCA	42
4.2.1	Compute the Principal Components	43
4.2.2	Plotting PCA	44
4.2.3	Interpreting the results	45
4.2.4	Graphical parameters with ggbiplot	45
4.2.5	Adding a new sample	46
4.2.6	Project a new sample onto the original PCA	47
4.3	A note on functions	48
4.4	Principal Coordinate Analysis	49
4.4.1	Distance calculations	49
4.4.2	Computing the components	50
4.4.3	Graphing the PCoA	50
4.4.4	More on Vegan	52
4.5	Wrap up and back to the biology	52
5	Writing Custom Scripts	55
5.1	Scaling Up – Saving Your Work as a Script	55
5.2	Loading in Other People's Scripts	56
5.3	Best Practices in R	56
5.4	What is a function?	57
5.5	Writing a Function in R	58

5.6	Blast Summary Function	59
5.6.1	Loops	61
5.6.2	Some Graphing Functions	65
5.6.3	Writing to file	66
5.6.4	Final Cleanup	68
6	R Lab 3: Building a Sliding Window Analysis of GC Content	71
6.1	Pseudocode	72
6.2	Filling in the Code	73
6.3	Adding a Plot!	75
6.4	Just for fun – overlapping windows	78
7	Answers to Labs	81
7.1	Lab 1	81
7.2	Lab 2	82
7.3	Lab 3	84



1. Using and Manipulating Basic R Data Types

1.1 General Pedagogy

We're going to start learning the statistics programming language R. There are other classes here that offer more experience with R, so we are going to focus on the programming aspects of the language, especially today. To start, we'll go over basics like accessing the environment, declaring variables, moving around, use commands, load data, how to get help, and install new modules.

This is probably similar to what you may have learned when starting in Unix command line. It is easier to learn a new language when you already know some basic programming concepts (like what a variable is and how it works), so much of what I cover will be in reference to Unix. Much the same way learning French helps you understand English better, learning new computer languages helps solidify unifying concepts. I'll focus on these unifying concepts, because I find that it makes reading and writing R scripts easier, and makes it easier in the future to pick up other languages (like Python). However, don't fret if this is your first language- everyone starts somewhere, and you'll have a head start in learning command line next!

1.2 Getting Started in R

Before we can start learning R, we will actually have to gain access to R. There are several ways:

1. **download it to your personal computer (<http://cran.mtu.edu/>)**

This is easy, it's not a big software package, and I generally prefer to be able to work on stuff off line. You should be able to figure this one out as there are numerous instruction sets online. However, this will be limited by the hardware on your computer - if you need lots of memory, processors, or don't want to leave a program running for DAYS on your computer for larger scale stuff, you should investigate other options below.

2. use Rstudio

A very useful program to look into is RStudio. RStudio is a slightly smaller version of R that makes viewing graphs, etc. much easier. It installs on your machine but requires you to first have R installed (see 1). I tend to use this version for everything I do.

3. use Rstudio online

This is what we will do for this course. RStudio Server has the advantage of having a really nice interface to work with, especially when you are learning the language. The software can be run on a virtual machine (VM), meaning I can set up all the libraries, etc. that you will need. I can provide you all with exactly the same hardware, software versions, etc. regardless of what is on your laptop making everyone's lives easier. I can also sign in as you and view your data, even if you are taking the course from a distance – which is super helpful. You can also use the publicly available image (on Jetstream) to run the same set up whenever you'd like.

4. R command line

Because everything is more fun on the command line . R is pre-installed on most clusters, and usually available as a module.

module load r – Loads R

R - Starts R

q() - quits

A nice feature of RStudio Server is that there is a terminal tab in the bottom left hand section. You can access the terminal and write any command you normally would in Unix without pulling up a terminal or signing in!

We are also going to be using files throughout these lessons, which are all available via the ncbi website - <https://ncbi.org/R%20for%20Biologists%20Workshop.php>. A zip file is available with all files separated by chapter. Each chapter's text is included as well.

Where am I?

While you would use `pwd` (present working directory) to determine our present working directory in Unix, R has a similar concept of "working directories" – where it expects files you are referring to directly to be found.

getwd() – reports what the working directory is currently (i.e.: `pwd` for R)

setwd(dir) – allows you to define a new working directory (i.e.: `cd dir` for R)

```
getwd();
setwd("~/");
```

In RStudio, you can also set your working directory in the GUI. To do this, navigate to where you would like to be within the "Files" tab on the lower right panel. Then, click "More" on the menu bar for that panel. This will bring up a drop down menu with the ability to select "Set as working directory". If you are running into issues with your files loading, this is the number one solution!

Accessing your data

You can load data into R Studio very easily. You can browse files pre-loaded onto the VM or onto your computer using the Files tab at the top of the bottom right pane. You can also load files in from your home computer onto your VM using the upload button on the top of the bottom right pane in Rstudio.

1.3 R is a Language

As I mentioned before, R is a language and it helps to think of it this way. There are concrete (ish) data that are akin to nouns – they are things – numbers, characters, variables. We call these data types in R. There are also actions that go along with these data akin to verbs – what things do – sort, subset, define. We call these functions. How they are used and how they are written are the grammar and syntax of the language. Just like any other language course, let's start with some basic nouns...

Data types in R

R has many data types, and we'll discuss how to add more. However, you can get by pretty easily by starting with the main four datatypes that are built into R:

scalars - "variables", which are usually of common types (string, BOOL, int, float, etc.)

vectors - sets of scalars that are all the same with order

matrix - vector of vector, grid of scalars - all the same data type!

dataframe - "table", like a matrix, but can be of different data types

There are some other uncommon data types built into R that we won't discuss:

arrays - multidimensional matrices

lists - like hashes in other languages or dictionaries in python

Some data we use in biology

These data types may seem a bit obscure until you start thinking about data you are already familiar with. For example:

sequences - vectors

observation data - data frames

counts - matrices

samfiles - data frames

vcfs - data frames

database tables - usually data frames

blast results - data frames

gene names - scalar

metagenome tables - data frames

1.4 Working with Scalars

Scalars are the simplest data type - numbers, strings, and Boolean values. This is what we generally use in Unix. However, note that spacing does not matter in R as it does in Unix (we'll talk about why

shortly):

```
mystr = "hi";  
mybol = FALSE;  
myfloat = 5.2;  
myint = 6;  
mystr = "MDR1";  
name = "Sheri";
```

Thought Questions

Do we use <- or = to assign variables?

Technically these are the same thing, so you can use either. I tend to use = for everything but function creation, simply because = is a common assigner across most languages.

Also, spacing does not matter! The spaces in bash are serving as an indicator of a chunk of information. In R, however, spaces aren't used for this purpose – punctuation is. In bash:

```
myint=6 #works  
myint = 6 #fails
```

but in R:

```
myfloat = 5.2;  
myfloat=5.2;
```

are identical to the system. This is because the name of the variable and the content is one chunk of information in bash – so no spaces. In R, it is looking for the punctuation – in this case "=" to understand what you are telling it.

How does R know what kind of data you have in a variable? Are all things treated the same?

Well, we cannot add and multiply letters, so there is clearly some "data typing". If you look closely at the commands we used, there are some clear indicators that R uses – strings use "", integers only contain [0-9]*, floats contain [0-9]*.[0-9]*, and Boolean values are in all caps. For reference, bash just assumes everything is a string, which is why math is weird in bash.

What happens if we don't give it the quotations when defining a string?

R thinks it's a variable! `myvar = gene_name` is perfectly legitimate code, but `gene_name` must be defined first.

Do I have to use the ; at the end of each command?

No, it is not required. I do it throughout the book however, to make it clear when something is a multi-line command and when a command ends.

Functions

Okay, that is super basic and won't get you far. You need to be able to do things (verbs!) with your variables! So, let's look at what functions look like in R, using printing to screen as our first example.

There are lots of ways to print things in R; one that may be super familiar to you is the `cat()` function – stands for catenate just like in Unix – and it does the same thing! It simply prints to the console what is in that variable ("noun"). Another is simply `"print()"`.

```
cat("Hi", name, "\n");  
print(name);
```



While this seems pretty basic, there's actually a quite bit to talk about here!!

First, you will notice the parentheses. If you see parentheses, it's a function. Anything within those parentheses are inputs into the function. Again, this is why white space doesn't matter in R. In bash, functions look like this:

```
cat $myint  
blast -query $query -db $db
```

A lone string is assumed to be a command or program in bash, and has to be a separate chunk – i.e. "cat" or "blast". Then each information chunk has to be served up separated by spaces so it can parse what is a chunk of information. In R, this is handled by commas and parentheses.

```
print(name);  
print( name );  
print ( name );
```

Usually, you should try to be consistent, since this is a community written language. Generally, keep the leading "(" with the function name at very least!

Another thing you may notice from the print command is that things that are printed to the terminal by default. Beware, R likes to print a lot (which can be annoying). For example, any value that is calculated but not stored somewhere (pushed into a variable) gets printed to terminal. We will get more into this next chapter.

Brief Commentary on Objects

Objects can be a bit of a difficult concept at first, but I will give you a brief introduction.

Imagine a car. While everyone will have a slightly different mental image, there will be common aspects to everyone's idea of a car. It will have certain characteristics – four wheels, doors, windshield, transmission, color, model, year, etc. It will also have certain basic functions – drive forward, go in reverse, park.

Imagine a dog. Same thing – everyone will have slightly different mental images of a dog – but they will all have similar characteristics and functions (run, drool, etc). Some things are shared – like both dogs and cars have colors. But other functions and characteristics are not necessarily shared between dogs and cars... you wouldn't expect to be able to put a dog in drive or a car to drool. You wouldn't

expect a transmission type of a dog or a breed of car. This is because they are different classes of objects.

This is how R and many other languages work – they have defined classes of data – scalars, matrices, vectors – that all have specific characteristics and functions that are built into that class. Every object of that class type has those features, while other classes may not. So R must know what class of object a piece of data is before it can do anything with it.

Thought Questions

How does it know what class a variable is?

We saw this above in scalars – it's part of the declaration. We'll see how to handle other classes of data objects (types of nouns) below.

Why don't we do this in bash?

One of the basic tenements of Unix is that everything is text. Obviously for a statistical language we want a bit more nuanced than that! R cares a lot about the different types of variables, because each has different functions and rules attached to them (because they are "objects").

How do we know what is permissible?

You can find out what the options are and what order you are to enter in the information by typing `?command`:

`?cat`

One reason I love RStudio is it automatically gives you a heads up on what is expected and prints out the information in its own little frame.

Also, just like in bash, there are defaults to the functions. For example, the `cat` function is as follows:

```
cat(... , file = "", sep = " ", fill = FALSE, labels = NULL, append = FALSE)
```

Any time you see something defined, such as `sep = " "` – it's a default! You can skip this part of the call if you want, which is why our first `cat()` call was so much shorter than all of these options!

This may seem like a trivial point – but it is critical! `?function` is going to help you read any code that you run across and also use new functions, look for options, and generally function within R. It is the dictionary to help you in learning the new language. Also, this let's you see the difference between a function and a variable immediately – if you don't see `()`, even empty ones like `q()`, it is not a function; it is a variable! Again, this is immensely useful in reading other people's code!!

WHEW! That was a lot of information coming from a simple print command! However, much of this applies to the language as a whole – think of it as grammar!

! Now that we are using ()'s and will be getting more complex with other brackets, you may run into a common problem - what happens when your prompt gives you a "+" instead of a ">"? If you see the "+", it means R is looking for the rest of the command - meaning you likely forgot a end quote, an end parentheses, etc. If you want to get back to the main prompt (>), hit esc. The + is there if you want to make multiple lines of code, which we will do much later!

1.5 Getting a bit more complicated – Vectors

Remember vectors are ordered lists of data, basically a group of scalars with distinct places in line.

Anatomy of a vector

Let's think of a string of DNA (primer) as an easy example:
ATCGCCCTG

The order of these nucleotides matters, right? So we can assign them numbers, as we can always expect them to be in the same order for this primer:

```
ATCGCCCTG  
123456789
```

Notice I started with 1. This is kind of odd for computational languages, but R is 1-indexed.

Creation of Vectors

So how do we replicate this in R? We define a vector:

```
myprimer = c("A", "C", "T", "G", "C", "C", "C", "T", "G");
```

Thought Questions

What is this c(...) thing? I see it all the time in code...

c() is a function that allows you to input a list. It (also) stands for catenate. This is needed any time you want to input a list of something into R for one option in the function. Because R is chunking information by "," you cannot use the same character to define a list of input for "file" or "labels" or whatnot. By wrapping it in the c function, you are telling R that this is really one chunk of information with several pieces.

DNA isn't the only time you'll use vectors. Sometimes you want to create your own, for example using numbers:

We can also define a new, ordered vector of numbers using the seq function:

```
seq(start, end, increment);  
seq(1, 10, 1); #prints "1 2 3 4 5 6 7 8 9 10"
```

! The # indicates a comment, which is ignored by the computer. These are helpful to use throughout code!

Or read a vector in from a file (read documentation on this one if you use it!), vect.txt:

file:

1 2 3 5 6

6

7

`scan(file="file.txt");`

`vect = scan(file=~/"vect.txt");`

`print(vect);`

You can do this with characters as well – i.e. file primer.txt:

file "primer.txt":

A C T G C C C T G

`vect2 = scan(file=~/"primer.txt"); #ERROR!`

Thought Questions

But you said you could do this!?!

What is the first thing you should do if you see an error? Look at the documentation! ?scan and see if you can figure out what went wrong!!

Okay I got it working...but this doesn't work for sequences that don't have spaces

You can import the data, then split it, then convert it to a vector but this gets complicated (remember - each verb is an additional function). We'll come back to this!!

Manipulating Vectors

Because vectors assume a numbered order – its part of the vector classification. So you can easily grab a specific nucleotide by requesting it's number in line:

`myprimer[1]; #prints A`

`myprimer[2]; #prints C`

Thought Questions

What's with the square brackets?

These are almost always an indication of locations in a vector or a matrix. It sets apart pointing to a portion of a multi-part variable from calling a function!

These seem handy... When might you use this?

It's a GREAT way to grab SNPs if you have the location in the genome...

You can also subset a vector in a similar way - by giving a range:

```
myprimer[1:4]; #print "A" "C" "T" "G"
```

Functions of Vectors

Vectors have defined lengths as well, as they are ordered sets:

```
length(myprimer); #gives you the length of your vector
```

This is handy to check files imported or anything you create – VERY highly recommended! Another REALLY useful function for checking almost all data type is `summary()`. Try **`summary(vect)`** and see what you get.

You can also combine functions together, such as using `length` and `seq` together:

```
seq(1, length(myprimer), 1); #prints "1 2 3 4 5 6 7 8 9"
```

1.6 Vectors of Vectors - Matrices

Matrices are 2-dimensional grids of scalars – a vector of vectors if you will. As a group of scalars or a vector of vectors – it follows that all "cells" in the matrix must be of the same data type!

1.6.1 Anatomy of a Matrix

So if we have a matrix:

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

We can see that this data also has inherent order data – this time in two dimensions:

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

I labeled the columns and rows in the same way R refers to them:

```
matrix[1,1] is 1
matrix[1,] is all of row 1
matrix[,1] is all of col 1
```

1.6.2 Creation

There are a couple of ways to generate matrices – let's look a few. Let's first generate the matrix above:

```
y = matrix(1:20, nrow=5, ncol=4);  
print(y);
```

We can also fill a matrix by rows instead of columns, using an option called `byrow`:

```
a = matrix(c(1,2,3,4), nrow=2, ncol=2, byrow=TRUE);  
print(a);
```

Thought Questions

How would you know about this option?

By looking up `?matrix`

We can also read it in from different files:

```
a = as.matrix(read.table("matrix.dat"));  
print(a);
```

```
a = as.matrix(read.table("matrixheader.data", header = TRUE, row.names = 1));  
print(a);
```

Note, just as we did above with `seq` and `length`, you can use a function inside a function – `as.matrix` makes sure the data is read in as a matrix and not a dataframe (default for `read.table`).

Thought Questions

How do you know the default?

`?read.table`



The standard separator for `read.table` is white space. To do a csv file (comma separate values), you can use `read.csv`, or in your `read.table` function you can set `sep = ","`.

Since you are reading in files, it is ALWAYS a good idea to check the data. You can get:

```
nrow(a)  #number of rows in a  
ncol(a)  #number of cols in a  
summary(a)  #summary of data
```

You can also click on the variable name in the Environment tab in the top right pane. This will bring up the variable information in the top left pane. If the matrix has row names or column names, they will now be listed.

1.6.3 Manipulating Matrices

We can also subset matrices just like vectors. This is extremely common place in use, as we usually deal with one column or one subset of rows for an analyses, be they sample groups or genes of interest, etc.

Thought Questions

Given the anatomy of a matrix, how would you get the following from matrix a:

Print the 3rd row?

4th column?

Element at row 3, col 4?

Grab just rows 1-4 and col 2-5?

`a[3,] -> 3rd row`

`a[,4] -> 4th column`

`a[3,4] -> element at row 3, col 4`

`a[1:3,2:4] -> submatrix of rows 1-4 and cols 2-5`

1.7 Data Frames

Data frames are like matrices, but each column can be a different type. This is likely the data type you will use most, since one column can be gene names (strings), one can be e-values of hits (numbers), another can be bit scores, etc. Columns can also be named, which really comes in handy.

We pretty much do not initiate data frames, we load them in from files.

