Workshop 4: Matrices

Matrices and arrays are one of the most useful structures in programming for both mathematical models and data. R has many powerful and computationally efficient functions for manipulating matrices.

Matrices

By learning how to use vectors, you've already mostly learned to use matrices. A vector is like a $1 \times n$ matrix (i.e., a matrix with one row), but not quite identical in R. To make a matrix, you can use the `matrix` command.

```
vals <- 1:6
m <- matrix(vals, nrow=2, ncol=3)
```

Note that, by default, R fills in a matrix "by column." If you want to fill in "by row", use the `byrow` option:

```
m <- matrix(vals, nrow=2, ncol=3, byrow=TRUE)
```

Of course, matrices do not need to hold only integer values:

```
vals <- runif(6)
m <- matrix(vals, nrow=2, ncol=3)
```

Note, that if you pass the wrong number of values in to your `matrix` command, R will "recycle" them (if there are not enough) or not use them all (if there are too many). It will print a warning, but be careful! E.g.,

```
matrix(1:4, nrow=2, ncol=3)
matrix(1:8, nrow=2, ncol=3)
```

The `dim` command will tell you the dimensions of your matrix. E.g.,

```
dim(matrix(1:6, nrow=2, ncol=3))
```

To access entries in a matrix, use square brackets (as you did with vectors). However, now you must specify both the row and column. E.g.,

```
m[2,3]
```

You could also access the entire 2nd row with

```
m[2,]
```

or the entire 3rd column with

```
m[,3]
```

You could also create a subset of a bigger matrix (i.e., one that only contains some of the rows and/or columns) using:

```
m <- matrix(runif(12), nrow=3, ncol=4)
m.subset <- m[c(1,2),c(1,3)] ## smaller subset matrix
```

Important cautionary note: If your subset matrix contains only 1 of the original rows or one of the original columns, R will automatically turn the output into a vector. This can be problematic if you are expecting the resultant object to be a matrix. To prevent this, use the `drop=FALSE` option. E.g., compare the output from the below

```
m[1,]
m[1,,drop=FALSE]
```

R has converted the first into a vector. For further clarification, try:

```
dim(m[1,])
dim(m[1,,drop=FALSE])
```

I try to always use the `drop=FALSE` option when subsetting matrices (and arrays which we will learn about below), because there is no downside.

You *can* access matrix entries using a single number (i.e., without specifying row/column), e.g.,

```
m[3]
```

however, only do this if you have a good reason to. For the above example, R is essentially turning the matrix into a vector, and then extracting the the 3rd element.

You can also make matrices by binding together vectors with `cbind` ("column bind") or `rbind` ("row bind").

```
a <- c(1,2,3)
b <- c(-3,-4,-5)
cbind(a,b)
rbind(a,b)
```

1. Create some matrices of your own.
2. Try some of the commands we learned in workshop 1 from the "vectors" section (e.g., try removing only a few specified rows or columns from a larger matrix using the minus sign).

Basic matrix operations

It is straightforward to perform basic algebraic operations with matrices in R. Let's first create two square matrices:

```
M <- matrix(sample(1:9), nrow=3)
N <- matrix(sample(1:9)/10, nrow=3)
```

Matrix addition/substraction is exactly as you'd expect (e.g., `M+N` and `M-N`). However, the standard $*$ symbol does not perform matrix multiplication, but instead multiplies corresponding elements of the two matrices (an often useful function, but not try matrix multiplication). To "matrix multiply" two matrices, use `M %*% N`.

More sophisticated matrix operations (that mostly are only applicable for square matrices) are:

```
t(M) # transpose
eigen(M) # eigenvalues and eigenvectors
solve(M) # inverse matrix
diag(M) # access the diagonal elements of a matrix
M[upper.tri(M)] # access the upper triangle values
M[lower.tri(M)] # access the lower triangle values
# and many more...
```

1. Compute the "dot product" of two vectors.
2. Confirm that matrix multiplication **is not** commutative.
3. Confirm that matrix multiplication **is** associative.
4. Confirm that matrix multiplying a matrix by its inverse yields the identity matrix. You may find the command `zapsmall` useful here. This command does exactly what it says - zaps small stuff - because R is doing everything numerically, often small rounding errors are introduced (e.g., `10e-17`) and `zapsmall` will get rid of these.

