Matrix models in R (lab 3)

©2010 Ben Bolker (modified by Steve Walker)

September 4, 2014

1 Basic matrix manipulation

R uses `matrix` to enter matrices:

```
> (X <- matrix(c(0, 1, 0, 0.1, 0, 0.4, 1.5, 0, 0.2), nrow = 3))

##      [,1] [,2] [,3]
## [1,]    0  0.1  1.5
## [2,]    1  0.0  0.0
## [3,]    0  0.4  0.2
```

You need to specify *either* `nrow` or `ncol`; R will figure out the other dimension of the matrix. You must encapsulate your values within `c()`, or you will (if you're lucky) get an error or (if not) get something other than what you expected. Also notice that, by default, R specifies matrices *by column*: if you want you can explicitly specify `byrow=TRUE` (doing this, and formatting your code so that the values for each row on are a separate line, is a good way to preserve your sanity when entering large matrices). For example:

```
> X <- matrix(c(0, 0.1, 1.5,
+               1,   0, 0,
+               0,   0.4, 0.2),
+             byrow = TRUE,
+             nrow = 3)
```

You may be wondering if vectors in R are row vectors or column vectors (if you don't know what those are, don't worry). The answer is "both and neither". Vectors are printed out as row vectors, but if you use a vector in an operation that succeeds or fails depending on the vector's orientation, R will assume that you want the operation to succeed and will proceed as if the vector has the necessary orientation. For example, R will let you add a vector of length 5 to a 5×1 matrix or to a 1×5 matrix, in either case yielding a matrix of the same dimensions.

Exercise 1. Assume that `X` is a matrix for a discrete-time linear multivariate deterministic model in recurrsion form (i.e. $\mathbf{n}(t+1) = \mathbf{X}\mathbf{n}(t)$). Calculate $\mathbf{n}(1)$ given that,

$$\mathbf{n}(0) = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

Hint: matrix multiplication in `R` is done with the `%*%` operator (Table 1). Now calculate $\mathbf{n}(t+2)$.

The conceptually simplest way to calculate the state at a given time step is – as always – to use a `for` loop,

```
> n0 <- rep(1, 3)
> n <- n0 # start the state at n0
> t <- 5 # target the fifth time-step
> for(i in 1:t) n <- X %*% n # update the state t times
> print(n) # print the state

##         [,1]
## [1,] 0.9172
## [2,] 1.2160
## [3,] 0.4858
```

Note that we deleted the old state each time we updated the recurrsion. If we want to save all of these intermediate values of the state, we need a

matrix of states rather than just a vector. This is because our multivariate state requires one matrix dimension for time and one for its multiple variables. For example,

```
> N <- matrix(0,t+1,3) # t+1 because we need room for the initial state
> N[1,] <- n0 # store the initial state
> for(i in 1 + (1:t)) N[i,] <- X %*% N[i-1,]
> print(N)

##        [,1]  [,2]   [,3]
## [1,] 1.0000 1.000 1.0000
## [2,] 1.6000 1.000 0.6000
## [3,] 1.0000 1.600 0.5200
## [4,] 0.9400 1.000 0.7440
## [5,] 1.2160 0.940 0.5488
## [6,] 0.9172 1.216 0.4858
```

Note that the final row contains the same numbers in $\mathbf{n}$.

The explicit closed form solution of $\mathbf{n}(t+1) = \mathbf{X}\mathbf{n}(t)$ is $\mathbf{n}(t) = \mathbf{S}\mathbf{D}^t\mathbf{S}^{-1}\mathbf{n}(0)$, where $\mathbf{S}$ is a matrix whose columns are the eigenvectors of $\mathbf{X}$ and $\mathbf{D}$ is a diagonal matrix with the eigenvalues of $\mathbf{X}$ on the diagonal. In R, $\mathbf{S}$ can be obtained by,

```
> (S <- eigen(X)$vectors)

##           [,1]             [,2]             [,3]
## [1,] 0.6420+0i -0.2784+0.5099i -0.2784-0.5099i
## [2,] 0.6764+0i  0.7431+0.0000i  0.7431+0.0000i
## [3,] 0.3611+0i -0.2133-0.2546i -0.2133+0.2546i
```

Similarly, the diagonal of $\mathbf{D}$ is,

```
> (d <- eigen(X)$values)

## [1]  0.9492+0.0000i -0.3746+0.6861i -0.3746-0.6861i
```

and to put these in a diagonal matrix to obtain $\mathbf{D}$,

```
> (D <- diag(d,nrow=length(d)))
```

```
##           [,1]             [,2]             [,3]
## [1,] 0.9492+0i  0.0000+0.0000i  0.0000+0.0000i
## [2,] 0.0000+0i -0.3746+0.6861i  0.0000+0.0000i
## [3,] 0.0000+0i  0.0000+0.0000i -0.3746-0.6861i
```

Note that there are complex numbers. Nevertheless the closed-form solution still works, even though we know that the state variable $\mathbf{n}$ can only take on real numbers. If the answers you get from the closed-form solution have imaginary parts, they will be exactly zero and therefore not relevant; you may want to make use of the `Re` function (Table 1), which extracts the real part.