1.7.1 Reading a Data Frame from a File

The most basic import of a Data Frame is the following:

```
mydata = read.table("BlastResults.txt");
```

but this will lack column names and row names! If there is no header line and you want to name columns at import, you can:

```
mydata = read.table("BlastResults.txt", col.names = c("GeneID", "e-value",...));  
mydata = read.table("BlastResults.txt", col.names = seq(1,12,1));
```

We can also set colnames and rownames after the fact, using:

```
colnames(mydata) = c("GeneID", "e-value"...);  
rownames(mydata) = c("MDR1", "MDR2"...);  
rownames(mydata) = mydata[,1];
```

However, if the first line is a header describing columns and columns have names – as is the case in ours:

```
mydata = read.table("BlastResults.txt", header=TRUE, row.names=1);
```

You can click on the file in the top right frame to see the file that was loaded. You can also check the file with:

```
summary(mydata); #gives you a summary of the data
names(mydata); #outputs column names
row.names(mydata); #outputs row names
head(mydata); #prints the top couple lines
```

1.7.2 Subsetting Data Frames

Subsetting dataframes can be done exactly the same way as a matrix, but it can also be done leveraging the names of the columns and rows!

Extracting columns:

```
bitscore = mydata["bitscore"];
bitscore = mydata[c("bitscore", "evalue")];
```

Extracting rows:

```
goodhits = subset(mydata, evalue < 1e-35); #evalue cannot have ""s
nrow(mydata);
nrow(goodhits);
```

Extracting matches:

```
#Define a set of genes:
gene_names = rownames(mydata)[1:2]

#Grab the hits of interest:
hitsofinterest = subset(mydata, rownames(mydata) %in% gene_names);

#Print the results:
print(hitsofinterest);

#Print the bitscore of the results:
hitsofinterest[c("bitscore", "length")];
```



subset() is very useful – I recommend reading up on it

1.8 Quiz! Reading R Syntax

Thought Questions

What does the following mean?

```
print(nrow(subset(mydata, rownames(mydata) %in% gene_names)));
```

How would do you start reading these chains?

R is nested – functions are contained within ‘()’, so you can break it down by each set (this is actually how the computer interprets them!). Find the inner most set and interpret:

```
subset(mydata, rownames(mydata) %in% gene_names – find the rows in mydata that  
match gene_names  
nrow() - get number of rows  
print() - print
```

So this line prints the number of rows that contain information on the genes listed in gene_names!

1.9 What about REALLY complex data types?

Remember how we talked about importing DNA vectors into R, and how that gets complicated? Most of the time, if you run into something that is going to be a common practice (like importing DNA strings!), someone made a package for it!

Packages are things you can download that add functionality to R. They contain functions written by other people, and packaged up all nicely so you can just load them, and then call the functions within. Packages will also include additional classes, such as "fasta" or "sam" that define classes of objects that can be really handy to deal with intuitively, rather than shoehorning them into the above data types.

In RStudio, installing and loading packages is easy. On the right frame, there is a section of tabs – one is labeled packages. This will list all the installed packages, which you can just click on or off. The code to do it manually will be written in the terminal, for your interest.

If you want to install a package, you can click "Install Package" and follow the prompts. This only works if you want code from the repositories (CRAN, Bioconductor). You can install packages from git, but this is much less common. If you want custom downloaded code, you will have to do it the command line way!

To download a package:

```
install.packages("packagename");
```



Note: You will see a LOT Of red text scrolling through your terminal - this is perfectly normal, even though it may look like an error. This is just the installation occurring!

Load a package:

```
library(packagename);
```

However, if you are doing this on the clusters, the lack of root access will make things fail if we don't direct the program to a location we have access to! In this example, I use my home directory on Indiana University's cluster - but the file path will have to be a different, existing path on your cluster!

```
install.packages("ggplot2", lib="/N/u/ss93/R/library/");  
library(ggplot2, lib.loc="/N/u/ss93/R/library/");
```

Packages only need to be installed once, but they need to be loaded every time you have a new R instance. This is much easier if you are working on your own machine, or in Rstudio.

When you are in the terminal or working with R on a cluster, you can (and should) define a variable called `R_LIB_USER`. This is often defined as `"/R/library"` but you can put your libraries anywhere. This becomes your default location for installing packages when you don't have permission to the main repository (which is VERY common on servers) – it also becomes the default location that R will look for packages if they aren't in the system wide installation. I recommend putting this in your `.bashrc` so that you don't have to define it every time!

This is worth Googling before trying to install anything on a cluster!

1.10 CRAN versus Bioconductor

I mention CRAN and bioconductor – what are they?

Basically, CRAN is an awesome general use repository of about 7000 packages. It holds a bunch of useful stuff with the purpose to basically hold software the open community writes. Bioconductor is very similar, in the fact that it is a giant repo of about 1000 packages, but it is only for genomics content. Because it has the general mission to make genomic analysis easier, it is managed differently than CRAN. I assume this is why they are separate (see: <http://blog.revolutionanalytics.com/2015/08/a-short-introduction-to-bioconductor.html>).

So what's different about them? Bioconductor has more requirements than CRAN packages do, such as 1) high quality documentation with 2) at least one vignette on how to run a genomic analysis demonstrating 3) its contribution to the better analysis of genomic data. It has a heavier focus on teaching computation and analysis to the community it serves. It also heavily encourages the use of its other packages, data structures, etc. to establish common input/output formatting between packages to prevent "reinvention of the wheel". There is a really cool network analysis of the resulting differences between CRAN and bioconductor: <https://www.r-bloggers.com/differences-in-the-network-structure-of-cran-and-bioconductor/>.

Both libraries are called the same, but the installation is a bit different. See bioconductor.org for more information (the installation procedure is currently changing, which is why I'm not writing it out here!)

1.11 Getting more help

With additional commands cropping up in new packages, you may need more help than just the `?command` trick.

Some other things you can do:

function; - NO parentheses, this will print the code for the function

```
cat;
```

methods(function); - will print all the versions of that function

```
methods(print);
```

Another option is to look at vignettes. These are build in examples that come with many packages and are required by bioconductor. To pull up the list of vignettes available in a package:

vignette(package="name of package");

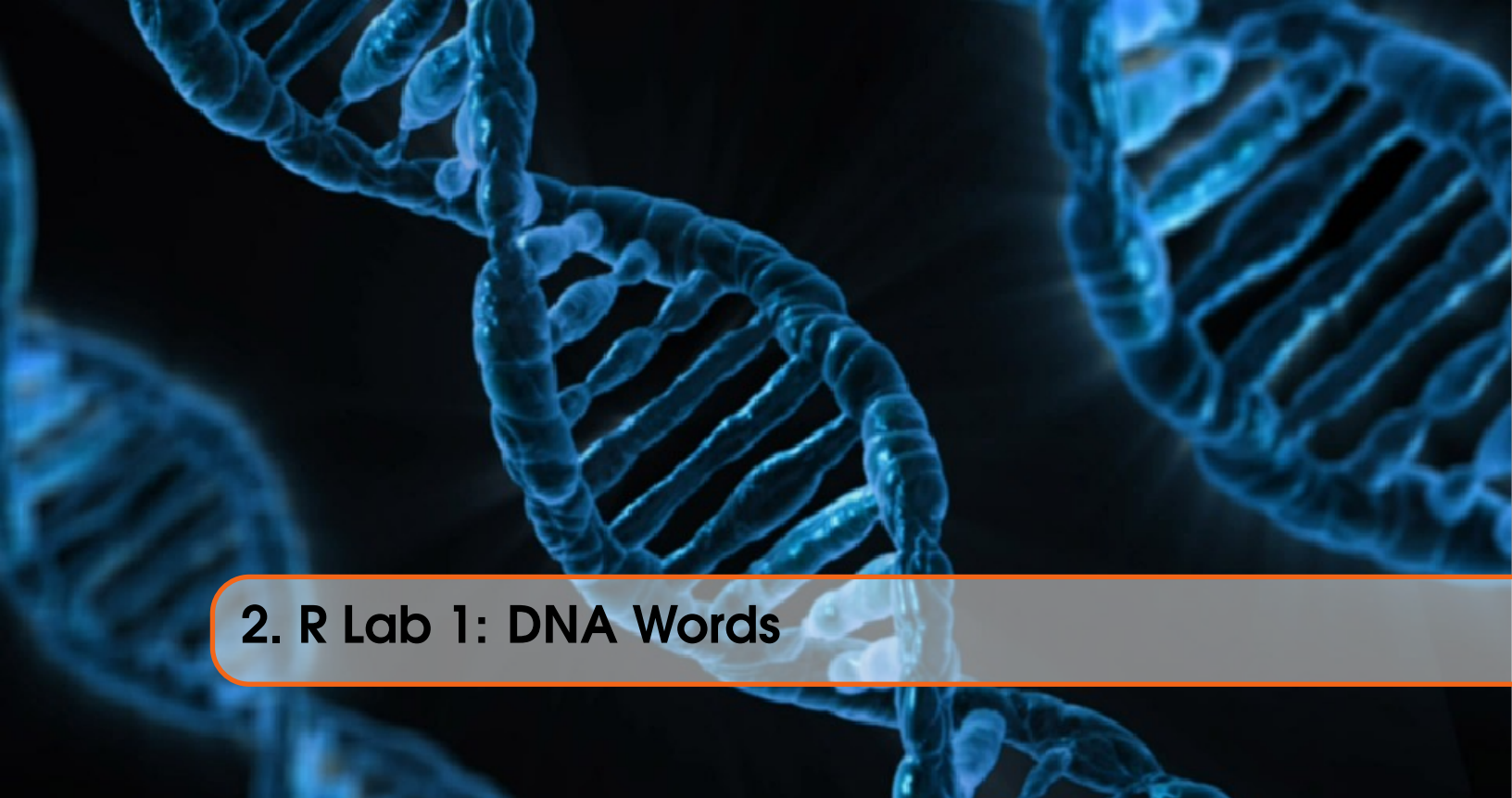
```
vignette(package="annotate"); #This will bring up a list of all the vignettes in "annotate"
```

You can then pull the actual vignette by name:

vignette("name of vignette");

```
vignette("GOusage"); #This will bring up the documentation on how to use this feature!
```

If that still doesn't work – Google!



2. R Lab 1: DNA Words

The purpose of this activity is to get you working in R and getting a feel for some of the syntax. This is definitely a case of "the more you put in, the more you get out". Think about the syntax you are using, and what it's doing. It's all well and good to be able to type the commands and answer some questions, but thinking about commands and options and how vectors work will help you get more familiar with R.

2.1 Install your packages

Some well-known bioinformatics packages for R are the Bioconductor set of R packages, which contains several packages with many R functions for analyzing biological datasets such as microarray data; and the SeqinR package, which contains R functions for obtaining sequences from DNA and protein sequence databases, and for analyzing DNA and protein sequences.

To use function from the SeqinR package, we first need to install the SeqinR ("seq in R") package.

#tell R you are looking in bioconductor repository

```
source("https://bioconductor.org/biocLite.R");
```

#install package

```
install.packages("seqinr");
```

#load package

```
library("seqinr");
```

2.2 Reading sequence data into R

First upload the `dengue.fasta` file to your computer. It is available on the course website. To do this, click the files tab on the right side, then upload.

Using the `SeqinR` package from bioconductor, you can easily read a DNA sequence from a FASTA file into R using the `read.fasta()` function from the `SeqinR` R package. This is similar to the `read.table` command we used before:

```
dengue = read.fasta("dengue.fasta");
```



Note that R expects the files that you read in to be in the working directory, so if you stored your file somewhere else, you will have to either move your working directory or route to the file (i.e. `/home/guest_user/dengue.fasta`).

Look at your new variable – you should always check values you read in. When we read out the new variable, `dengue`, we see that it has the sequence, but also some "attr"s. These are attributes of the sequence (which is an "object"). In this case, it has a name (from the header in the file), an annotation (first line of the sequence), and a class (type of data). If we were to read in two sequences, we could view any single sequence with:

```
dengue$NC_001477.1;
```

The `$` is the means by which we grab individual entries in a fasta file (subsets within the object). You can type the name of the object, type `"$"` and the list of entries will come up. Since `dengue.fasta` only had one fasta entry, you will only see one option.

If we wanted to get the sequence only, we can easily find how to do this by Googling "get sequence only in seqinr". This is a general format for finding functions – Googling (what you want to do) in (r if you don't know the package name, or the package name if you do!). Google will return as likely the first hit – `getSequence`. Look at that function.

```
#get the sequence in a vector
?getSequence
dengueseq = getSequence(dengue$NC_001477.1); #you have to tell it WHICH sequence!
print(dengueseq);
```

2.3 Length of a DNA sequence

Once you have retrieved a DNA sequence (`dengueseq`), we can obtain some simple statistics to describe that sequence, such as the sequence's total length in nucleotides.

Problem 2.1 How would you output the length of the sequence in `dengueseq`? **Store the output as `seqlength`.**

2.4 Base composition of a DNA sequence

An easy first analysis of any DNA sequence is to count the number of occurrences of the four different nucleotides ("a", "c", "g", and "t") in the sequence. This can be done using the `table()` function. For

example, to find the number of As, Cs, Gs, and Ts in the DEN-1 Dengue virus sequence (which you have put into vector variable `dengueseq`, using the commands above), you would type:

```
table(dengueseq);
```

Alternatively, you can find the value of the element of the table in column "g" by typing:

```
table(dengueseq)["g"];
```

Problem 2.2 Why is the `[]` after the function?

2.5 GC Content of DNA

GC content is the percent of a sequence that is G's and C's. This is a feature reported for most genomes, transcriptomes, etc. Since G's and C's have triple bonds (as opposed to the double bonds of A's and T's), this metric has bearing on stability of genomes and has been linked to other characteristics such as thermal stability of sequences (effecting PCR thermoprofiles) and optimal growth rates at high temperatures in prokaryotes. See <https://en.wikipedia.org/wiki/GC-content> for more information.

Problem 2.3 How would you manually calculate GC content given the above information? Can you do it using variables instead of the raw numbers? Store this value as `GCcontent`.

Problem 2.4 Can you find a function in the `seqinr` package that would do this for you? Where do you look?

2.6 DNA words

As well as the frequency of each of the individual nucleotides ("A", "G", "T", "C") in a DNA sequence, it is also interesting to know the frequency of longer DNA "words". The individual nucleotides are DNA words that are 1 nucleotide long, but we may also want to find out the frequency of DNA words that are 2 nucleotides long (ie. "AA", "AG", "AC", "AT", "CA", "CG", "CC", "CT", "GA", "GG", "GC", "GT", "TA", "TG", "TC", and "TT"), 3 nucleotides long (eg. "AAA", "AAT", "ACG", etc.), 4 nucleotides long, etc.

We already have the breakdown of 1 letter words in the genes ("a", "t", "c", "g"). We can also perform this with the `count` function.

Problem 2.5 Use the help options to `count` the two letter words in the sequence. **Save the output as `Count2words`.**

2.7 Over-represented and under-represented DNA words

It is interesting to identify DNA words that are two nucleotides long ("dinucleotides", i.e. "at", "ac", etc.) that are over-represented or under-represented in a DNA sequence. If a particular DNA word is over-represented in a sequence, it means that it occurs many more times in the sequence than you would have expected by chance. Similarly, if a particular DNA word is under-represented in a sequence, it means it occurs far fewer times in the sequence than you would have expected at random. The relative abundance and rarity of DNA words can have implications on the regulation, repair, and evolutionary

mechanism of a genome. Some of these have been linked to functional sites such as the origin of replication and regulation in viruses – so let's see what is going on in dengue.

A statistic called ρ (**Rho**) is used to measure how over- or under-represented a particular DNA word is. For a 2-nucleotide (dinucleotide) DNA word is calculated as:

$$\rho(xy) = f_{xy} / (f_x * f_y)$$

where f_{xy} and f_x are the frequencies (percent occurrence) of the DNA words xy and x in the DNA sequence under study. For example, the value of ρ for the DNA word "ta" can be calculated as:

$$\rho(ta) = f_{ta} / (f_t * f_a)$$

where f_{ta} , f_t and f_a are the frequencies of the DNA words "ta", "t" and "a" in the DNA sequence.

The idea behind the ρ statistic is that, if a DNA sequence had a frequency f_x of a 1-nucleotide DNA word x , and a frequency f_y of a 1-nucleotide DNA word y , then we expect the frequency of the 2-nucleotide DNA word xy to be $f_x * f_y$. That is, the frequencies of the 2-nucleotide DNA words in a sequence are expected to be equal the products of the specific frequencies of the two nucleotides that compose them. If this were true, then ρ would be equal to 1 (the null hypothesis of random). If we find that ρ is much greater than 1 for a particular 2-nucleotide word in a sequence, it indicates that that 2-nucleotide word is much more common in that sequence than expected (ie. it is over-represented). Likewise, if it is below 1, then that 2-nucleotide word is less common than expected by random.

Problem 2.6 Do the following:

- Output the number of occurrences of each 1-nucleotide word.
- Calculate the frequency of g (define as f_G) and c (define as f_C).
- Output the number of occurrences of each 2-nucleotide word.
- Calculate the frequency of gc (define as f_{GC}).
- Calculate rho for "gc" and save as **pGC**. Repeat this for rho for "cg" and save as **pCG**.

Problem 2.7 Can you find a function that would calculate this for you (do it by hand first to practice R syntax)?

Note that if the ratio of the observed to expected frequency of a particular DNA word is very low or very high, then we would suspect that there is a statistical under-representation or over-representation of that DNA word. However, to be sure that this over- or under-representation is statistically significant, we would need to do a statistical test.

