

Example Rmarkdown Notebook

Adam Lauretig

September 19, 2018

Matrix Algebra and Functions

There are five basic data structures in R: vectors, matrices, arrays, lists, and data.frames. We'll be going through each of these here, but if you want an in depth exploration of these I'd recommend Norman Matloff's *The Art of R Programming: A Tour of Statistical Software Design*.

Matrix basics

Up to this point, we've primarily *talked* about vectors. We've encountered other data types, but haven't used them. Vectors have length, but no width (they can only represent one variable at a time). Matrices are just collections of vectors (exactly like you learned in math camp). We can combine them by column using `cbind`, or by row, using `rbind`. We then access elements of matrix by `matrix[row, column]`.

```
vap <- voting.age.population <- c(3481823, 496387, 4582842, 2120139, 26955438, 3617942, 2673154)
```

```
total.votes <- tv <- c(NA, 238307, 1553032, 780409, 8899059, 1586105, 1162391, 258053, 122356)
```

```
m1 <- cbind(vap, tv) # Combined by column
```

```
m2 <- rbind(vap, tv) # combined by row
```

```
m2[1,2] # first row, second column
```

```
##      vap
```

```
## 496387
```

```
m1[,1] # the ith column
```

```
## [1] 3481823 496387 4582842 2120139 26955438 3617942 2673154
## [8] 652189 472143 14085749 6915512 995937 1073799 9600372
## [15] 4732010 2265860 2068253 3213141 3188765 1033632 4242214
## [22] 4997677 7620982 3908159 2139918 4426278 731365 1321923
## [29] 1870315 1012033 6598368 1452962 14838076 6752018 494923
## [36] 8697456 2697855 2850525 9612380 824854 3303593 594599
## [43] 4636679 17038979 1797941 487900 5841335 4876661 1421717
## [50] 4257230 392344
```

```
m1[1:5,1:2] # a submatrix
```

```
##      vap      tv
## [1,] 3481823    NA
## [2,] 496387 238307
## [3,] 4582842 1553032
## [4,] 2120139 780409
## [5,] 26955438 8899059
```

```
m2[1,1:10]
```

```
## [1] 3481823 496387 4582842 2120139 26955438 3617942 2673154
## [8] 652189 472143 14085749
```

```
m2[1:2, 1:10]

##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
## vap 3481823 496387 4582842 2120139 26955438 3617942 2673154 652189 472143
## tv      NA 238307 1553032 780409 8899059 1586105 1162391 258053 122356
##      [,10]
## vap 14085749
## tv 4884544
```

```
m2[, 1:10] # same as previous line since there are only two rows.
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
## vap 3481823 496387 4582842 2120139 26955438 3617942 2673154 652189 472143
## tv      NA 238307 1553032 780409 8899059 1586105 1162391 258053 122356
##      [,10]
## vap 14085749
## tv 4884544
```

```
class(m2)
```

```
## [1] "matrix"
```

However, we can also create matrices directly, we don't need to create vectors first:

```
#Another way to specify a matrix
matrix(1:10, nrow = 5)
```

```
##      [,1] [,2]
## [1,] 1    6
## [2,] 2    7
## [3,] 3    8
## [4,] 4    9
## [5,] 5   10
```

```
matrix(1:10, ncol = 2) #the same
```

```
##      [,1] [,2]
## [1,] 1    6
## [2,] 2    7
## [3,] 3    8
## [4,] 4    9
## [5,] 5   10
```

```
matrix(1:10, nrow = 5, ncol = 2) # the same
```

```
##      [,1] [,2]
## [1,] 1    6
## [2,] 2    7
## [3,] 3    8
## [4,] 4    9
## [5,] 5   10
```

```
matrix(1:10, nrow = 5, byrow = TRUE) ## not the same
```

```
##      [,1] [,2]
## [1,] 1    2
## [2,] 3    4
## [3,] 5    6
## [4,] 7    8
```

```
## [5,]      9     10
```

By default, R will fill each column of a matrix, and then move to the next one. If you specify `byrow = TRUE`, however, R will fill each row, and then move onto the next one.

Arrays and attributes

Arrays are a more general way to store data. Where a matrix can only have 2 dimensions (rows and columns), arrays can have an arbitrary number of dimensions, but this *will* increase the amount of memory they consume.

Let's examine a cube of dimensions $3 \times 4 \times 2$. One way of thinking of this is two 3×4 matrices stacked on top of each other:

```
a <- array(1:24, dim = c(3, 4, 2))
a
```

```
## , , 1
##
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
##
## , , 2
##
##      [,1] [,2] [,3] [,4]
## [1,]   13   16   19   22
## [2,]   14   17   20   23
## [3,]   15   18   21   24
```

Since this array has three dimensions, there are now three indices we can use to access the array:

```
a[, , 1]
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
```

```
a[, 1, ]
```

```
##      [,1] [,2]
## [1,]    1   13
## [2,]    2   14
## [3,]    3   15
```

```
a[1, , ]
```

```
##      [,1] [,2]
## [1,]    1   13
## [2,]    4   16
## [3,]    7   19
## [4,]   10   22
```

```
a[1, 1, ]
```

```
## [1]  1 13
```

```
a[, 1, 1]
```

```
## [1]  1 2 3
```

```
a[1, 1, 1]
```

```
## [1]  1
```

Notice that the 'dim' is assigned. This is an "attribute" of the array; attributes are some piece of data associated with the structure that isn't the data itself, and are used to make working with these data easier.

```
dim(a)
```

```
## [1] 3 4 2
```

```
attributes(a)
```

```
## $dim
```

```
## [1] 3 4 2
```

```
str(a)
```

```
## int [1:3, 1:4, 1:2] 1 2 3 4 5 6 7 8 9 10 ...
```

Matrices also have this attribute (`dim`), and also have an attribute `dimnames()`, which are strings (technically lists of strings, but we'll get to that in a minute), which allow you to label your observations.

```
dim(m1) # number of rows, number of columns
```

```
## [1] 51 2
```

```
attributes(m1) # there is another attribute here -- the columns have names
```

```
## $dim
```

```
## [1] 51 2
```

```
##
```

```
## $dimnames
```

```
## $dimnames[[1]]
```

```
## NULL
```

```
##
```

```
## $dimnames[[2]]
```

```
## [1] "vap" "tv"
```

```
dimnames(m1) # we can either assign or get the dimnames attribute
```

```
## [[1]]
```

```
## NULL
```

```
##
```

```
## [[2]]
```

```
## [1] "vap" "tv"
```

```
# The first part is the rownames (which we didn't assign)
```

```
dimnames(m2) # here the columns have no names
```

```
## [[1]]
```

```
## [1] "vap" "tv"
```

```
##
```

```
## [[2]]
```

```
## NULL
```

```
dimnames(m1)[[2]][1] <- "Dracula"
```

```
head(m1) # We have re-named the first column to have the name "Dracula"
```

```
##      Dracula      tv
```

```
## [1,] 3481823      NA
```

```
## [2,]  496387 238307
```

```
## [3,] 4582842 1553032
```

```
## [4,] 2120139  780409
```

```
## [5,] 26955438 8899059
```

```
## [6,] 3617942 1586105
```

```
dimnames(m1)[[2]][1]<-"vap" # all of this bracketing is because this is a list ... what's a  
head(m1)
```

```
##           vap           tv  
## [1,] 3481823          NA  
## [2,] 496387   238307  
## [3,] 4582842 1553032  
## [4,] 2120139  780409  
## [5,] 26955438 8899059  
## [6,] 3617942 1586105
```