Arrays

Just as a matrix is a 2D generalization of a vector, an array is a multi-dimensional generalization of a matrix. We will not use arrays much in this class. However, they are one of the most powerful data-structures in R, and useful for both modelling and data-storage. Mastering arrays is well worth the effort. Creating an array is similar to creating a matrix, but rather than specifying the row and column numbers, you specify the overall dimensions. For example, to create a $3 \times 2 \times 4$ array containing the values $1., \ldots 24$,

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

and now, because this array is three-dimensional, we would access elements using, for example

```
a[2,1,3]
```

As with matrices, you can apply many algebraic operations to arrays. While R is generally a "slow" language, it is computationally fast at algebraic operations with matrices and arrays, so exploiting this can be an efficient way to analyze a model.

1. Create your own array.
2. Try the `dim` command on your array.

Dimnames

One particularly helpful trick when working with matrices (and vectors or arrays) is to add names to your dimensions. This is kind of like adding "column names" to a spreadsheet or data-frame. For example, supposed I am using a matrix to store some measure of distance between a set of individuals (e.g., could be phylogenetic distance between a set of species). Let's make up a hypothetical matrix with such distances for a community of three species:

```
phy.mat <- matrix(0, nrow=3, ncol=3)
phy.dists <- runif(3)
phy.mat[upper.tri(phy.mat)] <- phy.dists
phy.mat[lower.tri(phy.mat)] <- phy.dists
```

Now, if I wanted to extract the distance between species 1 and species 2 (i.e., the [1,2] entry of the matrix), I could type

```
phy.mat[1,2]
```

However, suppose my species have names (e.g., 'dog', 'cat', 'mouse'). Let's add these names to our matrix:

```
sp.names <- c('dog', 'cat', 'mouse')
colnames(phy.mat) <- sp.names
rownames(phy.mat) <- sp.names
# or in one step
dimnames(phy.mat) <- list(sp.names, sp.names)
```

Now, when you view the matrix, you'll see that its much nicer to look at. The real power here lies in the fact that you can now access entries by name, rather than using row and column numbers. E.g.,

```
phy.mat['dog','cat']
```

This means you don't need to remember which row corresponds to which species! And, better yet, if later on your matrix rows or columns get re-arranged, any code that you have written to extract particular entries will still work! And, of course, the row names and column names may be different, depending on what the matrix represents.

1. Create a 5×4 matrix, and assign row names and column names to it.

2. Using these row and column names, rather than numbers, create a smaller subset matrix that contains only some of these rows/columns (e.g., in our example, we may drop the "dog" from the matrix).

Multi-variable models using matrix notation

Using matrix algebra can be a tidy way to code up a linear multi-variable model. In class, we looked at one such model (the dispersal model in lecture 3b). We came up with the following:

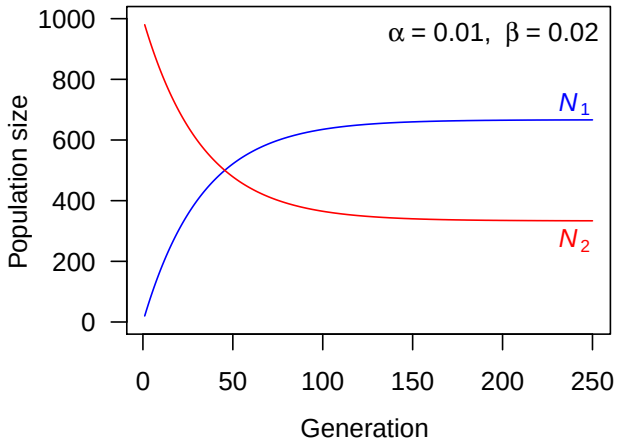
$$\begin{bmatrix} N_1[t+1] \\ N_2[t+1] \end{bmatrix} = \begin{bmatrix} 1-\alpha & \beta \\ \alpha & 1-\beta \end{bmatrix} \begin{bmatrix} N_1[t] \\ N_2[t] \end{bmatrix}$$

Or, written another way,

$$\vec{N}[t+1] = M \cdot \vec{N}[t]$$

If you define an initial population vector, $\vec{N}[0]$ and a transition matrix M (which requires specifying values for α and β, you can "iterate" this model by simply using a `for` loop to multiply the matrix (technically, the dot product) with this initial population vector many times (thus, calculating $\vec{N}[1]$, $\vec{N}[2]$, and so on).

1. Code up this model and ultimately create a figure like the one I displayed in class (below).



2. Repeat the above, but with $\alpha = 0.98$ and $\beta = 0.99$.

3. Update your above code by placing the code that runs the model inside a function that takes two arguments (`alpha` and `beta`). Now you should be able to easily create a figure for any new values of α and β, without copying and pasting many lines of code!

4. Create a plot using `image` where the x-axis is α (range 0 to 1) and the y-axis is β (range 0 to 1) and cells are shaded according to the value of N_1 after 100 generations.