In this case, we can use a **zscore** (standard score, a measure of the deviation from mean). See https://en.wikipedia.org/wiki/Standard_score if you are not familiar with zscores. Basically, the zscore is the number of standard deviations away the observation falls from the average. This should agree with the rho value, but will give you a statistical backing of HOW under- or over-represented a "word" is. In this case, let's use the "base" model, which looks at the individual bases, rather than codons.

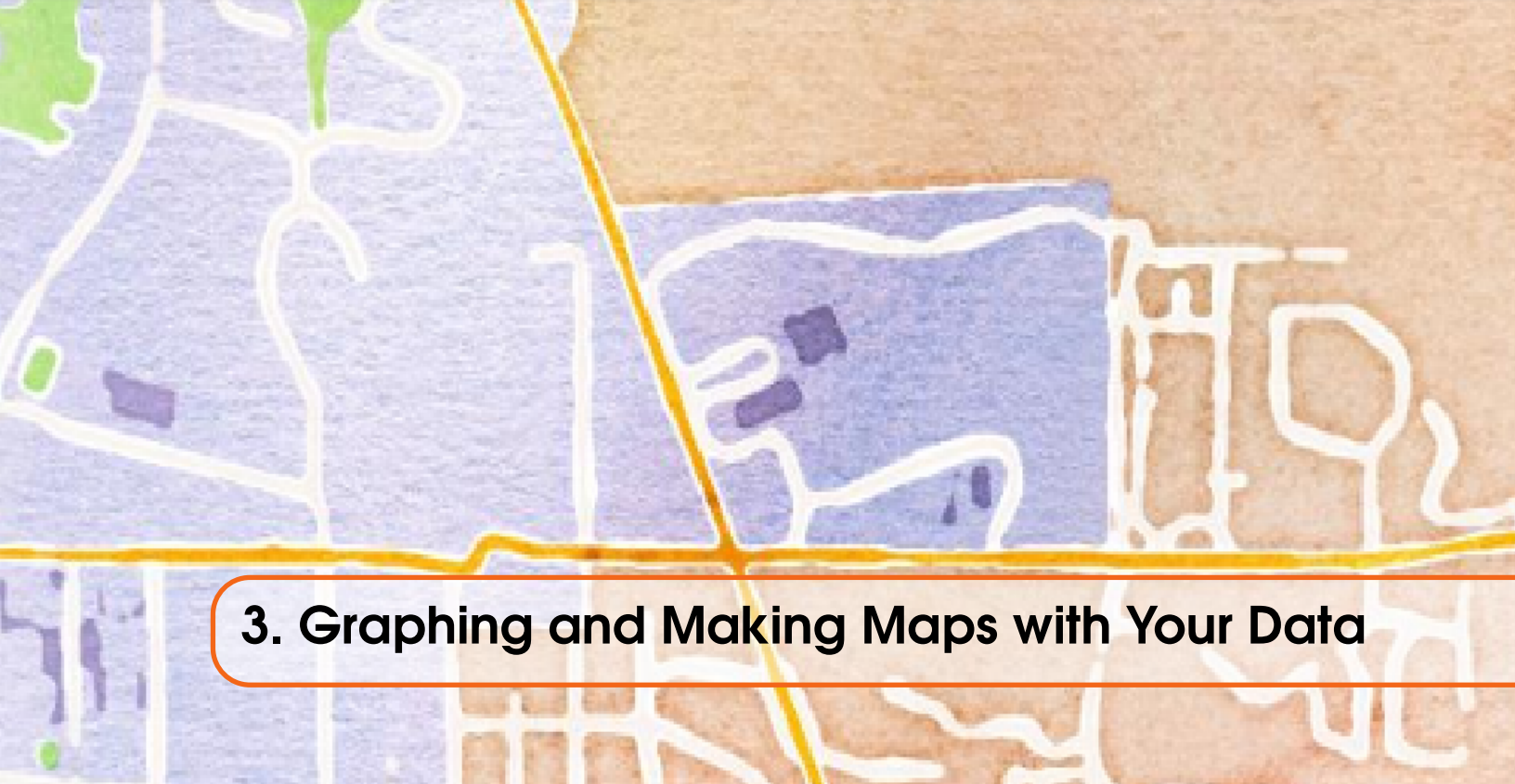
Problem 2.8 Using the command **zscore**, are the "gc" or "cg" words significantly over- or underrepresented?

Thought Questions**When is this useful?**

If we use three letter words, we are calculating the occurrence of codons. Knowing if codons are more or less common than random is a whole field of biological investigation - codon bias!

So why didn't we calculate that?

Because the long hand math of calculating rho is shorter if you use two-letter words. Go ahead and try looking at the codon values now that you know the easy-to-use functions! Make sure to use the right model in `zscore()`!



3. Graphing and Making Maps with Your Data

3.1 Graphing Basics

Graphing is a pretty convenient use for R, especially in Rstudio. `plot()` is the most generalized graphing function. If you give it all numeric data (numeric vs. numeric) it'll do a scatter plot. You can tell R to plot points and/or lines with `type = ""`.

```
plot(seq(0,10,1), seq(0,10,1));  
plot(seq(0,10,1), seq(0,10,1), type = "l");  
plot(seq(0,10,1), seq(0,10,1), type = "p");  
plot(seq(0,10,1), seq(0,10,1), type = "b");
```

A few other useful parameters are:

```
main="The Title"  
xlim=c(1,20) #define length of x axis  
ylim=c(1,20) #define length of y axis  
col="darkblue"
```

```
plot(seq(0,10,1), seq(0,10,1), main = "the Title", xlim=c(1,20), ylim=c(1,20), col="darkblue");
```

One additional useful function for graphing is `abline()`. This function allows you to give R a function to plot, so you can add a regression line to a plot.

```
abline(a=0, b=1, col="blue");
```

We can also use explicit plotting functions like `hist()`. To make a histogram of a column, you can

use the `hist()` function and point it to the variable, and the portion of that variable (by using `$`).

```
table=read.table(file="BlastResults.txt", header=TRUE, row.names=1);  
hist(table$bitscore);  
hist(table$bitscore, breaks=seq(0,900,1));  
hist(table$bitscore, breaks=seq(0,900,10));
```

3.2 Mapping

There is a lot you can do in making figures in R. We are going to explore the matter of location data further, mainly because it's intuitive and demonstrates the key concepts. There are a couple of useful libraries to load ahead of time, so let's start with that:

```
library(maptools);  
library(maps);  
library(mapdata);  
library(lattice);  
library(raster);
```

These are a great starting point for exploring mapping and include the Google maps library (`ggmaps`), which we will use to make plots of GIS data. It also automatically loads `ggplot2`, which is also a very very common plotting library.

Since it will come up a lot in this section, let's get a reminder of some REALLY useful syntax:

c() - define a vector

This is a means of telling R that you want to give it a vector of items to be used for one variable i.e. `xlim=c(0,10)`. `xlim` is one variable, but you want to pass it two integers here, so we make both integers into one vector - one variable holds the one vector. We couldn't do `xlim = 0, 10` as one variable cannot hold two discrete items. Also, that `", "` means second parameter in R!

Okay, now that that's fresh in our brains, let's do some graphing!

3.3 Making a World Map

First, let's grab some data. One of the great thing about maps is that there are a LOT of pre-existing datasets to use. Here's a trick especially helpful in loading data for mapping – the `data()` function! Type this function in slowly – when you get to the open parentheses, RStudio will give you a drop down list of a lot of data you can pull. Not all of it is mapping data, but this can be useful to you in the future.

```
data(wrld_simpl);
```

I said this data was pre-loaded, but where is it really coming from? We can find out using the documentation:

```
?data;
```

Turns out this function is a pre-installed "utils" library in R that loads specified datasets, including data that may come from other packages. Useful to know!

Let's get back to making our first plot. We can try our first plotting function and see what happens:

```
plot(wrld_simpl);
```

Oo.. pretty! Let's "zoom in" - almost everything in mapping is based on latitude and longitude. Unfortunately, we usually think of these as lat, long - but graphing programs want them as x and y. So, so zoom in to 130W to 60W by 25N to 53N, we have to do the following:

```
xlim=c(-130,-60); #longitude from -130 to -60 or 130-60W
ylim=c(25,53); #latitude from 25-53, or 25-50 N
plot(wrld_simpl,xlim=xlim,ylim=ylim);
```

Thought Questions

Why did we do this in three steps? You could just do

```
plot(wrld_simpl,xlim=c(-130,-60),ylim=c(25,53));
```

But since we don't always know the exact lat and long of where we want to "zoom" to, setting variables that we can quickly change before rerunning the same plot function saves on a lot of scrolling around. Also, it is a bit easier to read!

Since we know that plot works here, and xlim and ylim work, let's try the other common parameters – color and title. I'll also throw in a useful one that isn't listed in the help – bg. See if you can figure out what it does!

```
plot(wrld_simpl,xlim=xlim,ylim=ylim,col='olivedrab3', main='United States and Neighbors',
     bg='lightblue');
```

3.4 Mapping Points

Great, we have colorful maps that we can customize from pre-existing data. But you are likely going to want to map some points onto the maps. Conveniently, there is a points() function. If you look in the documentation (as you always should!), you will see it requires something to be input as an x value, but y is optional. It also gives you links to more parameters (very useful) and types of plotting (remember p for points, l for lines, b for both?). The documentation for points also lists a couple that might be useful – col and bg we've used, but what about pch? The description isn't very clear, we'll just have to experiment!

```
points(x=c(-83.53,-86.23,-86.5),y=c(41.65,41.7,39.17), pch=5, col='red');
```

```
#clear the map of points by redrawing the base map
```

```
plot(wrld_simpl,xlim=xlim,ylim=ylim,col='olivedrab3', main='United States and Neighbors',
     bg='lightblue');
points(x=c(-83.53,-86.23,-86.5),y=c(41.65,41.7,39.17), pch=24, col='blue');
```

So pch changes the shapes... we can Google "pch options in r" and get this link (<http://www.endmemo.com/program/R/pchsymbols.php>) to see some options. I like 19, but you can use whatever you'd like!

We can also draw lines with the points function by setting type = "l". Let's plot the borders of the province of Colorado because they're easy to draw. You will have to give the function FIVE sets of coordinates... why?

```
points(x=c(-109,-102,-102,-109,-109), y=c(37,37,41,41,37), type="l", col="purple");
```

Just like the pch options, you can also get a list of the color options by Googling "col options in r" to get this link (<http://www.stat.columbia.edu/tzheng/files/Rcolor.pdf>) – R has a LOT of pre-named colors for you to use!

3.5 Mapping with Objects

While data() is super useful for a lot of data types, getData() is another function that is similar but specifically useful for geographical data. It is part of the raster library - we know because this is listed in the top left of the help tab when you use ?getData. This function will grab data from the web as needed. There are datasets available for countries, elevation, climate, and a few other things. This is one to read up on and be familiar with if these are useful datasets for you!

Let's pull the USA and plot it again, using this function:

```
USA = getData('GADM', country="USA", level=1);
plot(USA);
```



GADM stands for global administrative boundaries here - you can also use 'climate', etc for different data types. Feel free to check it out using ?getData!

Hmm, that looks like it would fit in our world map... they're both listed as similar data types in our Environment tab, so let's try it!

```
plot(wrld_simpl);
```

A quick ?plot search gives us the option to look at the "plot" function in two packages – one is in raster, which is what we used to load the USA data. Let's try that first – which is great, because it tells us there is an add=TRUE option! Let's try that:

```
plot(USA, add=TRUE);
```

Very cool. We can use country specific data to fill in more information on the world map if we need it. We can do the same to add provenances to Canada:

```
CAN = getData('GADM', country="Canada", level=1);
plot(CAN, add=TRUE);
```

But what are we actually plotting in these cases? Let's look at the data:

USA;

We can see this variable is of "class: SpatialPolygonsDataFrame"... what does that mean? Remember that classes are types of things, with defined specific functions and characteristics. This makes sense for spatial polygon data (map shapes), as they would share specific characteristics (boundaries) that could be kind of complex to hand code each time. They also have functions that would be useful across all spatial data, like plot handling.

A bit down in the output for USA, we can see that the names of the states are listed in NAME_1, so let's use that to extract states. Since these names are a part of an object, we can refer to each state in this specific object (of the SpatialPolygonsDataFrame class) by using USA\$NAME_1:

For example, Indiana:

```
IN = USA[USA$NAME_1=="Indiana",];  
plot(IN,col="white");
```

Thought Questions

Why is there a "," after "Indiana"?

Because USA is a data frame – and we are subsetting that data frame, so we need to give it row, col. We want all of that column, so we can leave the col blank to grab all of them – just like we did in a normal data frame. SpatialPolygonsDataFrame is a super specific version of a dataframe, so you can work with it the same way!

Let's do the same for a couple other states that surround Indiana: Plot IL, OH, MI, and KY – remember to "add" them

```
IL = USA[USA$NAME_1=="Illinois",];  
plot(IL,col="grey", add=TRUE);
```

```
OH = USA[USA$NAME_1=="Ohio",];  
plot(OH,col="grey", add=TRUE);
```

```
MI = USA[USA$NAME_1=="Michigan",];  
plot(MI,col="grey", add=TRUE);
```

```
KY = USA[USA$NAME_1=="Kentucky",];  
plot(KY,col="grey", add=TRUE);
```

We can go one further - we can add cities, with information about the cities. Let's plot points in Toledo, OH; Notre Dame, IN; and Bloomington, IN.

```
points(x=c(-83.53,-86.23,-86.5),y=c(41.65,41.7,39.17), pch=19, col='red');
```

Most of the time, we want to add some sort of information into the points – make their size indicate

the population, for example. In order to do this, we need to build our own data frame to add that information. Let's do this with variables to make it more readable. All of this we've more or less seen before:

```
#define our matrix of coordinates
lon=c(-83.53,-86.23,-86.5);
lat=c(41.65,41.7,39.17);
location=c("Toledo","Notre Dame","Bloomington");
pop=c(278508,5973,84465);

#create our dataframe
df = data.frame(location, lat, lon, pop);
```

However, if you plot raw population, you can see there is a LARGE difference (The actual town of Notre Dame is very tiny). The dot sizes would end up covering a large portion of the map – so let's scale them. You can do this by dividing the square root of the population column (`df$pop`) by the maximum population value (in this case Toledo):

```
scaling = sqrt(df$pop)/max(df$pop);
```

Thought Questions

What is this scaling doing?

By taking the square root of the populations, we are reducing the actual distance between the measures without changing their relationship each other. For instance, 2,4,and 16 has a range of 14, but the square root of all of the values (1.4,2,4) only has a range of 3.6.

By dividing by the maximum number, we are representing all of the values as a percentage of our largest, thereby scaling the data to our own values. Technically, you can use `sqrt(max(df)$pop)` to scale to the max square root transformed populations size. However, as you will see next, we have to adjust these values anyway, so this additional `sqrt()` calculation doesn't really help anything.

Now we can plot our points. Note that `cex` is the sizing parameter in points!:

```
points(x=df$lon, y=df$lat, pch=25, col='red', cex=scaling);
```

Well, the scaling made the dots very very small. Let's increase them all by a factor of 5000.

```
points(x=df$lon, y=df$lat, pch=25, col='red', cex=scaling*5000);
```

Much better!

3.6 Using Real Data

We can also pull data from pre-made data frames from research. The data we are going to use is blood titer values for Canine Distemper in Wolves and Bears at Yellowstone National Park. This data was

pulled from the USGS, and then I added randomized GIS coordinates to the samples to demonstrate mapping. As a reminder, this data is available on the ncgas website:
<https://ncgas.org/R%20for%20Biologists%20Workshop.php>.

```
distemper = read.table("Canine distemper virus load with fake GIS.txt", header=TRUE);
```

Let's map Wyoming, Montana, and Idaho to start. Since we aren't centering on any one state, let's use the USA map, and crop the x and y lims:

```
plot(USA, ylim=c(40,48), xlim=c(-106,-114));
```

Thought Questions

How do you know what limits to use?

You can use Google maps, right click on where you want your line, and select "what's here". It will give you the GIS coordinates!

Now we can plot the points with this dataframe. Here we use `df$long`, etc. but you could just as easily use `df[,1]` to pull the first column. Using the `$` is consistent with what we did above with plotting states, but also removes the requirement of remembering the order. However, use what makes sense to you!

```
points(x=distemper$Long, y=distemper$Lat, col="blue", pch=19);
```

And we can also scale the dots via titer volume:

```
scaling = sqrt(distemper$Titer)/max(distemper$Titer);  
points(x=distemper$Long, y=distemper$Lat, col="blue", pch=19, cex=scaling*50);
```

3.7 Using Google Satellite Maps

There is a great package for using google to map your maps that uses the `ggplot2` grammar. It is a super popular plotting package, however there has been a bit of reorganization lately.

First off, you may also come across the `RGoogleMaps` package, but I do not recommend using it because it seems to have a grammar unique to that package (i.e. not compatible with base plotting or `ggplot2`) and has strange scaling behaviour. This makes it more difficult to learn and retain in memory.

Secondly, as of April 2018, Google moved all the map functions into their cloud service. You are required to add a payment method to use it, but you get \$300 free each year to use, which ends up being 1000s of maps, making it essentially free. You just have to go through the semi-annoying set up.

To set this up, do the following:

1. visit <https://console.cloud.google.com> and sign up for a google cloud account.
2. Then you have to create a project by clicking the "select a project" menu next to the "Google Cloud Platform" name at the top.
3. Create a new project, then click on the "APIs Services" menus on the left side.
4. A side menubar will appear, allowing you to select "Library".

