Lab 1. NUMERICS, ARITHMETIC, ASSIGNMENT AND VECTORS

R for Basic Math

All common arithmetic operations and mathematical functionality are ready to use at the console prompt. You can perform addition, subtraction, multiplication, and division with the symbols +, -, *, and /, respectively. You can create exponents (also referred to as powers or indices) using ^, and you control the order of the calculations in a single command using parentheses, ().

1.1 Arithmetic

In R, standard mathematical rules apply throughout and follow the usual left-to-right order of operations: parentheses, exponents, multiplication, division, addition, subtraction. Here's an example in the console:

```
> 2+3
[1] 5
> 14/6
[1] 2.333333
> 14/6+5
[1] 7.333333
> 14/(6+5)
[1] 1.272727
> 3^2
[1] 9
> 2^3
[1] 8
```

You can find the square root of any non-negative number with the **sqrt** function. You simply provide the desired number to **x** as shown here:

```
> sqrt(x=9)
[1] 3
> sqrt(9)
[1] 3
```

When using R, you'll often find that you need to translate a complicated arithmetic formula into code for evaluation (for example, when replicating a calculation from a textbook or research paper). The next examples provide a mathematically expressed calculation, followed by its execution in R:

$$10^2 + \frac{3 \times 60}{8} - 3$$

```
R> 10^2+3*60/8-3
[1] 119.5
```

$$\frac{5^3 \times (6-2)}{61-3+4}$$

```
R> 5^3*(6-2)/(61-3+4)
[1] 8.064516
```

$$2^{2+1} - 4 + 64^{-2^{2.25-\frac{1}{4}}}$$

```
R> 2^(2+1)-4+64^((-2)^(2.25-1/4))
[1] 16777220
```

$$\left(\frac{0.44 \times (1-0.44)}{34}\right)^{\frac{1}{2}}$$

```
R> (0.44*(1-0.44)/34)^(1/2)
[1] 0.08512966
```

You'll often see or read about researchers performing a **log** transformation on certain data. This refers to rescaling numbers according to the logarithm.

```
> log(x=243,base=3)
[1] 5
> log(243,3)
[1] 5
```

There's a particular kind of log transformation often used in mathematics called the natural log, which fixes the base at a special mathematical number – Euler's number. This is conventionally written as **e** and is approximately equal to 2.718.

```
> exp(3)
[1] 20.08554
> log(20.08554)
[1] 3
```

EXERCISE 1.1

a. Using R, verify that

$$\frac{6a + 42}{3^{4.2-3.62}} = 29.50556$$

when $a = 2.3$.

b. Which of the following squares negative 4 and adds 2 to the result?
   i. `(-4)^2+2`
   ii. `-4^2+2`
   iii. `(-4)^(2+2)`
   iv. `-4^(2+2)`

c. Using R, how would you calculate the square root of half of the average of the numbers 25.2, 15, 16.44, 15.3, and 18.6?

d. Find $\log_e 0.3$.

e. Compute the exponential transform of your answer to (d).

1.2 Vectors

Creating a Vector

The vector is the essential building block for handling multiple items in R. In a numeric sense, you can think of a vector as a collection of observations or measurements concerning a single variable, for example, the heights of 50 people or the number of coffees you drink daily. More complicated data structures may consist of several vectors. The function for creating a vector is the single letter **c**, with the desired entries in parentheses separated by commas.

```
> myvec<-c(1,2,3)
> myvec
[1] 1 2 3
```

Vector entries can be calculations or previously stored items (including vectors themselves).

```
> foo<-32.1
> myvec2<-c(3,-3,2,3.45,1e+03,64^0.5,2+(3-1.1)/9.44,foo)
> myvec2
[1]    3.000000   -3.000000    2.000000    3.450000 1000.000000    8.000000
[7]    2.201271   32.100000
```

This code created a new vector assigned to the object **myvec2**. Some of the entries are defined as arithmetic expressions, and it's the result of the expression that's stored in the vector. The last element, foo, is an existing numeric object defined as 32.1. Let's look at another example.

```
> myvec3<-c(myvec,myvec2)
> myvec3
 [1]    1.000000    2.000000    3.000000    3.000000   -3.000000    2.000000
 [7]    3.450000 1000.000000    8.000000    2.201271   32.100000
```

This code creates and stores yet another vector, myvec3, which contains the entries of **myvec** and **myvec2** appended together in that order.