R is flexible, and there are multiple ways to access dimnames:

```
# Another way to do this  
colnames(m1)
```

```
## [1] "vap" "tv"
```

```
# How would we rename the first column?  
colnames(m2)
```

```
## NULL
```

```
rownames(m1)
```

```
## NULL
```

```
rownames(m2)
```

```
## [1] "vap" "tv"
```

Lists

One downside to matrices and vectors is that every element in them must be the same type (all numerics, or all integers, or all character vectors). Lists offer a way around this restriction, they can combine multiple data types. Lists are a very flexible way to store data, and are maybe the most common data structure you'll encounter: many functions produce lists.

```
list.a <- list(m1, vap, 3) # m1 is a matrix, vap is a vector, 3 is an integer
list.a
```

```
## [[1]]
##           vap           tv
## [1,]  3481823          NA
## [2,]   496387   238307
## [3,]  4582842 1553032
## [4,]  2120139   780409
## [5,] 26955438 8899059
## [6,]   3617942 1586105
## [7,]  2673154 1162391
## [8,]    652189   258053
## [9,]    472143   122356
## [10,] 14085749 4884544
## [11,]  6915512 2143845
## [12,]   995937   348988
## [13,]  1073799   458927
## [14,]  9600372 3586292
## [15,]  4732010 1719351
## [16,]  2265860 1071509
## [17,]  2068253   864083
## [18,]  3213141 1370062
## [19,]  3188765   954896
## [20,]  1033632          NA
## [21,]  4242214 1809237
## [22,]  4997677 2243835
## [23,]  7620982 3852008
## [24,]  3908159 2217552
## [25,]  2139918          NA
## [26,]  4426278 2178278
## [27,]   731365   411061
## [28,]  1321923   610499
## [29,]  1870315   586274
## [30,]  1012033   418550
## [31,]  6598368 2315643
## [32,]  1452962   568597
## [33,] 14838076 4703830
## [34,]  6752018 2036451
## [35,]   494923   220479
## [36,]  8697456 4184072
## [37,]  2697855          NA
## [38,]  2850525 1399650
## [39,]  9612380          NA
## [40,]   824854   392882
## [41,]  3303593 1117311
## [42,]   594599   341105
## [43,]  4636679 1868363
```

```
## [44,] 17038979      NA
## [45,] 1797941  582561
## [46,]  487900  263025
## [47,]  5841335 2398589
## [48,]  4876661 2085074
## [49,]  1421717  473014
## [50,]  4257230 2183155
## [51,]   392344  196217
##
## [[2]]
## [1] 3481823  496387  4582842  2120139 26955438  3617942  2673154
## [8]  652189  472143 14085749  6915512  995937  1073799  9600372
## [15] 4732010  2265860  2068253  3213141  3188765  1033632  4242214
## [22] 4997677  7620982  3908159  2139918  4426278  731365  1321923
## [29] 1870315  1012033  6598368  1452962 14838076  6752018  494923
## [36] 8697456  2697855  2850525  9612380  824854  3303593  594599
## [43] 4636679 17038979  1797941  487900  5841335  4876661  1421717
## [50] 4257230   392344
##
## [[3]]
## [1] 3
```

We can make all sorts of lists, and can even create lists containing other lists!

```
vector1 <- c(1,2,3)
gospels <- c("matthew","mark","luke", "john")
my.matrix <- matrix(c(1:20), nrow=4)
my.data <- data.frame(cbind(vap, tv))
my.crazy.list <- list(vector1, gospels, my.matrix, TRUE, list.a)
my.crazy.list # we can combine anything we want -- we can even include other lists in our
```

```
## [[1]]
## [1] 1 2 3
##
## [[2]]
## [1] "matthew" "mark"      "luke"      "john"
##
## [[3]]
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    5    9   13   17
## [2,]    2    6   10   14   18
## [3,]    3    7   11   15   19
## [4,]    4    8   12   16   20
##
## [[4]]
## [1] TRUE
##
## [[5]]
## [[5]][[1]]
##      vap      tv
## [1,] 3481823      NA
## [2,]  496387  238307
## [3,] 4582842 1553032
## [4,] 2120139  780409
## [5,] 26955438 8899059
```



```

## [6,] 3617942 1586105
## [7,] 2673154 1162391
## [8,] 652189 258053
## [9,] 472143 122356
## [10,] 14085749 4884544
## [11,] 6915512 2143845
## [12,] 995937 348988
## [13,] 1073799 458927
## [14,] 9600372 3586292
## [15,] 4732010 1719351
## [16,] 2265860 1071509
## [17,] 2068253 864083
## [18,] 3213141 1370062
## [19,] 3188765 954896
## [20,] 1033632 NA
## [21,] 4242214 1809237
## [22,] 4997677 2243835
## [23,] 7620982 3852008
## [24,] 3908159 2217552
## [25,] 2139918 NA
## [26,] 4426278 2178278
## [27,] 731365 411061
## [28,] 1321923 610499
## [29,] 1870315 586274
## [30,] 1012033 418550
## [31,] 6598368 2315643
## [32,] 1452962 568597
## [33,] 14838076 4703830
## [34,] 6752018 2036451
## [35,] 494923 220479
## [36,] 8697456 4184072
## [37,] 2697855 NA
## [38,] 2850525 1399650
## [39,] 9612380 NA
## [40,] 824854 392882
## [41,] 3303593 1117311
## [42,] 594599 341105
## [43,] 4636679 1868363
## [44,] 17038979 NA
## [45,] 1797941 582561
## [46,] 487900 263025
## [47,] 5841335 2398589
## [48,] 4876661 2085074
## [49,] 1421717 473014
## [50,] 4257230 2183155
## [51,] 392344 196217
##
## [[5]][[2]]
## [1] 3481823 496387 4582842 2120139 26955438 3617942 2673154
## [8] 652189 472143 14085749 6915512 995937 1073799 9600372
## [15] 4732010 2265860 2068253 3213141 3188765 1033632 4242214
## [22] 4997677 7620982 3908159 2139918 4426278 731365 1321923
## [29] 1870315 1012033 6598368 1452962 14838076 6752018 494923
## [36] 8697456 2697855 2850525 9612380 824854 3303593 594599