5. Search for the Map Static API and enable.
6. The last thing you need to do is get an API key, which you will need to add into your R command. If you click "API Services" again, you will see a "Credentials" option. Click that.
7. You can create credentials here, or grab them again later if you forget (very helpful to know!).
8. Once you have a key listed, copy it with the handy copy button, and go back to R.

Finally, it is beneficial to install the latest version of this package. It's a bit annoying to keep doing, but since this is attached to the cloud service, it is worth guaranteeing it will work. Also note that this is a great example of how to load packages from github - which is becoming more common (though is more likely to be problematic than loading through bioconductor or CRAN).

```
if(!requireNamespace("devtools")) install.packages("devtools");
devtools::install_github("dkahle/ggmap", ref = "tidyup", force=TRUE);
```

```
#Load the library
library("ggmap")
```

Then you have to give it that key you copied from Google Cloud:

```
register_google(key="PUT YOUR KEY HERE");
```

You only have register once, but you should set your key each time to guarantee it will work. This now brings us to actually using the package and yet another means of grabbing existing maps is found in this package – using `get_map`. Let's grab a satellite image of Yellowstone:

```
?get_map;
Google = get_map(location = c(-110,44), zoom = 6, maptype = "satellite");
```



NOTE: sometimes this has errors due to being pulled from Google's servers – just wait a second and retry the command.

Now plot it:

```
ggmap(Google);
```

With this package, you have to pick a center point, and then work with zoom to get the map you want. The package is nice enough to print out the lat and long on the axes, so it's pretty quick to figure out if you are within the range you want.

We cannot add points to it the same way as before... We have to use `ggmaps'` syntax, but much of it should look familiar:

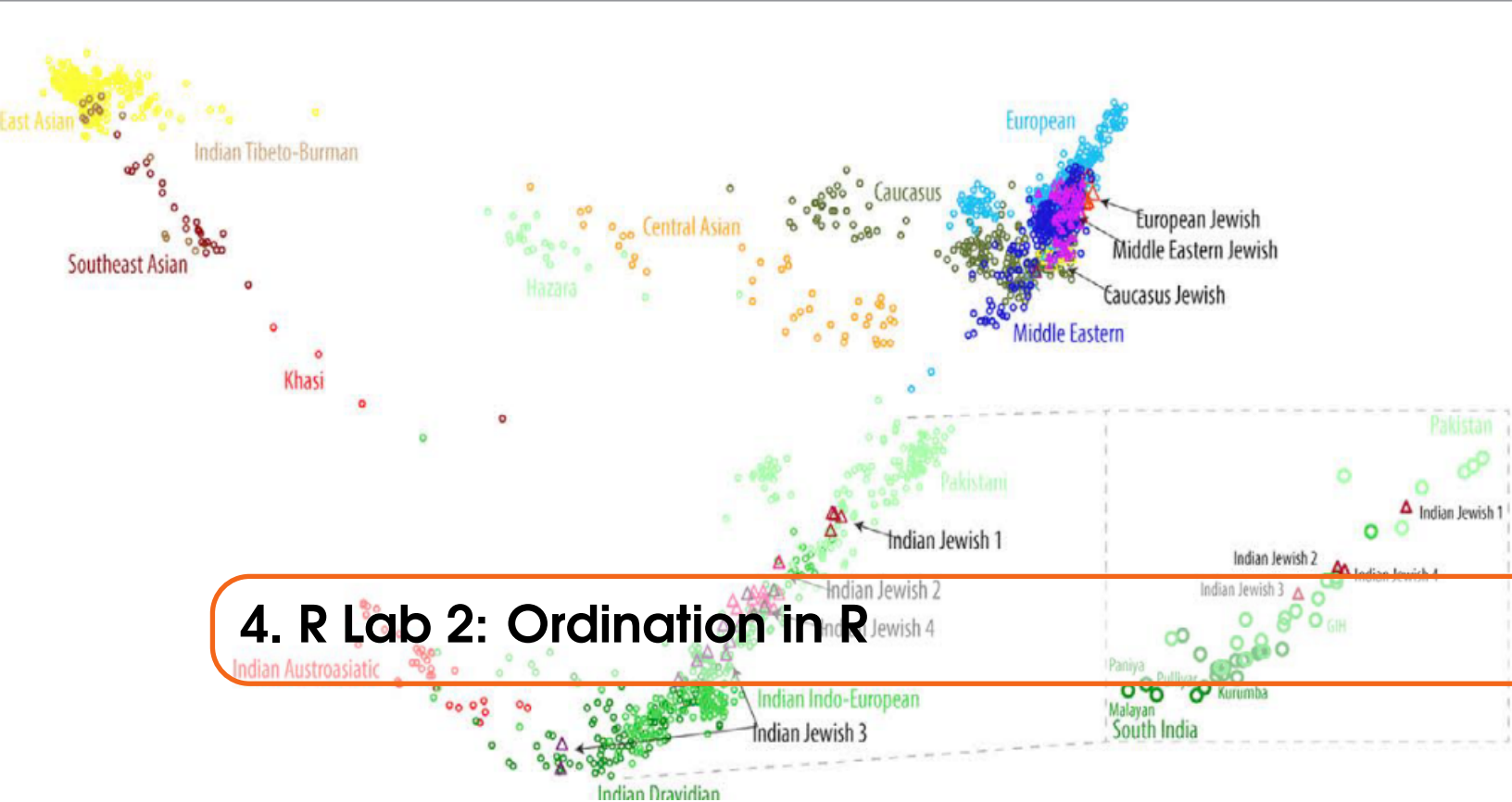
```
ggmap(Google) + geom_point(distemper, mapping = aes(x=distemper$Long, y=distemper$Lat),
  color="yellow", size = scaling*50);
```

3.8 One More Cool Thing

Just for fun, Google has a very pretty version of their maps, which you should try:

```
Google <- get_map(location = c(-110,44), zoom = 6, maptype = "watercolor");  
ggmap(Google) + geom_point(distemper, mapping = aes(x=distemper$Long, y=distemper$Lat),  
  color="darkorchid4", size = scaling*50);
```

Sometimes it really pays off to check out the options in a new function!!



4. R Lab 2: Ordination in R

This tutorial is largely taken from the following excellent tutorials:

- <https://www.datacamp.com/community/tutorials/pca-analysis-r>
- <https://stat.ethz.ch/pipermail/r-sig-ecology/2011-May/002125.html>
- <https://ourcodingclub.github.io/2018/05/04/ordination.html>
- https://rpubs.com/dillmcfarlan/R_microbiotaSOP

but with a lot of added detail and information to dovetail into our course better.

In this exercise, we'll do more graphing, but in not using GIS data. Instead we'll be doing principal component analysis (PCA) and principal coordinate analysis (PcoA) complete with plots. Principal Component Analysis (PCA) is a useful technique for exploratory data analysis, allowing you to better visualize the variation present in a dataset with many variables. It is particularly helpful in the case of "wide" datasets, where you have many variables for each sample. For example, it is very common to use these plots to visualize microbiome data or identify important ecological factors in niche partitioning.

So let's start with PCA in R.

4.1 Introduction to PCA

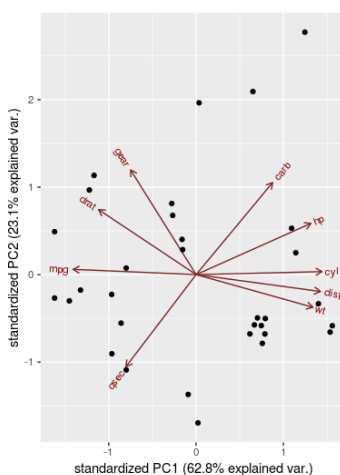
PCA is particularly handy when you're working with "wide" datasets. By "wide" we mean data that has a lot of characteristics (measurements) for each sample (row). These are usually in tables, which are wide due to the number of columns (measurements). But why is PCA a good tool to use with "wide" data?

Part of the problem with this data format is that it is very difficult to plot the data in its raw format – there are too many dimensions. The purpose of PCA is to visualize trends in the data by collapsing a large number of measures into a small number of dimensions, which hopefully help identify samples

that are similar to one another and which are largely different.

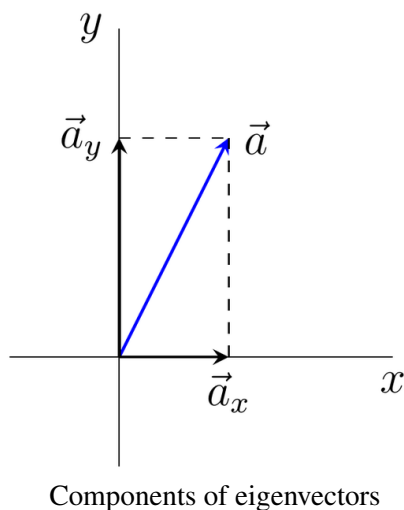
This isn't a statistics course, so I'll keep the explanation of how this is done brief (see the bottom for some more in depth resources). Basically, each math is done to combine measurements, each weighted differently, to create individual **principal components** that explain as much of the variation in the data as possible.

So for example, look at the graph below of car characteristics:



A basic PCA based on car metrics, taken from
<https://www.datacamp.com/community/tutorials/pca-analysis-r>

First, we can see the individual characteristics for the cars in the center, such as gear, mpg, etc. What these are specifically is not important at the moment - just know these are characteristics measured. When plotted, these characteristics are all spreading the data in different directions and by different magnitudes (shown by length of line – though these are largely similar). Each arrow (a) has a x (a_x) and y (a_y) component:

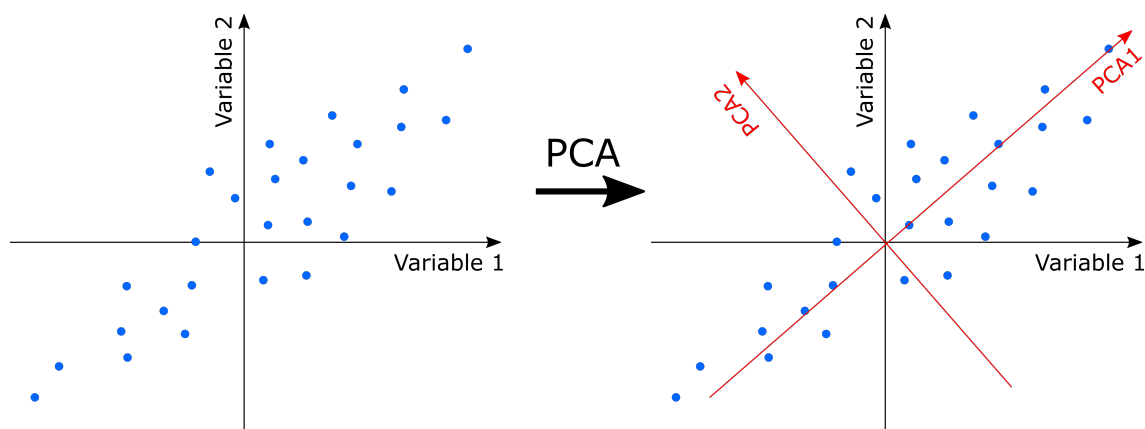


For instance, the mpg has a really small y component and a relatively large x component. This means that mpg contributed much more to PC2 (x) than PC1 (y). qsec, on the other hand, has relatively even x and y components, meaning it contributed similarly to PC1 and PC2. Both PC1 and PC2 are composed of some weighted value of each measurement (mpg was higher weighted than gear in PC1, for example). Since mpg and cyl have a large influence on PC1, which explains 62.8% of the data – these two measures are likely largely defining the variance in the data.

Another aspect of the data that becomes evident in PCA plotting is correlation – when several variables all contribute strongly to a variable, they are likely correlated. Mpg and cyl both contribute largely to PCA1, and are correlated – cars with more cylinders consume more gas.

4.1.1 Eigenvalues and Eigenvectors

You may see a lot of reference to eigenvalues and eigenvectors when looking into PCA plots. Simply put, an eigenvector is a direction, such as "vertical" or "45 degrees", while an eigenvalue is a number telling you how much variance there is in the data in that direction. The reddish brown lines in the plot above are vectors of the raw data; the combined data used to explain the data (PC1 and PC2) are eigenvectors, with eigenvalues (% variance explained). PCA calculates a bunch of eigenvectors and eigenvalues. The eigenvector with the highest eigenvalue is, therefore, the first principal component. Once the top 2-3 eigenvectors are selected, the plot is made rotating the data to make PC1 and PC2 horizontal and vertical.



Rotation of eigenvectors, taken from <https://ourcodingclub.github.io/2018/05/04/ordination.html>

Thought Questions

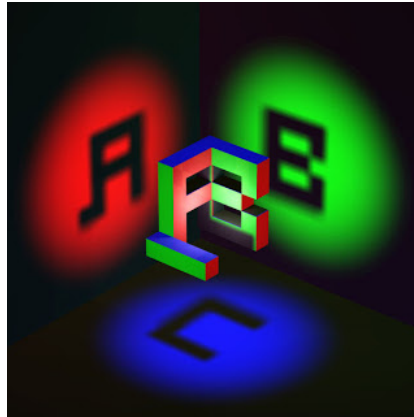
So wait, there are possibly more eigenvalues and eigenvectors to be found in one dataset?

That's correct! The number of eigenvalues and eigenvectors that are produced is equal to the number of dimensions the dataset has. In the example that you saw above, there were 9 variables, so the dataset was nine-dimensional (hard to plot!). That means that there are nine eigenvectors and eigenvalues. The highest two explained 85.9% of the variance, and they were plotted. The others are available to be viewed in R, but are not really that informative in this case.

We can also re-frame a dataset in terms of these other eigenvectors and eigenvalues without changing the underlying information. For example, you may see PC2 vs. PC3 in plots. Re-framing a

dataset regarding a set of eigenvalues and eigenvectors does not entail changing the data itself, you're just looking at it from a different angle (in nine-dimensional space...), which should represent the data better.

For example, consider this three dimensional shape:



Three dimensional object in three different two dimensional projections, taken from <https://en.wikipedia.org/wiki/Ambigram/media/File:3d-ambigram.jpg>

The shape of the object (data) does not change, but when you compress it into two dimensions, it looks different from different angles. The same applies here, but in more dimensions!

Now that you've seen some of the theory behind PCA, you're ready to see all of it in action!

4.2 A Simple PCA

In this section, we will build a PCA using a simple and easy to understand dataset and make the PCA plot from above. We will use the `mtcars` dataset, which is part of the `datasets` package in R – which we used to also graph the world map before. This isn't really a biological dataset, but it is one reasonably complex and has logical variables we can all relate to.

The `mtcar` dataset consists of data on 32 models of car, taken from an American motoring magazine (1974 Motor Trend magazine). For each car, you have 11 features, expressed in varying units (US units). They are as follows:

- **mpg**: Fuel consumption (Miles per (US) gallon): more powerful and heavier cars tend to consume more fuel.
- **cyl**: Number of cylinders: more powerful cars often have more cylinders
- **disp**: Displacement (cu.in.): the combined volume of the engine's cylinders
- **hp**: Gross horsepower: this is a measure of the power generated by the car
- **drat**: Rear axle ratio: this describes how a turn of the drive shaft corresponds to a turn of the wheels. Higher values will decrease fuel efficiency.
- **wt**: Weight (1000 lbs): pretty self-explanatory!
- **qsec**: 1/4 mile time: the cars speed and acceleration
- **vs** Engine block: this denotes whether the vehicle's engine is shaped like a "V", or is a more common straight shape.

- **am**: Transmission: this denotes whether the car's transmission is automatic (0) or manual (1).
- **gear**: Number of forward gears: sports cars tend to have more gears.
- **carb**: Number of carburetors: associated with more powerful engines

 Note that the units used vary and occupy different scales.

BEFORE YOU BEGIN: make sure `ggplot2` is not loaded, it will conflict with packages in this lab. You can either uncheck the box or you can run the following command:

```
detach ("package:ggplot2", unload =TRUE);
```

Problem 4.1 How would you grab this dataset (similar to how we grabbed `wrld_map`)?

4.2.1 Compute the Principal Components

You'll notice that this data is mixed numerical and categorical.

Problem 4.2 What data type do you think dataset is?

Because PCA works best with numerical data, you'll need to exclude the two categorical variables (`vs` and `am`).

Problem 4.3 How would you subset the data to remove `vs` and `am`?

Problem 4.4 Now that all of the data is the same type (numerical), how would you make this a matrix called `mtcars_matrix`?

You should now have a **matrix** of 9 columns (measures) and 32 rows (samples). We are now going to use the `prcomp()` function.

Problem 4.5 What package is this part of? How do you know?

Problem 4.6 How would you find some options for this function? What does `center` do? What does `scale` do?

Let's run the `prcomp()` function on our `mtcars_matrix` data, center it, and scale it. Let's also save the output as `mtcars.pca`.

```
mtcars.pca = prcomp(mtcars_matrix, center=TRUE, scale=TRUE);
```

Problem 4.7 What kind of object is `mtcars.pca`?

Let's take a look at this object with our handy `summary()` function. Remember, this function is great for giving some general information about tabular data in particular.

```
summary(mtcars.pca);
```

Importance of components: PC1 PC2 PC3 PC4 PC5 PC6 PC7 PC8 PC9

Standard deviation 2.3782 1.4429 0.71008 0.51481 0.42797 0.35184 0.32413 0.2419 0.14896

Proportion of Variance 0.6284 0.2313 0.05602 0.02945 0.02035 0.01375 0.01167 0.0065 0.00247
Cumulative Proportion 0.6284 0.8598 0.91581 0.94525 0.96560 0.97936 0.99103 0.9975 1.00000

As we expected, we will get 9 principal components, because we have nine measures. Each of these explains some portion of the total variation in the dataset. You can see that PC1 explains 62.8% of the data, and PC2 explains another 23.13%, for a cumulative percentage of 85.9%. This should seem very familiar!