Sequences, Repetition, Sorting and Lengths

Here I'll discuss some common and useful functions associated with R vectors: **seq**, **rep**, **sort** and **length**. Let's create an equally spaced sequence of increasing or decreasing numeric values. This is something you'll need often, for example when programming loops or when plotting data points. The easiest way to create such a sequence, with numeric values separated by intervals of 1, is to use the colon operator.

```
> 3:27
 [1]  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27
```

The example 3:27 should be read as "from 3 to 27 (by 1)." The result is a numeric vector just as if you had listed each number manually in parentheses with **c**. As always, you can also provide either a previously stored value or a (strictly parenthesized) calculation when using the colon operator:

```
> foo<-5.3
> bar<-foo:(-47+1.5)
> bar
 [1]   5.3   4.3   3.3   2.3   1.3   0.3  -0.7  -1.7  -2.7  -3.7  -4.7  -5.7  -6.7
[14]  -7.7  -8.7  -9.7 -10.7 -11.7 -12.7 -13.7 -14.7 -15.7 -16.7 -17.7 -18.7 -19.7
[27] -20.7 -21.7 -22.7 -23.7 -24.7 -25.7 -26.7 -27.7 -28.7 -29.7 -30.7 -31.7 -32.7
[40] -33.7 -34.7 -35.7 -36.7 -37.7 -38.7 -39.7 -40.7 -41.7 -42.7 -43.7 -44.7
```

Sequences with seq

You can also use the **seq** command, which allows for more flexible creations of sequences. This ready-to-use function takes in a **from** value, a **to** value, and a **by** value, and it returns the corresponding sequence as a numeric vector.

```
> seq(from=3,to=27,by=3)
[1]  3  6  9 12 15 18 21 24 27

> seq(from=3,to=27,length.out=40)
 [1]  3.000000  3.615385  4.230769  4.846154  5.461538  6.076923  6.692308  7.307692
 [9]  7.923077  8.538462  9.153846  9.769231 10.384615 11.000000 11.615385 12.230769
[17] 12.846154 13.461538 14.076923 14.692308 15.307692 15.923077 16.538462 17.153846
[25] 17.769231 18.384615 19.000000 19.615385 20.230769 20.846154 21.461538 22.076923
[33] 22.692308 23.307692 23.923077 24.538462 25.153846 25.769231 26.384615 27.000000
```

By setting **length.out** to 40, you make the program print exactly 40 evenly spaced numbers from 3 to 27.

For decreasing sequences, the use of **by** must be negative. Here's an example:

```
> foo <- 5.3
> myseq <- seq(from=foo,to=(-47+1.5),by=-2.4)
> myseq
 [1]   5.3   2.9   0.5  -1.9  -4.3  -6.7  -9.1 -11.5 -13.9 -16.3 -18.7 -21.1 -23.5
[14] -25.9 -28.3 -30.7 -33.1 -35.5 -37.9 -40.3 -42.7 -45.1
```

Repetition with rep

Sequences are extremely useful, but sometimes you may want simply to repeat a certain value. You do this using **rep.**

```
> rep(x=1,times=4)
[1] 1 1 1 1
> rep(x=c(3,62,8.3),times=3)
[1]  3.0 62.0  8.3  3.0 62.0  8.3  3.0 62.0  8.3
> rep(x=c(3,62,8.3),each=2)
[1]  3.0  3.0 62.0 62.0  8.3  8.3
> rep(x=c(3,62,8.3),times=3,each=2)
 [1]  3.0  3.0 62.0 62.0  8.3  8.3  3.0  3.0 62.0 62.0  8.3  8.3  3.0  3.0 62.0 62.0
[17]  8.3  8.3
```

The **rep** function is given a single value or a vector of values as its argument **x**, as well as a value for the arguments **times** and **each**. The value for **times** provides the number of times to repeat **x**, and **each** provides the number of times to repeat each element of **x**.