```

```
## [43] 4636679 17038979 1797941 487900 5841335 4876661 1421717
## [50] 4257230 392344
##
## [[5]][[3]]
## [1] 3
```

What if we want to access the attributes of our list?

```
str(my.crazy.list) # the str() function is useful for looking at the basic components
```

```
## List of 5
## $ : num [1:3] 1 2 3
## $ : chr [1:4] "matthew" "mark" "luke" "john"
## $ : int [1:4, 1:5] 1 2 3 4 5 6 7 8 9 10 ...
## $ : logi TRUE
## $ :List of 3
## ..$ : num [1:51, 1:2] 3481823 496387 4582842 2120139 26955438 ...
## .. ..- attr(*, "dimnames")=List of 2
## .. .. ..$ : NULL
## .. .. ..$ : chr [1:2] "vap" "tv"
## ..$ : num [1:51] 3481823 496387 4582842 2120139 26955438 ...
## ..$ : num 3
```

```
# of any complicated object like this
#str() will work with most types of objects
```

```
attributes(my.crazy.list) # lists has attributes, but we haven't set them
```

```
## NULL
```

```
length(my.crazy.list) # this reports the number of major sub-elements in the list
```

```
## [1] 5
```

```
dim(my.crazy.list) # this won't work for complicated lists
```

```
## NULL
```

```
names(my.crazy.list) <- c("one", "two", "three", "four", "five")
str(my.crazy.list) # now each part of the list has a name attribute
```

```
## List of 5
## $ one : num [1:3] 1 2 3
## $ two : chr [1:4] "matthew" "mark" "luke" "john"
## $ three: int [1:4, 1:5] 1 2 3 4 5 6 7 8 9 10 ...
## $ four : logi TRUE
## $ five :List of 3
## ..$ : num [1:51, 1:2] 3481823 496387 4582842 2120139 26955438 ...
## .. ..- attr(*, "dimnames")=List of 2
## .. .. ..$ : NULL
## .. .. ..$ : chr [1:2] "vap" "tv"
## ..$ : num [1:51] 3481823 496387 4582842 2120139 26955438 ...
## ..$ : num 3
```

```
my.crazy.list
```

```
## $one
## [1] 1 2 3
##
```

```

## $two
## [1] "matthew" "mark"      "luke"      "john"
##
## $three
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    5    9   13   17
## [2,]    2    6   10   14   18
## [3,]    3    7   11   15   19
## [4,]    4    8   12   16   20
##
## $four
## [1] TRUE
##
## $five
## $five[[1]]
##      vap      tv
## [1,] 3481823    NA
## [2,] 496387   238307
## [3,] 4582842 1553032
## [4,] 2120139  780409
## [5,] 26955438 8899059
## [6,] 3617942 1586105
## [7,] 2673154 1162391
## [8,] 652189  258053
## [9,] 472143  122356
## [10,] 14085749 4884544
## [11,] 6915512 2143845
## [12,] 995937  348988
## [13,] 1073799 458927
## [14,] 9600372 3586292
## [15,] 4732010 1719351
## [16,] 2265860 1071509
## [17,] 2068253  864083
## [18,] 3213141 1370062
## [19,] 3188765  954896
## [20,] 1033632    NA
## [21,] 4242214 1809237
## [22,] 4997677 2243835
## [23,] 7620982 3852008
## [24,] 3908159 2217552
## [25,] 2139918    NA
## [26,] 4426278 2178278
## [27,] 731365  411061
## [28,] 1321923  610499
## [29,] 1870315  586274
## [30,] 1012033  418550
## [31,] 6598368 2315643
## [32,] 1452962  568597
## [33,] 14838076 4703830
## [34,] 6752018 2036451
## [35,] 494923  220479
## [36,] 8697456 4184072
## [37,] 2697855    NA
## [38,] 2850525 1399650

```

```
## [39,] 9612380 NA
## [40,] 824854 392882
## [41,] 3303593 1117311
## [42,] 594599 341105
## [43,] 4636679 1868363
## [44,] 17038979 NA
## [45,] 1797941 582561
## [46,] 487900 263025
## [47,] 5841335 2398589
## [48,] 4876661 2085074
## [49,] 1421717 473014
## [50,] 4257230 2183155
## [51,] 392344 196217
##
## $five[[2]]
## [1] 3481823 496387 4582842 2120139 26955438 3617942 2673154
## [8] 652189 472143 14085749 6915512 995937 1073799 9600372
## [15] 4732010 2265860 2068253 3213141 3188765 1033632 4242214
## [22] 4997677 7620982 3908159 2139918 4426278 731365 1321923
## [29] 1870315 1012033 6598368 1452962 14838076 6752018 494923
## [36] 8697456 2697855 2850525 9612380 824854 3303593 594599
## [43] 4636679 17038979 1797941 487900 5841335 4876661 1421717
## [50] 4257230 392344
##
## $five[[3]]
## [1] 3
```

But this can be quite convoluted. Instead, when we create our list, we can give each element a name:

```
my.crazy.list <- list(one=vector1,two=gospels, three=my.matrix, four=TRUE, five=list.a)
str(my.crazy.list)
```

```
## List of 5
## $ one : num [1:3] 1 2 3
## $ two : chr [1:4] "matthew" "mark" "luke" "john"
## $ three: int [1:4, 1:5] 1 2 3 4 5 6 7 8 9 10 ...
## $ four : logi TRUE
## $ five :List of 3
## ..$ : num [1:51, 1:2] 3481823 496387 4582842 2120139 26955438 ...
## .. ..- attr(*, "dimnames")=List of 2
## .. .. ..$ : NULL
## .. .. ..$ : chr [1:2] "vap" "tv"
## ..$ : num [1:51] 3481823 496387 4582842 2120139 26955438 ...
## ..$ : num 3
```

```
names(my.crazy.list)
```

```
## [1] "one" "two" "three" "four" "five"
```