Again, this means that by just knowing where a sample sits on the first two components, you get a really good idea of where it sits in relation to all the other data.

4.2.2 Plotting PCA

Now it's time to plot our PCA. Let's make a biplot, which includes both the position of each sample in terms of PC1 and PC2 and also will show us how the initial variables map onto this. This will result in the plot we talked about in the introduction.

ggbiplot is a package related to ggplot2, the standard basic plotting that we talked about before, but specific to the graphing of PCAs. A biplot allows for the plotting of points in relation to PC1 and PC2, to see the samples' relation to each other, as well as the plotting of the raw data vectors (reddish brown arrows) to see how each measure contributes to the distribution. This is standard practice for PCA.

Problem 4.8 How would you check if ggbiplot is installed? How would you install it and enable it? Installing from CRAN won't work – what should you do next?

ggbiplot takes a pca object, which we have! Let's feed it our data and see what happens by default:

```
ggbiplot(mtcars.pca);
```

You should see the car PCA, which will look very familiar. Because we centered our data, we see all vectors coming from the center point. We can see all the points, and get an idea of the distribution of the data, but not a great idea of where any one sample sits or what that cluster under wt might be.

Problem 4.9 How would you add the row names of mtcars to be labels to the ggbiplot?

Now you can see which cars are similar to one another. For example, the Maserati Bora, Ferrari Dino and Ford Pantera L all cluster together at the top. This makes sense, as all of these are sports cars with larger engines, lower fuel efficiency, more gears, and higher weight. You can also see that the Camero and Plymouth Duster 360 have terrible mpg and high horse power (they're 1970s muscle cars!) where as the Civic was a great efficiency car even back then.



You can also blot this with biplot(), a similar plotting function to plot(), meaning it uses the same parameters (xlab, main, col, etc). However, ggbiplot adds a lot of cool features that we'll go through below. But if you want a basic plot, biplot(mtcars.pca); will get you there with simpler formats to make it pretty! Remember, plot() family is easier, ggplot() family is more feature rich!

But let's look at the data further – there is more to learn and see here!

4.2.3 Interpreting the results

Sometimes it is nice to see what sample groups look like on the PCA. This is particularly helpful if you have species sets, geographical sets, treatment groups, basically any a prior group assignment.

For this data, let's make some groups using origin of manufacture. We can group these cars into three major regions – US, Japan, and Europe. Just like with GIS mapping, we can make a vector of this information and then look for an option in the function to allow us to add the groups.

First, let's make the vector – you can trust me on the origins:

```
mtcars.country=c("Japan","Japan","Japan","US","US","US","US","Europe","Europe","Europe",
  "Europe","Europe","Europe","Europe","US","US","US","Europe","Japan","Japan","Japan","US",
  "US","US","US","Europe","Europe","Europe","US","Europe","Europe","Europe");
```

Problem 4.10 How would you get ggbiplot to use these groups and form ellipses around them?

Now we see something interesting: the American cars form a distinct cluster to the right. Looking at the axes, you see that the American cars are characterized by high values for cyl, disp, and wt (this was the muscle car era in America after all!). Japanese cars, on the other hand, are characterized by high mpg. European cars are somewhat in the middle and less tightly clustered than either group.

We could also look at other PC's that explain less information and look to see how the groups line up with those eigenvectors. Let's have a look at PC3 and PC4. Is there an option in ggbiplot to choose which PC? There is!

Problem 4.11 Can you find the option to choose with Principal Component?

We don't see much here, but this isn't too surprising. PC3 and PC4 explain very small percentages of the total variation, so it would be surprising if you found that they were very informative and separated the groups or revealed apparent patterns. This is a good thing to try when exploring your data though, as you may have less heavily weighted "top" PCs.

4.2.4 Graphical parameters with ggbiplot

Since this an get pretty busy with all the circles, arrows, and names, please note there we can also remove the biplot arrows altogether with var.axes:

```
ggbiplot(mtcars.pca,ellipse=TRUE,choices=c(3,4), labels=rownames(mtcars), groups=mtcars.country,
  var.axes=FALSE);
```

As ggbiplot is based on the ggplot function, you can use the same set of graphical parameters to alter your biplots as you would for any ggplot. Think of this like the shared vocabulary in the various plot functions we used while mapping, but in this case a shared vocabulary for most "gg" plotting packages:

Attribute	plot()	ggplot()
color	col=	scale_colour_manual()
title	main=	theme() or ggtitle()
legend	legend()	theme()

ggplot does have a bit more complex syntax, which we saw while working with ggmaps (Google mapping package). Generally, you construct your plot first (as we have been doing) and then add in aspects you want to tweak – just like we made the ggmap and then added (with "+") the points we wanted to see.

Google maps example from before:

```
ggmap(Google) + geom_point(distemper, mapping = aes(x=distemper$Long, y=distemper$Lat),
  color="darkorchid4", size = scaling*50);
```

Mapping the PCA with similar syntax - + is used to add scale_colour_manual (just like col=) and ggtitle (just like main=):

```
ggbiplot(mtcars.pca,ellipse=TRUE, obs.scale = 1, var.scale = 1, labels=rownames(mtcars),
  groups=mtcars.country)
+ scale_colour_manual(name="Origin", values= c("forest green", "red3", "dark blue"))
+ ggtitle("PCA of mtcars dataset")+ theme_minimal()
+ theme(legend.position = "bottom");
```



This command is only split for the sake of printing - it must be all on the same line when you run the code.

4.2.5 Adding a new sample

Okay, so let's say we want to add a new sample to our dataset. This is a very special car, with stats unlike any other. It's super-powerful, has a 60-cylinder engine, amazing fuel economy, no gears (continuous transmissions are now a thing!) and is very light. It's a "spacecar", from Jupiter.

Can we add it to our existing dataset and see where it places in relation to the other cars? There are two options here – recalculating, which requires adding the new sample and then repeating the PCA and plotting; or projecting the sample directly onto our already created plot. The first option takes into account the **variance including the spacecar**, the latter places it within the variation **without including the variance introduced by the spacecar**.

Let's start with the first option. We will have to add the spacecar to the matrix, first by making the sample entry, and then binding it to the matrix that already exists – this is a super helpful method to know! We'll also have to do this for the group data.

```
#make a spacecar entry
```

```
spacecar <- c(1000,60,50,500,0,0.5,2.5,0,0);
```

```
#bind the entry to the matrix – similar to how we did data.frame(name, lat, long, pop) before!
```

```
mtcarsplus <- rbind(mtcars_matrix, spacecar);
```

```
#add to the matrix of countries – using c to concatenate!
```

```
mtcars.countryplus <- c(mtcars.country, "Jupiter");
```

And then we redo what we did before:

```
mtcarsplus.pca <- prcomp(mtcarsplus, center = TRUE, scale. = TRUE);
ggbiplot(mtcarsplus.pca, ellipse=TRUE, labels=rownames(c(mtcars, "supercar")),
         groups=mtcars.countryplus)
```

Yikes! That looks awful. Our nice PCA plot has changed dramatically – because we are including a HIGHLY variable sample.

When you consider this result in a bit more detail, it actually makes perfect sense. In the original dataset, you had strong correlations between certain variables (for example, cyl and mpg), which contributed to PC1 and separated our groups from one another along this axis. However, when you perform the PCA with the extra sample, the same correlations are not present, which warps the whole dataset. In this case, the effect is particularly strong because your extra sample is an extreme outlier in multiple respects.

If you want to see how the new sample compares to the groups produced by the initial PCA, you need to **project** it onto that PCA, the second option we discussed.

4.2.6 Project a new sample onto the original PCA

Projecting means is that the principal components are defined without relation to our new spacecar sample, then we compute where the spacecar is placed in relation to the other samples by applying the transformations that your PCA has produced. You can think of this as, instead of getting the mean of all the samples and allowing spacecar to skew this mean, you get the mean of the rest of the samples and look at spacecar in relation to this.

But how do we scale the values the same way our magic `prcomp()` function did? Well, two things happened – we scaled all the values to make eigenvalues, and we rotated the graph to make the PC1 and PC2 the x and y axis. So we need to do both!

The scaling vector is nicely stored in the "center" variable within the `pca` object.

Problem 4.12 How do we refer to `mtcar.pca`'s attribute, `center`?

Let's use the `scale()` function to scale the `supercar` vector using the `mtcars.pca$center`:

```
#save the pca as a new pca
mtcars.plusproj.pca = mtcars.pca;

#scale the spacecar vector by the information stored in the center.
#Note, supercar has to be transposed to do the matrix math, which is what t() is doing!
s.sc = scale(t(supercar), center= mtcars.pca$center);
```

We also have to rotate the sample to match up with the vertical and horizontal eigenvectors. Remember we said that we aren't changing the data, just changing the angle? We have to rotate it to match the rotation we did when plotting the initial function. Remember that ABC image above? We want to make sure we are viewing the data from the same angle as the graph! The `pca` object store the rotation it did in `mtcars.pca$rotation`:

```
s.pred = s.sc %*% mtcars.pca$rotation;
```

Thought Questions

What is %*%? It's matrix multiplication! We are multiplying `s.sc` (the vector) by the rotation factor in the `pca`.

Then we just add the `pca` object – which just places it on the graph without changing the calculation of the eigenvectors – and re-graph!

```
#add supercar to the matrix of vectors in the pca
mtcars.plusproj.pca$x = rbind(mtcars.plusproj.pca$x, s.pred);
```

```
#plot, adding supercar to the label
ggbiplot(mtcars.plusproj.pca,ellipse=TRUE, labels=rownames(c(mtcars, "supercar")),
  groups=mtcars.countryplus);
```

This result is drastically different. Note that all the other samples are back in their initial positions, while `spacecar` is placed somewhat near the middle. Our extra sample is no longer skewing the overall distribution, but it can't be assigned to a particular group – it is so far in another direction it doesn't show up well in this view.

Thought Questions

So which is better, the projection or the re-computation of the PCA? As always, it depends on the question that you want to answer. The re-computation shows the outlier, but makes interpretation of the rest of the data harder. If you are interested in the outlier or demonstrating the variance in the whole set, consider this. The projection makes it clear that the new sample is not part of any existing group, but doesn't warp your other data. This could be useful in checking if subsampling is sufficient (won't be warped by additional data), etc. However, since these are exploratory analyses, performing both approaches is often useful. This type of exploratory analysis is often a good starting point before you dive more deeply into a dataset.

4.3 A note on functions

There are numerous functions in R to do PCA, in slightly different ways:

- **prcomp()** which is a Singular value decomposition of variance/co-variance matrix where variance is calculated using $N-1$ and by default, data are not scaled
- **princomp()** which is an eigen analysis of the correlation of variance/co-variance matrix where variance calculated differently (N). It has no option for scaling data (need to do this before), but there is an option, `cor`, that specifies if you use correlation or co-variance matrix
- **rda()** is a vegan package function (shown later) used for constrained ordinations. It is similar to `prcomp()`, species and site scores are re-scaled according to Legendre, P. and Legendre, L. (1998) Numerical Ecology. 2nd English ed. Elsevier

- **capscale()** is a vegan package function to do Constrained Analysis of Principal Coordinates (CAP), which is an ordination method similar to Redundancy Analysis (rda), but it allows non-Euclidean dissimilarity indices, such as Manhattan or Bray–Curtis distance.

If you don't know when to use what, it's a great chance to talk to a statistician! Just know there are different ways with different applications and you should consider it when doing your analyses!

4.4 Principal Coordinate Analysis

There are many versions of multidimensional scaling of data matrices. PCA is one, PCoA is another common one we will cover. Correspondence analysis (CA) is another that is common in expression data. Non-metric dimensional scaling (nMDS) is also common, but computationally expensive. They are all very similar, only their distance really matters. PCA preserves Euclidean distance between numerical data in multiple dimensions, CA preserves chi-square distances between data, and PCoA first creates a Euclidean distance from whatever similarity/distance measure provided between samples then calculates eigenvectors. nMDS doesn't preserve any distance, but the relationships are maximized to best-fit through iterations of adjustments of their locations in n-dimensional space.

If that is really confusing, a good rule is PCA is for quantitative data, PCoA and nMDS is for categorical data, CA is better for expression data. (See https://www.researchgate.net/post/How_to_choose_ordination_method_such_as_PCA_CA_PcoA_and_NMDS and <https://academic.oup.com/femsec/article/62/2/142/434668> for a great summary and review of these topics). There are many options of packages when producing ordination graphs. However, similar steps are taken to do a PCoA as we did in a PCA – the first of which is to produce a distance matrix, since you are converting qualitative data into a matrix of distances between the samples. Then you compute the eigenvectors from that distance matrix, and finally graph.

Let's use the full mtcars dataframe, as we can now include qualitative data!

4.4.1 Distance calculations

There are several different ways to do this (see: <http://www.gastonsanchez.com/visually-enforced/how-to/2013/01/23/MDS-in-R/>). The `dist()` function comes with the stats package we've been using, so that's a good place to start. `dist()` allows several different methods of distance to be used, and we can feel free to use whichever is appropriate for our needs – remember PCoA isn't constrained to Euclidean distance like PCA.

Problem 4.13 How could you tell which options you have in `dist()`?

Another method with more options that may be of ecological use is the `vegdist()` function in the vegan package. This is a very popular package to use, so I am going to show you this package. Let's load it first:

```
library(vegan);
```

Problem 4.14 How could you tell which distance options you have in `vegdist()`? Let's use the `vegdist()` function, and produce a Euclidean distance matrix of our car data.

4.4.2 Computing the components

The next step is to perform the analysis to produce the eigenvectors. If you search in the help section for "pcoa" it will tell you it is available in the "ape" library, which is pre-installed for you. ape stands for "Analyses of Phylogenetics and Evolution" and is another common package we use in biology. However, it has limited handling of ordination, so we're going to stick with vegan. However, no other package is coming up...

That is because the vegan package uses a different name for the analysis – weighted classical multidimensional scaling. If you look at the help on the wcmdscale() function, you will see this:

"Weighted classical multidimensional scaling, also known as weighted principal coordinates analysis."

This function is also based on a stats function called cmdscale(), which is similar except that it doesn't weight the values and handles negatives differently. We can elect to not input weights, which makes wcmdscale() identical to cmdscale(), which doesn't allow for vegan's features. So we'll use wcmdscale().

While annoying, at least you now know!

Let's do a WCMDS/PCoA on our new distance matrix, with eigenvalues returned.

Problem 4.15 Can you figure out how to do this from the help section? Call your new variable vpcoa

As recommended, let's look at a summary of this object:

```
summary(vpcoa);
```

You should see two major things here: First, you should see a "points" attribute – which holds all the points to be plotted on the dimensions. You should also see a "GOF" attribute – which we will talk about shortly.



IF YOU DO NOT SEE A POINTS AND GOF LISTING – revisit the options in vpcoa and make sure you tell it to return eigenvalues!!

4.4.3 Graphing the PCoA

Now for the part we've all been waiting for – graphing. Again, there are multiple options (such as vegans's ordiplot()) but the one I like best is good ol' plot(). Since plot is so intuitive, let's just try the default:

```
plot(vpcoa, type="p") #points by using type="p";
```

Ok... that's pretty simple. But we're missing a lot here, like labels, group coloring, and the nice arrows that were so helpful in our interpretation. First, let's handle labeling those points using the ordilabel() function. Many functions in vegan start with ordi-, so if you are looking for something, start typing ordi into the help for a full list! Use help to investigate these commands before just typing them in!

```
ordilabel(vpcoa, labels=rownames(mtcars));
points(vpcoa$points, display="sites", pch=20, col=c("blue", "green", "red")[df$mtcars.country]);
legend(0, -0.05, legend=c("Japan", "US", "Europe"), col=c("green", "red", "blue"), pch=20);
```

Okay! Looking better! Now for those useful arrows! Vegan has a wonderful function called `envfit()` - "The function fits environmental vectors or factors onto an ordination." That sounds useful!

```
fit=envfit(vpcoa,mtcars);
```

We can now add this to our graph (and only plot significant values!):

```
plot(fit, col="black", p.max=0.05);
```

Great! Now let's revisit how to fit ellipses onto our data to see trends. Vegan has a function called `ordiellipse()` for just this occasion:

```
ordiellipse(vpcoa, groups=df$mtcars.country, display="sites", kind="ehull", conf=0.99, label=FALSE,
col="green", draw="polygon", alpha=200, show.groups = c("Japan"), border=FALSE);
ordiellipse(vpcoa, groups=df$mtcars.country, display="sites", kind="ehull", conf=0.99, label=FALSE,
col="red", draw="polygon", alpha=200, show.groups = c("US"), border=FALSE);
ordiellipse(vpcoa, groups=df$mtcars.country, display="sites", kind="ehull", conf=0.99, label=FALSE,
col="blue", draw="polygon", alpha=200, show.groups = c("Europe"), border=FALSE);
```

YIKES! That is a bit overwhelming. Thankfully the "kind" option allows us to display ellipses that are not all inclusive, but also standard error or standard deviation.

Problem 4.16 Take a crack at this yourself! Remake the graph, and add standard error ellipses.

Much nicer! Also, notice that this supports our previous PCA based finding that US cars are distinct and higher in wt, cyl, and hp; while Japanese cars were more efficient.