As with **seq**, you can include the result of **rep** in a vector of the same data type, as shown in the following example:

```
> foo <- 4
> c(3,8.3,rep(x=32,times=foo),seq(from=-2,to=1,length.out=foo+1))
 [1]  3.00  8.30 32.00 32.00 32.00 32.00 -2.00 -1.25 -0.50  0.25  1.00
```

Sorting with sort

Sorting a vector in increasing or decreasing order of its elements is another simple operation that crops up in everyday tasks. The conveniently named sort function does just that.

```
> sort(x=c(2.5,-1,-10,3.44),decreasing=FALSE)
[1] -10.00  -1.00   2.50   3.44
> sort(x=c(2.5,-1,-10,3.44),decreasing=TRUE)
[1]   3.44   2.50  -1.00 -10.00
> foo <- seq(from=4.3,to=5.5,length.out=8)
> foo
[1] 4.300000 4.471429 4.642857 4.814286 4.985714 5.157143 5.328571 5.500000
> bar <- sort(x=foo,decreasing=TRUE)
> bar
[1] 5.500000 5.328571 5.157143 4.985714 4.814286 4.642857 4.471429 4.300000
> sort(x=c(foo,bar),decreasing=FALSE)
 [1] 4.300000 4.300000 4.471429 4.471429 4.642857 4.642857 4.814286 4.814286 4.985714
[10] 4.985714 5.157143 5.157143 5.328571 5.328571 5.500000 5.500000
```

Finding a Vector Length with length

I'll round off this section with the length function, which determines how many entries exist in a vector given as the argument **x**.

```
> length(x=c(3,2,8,1))
[1] 4
> length(x=5:13)
[1] 9
> foo <- 4
> bar <- c(3,8.3,rep(x=32,times=foo),seq(from=-2,to=1,length.out=foo+1))
> length(bar)
[1] 11
```

EXERCISE 1.2

a. Create and store a sequence of values from 5 to −11 that progresses in steps of 0.3.

b. Overwrite the object from (a) using the same sequence with the order reversed.

c. Repeat the vector `c(-1,3,-5,7,-9)` twice, with each element repeated 10 times, and store the result. Display the result sorted from largest to smallest.

d. Create and store a vector that contains, in any configuration, the following:
   i. A sequence of integers from 6 to 12 (inclusive)
   ii. A threefold repetition of the value 5.3
   iii. The number −3
   iv. A sequence of nine values starting at 102 and ending at the number that is the total length of the vector created in (c)

e. Confirm that the length of the vector created in (d) is 20.

1.3 Subsetting and Element Extraction

In all the results you have seen printed to the console screen so far, you may have noticed a curious feature. Immediately to the left of the output there is a square-bracketed [1]. When the output is a long vector that spans the width of the console and wraps onto the following line, another square bracketed number appears to the left of the new line. These numbers represent the index of the entry directly to the right. Quite simply, the index corresponds to the position of a value within a vector, and that's precisely why the first value always has a [1] next to it (even if it's the only value and not part of a larger vector).

These indexes allow you to retrieve specific elements from a vector, which is known as **subsetting**.

```
> myvec <- c(5,-2.3,4,4,4,6,8,10,40221,-8)
> length(x=myvec)
[1] 10
> myvec[1]
[1] 5
> length(x=myvec)
[1] 10
> myvec[1]
[1] 5
> foo<-myvec[2]
> foo
[1] -2.3
> myvec[length(x=myvec)]
[1] -8
```

Because **length(myvec)** results in the final index of the vector (in this case, 10), entering this phrase in the square brackets extracts the final element, -8. Similarly, you could extract the second-to-last element by subtracting 1 from the length; let's try that, and also assign the result to a new object:

```
> myvec.len <- length(myvec)
> bar <- myvec[myvec.len-1]
> bar
[1] 40221
```

As these examples show, the index may be an arithmetic function of other numbers or previously stored values. You can assign the result to a new object in your workspace in the usual way with the <- notation. Using your knowledge of sequences, you can use the colon notation with the length of the specific vector to obtain all possible indexes for extracting a particular element in the vector:

```
> 1:myvec.len
 [1]  1  2  3  4  5  6  7  8  9 10
```

You can also delete individual elements by using **negative** versions of the indexes supplied in the square brackets. Continuing with the objects **myvec**, **foo**, **bar** and **myvec.len** as defined earlier, consider the following operations:

```
> myvec[-1]
[1]    -2.3     4.0     4.0     4.0     6.0     8.0    10.0 40221.0    -8.0
```