Manipulating lists is similar to other manipulations in R, the new one is using double brackets `[[]]` to access an element of a list.

```
# there are several ways to access/add to/subtract from a list
my.crazy.list[[1]]
```

```
## [1] 1 2 3
```

```

my.crazy.list$one

## [1] 1 2 3
my.crazy.list[1]

## $one
## [1] 1 2 3
my.crazy.list["one"]

## $one
## [1] 1 2 3
my.crazy.list$dracula <- "dracula"
my.crazy.list # now we have added another element

## $one
## [1] 1 2 3
##
## $two
## [1] "matthew" "mark"      "luke"      "john"
##
## $three
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    5    9   13   17
## [2,]    2    6   10   14   18
## [3,]    3    7   11   15   19
## [4,]    4    8   12   16   20
##
## $four
## [1] TRUE
##
## $five
## $five[[1]]
##           vap      tv
## [1,]  3481823      NA
## [2,]  496387   238307
## [3,]  4582842 1553032
## [4,]  2120139  780409
## [5,] 26955438 8899059
## [6,]  3617942 1586105
## [7,]  2673154 1162391
## [8,]   652189  258053
## [9,]   472143  122356
## [10,] 14085749 4884544
## [11,]  6915512 2143845
## [12,]   995937  348988
## [13,]  1073799  458927
## [14,]  9600372 3586292
## [15,]  4732010 1719351
## [16,]  2265860 1071509
## [17,]  2068253  864083
## [18,]  3213141 1370062
## [19,]  3188765  954896
## [20,]  1033632      NA

```

```
## [21,] 4242214 1809237
## [22,] 4997677 2243835
## [23,] 7620982 3852008
## [24,] 3908159 2217552
## [25,] 2139918      NA
## [26,] 4426278 2178278
## [27,] 731365  411061
## [28,] 1321923 610499
## [29,] 1870315 586274
## [30,] 1012033 418550
## [31,] 6598368 2315643
## [32,] 1452962 568597
## [33,] 14838076 4703830
## [34,] 6752018 2036451
## [35,] 494923  220479
## [36,] 8697456 4184072
## [37,] 2697855      NA
## [38,] 2850525 1399650
## [39,] 9612380      NA
## [40,] 824854  392882
## [41,] 3303593 1117311
## [42,] 594599  341105
## [43,] 4636679 1868363
## [44,] 17038979      NA
## [45,] 1797941 582561
## [46,] 487900  263025
## [47,] 5841335 2398589
## [48,] 4876661 2085074
## [49,] 1421717 473014
## [50,] 4257230 2183155
## [51,] 392344  196217
##
## $five[[2]]
## [1] 3481823 496387 4582842 2120139 26955438 3617942 2673154
## [8] 652189 472143 14085749 6915512 995937 1073799 9600372
## [15] 4732010 2265860 2068253 3213141 3188765 1033632 4242214
## [22] 4997677 7620982 3908159 2139918 4426278 731365 1321923
## [29] 1870315 1012033 6598368 1452962 14838076 6752018 494923
## [36] 8697456 2697855 2850525 9612380 824854 3303593 594599
## [43] 4636679 17038979 1797941 487900 5841335 4876661 1421717
## [50] 4257230 392344
##
## $five[[3]]
## [1] 3
##
##
## $dracula
## [1] "dracula"

# We can repeat this accessing method
my.crazy.list[[3]][1,] # first row of my.matrix

## [1] 1 5 9 13 17
```

```
my.matrix[1,] #the same
```

```
## [1] 1 5 9 13 17
```

However, you cannot do math on lists directly (note that this is set to `eval = FALSE`, since if we ran it, it throws an error and the document doesn't compile):

```
my.crazy.list + 2 # not so much
```

```
my.crazy.list[[3]] + 2
```

Matrix operations

Matrices are the workhorses of computational statistics. And just like there are special ways of manipulating matrices in mathematics, there are special operators in R for working with them. Unless you tell R explicitly, however, it *will* operate element-wise on a matrix.

```
# A couple of matrices
H3 <- matrix(c(1, 1/2, 1/3, 1/2, 1/3, 1/4, 1/3, 1/4, 1/5), nrow=3)
H3

##           [,1]      [,2]      [,3]
## [1,] 1.0000000 0.5000000 0.3333333
## [2,] 0.5000000 0.3333333 0.2500000
## [3,] 0.3333333 0.2500000 0.2000000

1/cbind(seq(1,3), seq(2, 4), seq(3,5)) # most basic function continue to be "element wise"

##           [,1]      [,2]      [,3]
## [1,] 1.0000000 0.5000000 0.3333333
## [2,] 0.5000000 0.3333333 0.2500000
## [3,] 0.3333333 0.2500000 0.2000000

H3+1

##           [,1]      [,2]      [,3]
## [1,] 2.0000000 1.5000000 1.3333333
## [2,] 1.5000000 1.3333333 1.2500000
## [3,] 1.3333333 1.2500000 1.2000000

H3*2

##           [,1]      [,2]      [,3]
## [1,] 2.0000000 1.0000000 0.6666667
## [2,] 1.0000000 0.6666667 0.5000000
## [3,] 0.6666667 0.5000000 0.4000000

H3^2

##           [,1]      [,2]      [,3]
## [1,] 1.0000000 0.2500000 0.1111111
## [2,] 0.2500000 0.1111111 0.0625000
## [3,] 0.1111111 0.0625000 0.0400000

mean(H3) # others will treat the matrix as a vector no matter what

## [1] 0.4111111

rowSums(H3) # others work on matrices in particular ways (more on this later)

## [1] 1.8333333 1.0833333 0.7833333

colSums(H3)

## [1] 1.8333333 1.0833333 0.7833333

rowMeans(H3)

## [1] 0.6111111 0.3611111 0.2611111

colMeans(H3)

## [1] 0.6111111 0.3611111 0.2611111
```



```
# logicals too
H3==1

##      [,1] [,2] [,3]
## [1,]  TRUE FALSE FALSE
## [2,] FALSE FALSE FALSE
## [3,] FALSE FALSE FALSE

H3 == c(1,2,3) #what's going on here?
```

```
##      [,1] [,2] [,3]
## [1,]  TRUE FALSE FALSE
## [2,] FALSE FALSE FALSE
## [3,] FALSE FALSE FALSE

H3 == H3
```

```
##      [,1] [,2] [,3]
## [1,]  TRUE TRUE TRUE
## [2,]  TRUE TRUE TRUE
## [3,]  TRUE TRUE TRUE
```

Some functions are exact translations of math:

```
# Some work like they do in the math books
det(H3) # the determinant -- hard for you ... easy in R
```

```
## [1] 0.000462963
```

```
diag(H3) # get the diagonal elements of a matrix
```

```
## [1] 1.0000000 0.3333333 0.2000000
```

```
diag(1, nrow=3) # make a 3by3 identity matrix
```

```
##      [,1] [,2] [,3]
## [1,]    1    0    0
## [2,]    0    1    0
## [3,]    0    0    1
```

```
t(H3) # matrix transpose
```

```
##      [,1] [,2] [,3]
## [1,] 1.0000000 0.5000000 0.3333333
## [2,] 0.5000000 0.3333333 0.2500000
## [3,] 0.3333333 0.2500000 0.2000000
```