So, notice that this has no variance listed on the "dimensions". This is a result of the nature of how PCoA is done – the analysis is only getting a distance matrix, not a set of the original values. When reporting a PCoA, a goodness of fit value is usually reported. Since we saw "GOF" is an attribute of `vpcoa`, we can list just how we've listed all attributes – with \$!

```
vpcoa$GOF;
```

And now you have a R^2 value for each dimension!

Speaking of attributes, there is also a "weights" listed. . . this should read all 1's – meaning this is essentially identical to `cmdscale()`.

Problem 4.17 Try proving this to yourself by relotting the PCoA using `cmdscale()` rather than `wcmdscale()`.

4.4.4 More on Vegan

A major benefit of vegan is that it is very easy to explore with different ordinations, without having to change the code much. Try these out, they code should look very similar to the above PCoA code! Note, like PCoA, there isn't "variance explained", instead there is "stress" measures and R^2 fitting values.

Non-metric dimensional scaling:

```
d=vegdist(mtcars,methods="euclidean");
mds=metaMDS(d);
points(mds, display="sites", pch=20, col=c("blue", "green", "red")[df$mtcars.country]);
legend(0, -0.05, legend=c("Japan", "US", "Europe"), col=c("green", "red", "blue"), pch=20);
fit=envfit(mds,mtcars);
plot(fit, col="black", p.max=0.05);
```

```
# To plot stress:
stressplot(mds);
```

PCA: This is a slightly different type of PCA – it is a single value decomposition (SVD) based PCA, whereas the prcomp() version is a spectral/eigenvalue based PCA. Vegan specifically uses a redundancy analysis PCA – hence the rda() function. See <http://www.flutterbys.com.au/stats/tut/tut14.2.html> for more details.

```
vpca=rda(mtcars_matrix) #remember all quantitative for PCA
plot(vpca, display="sites") #see below
points(vpca, display="sites", pch=20, col=c("blue", "green", "red")[df$mtcars.country]);
fit=envfit(vpca,mtcars);
plot(fit, col="black", p.max=0.05);
summary(vpca) #will give the proportion explained, which can be added with x/ylab;
round(cumsum(100*vpca$CA$eig/sum(vpca$CA$eig)),2);
round((100*vpca$CA$eig/sum(vpca$CA$eig)),2);
```

The reason I specify only sites here is that we use the fit option to have prettier vectors of the environmental factors, scaled for readability. This should look VERY similar to the first plot we looked at! You can add ellipses to these the same way as above as well.

Some vegan resources:

https://rpubs.com/dillmcfarlan/R_microbiotaSOP
<https://www.fromthebottomoftheheap.net/2012/04/11/customising-vegans-ordination-plots/>
<https://cran.r-project.org/web/packages/vegan/vignettes/FAQ-vegan.html>

4.5 Wrap up and back to the biology

All of this process is to help us be able to look at the data in a variety of different "dimensions". In this case, we have **nine** dimensions (each eigenvector), but we can identify and concentrate on the most important ones. We can also form groups, map variables, etc. to let us get a handle on the structure of the data. This process also further demonstrated the two different graphing families – plot() and ggplot() syntax.

In this exercise, we have been working largely with non-biological data – because it is easy to approach for everyone and the data ships with R. This process of looking at data structure is common across many applications. We do this in population genomics to look at the effect of SNPs versus populations (PCoA – this is presence/absence data), metagenomic samples to look at the different community compositions (again PCoA – this is again presence/absence data), and ecological information looking at how physical factors such as river characteristics on biological factors such as turtle shell morphology (PCA – these are all continuous measures).

5. Writing Custom Scripts

5.1 Scaling Up – Saving Your Work as a Script

A handy feature in RStudio is the ability to construct a script as you go. Thus far we've been working almost exclusively in the console, where our commands and our output are all together. This is great for learning and working out what you want to do, but it can be a bit of a bear to rerun scripts. This is where the "Source" window shines.

To pull up a source window in RStudio Server (it is default on RStudio desktop I believe), go to the "View" tab on the menu bar, and select "Move Focus to Source" (or hit ctrl and the 1 key). This will pull up a blank window at the top of the console.

You can think of this as a place to save commands that work for whatever you are currently working on. As a basic example, let's say we were trying to figure out how to define a sequence from 1-10 as a vector, and then print the first four entries (remember how to do this?)

In the console, you may begin working to figure out exactly what it is you need to do, especially if you don't remember. Eventually you will come up with something similar to:

```
vector=seq(1,10,1);  
print(vector[1:4]);
```

Great! Now that you know that works, put that code in the source window. You can now rerun this code line by line or as a whole. To run just one line, you can click the end of the line, then the button that has a box with a green arrow. To run multiple lines, highlight what you'd like to run, and then click the same button. The output will still print out to the console, but it is much easier to rerun and troubleshoot.

You can save any R code on a cluster terminal by clicking the blue floppy disk icon in the menu bar of the source window. I'm saving this example as demo.R. ".R" is a typical extension for R code, but you can technically make it whatever you'd like. Sometimes you will see .RScript as well.

To run this on a terminal, we can click the terminal tab at the top of our console. If you type "ls", you can see a list of all the files you have saved to your virtual machine that is running RStudio. demo.R should show up.

To run it, type:

```
Rscript demo.R
```

Notice, this only outputs what you explicitly print in the script!

5.2 Loading in Other People's Scripts

Another use for source is to enable customization of longer scripts, allowing you to run all of the script to a certain point, change a line, and keep working on that line until it does what you'd like – all without having to type in all the prior commands each time!

But in order to do that... we have to first learn to import scripts.

First, click the x on your demo.R tab to close that script. Your source window may disappear, but don't worry – we're going to load it right back up.

load the script in RStudio:

File -> open file -> demo.R and use it as we have been.

If you were on the cluster, you could also do the following (just as before):

```
Rscript ~/demo.R;
```

but notice – this will not allow you to change things as you go. To do that, you'd have to edit the file and rerun it, and repeat. This is why we focus on running things in RStudio to start – it is much much easier to develop code in the RStudio than on the cluster.

Thought Questions

But why run it on the cluster then?

RStudio is limited by the machine it is on – it may not have the power to run all of your data if it is on your laptop! It is common practice to use smaller subsets of your data to work out the code using RStudio. Then you can move your saved script of working code to the cluster and run it against the full dataset with more resources (high ram, etc).

5.3 Best Practices in R

Since you can only get so far with the basic commands that we've been working with thus far, we are going to be talking about how to scale this up to writing full programs in R – including functions! Now, you may not need to do this for a while, but going through the process of making a function will help you understand their structure, and how to read any new ones that come your way – via other packages or labmates/postdocs/etc!

5.4 What is a function?

Functions are the verbs we've been using – cat, print, load, etc. These are usually baked into a class that you load via bioconductor or CRAN, but you can write them yourselves. R is actually largely a community written language – and for better or worse – you may have to read other people's code to figure out what is going on, especially in CRAN packages. You may also want to expand R's ability to do something (adding your own verbs!) as you use new or odd data.

One reason I like to introduce writing in functions early is that it makes any code you write much more reusable! If your scripts are well organized and have discrete actions, these can be pulled out and used elsewhere. I reuse the code to graph a sliding window all the time – something I wrote in the first two weeks of learning R. (Spoiler – we'll be doing this in Chapter 6!)

Let's look at a function to learn some anatomy (**assuming seqinr is loaded!**):

`?count`

`count(seq, wordsize, start = 0, by = 1, freq = FALSE, alphabet = s2c("acgt"), frame = start)`

R functions can take a number of required parameters ("options") and/or a number of optional parameters, which are listed in the help. Remember, anything that is not followed by an = has no predefined value. So in the case of count, you need to specify the sequence and the wordsize every time – there is no default. However, start, by, freq, alphabet, and frame can all be ignored if you want to use the default values – which are all listed after the equal sign. To make things easy to read, it is generally recommended to put all the required options up front, followed by the optional parameters.

So, to make a function, we will need:

- 1) a name
- 2) required and/or optional parameters
- 3) code

1) A name

Function and variable names can have a "." as part of the name in R, however, in most languages "." has other meanings. I wouldn't get used to naming functions with "."s in them as a matter of consistency.

2) Parameters - See above

3) Code

This is the most flexible part, but a couple of guidelines: A function should encapsulate a single "idea" - it shouldn't be too long, or try to do too much. It should be a single "verb", not an entire complex sentence.

Only assume the existence of data given to it as a parameter ("option") - don't refer to any outside variables; they may not exist! You should be able to copy and paste an entire function to use in another program.

Make sure to write comments to document what functions do, what parameters are, and what they return. Since these functions won't have a help file, you will need to put the information in the code! Comment lines start with # in R.

5.5 Writing a Function in R

Ok, now that we know some basic rules, let's talk write a basic function. Let's make a super simple function to start – multiply two numbers and add an optional one.

- 1) **name** – call it myFunction
- 2) **parameters** - requires two numbers, optional third
- 3) **code** - basically `num1*num2+optnum3`

To define a function, the first line looks a lot like the results you look for in `?cat`, etc.:

```
myFunction = function(num1, num2, optnum3 = 0)
```

The only difference is the `"= function(...)"`. This is similar to how we have defined matrices, vectors, etc. It is called a "constructor" function – it constructs what it is labeled! In this case, it makes a function. Now, when you hit enter, you will see that R is expecting more (gives you the `"+"` instead of a `">"`). It wants code!

So let's give it some. Functions are wrapped in `"{"` - R's punctuation based marker for a the code equivalent of a "paragraph". It groups all the code going into this function together.

```
myFunction = function(num1, num2, optnum3 = 0)
{
  sum = num1 * num2 + optnum3;
  return(sum);
}
```

Thought Questions

What is this return thing?

When functions run, they are kind of in their own space – the end product has to be returned if you want to be able to save the value. This isn't always required, such as in "make a graph" functions that have special output. But if you want to be able to do:

```
myResult = myFunction(1,2);
```

You need to return something for R to save in myResult.

Why is there no output when this function is run?

You are simply defining a function – not actually using it. The lack of output is a good thing – it means it loaded without issue and is ready to use!

Now our function, myFunction, can be called in several different ways (even if all the parameters aren't used):

```
result = myFunction(5, 6, 2);
result = myFunction(5,6);
result = myFunction(5,6, optnum3 = 4);
```

5.6 Blast Summary Function

Let's do something a bit more complex! Let's start a script file, and begin building a program that "summarizes a table".

1) name: table_summary

2) parameters: Let's plan to give the function a file, and have it default to giving the average to each column, but optionally also give histograms in a pdf (default named out.pdf) for each column as well. (Don't worry if you know how to do this quite yet!).

3) code: We'll build this as we go! But let's start with some "pseudo code" - a logical plan

In the source file section, let's lay out the basic outline:

```
table_summary = function(file, out="out.pdf", hist=FALSE)
{
    #code here
}
```

Okay, now let's fill out that plan a bit more:

```
table_summary = function(file, out="out.pdf", hist=FALSE)
{
    #read in file

    #print averages

    #make histograms if asked
    #save as pdf
}
```

If you run this, you shouldn't get errors – the function is there, but it doesn't do anything yet! Before we start building the parts, let's first check to make sure the inputs are reading correctly – debugging as you go is ALWAYS A GOOD IDEA!!!

A simple way to debug is print statements – print variables you have to make sure they are outputting what you think they should- in this case the information we are giving the function:

```
table_summary = function(file, out="out.pdf", hist=FALSE)
{
    #check input
    print(out);
    print(hist);

    #read in file

    #print averages
```

```
#make histograms if asked
#save as pdf
}
```

Thought Questions

Huh... when I run that... I don't get any output...

Remember, you are still just defining the function – you need to call it to have it actually do anything!

```
table_summary("BlastResults.txt");
table_summary("Wetland Data.txt");
```

Now, it doesn't matter what table you give it, so I have two examples – results from blast (BlastResults.txt) and some physical pond data from the USGS (Wetland Data.txt). Feel free to use either, but I will test with BlastResults.txt

Great! Just what we expected! Now we can remove those tests so we don't see that every time – or we can comment them out for use later.

Now to do the first part of the code – reading in the file. Good news – we already know how to do that! You can use functions inside your function!

NOTE: If you are going to use something that is part of a non-default package (GC count for instance!) you should put a comment in the function saying what libraries are necessary – or load them in the function!

```
table_summary = function(file, out="out.pdf", hist=FALSE)
{
  #check input
  #print(out);
  #print(hist);

  #read in file
  table = read.table(file=file, header=TRUE, row.names=1);
  print(table);

  #print averages

  #make histograms if asked
  #save as pdf
}
table_summary("BlastResults.txt");
```

table_summary("BlastResults.txt") or table_summary("Wetland Data.txt") should now output a part of that table. Success!

5.6.1 Loops

For the next two parts, we will need to talk about loops. Loops are super helpful features to know how to read, but also to know how to run. Loops allow you to build a bit of logic into your code – if something is true, run this; for each column, run this.

If and for loops are very common across all languages. Conveniently, an if loop is similar across all languages, because it is a basic logic function. The major differences are typically in syntax details.

Let's look at an example in R. You can type this in your console, since it's just code we are using to learn how this works, but it can be hard to do this with multiple lines – so let's open another script and type it in there:

File New File R Script

```
var=11;

if (var < 10) {
  print("var is less than 10");
} else if (var == 10) {
  print("var is 10");
} else {
  print("var is greater than 10");
}
```

Note that the "else if" and "else" are on the same line as the closing bracket, but not contained within the preceding clause. Also note else if can be a number of different variations in different languages (elseif, else if, elsif, etc.).

Other than having to probably refresh your mind on what the syntax is, if statements are pretty intuitive. Let's add one into our blast_summary function to handle what to do if histograms are requested.

```
table_summary = function(file, out="out.pdf", hist=FALSE)
{
  #check input
  #print(hist);
  #print(out);

  #read in file
  read.table(file=file, header=TRUE, row.names=1);

  #print(table);

  #print averages

  #make histograms if asked
  if(hist==TRUE) {
    print("We will print a histogram soon!");
  }
}
```

```

    #save as pdf
}

table_summary("BlastResults.txt");
table_summary("BlastResults.txt", hist=TRUE);

```

You should see the loop work only if hist is TRUE! We'll leave it at this for now, let's look at the other loop we want – for each column.

For loops allow us to do something for each element in a set... each element in a vector, each column in a matrix, or each line in a file. Let's use that other script and replace our demo if loop with a demo for loop:

```

vector = c(0,1,2,3,4);
for (element in vector) #NOTE you can name the elements whatever you want!
{
    cat("Element is", element, "\n");
}

```

Thought Questions

How does R know what the "elements" are of the variable we are referring to?

R knows what the 'chunks' of information are for each class of object – it is specific to each class of variable – matrix, vector, etc.! Even though the function name is the same – the code is different pending what kind of variable you are giving the for function.

So, that's super useful. Let's add TWO for loops to our code we are building – one to loop through and make a histogram for each column, and one to output the average of each column.

```

table_summary = function(file, out="out.pdf", hist=FALSE)
{
    #check input
    #print(hist)
    #print(out)

    #read in file
    table=read.table(file=file, header=TRUE, row.names=1);
    #print(table);

    #print averages
    for (i in 3:ncol(table))
    {
        #averages coming soon
        print(i);
    }

    #make histograms if asked
    if(hist==TRUE) {

```

```

    print("We will print a histogram soon!");
    for (i in 3:ncol(table))
    {
        print(i);
    }

    #save as pdf
}
}

```

Thought Questions

Why do we start at column 3?

Both tables have non-data columns for the first columns. In BlastResults.txt, column 1 and 2 are gene names and is the DB_hit – a string of text. We cannot average this, and it will spit out an error if you try. In Wetland Data.txt, this is date information, and not terribly helpful to average.

Let's finish up this averaging section before we move onto graphing. I wonder how I can find how to perform an average... specifically a mean... Oh, look there's a super easy function listed in the help docs! Let's test in on the console, using just the first data column:

```

table = read.table(file="BlastResults.txt", header=TRUE, row.names=1);
mean(table[,3]);

```

Easy enough! Let's add that to our function, but using "i" instead of hard coding "3":

```

table_summary = function(file, out="out.pdf", hist=FALSE)
{
    #check input
    #print(hist)
    #print(out)

    #read in file
    table=read.table(file=file, header=TRUE, row.names=1);
    #print(table);

    #print averages
    for (i in 3:ncol(table))
    {
        #averages coming soon
        #print(i);
        avg = mean(table[,i]);
        print(avg);
    }

    #make histograms if asked
    if(hist==TRUE) {

```

```

        print("We will print a histogram soon!")
        for (i in 3:ncol(table))
        {
            #print(i);
        }

        #save as pdf
    }
}

table_summary("BlastResults.txt");

```

Before we move on, let's at least clean up that output so we can see what we are actually averaging:

```

table_summary = function(file, out="out.pdf", hist=FALSE)
{
    #check input
    #print(hist);
    #print(out);

    #read in file
    table=read.table(file=file, header=TRUE, row.names=1);
    #print(table);

    #print averages
    for (i in 3:ncol(table))
    {
        #averages coming soon
        #print(i);
        avg = mean(table[,i]);
        name = colnames(table[i]);
        cat(name); #cat won't put this on a new line
        print(avg); #this will print after the cat output and then add a new line
    }

    #make histograms if asked
    if(hist==TRUE)
    {
        print("We will print a histogram soon!");
        for (i in 3:ncol(table))
        {
            #print(i);
        }

        #save as pdf
    }
}

```

```
table_summary("BlastResults.txt");
```

Whoohoo! Our function runs in default mode! Now let's get that graphing part done!