This line produces the contents of **myvec** without the first element. Similarly, the following code assigns to the object **baz** the contents of **myvec** without its second element:

```
> baz <- myvec[-2]
> baz
[1]     5     4     4     4     6     8    10 40221    -8
```

Again, the index in the square brackets can be the result of an appropriate calculation, like so:

```
> qux <- myvec[-(myvec.len-1)]
> qux
[1]  5.0 -2.3  4.0  4.0  4.0  6.0  8.0 10.0 -8.0
```

As with most operations in R, you are not restricted to doing things one by one. You can also subset objects using **vectors of indexes**, rather than individual indexes. Using **myvec** again from earlier, you get the following:

```
> myvec[c(1,3,5)]
[1] 5 4 4
```

Another common and convenient subsetting tool is the colon operator, which creates a sequence of indexes. Here's an example:

```
> 1:4
[1] 1 2 3 4
> foo <- myvec[1:4]
> foo
[1]  5.0 -2.3  4.0  4.0
```

The order of the returned elements depends entirely upon the index vector supplied in the square brackets. For example, using foo again, consider the order of the indexes and the resulting extractions, shown here:

```
> length(x=foo):2
[1] 4 3 2
> foo[length(foo):2]
[1]  4.0  4.0 -2.3
```

Here you extracted elements starting at the end of the vector, working backward. You can also use **rep** to repeat an index, as shown here:

```
> indexes <- c(4,rep(x=2,times=3),1,1,2,3:1)
> indexes
 [1] 4 2 2 2 1 1 2 3 2 1
> foo[indexes]
 [1]  4.0 -2.3 -2.3 -2.3  5.0  5.0 -2.3  4.0 -2.3  5.0
```

This is now something a little more general than strictly "subsetting" – by using an index vector, you can create an entirely new vector of any length consisting of some or all of the elements in the original vector. As shown earlier, this index vector can contain the desired element positions in any order and can repeat indexes.

You can also return the elements of a vector after deleting more than one element. For example, to create a vector after removing the first and third elements of foo, you can execute the following:

```
> foo[-c(1,3)]
[1] -2.3  4.0
```

Note that it is not possible to mix positive and negative indexes in a single index vector. Sometimes you'll need to overwrite certain elements in an existing vector with new values. In this situation, you first specify the elements you want to overwrite using square brackets and then use the assignment operator to assign the new values. Here's an example:

```
> bar <- c(3,2,4,4,1,2,4,1,0,0,5)
> bar
 [1] 3 2 4 4 1 2 4 1 0 0 5
> bar[1] <- 6
> bar
 [1] 6 2 4 4 1 2 4 1 0 0 5
```

This overwrites the first element of **bar**, which was originally 3, with a new value, 6. When selecting multiple elements, you can specify a single value to replace them all or enter a vector of values that's equal in length to the number of elements selected to replace them one for one. Let's try this with the same bar vector from earlier.

```
> bar[c(2,4,6)] <- c(-2,-0.5,-1)
> bar
 [1]  6.0 -2.0  4.0 -0.5  1.0 -1.0  4.0  1.0  0.0  0.0  5.0
```

Here you overwrite the second, fourth, and sixth elements with -2, -0.5, and -1, respectively; all else remains the same. By contrast, the following code overwrites elements 7 to 10 (inclusive), replacing them all with 100:

```
> bar[7:10] <- 100
> bar
 [1]   6.0  -2.0   4.0  -0.5   1.0  -1.0 100.0 100.0 100.0 100.0   5.0
```

EXERCISE 1.3

a. Create and store a vector that contains the following, in this order:
   - A sequence of length 5 from 3 to 6 (inclusive)
   - A twofold repetition of the vector `c(2,-5.1,-33)`
   - The value $\frac{7}{42} + 2$

b. Extract the first and last elements of your vector from (a), storing them as a new object.

c. Store as a third object the values returned by omitting the first and last values of your vector from (a).

d. Use only (b) and (c) to reconstruct (a).

e. Overwrite (a) with the same values sorted from smallest to largest.

f. Use the colon operator as an index vector to reverse the order of (e), and confirm this is identical to using `sort` on (e) with `decreasing=TRUE`.

g. Create a vector from (c) that repeats the third element of (c) three times, the sixth element four times, and the last element once.

h. Create a new vector as a copy of (e) by assigning (e) as is to a newly named object. Using this new copy of (e), overwrite the first, the fifth to the seventh (inclusive), and the last element with the values 99 to 95 (inclusive), respectively.