Exercise 2. Verfiy that the explicit time-dependent solution using eigenvectors and eigenvalues of $\mathbf{X}$ gives the correct value of $\mathbf{n}(5)$. Verify also by explicitly multiplying $\mathbf{X}$ by itself five times (i.e. by calculating $\mathbf{XXXXX}\mathbf{n}(0)$).

There is something amazing about the first eigenvector. This first, dominiant, eigenvector is approximately a constant multiple of the state vector after the model has been run a large number of time steps,

```
> Re(S %*% D^100 %*% solve(S) %*% n0) # state after 100 steps

##          [,1]
## [1,] 0.006848
## [2,] 0.007214
## [3,] 0.003852

> Re(S[,1]) # first eigen vector

## [1] 0.6420 0.6764 0.3611
```

To see the proportionality more easily, we define a normalization function,

```
> normalize <- function(x) {
+   x/sum(x)
+ }
> normalize(Re(S %*% D^100 %*% solve(S) %*% n0))

##        [,1]
## [1,] 0.3822
```

```
## [2,] 0.4027
## [3,] 0.2150

> normalize(Re(S[,1]))

## [1] 0.3822 0.4027 0.2150
```

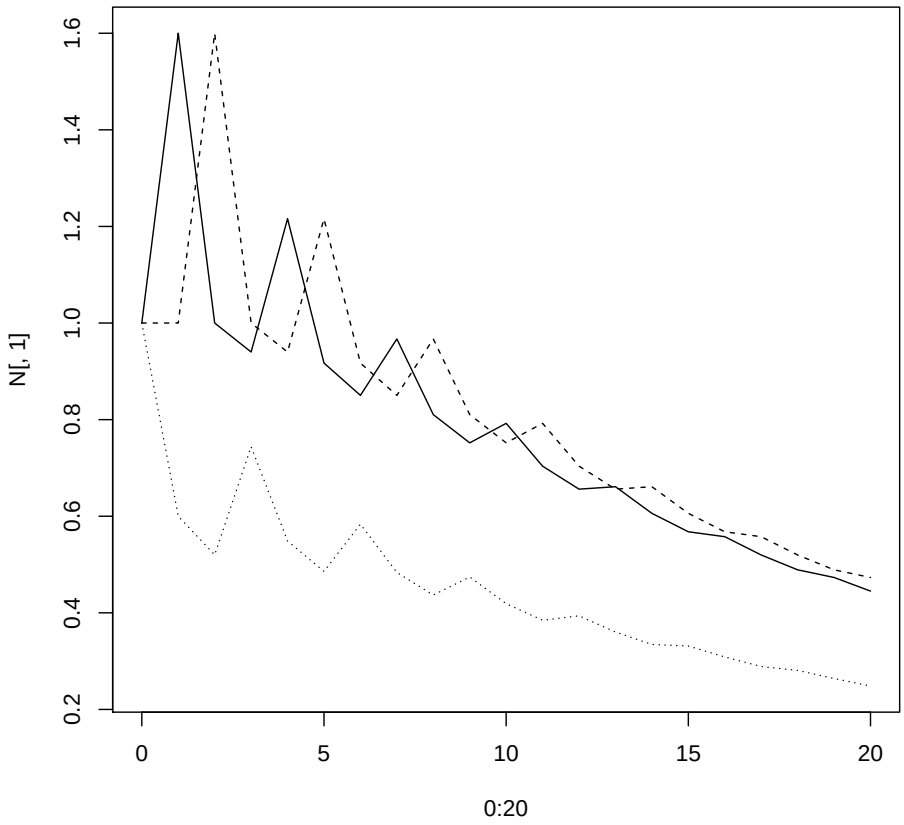
Because this is a linear system it has a fixed point at $\{0, 0, 0\}$. The eigenvalues tell us about the stability of this point. In particular, if all of the eigenvalues have an absolute value (or modulus for complex eigenvalues) of less than one, then the fixed point at the origin is stable. For example, our example system is stable because,

```
> all(Mod(d) < 1)

## [1] TRUE
```

To plot the first 20 time steps of this model we modify the `for` loop above,

```
> N <- matrix(0,20+1,3)
> N[1,] <- n0
> for(i in 2:21) N[i,] <- X %*% N[i-1,]
> plot(0:20, N[,1], ylim = range(N), type = "n")
> for(i in 1:3) lines(0:20, N[,i], lty = i)
```



Exercise 3. Modify this code to add (a) an x-axis label, 'time-step', (b) y-axis label, 'state', and (c) legend identifying the different line types used for the three different states.

Exercise 4. Show that starting with initial values of either $\{1, 1, 1\}$ or $\{1, 0, 0\}$ gets you to the same state after 100 time steps (you can either use a `for` loop or the `%^%` operator from the `expm` package), and these states are the same as the first eigenvector. Don't forget to normalize!