Thought Questions

Other Useful Statistical Functions

Some useful functions you may want, that we will not really go into in this example – but it helps to know their names!

lm() - used to fit linear models

aov(deptvar indept1*indept2*indept3) - ANOVA (use + signs if you don't want interaction of factors).

t.test() - t-test

hclust() - hierarchical clustering

5.6.2 Some Graphing Functions

Remember that `plot()` is the most generalized graphing function. If you give it all numeric data (numeric vs. numeric) it'll do a scatter plot. Let's review some `plot()` mechanics:

```
plot(seq(0,10,1), seq(0,10,1), main = "the Title", xlim=c(1,20), ylim=c(1,20), col="darkblue");
```

But we can also plot other functions, such as `hist()`, which is of particular interest in this function. We can define breaks if we want to, but we can also have the function figure out what makes:

```
table=read.table(file="BlastResults.txt", header=TRUE, row.names=1);
hist(table$bitscore);
hist(table$bitscore, breaks=seq(0,900,1));
hist(table$bitscore, breaks=seq(0,900,10));
```

Let's add the code for histograms to our function! We should also remove the code to print the ugly histogram information:

```
table_summary = function(file, out="out.pdf", hist=FALSE)
{
  #check input
  #print(hist);
  #print(out);

  #read in file
  table=read.table(file=file, header=TRUE, row.names=1);
  #print(table);

  #print averages
  for (i in 3:ncol(table))
```

```

{
  #averages coming soon
  #print(i);
  avg = mean(table[,i]);
  name = colnames(table[i]);
  cat(name);
  print(avg);
}

#make histograms if asked
if(hist==TRUE)
{
  #print("We will print a histogram soon!");
  for (i in 3:ncol(table))
  {
    #print(i);
    hist(table[,i], main=colnames(table[i]));
  }

  #save as pdf
}
}

table_summary("BlastResults.txt", hist=TRUE);
table_summary("Wetland Data.txt", hist=TRUE);

```

See all the histograms popping up on the side panel? You can scroll through them with the arrows, or save them to a file individually. But... we want to have these write to a file.

5.6.3 Writing to file

To direct all output to a file, you use the sink function (opposite of source)!

```

sink("myfile", append=FALSE, split=FALSE);
sink(); #return output to the terminal

```

But sink() will not redirect graphic output, which we are interested in doing for our hist function! To redirect graphic output use one of the following functions. You will have to use dev.off() to return output to the terminal – if you forget, the file will be garbled.

Function	Output to
pdf("mygraph.pdf")	pdf file
win.metafile("mygraph.wmf")	windows metafile
png("mygraph.png")	png file
jpeg("mygraph.jpg")	jpeg file
bmp("mygraph.bmp")	bmp file
postscript("mygraph.ps")	postscript file

Just like in sink, we have to turn off the function to make sure we return our output to the console. To do this we use **dev.off()**. This is critical to functions where you are producing a non-text format, as the lack of an End of File (EOF) marker can cause the output to be unreadable!

Use a full path in the file name to save the graph outside of the current working directory. Let's output our histograms to a pdf that we can name with the out parameter:

```
table_summary = function(file, out="out.pdf", hist=FALSE)
{
  #check input
  #print(hist);
  #print(out);

  #read in file
  table=read.table(file=file, header=TRUE, row.names=1);
  #print(table);

  #print averages
  for (i in 3:ncol(table))
  {
    #averages coming soon
    #print(i);
    avg = mean(table[,i]);
    name = colnames(table[i]);
    cat(name);
    print(avg);
  }

  #make histograms if asked
  if(hist==TRUE) {
    #print("We will print a histogram soon!");
    pdf(out);
    for (i in 3:ncol(table))
    {
      #print(i);
      hist(table[,i], main=colnames(table[i]));
    }

    #save as pdf
    dev.off();
  }
}

table_summary("BlastResults.txt", hist=TRUE);
```

Now look in the files tab – you have a new pdf that you can view! This should be saved as "out.pdf", but we could define a different name by using:

```
table_summary("BlastResults.txt", hist=TRUE, out="different_name.pdf");!
```



Note that we put the pdf() and dev.off() outside the loop. If we don't the file will open and close each time the loop executes and we won't save all the information!

5.6.4 Final Cleanup

Let's remove all the print statements and make it pretty – add comments where you think you might need to remember things, etc. Here's mine – yours may look different!

```
table_summary = function(file, out="out.pdf", hist=FALSE)
{
  #read in file
  table=read.table(file=file, header=TRUE, row.names=1);

  #print averages
  for (i in 3:ncol(table))
  {
    avg = mean(table[,i]);
    name = colnames(table[i])
    cat(name);
    print(avg);
  }

  #make histograms if asked, save as pdf
  if(hist==TRUE)
  {
    pdf(out);
    for (i in 3:ncol(table)) #col 1,2 are text
    {
      hist(table[,i], main=colnames(table[i]));
    }
    dev.off();
  }
}
```

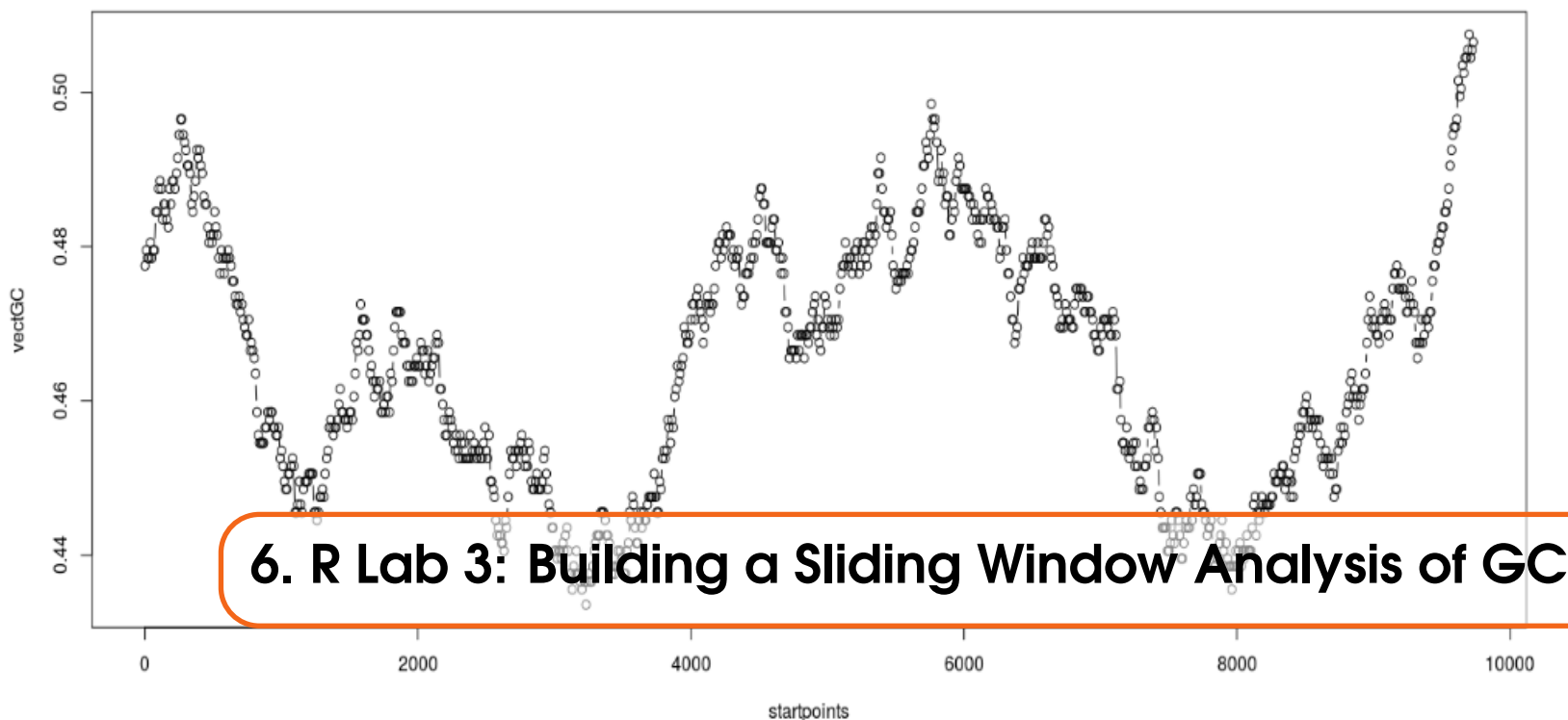
You can now save it, close the file, and source it, move it to another RStudio installation on a different computer, or load it via terminal!

You can also easily change things – you can change mean to median, or highest, or lowest. You can use any file you'd like – try it with both example files. You could have it graph bar graphs, etc. It's very flexible!

Also, this is a good primer on how to build code – make a logical plan, and build and test in pieces!!

That's the basics of R – you can now handle default data types, know how to find more libraries to add more data types, get and read the help, and write real R code! While this may not all sink in right away, you can use these notes as a reminder at any time. It takes time, and you will be googling your way through it for years, but at least you can now read and get help!

The next chapter will help you practice making your own function with a little less hand-holding!



Let's dig into the GC content a bit further. In order to study local variation in GC content within a genome sequence, we could calculate the GC content for small chunks of the genome sequence. The DEN-1 Dengue virus genome sequence is 10735 nucleotides long (how would you find this?).

First, you will have to get the sequence again (**make sure you load seqinr!**):

```
dengue = read.fasta("dengue.fasta");  
  
#grab only the sequence as a vector  
dengueseq = getSequence(dengue$NC_001477.1);
```

To find it's GC content, we can do the following:

```
GC(dengueseq);
```

If we wanted to look at variation within the sequence, we could calculate the GC content of each range of the DEN-1 Dengue virus genome, for example, for each 1789bp chunk of the genome sequence (the genome is evenly divisible into 1789bp ranges). The process of looking at small sections ("windows") in order is referred to as a "sliding window" analyses. Often these "windows" overlap, but for simplicity, we are going to use largely non-overlapping windows in this case study.

```
# Calculate the GC content of nucleotides starting at 1 and including the next  
#1789bp (or to the 1790th) of the Dengue genome:  
GC(dengueseq[1:1790]);
```

0.4689771

Now, calculating the next "window" requires you to know the endpoint of the second range. We could do some quick addition – or we could make our mathematical language do it for us!

```
# Calculate the GC content of nucleotides 1790-3579 of the Dengue genome
GC(dengueseq[1790:(1790+1789)]);
```

0.4561697

Notice that you can use expressions to replace the start and end range of a vector subset.

Problem 6.1 Calculate the rest of the windows manually to get practice using ranges of vectors. How do you know when to stop? Do we know a function to identify the length of a sequence?

From the output of the above calculations, we see that the region of the DEN-1 Dengue virus genome from nucleotides 1-1790 has a GC content of 46.9%, while the region of the Dengue genome from nucleotides 1790-1789 has a GC content of about 45.6%. The rest of the windows you calculated are slightly different as well. Thus, there seems to be some local variation in GC content within the Dengue genome sequence. GC content can be an indicator of protein coding regions, epigenetic hotspots, or other architectural points of interest. So let's make a means of investigating this variation, building a function, so we could feed it any sequence.

6.1 Pseudocode

You can see that it is a bit tedious to type in the individual ranges after a while – imagine if I gave you a mammalian genome. Or worse, a tree. This would take forever. Thankfully, we learned a major trick early that will help us out – the **for loop**!

Let's make a "gcByRange" function that takes a sequence (i.e. dengueseq) and a size (i.e. 1789bp), and returns the GC content **for each** window.

To start, we need a function outline. Remember that functions look like this:

```
NAME = function(reqparameter1, reqparameter2, optparameter=TRUE)
{
  #CODE
}
```

Start the function header. You may want to start a source tab as we will be building out this function as we go!

```
gcByRange = function(sequence, size)
{
  #CODE
}
```

Problem 6.2 What's the first thing you should do? Test the inputs and the function! Add code in the #CODE section to **print out the sequence and window size** (size of chunk being analyzed) when you run:

```
gcByRange(dengueseq, 1789);
```

Great! You have a working, albeit kind of pointless, function! Let's get some pseudocode in there to outline some functionality.

```
gcByRange = function(sequence, size)
{
  #for each window
  #subset sequence vector into correct range
  #print gc content of that "window"
}
```

6.2 Filling in the Code

Good news is both steps to this process sound like things we know how to do! Let's start with that subsetting part, we'll worry about the for loop here in a second.

When you calculated GC content for the first range (1-2147), it looked something like this:

```
GC(dengueseq[1:1790]);
```

When you did the next chunk, it was something similar:

```
GC(dengueseq[1790:3579]);
```

A pattern starts to emerge: **GC(dengueseq[STARTPOINT : (STARTPOINT+1789)]**

If we could only make a list of start points, starting at 1 and continuing to a set length, incrementing by 1789...

Wow, that sounds a LOT like **seq()**. Due to inclusiveness in this function, we want the end point to be the length of dengueseq – 1. Otherwise we end up with a startpoint at the last bp of dengueseq, which will cause errors.

Problem 6.3 Armed with that information complete the following (replace the ?s), so that the start-points start at 1 and continue through the length of dengueseq minus 1, by intervals of size:

```
gcByRange = function(sequence, size)
{
  #for each window

  #subset sequence vector into correct range
  startpoints=seq( ?, ? , ? );
  print(startpoints);

  #print gc content of that "window"
}
gcByRange(dengueseq, 1789);
```

You should see the following output:

```
[1] 1 1790 3579 5368 7157 8946
```

So there is a pattern we could use **for each** calculation. So let's make that a for loop with this pattern! Let's add in that simple print statement so we can see what is happening.

Problem 6.4 Complete the following (replace the ?s):

```
gcByRange = function(sequence, size)
{
  #define start of windows
  startpoints=seq( ? , ? , ? );
  #print(startpoints);

  #for each window
  for ( i in 1:length(startpoints)) {

    #subset sequence vector into correct range, and grab GC
    gc = GC(denguseq[ ? : ? ]);

    #then print gc content of that "window"
    print(gc);
  }
}
```



Pro Tip: If you want to start by hard coding in some of the numbers – i.e. `gc=GC(denguseq[startpoint[1]:(startpoint[1]+1789)])`, you can! Your loop will just repeat the same output for each loop. You can then change the 1 to `startpoint[i]`, etc. and test as you make changes. There is NO NEED to make it run on the first try!

Now you should get a full set of ranges when you run:

```
gcByRange(denguseq, 1789);
```

Thought Questions

Okay, now test with other ranges... oh no! It breaks! What's wrong??

What is happening is that if the last window is shorter than the window size, we end up telling R to calculate the gc content over a range that continues past the endpoint of the sequence. R doesn't know what you want to do here, so it just errors out. How would you test to see that this is happening?

To fix our code so it doesn't break, after we create the list of startpoints (stored in a vector), we want to check to see **if** the last window is too small, and **if** so, remove it. I'll be nice and give you the

code for that, but see if you can read and understand how that works! Also, this is the full code – your’s may look a bit different, but this is an example of a completed version of the function we built here!

```
gcByRange=function(sequence,size) {
  #define start of windows
  startpoints=seq( ? , ? , ? );
  #print(startpoints);

  #determine start of last window size
  n=startpoints[length(startpoints)];

  #determine length of last window
  lastsize = length(sequence[n:length(sequence)]);

  #if last segment is too small, remove the last vector entry
  if (lastsize < size) {
    startpoints=startpoints[1:(length(startpoints)-1)];
  }

  #for each window, calculate and print the gc content
  for (i in 1:(length(startpoints)))
  {
    gc=GC(sequence[ ? : ? ]);
    print(gc);
  }
}
```

Now the following will give you different gc contents!

```
gcByRange(denguseq, 1789);
gcByRange(denguseq, 45);
gcByRange(denguseq, 3000);
```

 Save your work!

You may want to use or reference code you write later, so it’s a good practice to have some library of your former work.

6.3 Adding a Plot!

It is common to use the data generated from a sliding window analysis to create a sliding window plot of GC content. To create a sliding window plot of GC content, you plot the local GC content in each window of the genome versus the nucleotide position of the start of each window.

Good news is, we don’t have much to change!

Start a new source tab and copy and paste (it’s okay **within** R) the code you had above. First let’s make a plan:

- 1) **name:** keeping it the same!
- 2) **parameters:** sequence and window size
- 3) **code:** We just have to save the gc output we generated above and then plot it afterwards. The easiest way to do this is below:

```
#for each window
#subset sequence vector into correct range and SAVE gc
#print gc for fun
#plot outside of loop!
```

So... how do we save an **ordered set of numbers**? You got it! A vector! But we need to define a vector first and then fill it. The trick here is that we know how long of a vector we'll need – it's the number of windows, which is the number of startpoints! Note, this code will not run as "startpoints" is not defined!

```
...
vectGC = vector(length=length(startpoints));
...
```

Problem 6.5 Now, we want these to be numbers, so is there a parameter in the vector function that will allow us to define what scalar the vector is composed of? Complete the code(replace the ?):

```
...
vectGC = vector(length = length(startpoints), ? );
...
```

Problem 6.6 If you print vectGC, you should see a set of 0's – default place holders for numbers. Let's bake this vector build into our function. Complete the code(replace the ?):

```
gcByRange = function(sequence, size)
{
  #define start of windows
  startpoints=seq( ? , ? , ? );
  #print(startpoints);

  #determine start of last window size
  n=startpoints[length(startpoints)];

  #determine length of last win.last
  lastsize = length(sequence[n:length(sequence)]);

  #if last segment is too small, remove the last entry from the vector
  if (lastsize<size) {
    startpoints=startpoints[1:(length(startpoints)-1)];
  }
}
```

```

#this has to come after startpoints is at its final length!
vectGC= vector(length = length(startpoints, ? ));

#for each window, calculate and print the gc content
for (i in 1:(length(startpoints)))
{
  gc=GC(sequence[ ? : ? ]);
  print(gc);
  #save GC into vectGC
}

#plot GC versus genomic position
}

```

Problem 6.7 Next, how do we save a variable (gc) into the vector at a specific point (slot "i" in the vector)? Add it to your function!