1.4 Vector-Oriented Behavior

Vectors are so useful because they allow R to carry out operations on multiple elements simultaneously with speed and efficiency. This *vectororiented*, *vectorized* or *element-wise* behavior is a key feature of the language, one that you will briefly examine here through some examples of rescaling measurements.

```
> foo <- 5.5:0.5
> foo
[1] 5.5 4.5 3.5 2.5 1.5 0.5
> foo-c(2,4,6,8,10,12)
[1]   3.5   0.5  -2.5  -5.5  -8.5 -11.5
```

The situation is made more complicated when using vectors of different lengths, which can happen in two distinct ways. The first is when the length of the longer vector can be evenly divided by the length of the shorter vector. The second is when the length of the longer vector cannot be divided by the length of the shorter vector – this is usually unintentional on the user's part. In both of these situations, R essentially attempts to replicate, or recycle, the shorter vector by as many times as needed to match the length of the longer vector, before completing the specified operation.

As an example, suppose you wanted to alternate the entries of foo shown earlier as negative and positive. You could explicitly multiply foo by c(1,-1,1,-1,1,-1), but you don't need to write out the full latter vector. Instead, you can write the following:

```
> bar <- c(1,-1)
> foo*bar
[1]  5.5 -4.5  3.5 -2.5  1.5 -0.5
```

Here bar has been applied repeatedly throughout the length of foo until completion.

```
> baz <- c(1,-1,0.5,-0.5)
> foo*baz
[1]  5.50 -4.50  1.75 -1.25  1.50 -0.50
Warning message:
In foo * baz :
  longer object length is not a multiple of shorter object length
```

Here you see that R has matched the first four elements of foo with the entirety of **baz**, but it's not able to fully repeat the vector again. The repetition has been attempted, with the first two elements of **baz** being matched with the last two of the longer foo, though not without a protest from R, which notifies the user of the unevenly divisible lengths.

You can consider single values to be vectors of length 1, so you can use a single value to repeat an operation on all the values of a vector of any length. Here's an example, using the same vector foo:

```
> qux <- 3
> foo+qux
[1] 8.5 7.5 6.5 5.5 4.5 3.5
```

Another benefit of vector-oriented behavior is that you can use vectorized functions to complete potentially laborious tasks. For example, if you want to sum or multiply all the entries in a numeric vector, you can just use a built-in function.

```
> foo
[1] 5.5 4.5 3.5 2.5 1.5 0.5
```

You can find the sum of these six elements with

```
> sum(foo)
[1] 18
```

and their product with

```
> prod(foo)
[1] 162.4219
```

Far from being just convenient, vectorized functions are faster and more efficient than an explicitly coded iterative approach like a loop. The main takeaway from these examples is that much of R's functionality is designed specifically for certain data structures, ensuring neatness of code as well as optimization of performance.

Lastly, as mentioned earlier, this vector-oriented behavior applies in the same way to overwriting multiple elements. Again using foo, examine the following:

```
> foo
[1] 5.5 4.5 3.5 2.5 1.5 0.5
> foo[c(1,3,5,6)] <- c(-99,99)
> foo
[1] -99.0   4.5  99.0   2.5 -99.0  99.0
```

You see four specific elements being overwritten by a vector of length 2, which is recycled in the same fashion you're familiar with. Again, the length of the vector of replacements must evenly divide

the number of elements being overwritten, or else a warning similar to the one shown earlier will be issued when R cannot complete a full-length recycle.

EXERCISE 1.4

a. Convert the vector `c(2,0.5,1,2,0.5,1,2,0.5,1)` to a vector of only `1`s, using a vector of length 3.

b. The conversion from a temperature measurement in degrees Fahrenheit F to Celsius C is performed using the following equation:

$$C = \frac{5}{9}(F - 32)$$

Use vector-oriented behavior in R to convert the temperatures 45, 77, 20, 19, 101, 120, and 212 in degrees Fahrenheit to degrees Celsius.

c. Use the vector `c(2,4,6)` and the vector `c(1,2)` in conjunction with `rep` and `*` to produce the vector `c(2,4,6,4,8,12)`.

d. Overwrite the middle four elements of the resulting vector from (c) with the two recycled values `-0.1` and `-100`, in that order.