To access only the lower triangle of a matrix, use the `lower.tri()` function, and some indexing:

```
Hnew<-H3
Hnew[lower.tri(H3, diag=TRUE)] # extract the lower triangular elements of H3
```

```
## [1] 1.0000000 0.5000000 0.3333333 0.3333333 0.2500000 0.2000000
```

```
# why can we just use lower.tri()?
```

```
lower.tri(H3, diag=TRUE)
```

```
##      [,1] [,2] [,3]
## [1,]  TRUE FALSE FALSE
## [2,]  TRUE  TRUE FALSE
## [3,]  TRUE  TRUE  TRUE
```

To get the trace, we'll need to write a function:

```
trace<- function(data) {  
  (sum(diag(data)))  
}  
trace(H3)
```

```
## [1] 1.533333
```

To multiply matrices we use `%*%`, the matrix multiplication operator:

```
t(H3) %*% H3
```

```
##           [,1]      [,2]      [,3]  
## [1,] 1.361111 0.750000 0.525000  
## [2,] 0.750000 0.423611 0.300000  
## [3,] 0.525000 0.300000 0.213611
```

```
c(1,2,3) %*% c(1,2,3) # dot product
```

```
##           [,1]  
## [1,]      14
```

```
matrix(c(1,2,3), ncol=1) %*% c(1,2,3) # outer product
```

```
##           [,1] [,2] [,3]  
## [1,]      1   2   3  
## [2,]      2   4   6  
## [3,]      3   6   9
```

To invert a matrix, we use the `solve()` command, which can also be used to solve a linear system:

```
solve(H3)
```

```
##           [,1] [,2] [,3]  
## [1,]      9  -36   30  
## [2,]  -36   92 -180  
## [3,]   30 -180   180
```

```
invH3<-solve(H3)  
H3 %*% invH3 ## close enough?
```

```
##           [,1]      [,2] [,3]  
## [1,] 1.000000e+00 0.000000e+00 0  
## [2,] 8.881784e-16 1.000000e+00 0  
## [3,] 0.000000e+00 -7.105427e-15 1
```

```
# solving a linear system:
```

```
b<-c(1,2,3)
```

```
solve(H3, b)
```

```
## [1]    27 -192   210
```

Data analysis - Roll your own linear model

You want to know: is the presidential vote share positively related to GDP growth? One way to test this is with a linear regression model:

```
library(foreign)
vote <- read.dta("votegdp.dta") # since this is in the same folder as the markdown document

lm(vote~q2gdp, data=vote) # sure is

##
## Call:
## lm(formula = vote ~ q2gdp, data = vote)
##
## Coefficients:
## (Intercept)          q2gdp
##      49.144         0.765

coefficients <- lm(vote~q2gdp, data=vote)$coefficients
```

But what is this doing? First, adding a constant column to the data, for our y intercept (order matters here)! Then

```
constant <- rep(1, nrow(vote))
X <- cbind(constant, vote$q2gdp)
```

Then, removing the rows which have missing values:

```
X <- X[!is.na(vote$vote),]
# na.omit(vote) # an alternative way to get rid of NA's in advance
class(X) # it's a matrix (not a data frame) so we can use our solution

## [1] "matrix"

is.matrix(X) # alternative approach

## [1] TRUE
```

Creating our Y variable:

```
Y <- cbind(vote$vote[!is.na(vote$vote)])
```

and finally, solving $(X'X)^{-1}(X'y)$ (the OLS equation), and checking whether our results are the same as `lm()`:

```
B <- solve((t(X)%*%X))%*%(t(X)%*%Y)
B[1]-coefficients[1] # about zero

## (Intercept)
## -7.105427e-15

B[2] - coefficients[2] # about zero

## q2gdp
## 2.220446e-16
```

Flow control and functions

You will find that for many tasks your R scripts will start to get *long*. Complex tasks will start turning into complex code.

TIPS:

1. If you find yourself copying and pasting more than 2-3 times – think about writing a loop or a function instead.
2. If you ever spend more than 20 minutes manually reshaping, editing, copying/pasting data that is already encoded and on a computer — then somewhere a fairy is killed.
3. Some combination of the basic skills in today's lessons can be used to solve most of these kinds of problems – although it may take time work out how.

`if(){} , else(){} , ifelse(){}`

Let's start with `if()`, which works as follows: `if(condition){commands}`. The input in the parenthesis needs to be something that returns a logical, and you can put anything in the braces you want:

```
if(TRUE){print("I got here")} #
```

```
## [1] "I got here"
```

```
if(FALSE){print("I also got here")} #
```

You can combine `if()` with an `else()` command. Everything in the `else{}` braces will be executed when the condition is false

```
x <- 3
if(x>2){
  print("X is larger than 2")
} else { # notice that these are on the same line
  print("X is 2 or smaller")
}
```

```
## [1] "X is larger than 2"
```

```
x <- (-3)
if(x>2){
  print("X is larger than 2")
} else { # notice that these are on the same line
  print("X is 2 or smaller")
}
```

```
## [1] "X is 2 or smaller"
```

However, `if()` and `else()` do not play nicely with vectors, so instead we'll use `ifelse()`, a *vectorized* version of these two commands:

```
# This will throw an error
if (c(1,2)>2){
  print("This won't work")
}
```

try the `ifelse()` command instead

```
x <- c(0,2)
ifelse(x > 1, "yes", "no") # but you can only put in values in here, not a bunch of instructions
```

```
## [1] "no" "yes"
```

```
# beware though ... if your outputs are vectors it will work element-wise
yes <- c("yes1", "yes2")
no <- c("no1", "no2")
ifelse(x>1, yes, no)
```

```
## [1] "no1" "yes2"
```

Note that the braces are not technically necessary if you have only a one line command, but not using them is like writing without punctuation. Someone can figure out what you're saying, but it makes life harder than it has to be.