Now all we have to do is plot gc. We want the x to be the startpoint of the window, the y to be the gc content, and the plot to have both lines and points.

Problem 6.8 See if you can figure this out and add it to your function!

Ok, time to clean it up – I like to put as many of the definitions at the top as possible, but this is not required.

```

gcByRange = function(sequence, size)
{
  #define start of windows
  startpoints=seq( ? , ? , ? );
  n=startpoints[length(startpoints)];
  lastsize = length(sequence[n:length(sequence)]);

  #if last segment is too small, remove the last entry from the vector
  if (lastsize<size) {
    startpoints=startpoints[1:(length(startpoints)-1)];
  }

  vectGC= vector(length = length(startpoints, ? ));

  #for each window, calculate and print the gc content
  for (i in 1:(length(startpoints)))
  {
    gc=GC(sequence[ ? : ? ]);
    print(gc);
    vectGC[i]=gc;
  }
}

```

```
#plot GC versus genomic position
plot(startpoints, vectGC, type="b");
}
```

Save your work! Test it with some numbers or a different sequence.

```
gcByRange(dengueseq,400);
gcByRange(dengueseq,50);
gcByRange(treeseq,4670); #this sequence name is made up
```

You now have a prototype of a sliding window plotting function. You can change what exactly you are graphing very easily – maybe you care about certain "words". You can easily change make a similar function to take a vector of numbers and plot the average of those numbers (such as if you had snp occurrence data). You have a lot of power knowing only the basics of R!!

6.4 Just for fun – overlapping windows

How would you make the windows overlap? This is a bit more advanced, but not by much! You'd just have to rearrange a bit to:

- 1) add a parameter for step size between window size
- 2) define you startpoints by step not size
- 3) you'd have to check each window to see if it was too short, not just the lastsize
- 4) if the last window is too short, you need to shorten the startpoints vector to remove all remaining windows, and the vectGC to remove all remaining 0's for those windows – otherwise (else!) proceed as normal.

Can you read the following function and see where those changes occurred? This was not something I changed in one go – it was slow changes, testing along the way. I included it for completion, and for your reference in thinking about how to do this! Also, look how far you've come!!

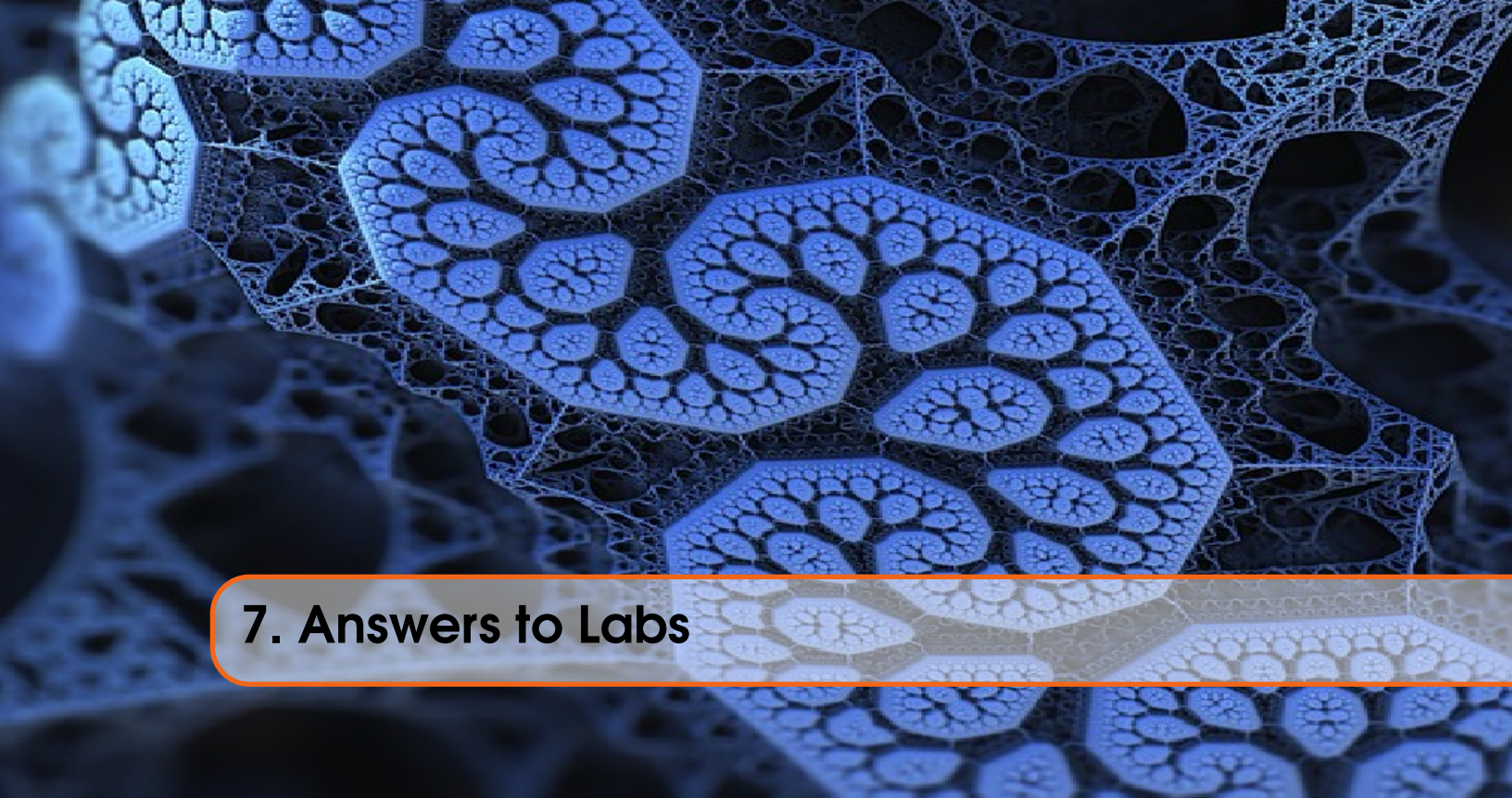
```
gcByRange=function(sequence,size,step) {

  #define start of windows, and initialize the gc storage vector
  startpoints=seq(1,length(sequence)-1,step);
  vectGC=vector(length=length(startpoints), mode="numeric");

  #for each window, calculate and print the gc content
  for (i in 1:(length(startpoints)))
  {
    #determine size of current window
    n=startpoints[i];
    lastsize = length(sequence[n:length(sequence)]);

    #if current window is too small, remove the remaining vector entries
    if (lastsize < size) {
```

```
        startpoints=startpoints[1:(i-1)]
        vectGC=vectGC[1:(i-1)]
        break; #this will end a loop!
    } else { #proceedas normal
        gc=GC(sequence[startpoints[i]:(startpoints[i]+size)])
        vectGC[i]=gc;
        print(gc)
    }
}
#plot GC by start location
plot(startpoints, vectGC, type ="b");
}
```

7. Answers to Labs

7.1 Lab 1

2.1: How would you output the length of the sequence in dengueseql? Store the output as seqlength.
`seqlength=length(dengueseql);`

2.2: Why is the [] after the function? `table(dengueseql)` is a list - think of it as a named vector. So the `["g"]` grabs the "g" entry in the table.

2.3: How would you manually calculate GC content given the above information? Can you do it using variables instead of the raw numbers? Store the value as GCcontent.

`GCcontent=(table(dengueseql)["g"]+table(dengueseql)["c"])/length(dengueseql);`

2.4: Can you find a function in the `seqinr` package that would do this for you? Where do you look?
`GC(dengueseql);`

2.5: Use the help options to count the two letter words in the gene. Save the output as Count2words.
`Count2words = count(dengueseql, 2);`

2.6: Do the following:

- Output the number of occurrences of each 1-nucleotide word.
- Calculate the frequency of g (define as fG) and c (define as fC).
- Output the number of occurrences of each 2-nucleotide word.
- Calculate the frequency of gc (define as fGC).
- Calculate rho for "gc" and save as pGC. Repeat this for rho for "cg" and save as pCG.

```
a. count(dengueseq, 1);
b. count(dengueseq, 1)["g"]; count(dengueseq, 1)["c"];
c. count(dengueseq, 2);
d. count(dengueseq, 2)["gc"];
e. pGC=count(dengueseq,2)["gc"] / ( count(dengueseq, 1)["g"] * count(dengueseq, 1)["c"] );
pGC=count(dengueseq,2)["cg"] / ( count(dengueseq, 1)["g"] * count(dengueseq, 1)["c"] );
```

2.7: Can you find a function that would calculate this for you (do it by hand first to practice R syntax)?
`rho()`;

2.8: Using the command `zscore`, are the "gc" or "cg" words significantly over- or underrepresented?

You should get the following results using the "base" model:

```
rho(gc) = 0.8651367
zscore(gc) = -4.2314011
rho(cg) = 0.4516013
zscore(cg) = -17.2062694
```

For rho, the null hypothesis is 1, with anything less than 1 is underrepresented and anything greater than 1 is overrepresented. So, both of these dinucleotide "words" are underrepresented.

Zscore tells you how many standard deviations away from average the score is. So "cg" is far more underrepresented than "gc".

7.2 Lab 2

4.1: How would you grab this dataset (similar to how we grabbed `wrld_map`)?

```
data(mtcars);
```

4.2: What data type do you think dataset is?

```
dataframe
```

4.3: How would you subset the data to remove `vs` and `am`?

```
mtcars=mtcars[, c(1:7,10:11)];
```

4.4: Now that all of the data is the same type (numerical), how would you make this a matrix called `mtcars_matrix`?

```
mtcars_matrix=as.matrix(mtcars);
```

4.5: What package is this part of? How do you know?

```
stats
```

4.6: How would you find some options for this function? What does `center` do? What does `scale` do?

```
?prcomp
```

`center`:
 a logical value indicating whether the variables should be shifted to be zero centered. Alternately, a vector of length equal the number of columns of `x` can be supplied. The value is passed to `scale`.

Scale.:

a logical value indicating whether the variables should be scaled to have unit variance before the analysis takes place. The default is FALSE for consistency with S, but in general scaling is advisable. Alternatively, a vector of length equal the number of columns of x can be supplied. The value is passed to scale.

4.7: What kind of object is mtcars.pca?

List of 5 variables

4.8: How would you check if ggbiplot is installed? How would you install it and enable it? Installing from CRAN won't work – what should you do next?

CRAN won't work – what should you do next?

Google it! Ggbiplot in R 3.4 works `␣`.

```
library(devtools);
install_github("vqv/ggbiplot");
library(ggbiplot);
```

4.9: How would you add the row names of mtcars to be labels to the ggbiplot?

```
ggbiplot(mtcars.pca, labels=rownames(mtcars));
```

4.10: How would you get ggbiplot to use these groups and form ellipses around them?

```
ggbiplot(mtcars.pca, ellipse=TRUE, labels=rownames(mtcars), groups=mtcars.country);
```

4.11: Can you find it?

```
ggbiplot(mtcars.pca, ellipse=TRUE, choices=c(3,4), labels=rownames(mtcars), groups=mtcars.country);
```

4.12: How do we refer to mtcars.pca's attribute, center?

```
mtcars.pca$center;
```

4.13: How could you tell which options you have in dist()?

```
?dist();
```

4.14: How could you tell which distance options you have in vegdist()? Let's use the vegdist() function, and produce a Euclidean distance matrix of our car data.

```
?vegdist();
dist_matrix=vegdist(mtcars);
```

4.15: Can you figure out how to do this from the help section? Call your new variable vpcoa `vpcoa=wcmdscale(dist_matrix, eig=TRUE)`

4.16: Take a crack at this yourself! Remake the graph, and add standard error ellipses.

```
plot(vpcoa);
points(vpcoa, display="sites", pch=20, col=c("blue", "green", "red")[df$mtcars.country]);
legend(0, -0.05, legend=c("Japan", "US", "Europe"), col=c("green", "red", "blue"), pch=20);
ordilabel(vpcoa, labels=rownames(mtcars));
```

```
ordiellipse(vpcoa, groups=df$mtcars.country, display="sites", kind="se", conf=0.99, label=FALSE,
col="green", draw="polygon", alpha=200, show.groups = c("Japan"), border=FALSE);
ordiellipse(vpcoa, groups=df$mtcars.country, display="sites", kind="se", conf=0.99, label=FALSE,
col="red", draw="polygon", alpha=200, show.groups = c("US"), border=FALSE);
ordiellipse(vpcoa, groups=df$mtcars.country, display="sites", kind="se", conf=0.99, label=FALSE,
col="blue", draw="polygon", alpha=200, show.groups = c("Europe"), border=FALSE);
```

4.17: Try proving this to yourself by relotting the PCoA using `cmdscale()` rather than `wcmdscale()`.

```
ord = cmdscale(dist_matrix, eig = TRUE);
plot(ord$points);
```

7.3 Lab 3

6.1: Calculate the rest of the windows manually to get practice using ranges of vectors. How do you know when to stop? Do we know a function to identify the length of a sequence?

```
start end
1 1790
1790 3579
3579 5368
5368 7157
7157 8946
8946 10736
```

You should stop at the end of the sequence - defined by `length(dengueseq)`!

6.2: Add code in the #CODE section to **print out the sequence and window size** (size of chunk being analyzed).

```
gcByRange = function(sequence, size)
{
  print(sequence);
  print(size);
}
```

6.3: Armed with that information complete the following (replace the ?s):

```
gcByRange = function(sequence, size)
{
  #for each window
  startpoints=seq(1,length(sequence)-1,size);
  print(startpoints);

  #subset sequence vector into correct range

  #print gc content of that "window"
}
gcByRange(dengueseq, 1789);
```

6.4: Complete the following (replace the ?s):

```
gcByRange = function(sequence, size)
{
  #define start of windows
  startpoints=seq( 1 , length(sequence)-1 , size );
  #print(startpoints);

  #for each window
  for ( i in 1:length(startpoints)) {

    #subset sequence vector into correct range, and grab GC
    gc = GC( sequence[startpoints[i] : (startpoints[i]+size) ] );
    #then print gc content of that "window"
    print(gc);
  }
}
```

6.5: Now, we want these to be numbers, so is there a parameter in the vector function that will allow us to define what scalar the vector is composed of? Complete the code(replace the ?)

```
vectGC = vector(length = length(startpoints), mode="numeric");
```

6.6: If you print vectGC, you should see a set of 0's - default place holders for numbers. Let's back this vector build into our function. Complete the code(replace the ?):

```
gcByRange = function(sequence, size)
{
  #define start of windows
  startpoints=seq( 1 , length(sequence)-1 , size );
  #print(startpoints);

  #determine start of last window size
  n=startpoints[length(startpoints)];

  #determine length of last window
  lastsize=length(sequence[n:length(sequence)]);

  #if last segment is too small, remove the last entry from the vector
  if (lastsize<size) {
    startpoints=startpoints[1:(length(startpoints)-1)];
  }
  vectGC= vector(length = length(startpoints), mode="numeric");

  #for each window, calculate and print the gc content
  for (i in 1:(length(startpoints)))
  {
    gc = GC( sequence[startpoints[i] : (startpoints[i]+size) ] );
```

```

        print(gc);
        #save GC
    }

    plot GC versus genomic position
}

```

6.7: Next, how do we save a variable (gc) into the vector at a specific point (slot "i" in the vector)? Add it to your function!

```
vectGC[i]=gc;
```

```

gcByRange = function(sequence, size)
{
    #define start of windows
    startpoints=seq( 1 , length(sequence)-1 , size );
    #print(startpoints);

    #determine start of last window size
    n=startpoints[length(startpoints)];

    #determine length of last window
    lastsize=length(sequence[n:length(sequence)]);

    #if last segment is too small, remove the last entry from the vector
    if (lastsize<size) {
        startpoints=startpoints[1:(length(startpoints)-1)];
    }
    vectGC= vector(length = length(startpoints), mode="numeric");

    #for each window, calculate and print the gc content
    for (i in 1:(length(startpoints)))
    {
        gc = GC( sequence[startpoints[i] : (startpoints[i]+size) ]);
        print(gc);
        #save GC
        vectGC[i]=gc;
    }

    plot GC versus genomic position
}

```

6.8: See if you can figure this out and add it to your function!

```

gcByRange = function(sequence, size)
{
    #define start of windows

```

```
startpoints=seq( 1 , length(sequence)-1 , size );
#print(startpoints);

#determine start of last window size
n=startpoints[length(startpoints)];

#determine length of last window
lastsize=length(sequence[n:length(sequence)]);

#if last segment is too small, remove the last entry from the vector
if (lastsize<size) {
  startpoints=startpoints[1:(length(startpoints)-1)];
}
vectGC= vector(length = length(startpoints, mode="numeric"));

#for each window, calculate and print the gc content
for (i in 1:(length(startpoints)))
{
  gc = GC( sequence[startpoints[i] : (startpoints[i]+size) ]);
  print(gc);
  #save GC
  vectGC[i]=gc;
}

#plot GC versus genomic position
plot(startpoints, vectGC, type="b");
}
```