1.5 MATRICES AND ARRAYS

The matrix is an important mathematical construct, and it's essential to many statistical methods. You typically describe a matrix **A** as an *m* x *n* matrix; that is, **A** will have exactly *m* rows and *n* columns.

To create a matrix in R, use the aptly named **matrix** command, providing the entries of the matrix to the data argument as a vector:

```
> A <- matrix(data=c(-3,2,893,0.17),nrow=2,ncol=2)
> A
     [,1]   [,2]
[1,]   -3 893.00
[2,]    2   0.17
```

You must make sure that the length of this vector matches exactly with the number of desired rows (**nrow**) and columns (**ncol**). You can elect not to supply **nrow** and **ncol** when calling matrix, in which case R's default behavior is to return a single-column matrix of the entries in data.
For example, **matrix(data=c(-3,2,893,0.17))** would be identical to **matrix(data=c(-3,2,893,0.17),nrow=4,ncol=1)**.

It's important to be aware of how R fills up the matrix using the entries from data. Looking at the previous example, you can see that the 2 x 2 matrix A has been filled in a *column-by-column* fashion when reading the data entries from left to right. You can control how R fills in data using the argument **byrow**, as shown in the following examples:

```
> matrix(data=c(1,2,3,4,5,6),nrow=2,ncol=3,byrow=FALSE)
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

Now, let's repeat the same line of code but set **byrow=TRUE**.

```
> matrix(data=c(1,2,3,4,5,6),nrow=2,ncol=3,byrow=TRUE)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

If you have multiple vectors of equal length, you can quickly build a matrix by binding together these vectors using the built-in R functions, **rbind** and **cbind**.

```
> rbind(1:3,4:6)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6

> cbind(c(1,4),c(2,5),c(3,6))
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

Here, you have three vectors each of length 2. You use cbind to glue together these three vectors in the order they were supplied, and each vector becomes a column of the resulting matrix.

```
> mymat <- rbind(c(1,3,4),5:3,c(100,20,90),11:13)
> mymat
     [,1] [,2] [,3]
[1,]    1    3    4
[2,]    5    4    3
[3,]  100   20   90
[4,]   11   12   13
> dim(mymat)
[1] 4 3
> nrow(mymat)
[1] 4
> ncol(mymat)
[1] 3
```

Extracting and subsetting elements from matrices in R is much like extracting elements from vectors. The only complication is that you now have an additional dimension. Element extraction still uses the square-bracket operator, but now it must be performed with both a row and a column position, given strictly in the order of [*row,column*]. Let's start by creating a 3 x 3 matrix.

```
> A <- matrix(c(0.3,4.5,55.3,91,0.1,105.5,-4.2,8.2,27.9),nrow=3,ncol=3)
> A
     [,1]  [,2] [,3]
[1,]  0.3  91.0 -4.2
[2,]  4.5   0.1  8.2
[3,] 55.3 105.5 27.9
```

To tell R to "look at the third row of A and give me the element from the second column," you execute the following:

```
> A[3,2]
[1] 105.5
```

To extract an entire row or column from a matrix, you simply specify the desired row or column number and leave the other value blank. It's important to note that *you must still include* the comma that separates the row and column numbers – this is how R distinguishes between a request for a row and a request for a column. The following returns the second column of A:

```
> A[,2]
[1]  91.0   0.1 105.5
```

The following examines the first row:

```
> A[1,]
[1]  0.3 91.0 -4.2
```

Note that whenever an extraction (or deletion, covered in a moment) results in a single value, single row, or single column, R will always return stand-alone vectors comprised of the requested values. You can also perform more complicated extractions, for example requesting whole rows or columns, or multiples rows or columns, where the result must be returned as a new matrix of the appropriate dimensions. Consider the following subsets:

```
> A[2:3,]
     [,1]  [,2] [,3]
[1,]  4.5   0.1  8.2
[2,] 55.3 105.5 27.9

> A[,c(3,1)]
     [,1] [,2]
[1,] -4.2  0.3
[2,]  8.2  4.5
[3,] 27.9 55.3

> A[c(3,1),2:3]
      [,1] [,2]
[1,] 105.5 27.9
[2,]  91.0 -4.2
```

The first command returns the second and third rows of A, and the second command returns the third and first columns of A. The last command accesses the third and first rows of A, in that order, and from those rows it returns the second and third column elements.

You can also identify the values along the diagonal of a square matrix (that is, a matrix with an equal number of rows and columns) using the **diag** command.