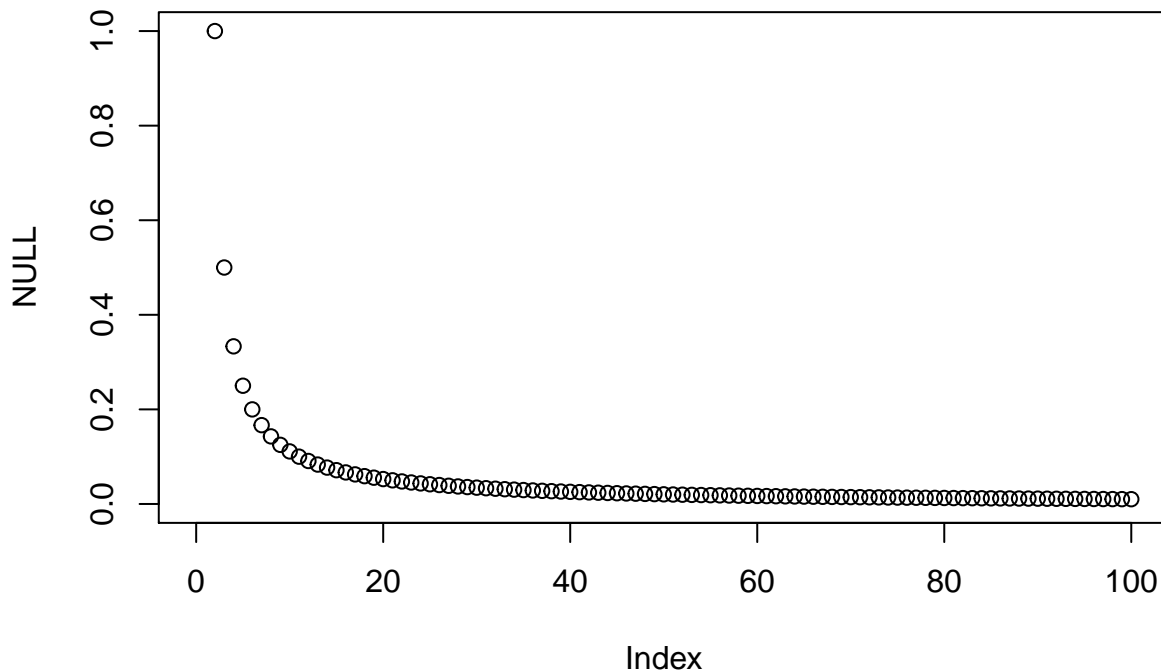
```
x<-3
if (x > 2) y <- 2*x else y <- 3*x
y
```

[1] 6

repeat, break

repeat{ Execute these commands over and over again } You had better include a break command in there, or you are not going to be happy. The 'break' command will stop the repeat (it will also work for the for() and while() loops below).

```
plot(NULL, xlim=c(0,100), ylim=c(0,1)) # make a blank plot with the limits set by those vec
x <- 0
repeat {
  y <- 1/x
  x <- x+1
  points(x, y)
  if (x == 100) { break }
}
```



Example: make the Fibonacci series less than 300:

```
# Example: make the Fibonacci series less than 300
Fib1 <- 1
Fib2 <- 1
Fibonacci <- c(0, Fib2)
repeat{
  Fibonacci <- c(Fibonacci, Fib2)
  oldFib2 <- Fib2
  Fib2 <- Fib1 + Fib2
  Fib1 <- oldFib2
  if (Fib2 > 300) {break}
}
Fibonacci
```

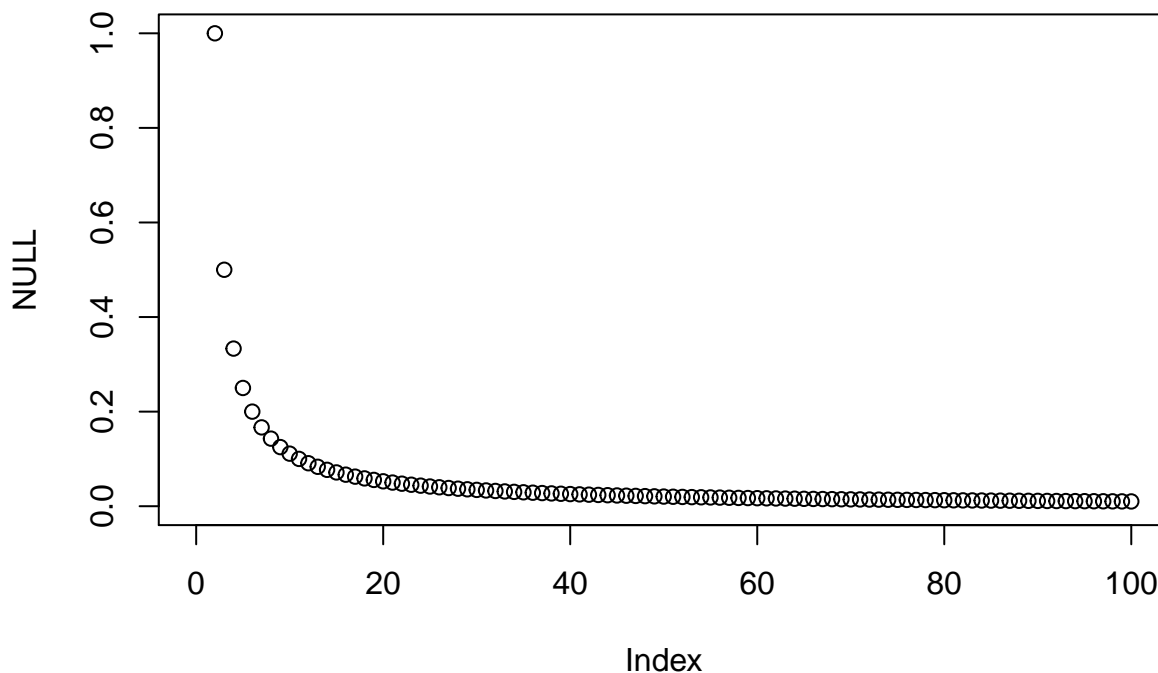
```
## [1] 0 1 1 2 3 5 8 13 21 34 55 89 144 233
```

****Tip:**** Save your files before running any repeat. *Better yet... **don't** use repeat.* You should write for-loops that can stop on their own.

while()

A `while()` loop is just a repeat, where the `if (condition) {break}` is specified at the top. Similar to the above:

```
plot(NULL, xlim=c(0,100), ylim=c(0,1))
x <- 0
while(x < 100) {
  y <- 1/x
  x <- x+1
  points(x,y)
}
```



```
# Example 2
Fib1 <- 1
Fib2 <- 1
Fibonacci <- c(0, Fib2)
while(Fib2 <= 300){
  Fibonacci <- c(Fibonacci, Fib2)
  oldFib2 <- Fib2
  Fib2 <- Fib1 + Fib2
  Fib1 <- oldFib2
}
Fibonacci
```

```
## [1] 0 1 1 2 3 5 8 13 21 34 55 89 144 233
```

Example, using the Fearon and Laitin data

Let's say we want to calculate the average number of civil-wars onsets for each level of democratization (polity2)