Download the data file `horn.txt` from the website (should already be done if you have downloaded the entire repository).

```
> f <- file.choose()  ## on MacOS or Windows this should open a dialog box
```

```
> (X <- as.matrix(read.table("horn.txt")))

##                   BTA GB SF BG SG WO OK HI TU RM BE
## Big-toothed aspen   3  5  9  6  6  0  2  4  2 60  3
## Gray birch          0  0 47 12  8  2  8  0  3 17  3
## Sassafras           3  1 10  3  6  3 10 12  0 37 15
## Blackgum            1  1  3 20  9  1  7  6 10 25 17
## Sweetgum            0  0 16  0 31  0  7  7  5 27  7
## White oak           0  0  6  7  4 10  7  3 14 32 17
## Red oaks            0  0  2 11  7  6  8  8  8 33 17
## Hickories           0  0  1  3  1  3 13  4  9 49 17
## Tulip tree          0  0  2  4  4  0 11  7  9 29 34
## Red maple           0  0 13 10  9  2  8 19  3 13 23
## Beech               0  0  0  2  1  1  1  1  8  6 80
```

Notice that the row sums are all equal to 100 — these data are in percentages, we want the *column sums* to be 1:

```
> rowSums(X)

## Big-toothed aspen        Gray birch         Sassafras
##               100               100               100
##          Blackgum          Sweetgum         White oak
##               100               100               100
##          Red oaks         Hickories        Tulip tree
##               100               100               100
##         Red maple             Beech
##               100               100

> X <- t(X)/100  ## transpose and divide by 100
> colSums(X)

## Big-toothed aspen        Gray birch         Sassafras
##                 1                 1                 1
##          Blackgum          Sweetgum         White oak
##                 1                 1                 1
##          Red oaks         Hickories        Tulip tree
##                 1                 1                 1
##         Red maple             Beech
##                 1                 1
```

This matrix contains the parameters for a linear model of temporal changes in the species composition of a forest. The changes in the abundances of the species depend on the current composition of species. Therefore, if we start with a forest containing only big-toothed aspen and nothing else, the composition after one time step would be,

```
> n0 <- numeric(nrow(X))
> n0[1] <- 1
> X %*% n0

##     [,1]
## BTA 0.03
## GB  0.05
## SF  0.09
## BG  0.06
## SG  0.06
## WO  0.00
## OK  0.02
## HI  0.04
## TU  0.02
## RM  0.60
## BE  0.03
```

The idea here is that big-toothed aspen facilitates the establishment of certain other species.

Exercise 6* (this will be homework). Determine the long-run composition of the forest using a `for` loop, the explicit closed-form eigenvector-based solution, *or* the `%^%` operator in the `exmp` package, and compare it with the dominant eigenvector (don't forget to normalize!). Sort the normalized vector of the long-run composition in decreasing order and produce a bar plot of the state variable (HINT: type `?sort` and `?barplot` to see these help files). The results of a `for` loop or the `%^%` operator will have names, which will be convenient for your bar plot). What is the dominant species? How many species will be present at frequencies of $> 5\%$? Is the origin a stable fixed point in this model (HINT: What happens if one of the eigenvalues has modulus of exactly one? Do your best!)

`matrix(v,nrow=m,ncol=n, byrow=FALSE)`	$m \times n$ matrix using the values in `v`
`t(X)`	transpose (exchange rows and columns) of matrix `X`
`dim(X)`	dimensions of matrix `X`. `dim(X)[1]`=# rows, `dim(X)[2]`=# columns
`data.entry(A)`	call up a spreadsheet-like interface to edit the values in `A`
`diag(v,n)`	diagonal $n \times n$ matrix with v on diagonal, 0 elsewhere (`v` is 1 by default, so `diag(n)` gives an $n \times n$ identity matrix)
`cbind(a,b,c,...)`	combine compatible objects by attaching them along columns
`rbind(a,b,c,...)`	combine compatible objects by attaching them along rows
`as.matrix(x)`	convert an object of some other type to a matrix, if possible. This is particularly handy for converting the results of `read.table`, which returns a data frame instead of a matrix
`outer(v,w)`	"outer product" of vectors `v`, `w`: the matrix whose $(i,j)^{\text{th}}$ element is `v[i]*w[j]`
`eigen(X)`	compute eigenvalues and eigenvectors of a matrix: the results is a list with the eigenvalues stored as `$values` and the eigenvectors as columns of the matrix `$vectors` (the eigenvectors are normalized so that their sum of squares is 1)
`X%*%Y`	multiply matrices `X` and `Y`
`X%^%k`	raise matrix `X` to the `k`th power (in the `expm` package — you will need to install and load it first)
`expm(X)`	exponentiate `X` (`expm` package)
`solve(X)`	matrix inverse
`Re(X)`	real part of a complex matrix (or vector)
`Im(X)`	imaginary part of a complex matrix (or vector)
`Mod(X)`	element-wise modulus of a complex matrix (or vector)

Table 1: Some important functions for creating and working with matrices. Many of these have additional optional arguments; use the help system for full details.