```
> diag(A)
[1]  0.3  0.1 27.9
```

To delete or omit elements from a matrix, you again use square brackets, but this time with negative indexes. The following provides A without its second column:

```
> A[,-2]
     [,1] [,2]
[1,]  0.3 -4.2
[2,]  4.5  8.2
[3,] 55.3 27.9
```

The following removes the first row from A and retrieves the third and second column values, in that order, from the remaining two rows:

```
> A[-1,3:2]
     [,1]  [,2]
[1,]  8.2   0.1
[2,] 27.9 105.5
```

The following produces A without its first row and second column:

```
> A[-1,-2]
     [,1] [,2]
[1,]  4.5  8.2
[2,] 55.3 27.9
```

Lastly, this deletes the first row and then deletes the second and third columns from the result:

```
> A[-1,-c(2,3)]
[1]  4.5 55.3
```

Note that this final operation leaves you with only the last two elements of the first column of A, so this result is returned as a stand-alone vector rather than a matrix.

To overwrite particular elements, or entire rows or columns, you identify the elements to be replaced and then assign the new values.

```
> B <- A
> B
     [,1]  [,2] [,3]
[1,]  0.3  91.0 -4.2
[2,]  4.5   0.1  8.2
[3,] 55.3 105.5 27.9
```

The following overwrites the second row of B with the sequence 1, 2 and 3:

```
> B[2,] <- 1:3
> B
     [,1]  [,2] [,3]
[1,]  0.3  91.0 -4.2
[2,]  1.0   2.0  3.0
[3,] 55.3 105.5 27.9
```

The following overwrites the second column elements of the first and third rows with 900:

```
> B[c(1,3),2] <- 900
> B
     [,1] [,2] [,3]
[1,]  0.3  900 -4.2
[2,]  1.0    2  3.0
[3,] 55.3  900 27.9
```

Next, you replace the third column of B with the values in the third *row* of B.

```
> B[,3] <- B[3,]
> B
     [,1] [,2]  [,3]
[1,]  0.3  900  55.3
[2,]  1.0    2 900.0
[3,] 55.3  900  27.9
```

To try R's vector recycling, let's now overwrite the first and third column elements of rows 1 and 3 (a total of four elements) with the two values -7 and 7.

```
> B[c(1,3),c(1,3)] <- c(-7,7)
> B
     [,1] [,2] [,3]
[1,]   -7  900   -7
[2,]    1    2  900
[3,]    7  900    7
```

The vector of length 2 has replaced the four elements *in a column-wise fashion*. The replacement vector c(-7,7) overwrites the elements at positions (1;1) and (3;1), in that order, and is then repeated to overwrite (1;3) and (3;3), in that order.

To highlight the role of index order on matrix element replacement, consider the following example:

```
> B[c(1,3),2:1] <- c(65,-65,88,-88)
> B
     [,1] [,2] [,3]
[1,]   88   65   -7
[2,]    1    2  900
[3,]  -88  -65    7
```

The four values in the replacement vector have overwritten the four specified elements, again in a column-wise fashion. In this case, because I specified the first and second columns in reverse order, the overwriting proceeded accordingly, filling the second column before moving to the first. Position (1;2) is matched with 65, followed by (3;2) with -65; then (1;1) becomes 88, and (3;1) becomes -88.

If you just want to replace the diagonal of a square matrix, you can avoid explicit indexes and directly overwrite the values using the **diag** command.