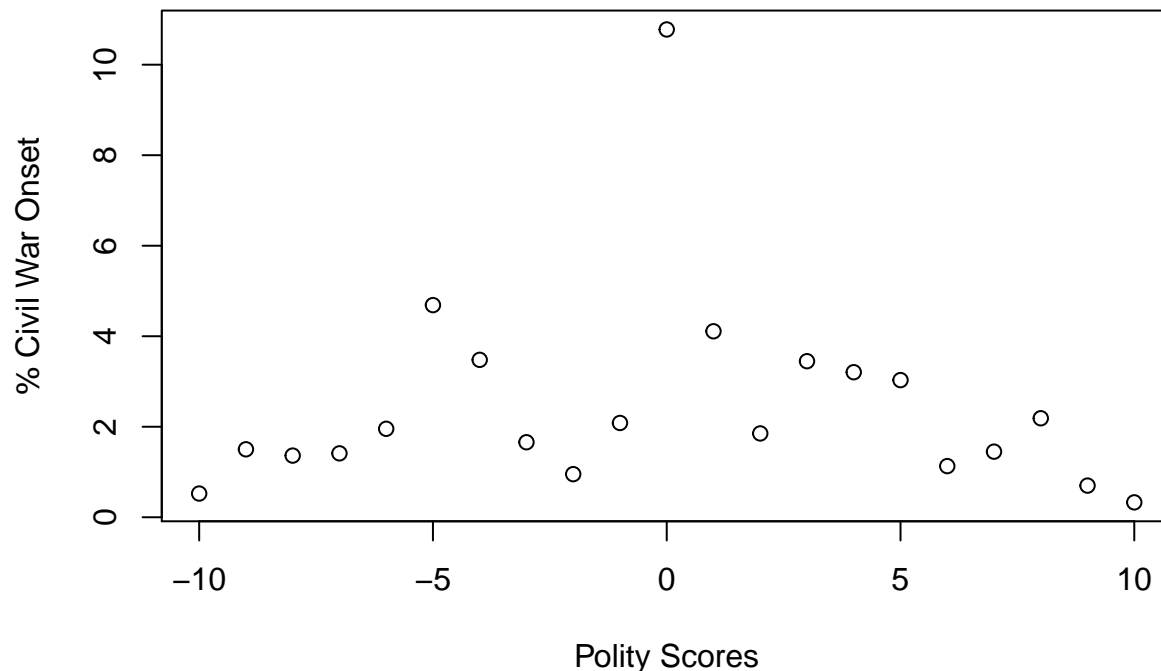

```

library(foreign)
civilw <- read.dta("repdata.dta")
democracy <- min(civilw$polity2, na.rm=T)
output_vector <- NULL
index <- 1
while(democracy <= max(civilw$polity2, na.rm=T)){
  output_vector[index] <- mean(civilw$onset[civilw$polity2 == democracy], na.rm=T)
  democracy <- democracy + 1 # DON'T FORGET THIS LINE OR YOU WILL BE IN AN ENDLESS LOOP
  index <- index + 1 # this line is needed to index forward
}
output_vector*100 # civil wars start this percent of the time at each level of democracy

## [1] 0.5235602 1.5015015 1.3623978 1.4115899 1.9543974 4.6875000
## [7] 3.4782609 1.6574586 0.9523810 2.0833333 10.7784431 4.1095890
## [13] 1.8518519 3.4482759 3.2051282 3.0303030 1.1299435 1.4492754
## [19] 2.1875000 0.6993007 0.3289474

plot(c(min(civilw$polity2, na.rm=T):max(civilw$polity2, na.rm=T)), output_vector*100,
     xlab="Polity Scores", ylab="% Civil War Onset")

```



In general, this is a pretty hack-y way to calculate this, and you're better off using something like `data.table`, as discussed in lab 2.

for-loops, next ()

The while loop still requires a lot of attention to indexing. Also ... in many instances all we want to do is increment by 1. So programmers put together a “for loop.”

for (name in vector) {execute these commands on each value of the vector} whatever you put into the name slot will become a “local” variable

```
for (monkey in c("spider", "howler", "wurst")) {print(monkey)}
```

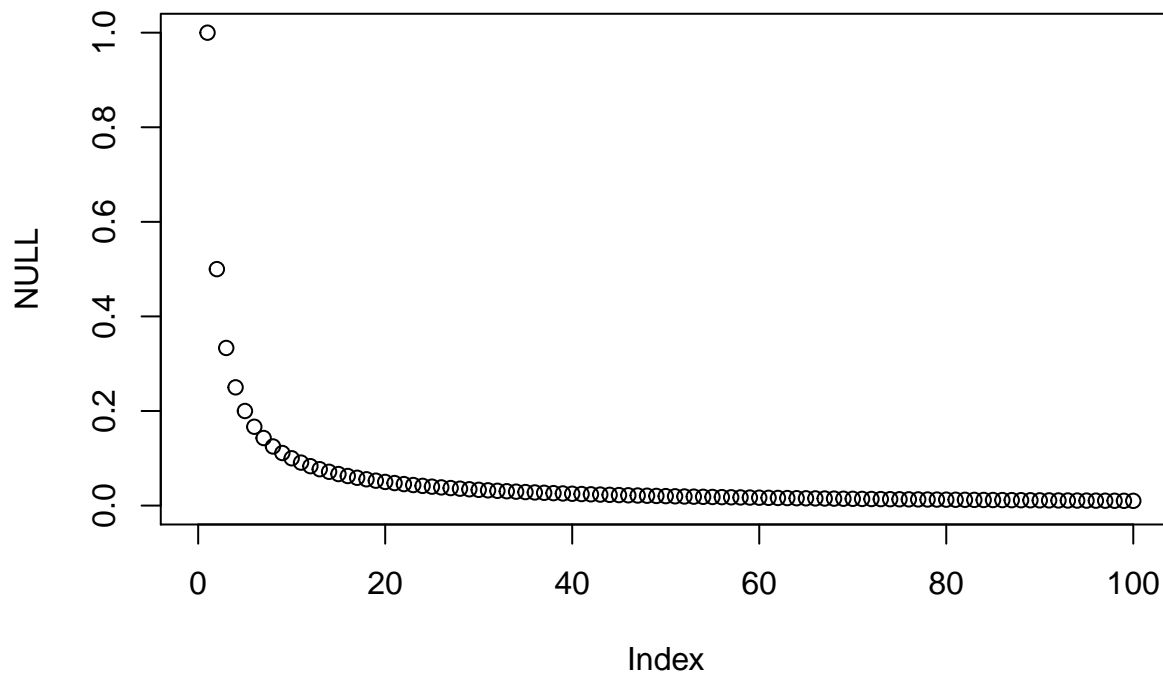
```
## [1] "spider"  
## [1] "howler"  
## [1] "wurst"
```

```
# or more commonly  
for (i in 1:20) {  
  print(i)  
}
```