```
> diag(B) <- rep(x=0,times=3)
> B
     [,1] [,2] [,3]
[1,]    0   65   -7
[2,]    1    0  900
[3,]  -88  -65    0
```

EXERCISE 1.5

a. Construct and store a 4×2 matrix that's filled row-wise with the values 4.3, 3.1, 8.2, 8.2, 3.2, 0.9, 1.6, and 6.5, in that order.

b. Confirm the dimensions of the matrix from (a) are 3×2 if you remove any one row.

c. Overwrite the second column of the matrix from (a) with that same column sorted from smallest to largest.

d. What does R return if you delete the fourth row and the first column from (c)? Use `matrix` to ensure the result is a single-column matrix, rather than a vector.

e. Store the bottom four elements of (c) as a new 2×2 matrix.

f. Overwrite, in this order, the elements of (c) at positions $(4,2)$, $(1,2)$, $(4,1)$, and $(1,1)$ with $-\frac{1}{2}$ of the two values on the diagonal of (e).

1.6 Matrix Operations and Algebra

Matrix Transpose

```
> A <- rbind(c(2,5,2),c(6,1,4))
> A
     [,1] [,2] [,3]
[1,]    2    5    2
[2,]    6    1    4
> t(A)
     [,1] [,2]
[1,]    2    6
[2,]    5    1
[3,]    2    4
```

If you "transpose the transpose" of A, you'll recover the original matrix.

```
> t(t(A))
     [,1] [,2] [,3]
[1,]    2    5    2
[2,]    6    1    4
```

Scalar Multiple of a Matrix

```
> A <- rbind(c(2,5,2),c(6,1,4))
> a<-2
> a*A
     [,1] [,2] [,3]
[1,]    4   10    4
[2,]   12    2    8
```

Matrix Addition and Subtraction

```
> A <- cbind(c(2,5,2),c(6,1,4))
> A
     [,1] [,2]
[1,]    2    6
[2,]    5    1
[3,]    2    4
> B <- cbind(c(-2,3,6),c(8.1,8.2,-9.8))
> B
     [,1] [,2]
[1,]   -2  8.1
[2,]    3  8.2
[3,]    6 -9.8
> A-B
     [,1] [,2]
[1,]    4 -2.1
[2,]    2 -7.2
[3,]   -4 13.8
```

Matrix Multiplication

```
> A <- rbind(c(2,5,2),c(6,1,4))
> dim(A)
[1] 2 3
> B <- cbind(c(3,-1,1),c(-3,1,5))
> dim(B)
[1] 3 2
```

This confirms the two matrices are compatible for multiplication, so you can proceed.

```
> A%*%B
     [,1] [,2]
[1,]    3    9
[2,]   21    3
```

You can show that matrix multiplication is noncommutative using the same two matrices. Switching the order of multiplication gives you an entirely different result.

```
> B%*%A
     [,1] [,2] [,3]
[1,]  -12   12   -6
[2,]    4   -4    2
[3,]   32   10   22
```

Matrix Inversion

Some square matrices can be *inverted*. The inverse of a matrix A is denoted A^{-1}.

```
> A <- matrix(data=c(3,4,1,2),nrow=2,ncol=2)
```

```
> A
     [,1] [,2]
[1,]    3    1
[2,]    4    2
> solve(A)
     [,1] [,2]
[1,]    1 -0.5
[2,]   -2  1.5
```

You can also verify that the product of these two matrices (using matrix multiplication rules) results in the 2 x 2 identity matrix.

```
> A%*%solve(A)
     [,1] [,2]
[1,]    1    0
[2,]    0    1
```

EXERCISE 1.6

a. Calculate the following:

$$\frac{2}{7}\left(\begin{bmatrix}1 & 2\\ 2 & 4\\ 7 & 6\end{bmatrix} - \begin{bmatrix}10 & 20\\ 30 & 40\\ 50 & 60\end{bmatrix}\right)$$

b. Store these two matrices:

$$A = \begin{bmatrix}1\\ 2\\ 7\end{bmatrix} \qquad B = \begin{bmatrix}3\\ 4\\ 8\end{bmatrix}$$

Which of the following multiplications are possible? For those that are, compute the result.

i. $A \cdot B$
ii. $A^\top \cdot B$
iii. $B^\top \cdot (A \cdot A^\top)$
iv. $(A \cdot A^\top) \cdot B^\top$
v. $[(B \cdot B^\top) + (A \cdot A^\top) - 100I_3]^{-1}$

c. For

$$A = \begin{bmatrix}2 & 0 & 0 & 0\\ 0 & 3 & 0 & 0\\ 0 & 0 & 5 & 0\\ 0 & 0 & 0 & -1\end{bmatrix},$$

confirm that $A^{-1} \cdot A - I_4$ provides a 4×4 matrix of zeros.