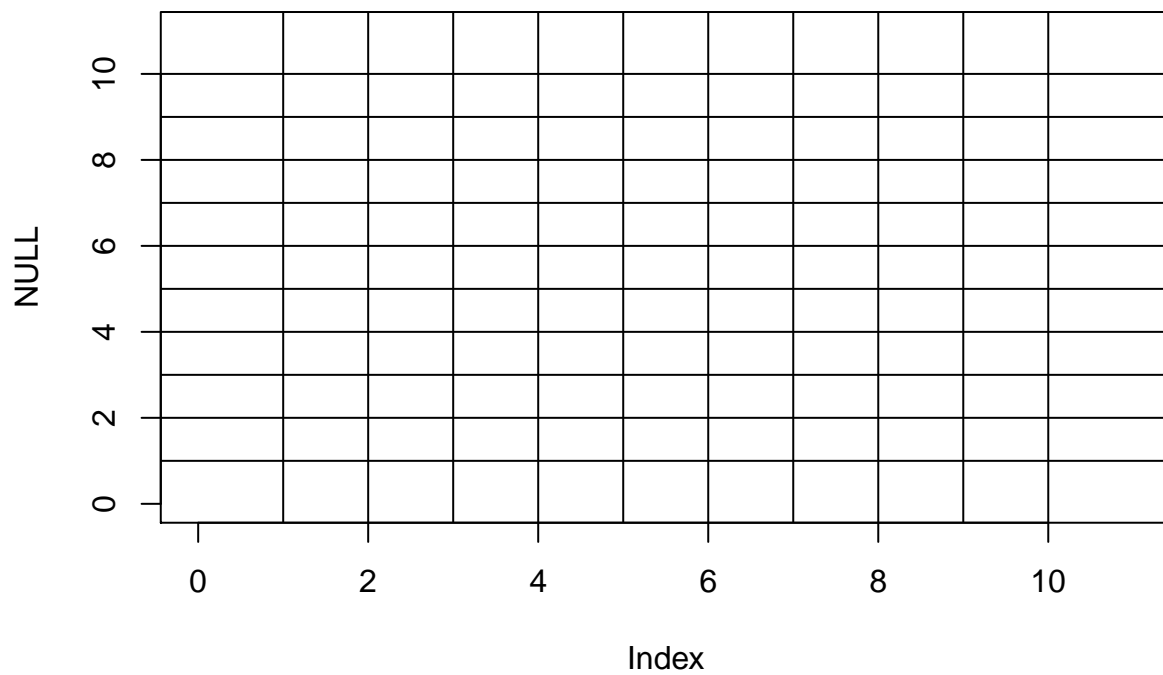
```
## [1] 1  
## [1] 2  
## [1] 3  
## [1] 4  
## [1] 5  
## [1] 6  
## [1] 7  
## [1] 8  
## [1] 9  
## [1] 10  
## [1] 11  
## [1] 12  
## [1] 13  
## [1] 14  
## [1] 15  
## [1] 16  
## [1] 17  
## [1] 18  
## [1] 19  
## [1] 20
```

Example of for-loops

```
plot(NULL, xlim=c(0,100), ylim=c(0,1))  
for (i in 0:100) {points(i, 1/i)}
```

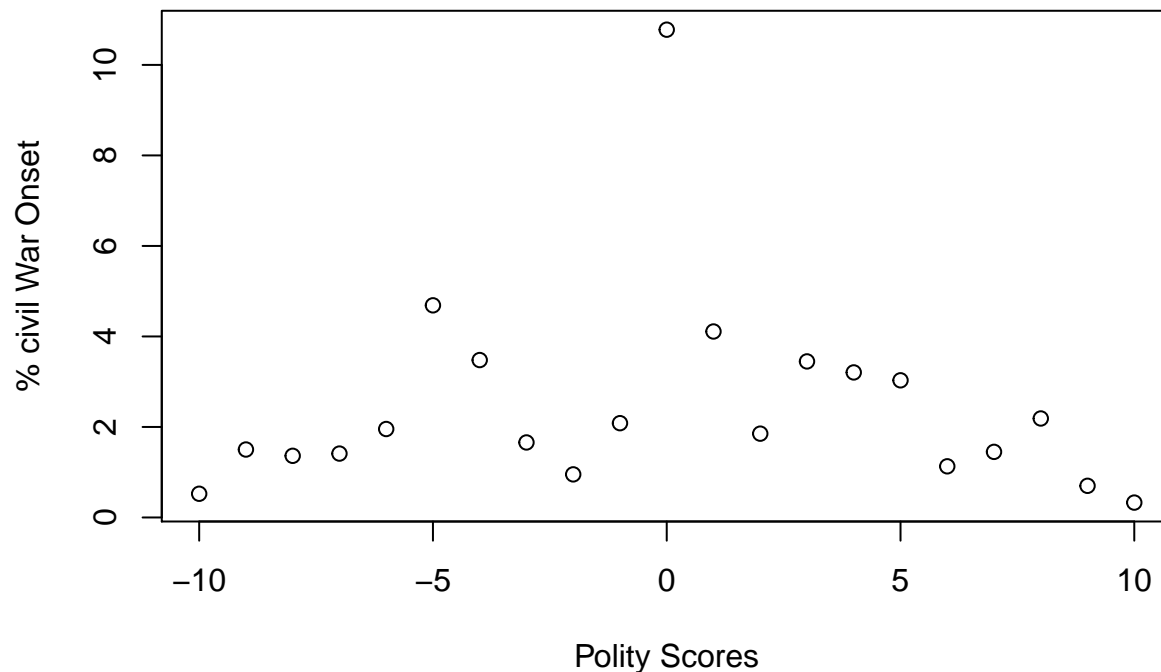


```
plot(NULL, xlim=c(0,11), ylim=c(0,11))
for(i in 1:10){
  abline(v=i)
  abline(h=i)
}
```



```
output <- NULL
for(i in 1:20){
  output[i] <- 0+i
}
```

```
library(foreign)
civilw <- read.dta("repdata.dta")
output_vector <- NULL # Still need to instantiate the variable or you cannot index it
democracy <- sort(unique(civilw$polity2))
for (i in min(democracy):max(democracy)){
  output_vector <- c(output_vector, mean(civilw$onset[civilw$polity2 == democracy[i+1]]), i)
}
plot(c(min(democracy):max(democracy)), output_vector,
     xlab="Polity Scores", ylab="% civil War Onset")
```



Sometimes you might not want to execute the commands for every element in the vector use the next command to skip (you can also use the break)

```
some.odds <- NULL
for (i in 1:200){
  if (i%%2 != 0) {
    some.odds<-c(some.odds, i)
  } else { next }
}
some.odds
```

```
## [1] 1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33
## [18] 35 37 39 41 43 45 47 49 51 53 55 57 59 61 63 65 67
## [35] 69 71 73 75 77 79 81 83 85 87 89 91 93 95 97 99 101
## [52] 103 105 107 109 111 113 115 117 119 121 123 125 127 129 131 133 135
## [69] 137 139 141 143 145 147 149 151 153 155 157 159 161 163 165 167 169
## [86] 171 173 175 177 179 181 183 185 187 189 191 193 195 197 199
```