CSC 121: Computer Science for Statistics

Radford M. Neal, University of Toronto, 2017

http://www.cs.utoronto.ca/∼radford/csc121/

Week 5

Making Vectors by Repetition

As you may recall from a previous lab exercise, you can make a vector in R by repeating a single value or a vector of values. For example:

```
> rep (5,10)
 [1] 5 5 5 5 5 5 5 5 5 5
> rep (c(8,1,2), 5)
 [1] 8 1 2 8 1 2 8 1 2 8 1 2 8 1 2
> rep (c("fred","mary"), 3)
[1] "fred" "mary" "fred" "mary" "fred" "mary"
```

Instead of saying how many times to repeat, you can instead say what the final length should be:

```
> rep (c(8,1,2), length=10)
 [1] 8 1 2 8 1 2 8 1 2 8
```

Another option is to say how many times each element should be repeated immediately:

```
> rep (c(8,1,2), each=3)
[1] 8 8 8 1 1 1 2 2 2
```

Making Sequence Vectors

You've seen that you can create a vector consisting of a sequence of consecutive integers like this:

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

If the first operand of `:` is greater than the second, the sequence it creates will go backwards:

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

The `seq` function is more flexible. It can create sequences of numbers that differ by an amount other than one:

```
  > seq (1, 2, by=0.1)
   [1] 1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.0
  > seq (1.1, by=0.01, length=13)
   [1] 1.10 1.11 1.12 1.13 1.14 1.15 1.16 1.17 1.18 1.19 1.20 1.21 1.22
```

Combining Ways of Creating Vectors

We can use the various ways of creating vectors that we've seen in combination.

For example:

```
> c (1:5, 5:1)
 [1] 1 2 3 4 5 5 4 3 2 1
> rep (1:5, 3)
 [1] 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5
> c (seq(1,2,by=0.2), rep(2,5))
 [1] 1.0 1.2 1.4 1.6 1.8 2.0 2.0 2.0 2.0 2.0 2.0
```

Note: The `c` function *combines* single values or vectors to make a bigger vector. If you already have the vector you want, you don't have to use `c`!

For example, the use of `c` in all the following is unnecessary.

```
> c(5)
[1] 5
> c(1:5)
[1] 1 2 3 4 5
> rep(c(5),3)
[1] 5 5 5
```

Matrices

In R, the elements of a vector can be arranged in a two-dimensional array, called a *matrix*.

You can create a matrix with the `matrix` function, giving it a vector of data to fill the matrix (down columns), which is repeated automatically if necessary:

```
 > matrix (3, nrow=2, ncol=2)
      [,1] [,2]
 [1,]    3    3
 [2,]    3    3
 > matrix (1:6, nrow=2, ncol=3)
      [,1] [,2] [,3]
 [1,]    1    3    5
 [2,]    2    4    6
```

You can fill in the data by row instead if you like:

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

Treating Matrices Mathematically

R has operators that treat a matrix in the mathematical sense as in linear algebra. For example, you can do matrix multiplication with the `%*%` operator:

```
> A <- matrix(c(2,3,1,5),nrow=2,ncol=2); A
     [,1] [,2]
[1,]    2    1
[2,]    3    5
> B <- matrix(c(1,0,2,1),nrow=2,ncol=2); B
     [,1] [,2]
[1,]    1    2
[2,]    0    1
> A %*% B       # This multiplies A and B as matrices
     [,1] [,2]
[1,]    2    5
[2,]    3   11
> A * B         # This just multiplies element-by-element
     [,1] [,2]
[1,]    2    2
[2,]    0    5
```

Treating Matrices Just as Arrays of Data

You can instead just consider a matrix to be a convenient way of laying out your data, not as an object in linear algebra.

For this purpose, it’s useful that you can create matrices with data other than numbers:

```
> matrix (c(TRUE,FALSE,TRUE), nrow=3, ncol=3)
      [,1]  [,2]  [,3]
[1,]  TRUE  TRUE  TRUE
[2,] FALSE FALSE FALSE
[3,]  TRUE  TRUE  TRUE

> matrix (c("abc","xyz"), nrow=3, ncol=2)
     [,1]  [,2]
[1,] "abc" "xyz"
[2,] "xyz" "abc"
[3,] "abc" "xyz"
```

Indexing Elements of a Matrix

You can get or change elements in a matrix by using `[...]` with *two* subscripts, the first identifying the row of the element, the second the column.

For example:

```
> X <- matrix (1:6, nrow=2, ncol=3); X
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> X[1,3]
[1] 5
> A <- matrix (0, nrow=3, ncol=3)
> A[2,1] <- 5
> A[1,3] <- 7
> A[3,3] <- 9
> A
     [,1] [,2] [,3]
[1,]    0    0    7
[2,]    5    0    0
[3,]    0    0    9
```

Extracting Rows and Columns of a Matrix

You can also use `[...]` to extract an entire row or column of a matrix, by just omitting one of the two subscripts.

For example:

```
> X <- matrix (1:6, nrow=2, ncol=3); X
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

> X[1,]          # get the first row
[1] 1 3 5
> X[,2]          # get the second column
[1] 3 4
> X[1,] + X[2,]  # add the first and second rows
[1]  3  7 11
```

Combining Matrices with `cbind` and `rbind`

You can put two matrices with the same number of rows together with `cbind`:

```
> X <- matrix (1:6, nrow=2, ncol=3)
> Y <- matrix (3, nrow=2, ncol=4)
> cbind(X,Y)
     [,1] [,2] [,3] [,4] [,5] [,6] [,7]
[1,]    1    3    5    3    3    3    3
[2,]    2    4    6    3    3    3    3
```

Similarly, `rbind` can put together two matrices with the same number of columns.

You can also use `cbind` or `rbind` to combine a matrix with a vector, which is treated like a matrix with one row or one column:

```
> rbind(X,c(10,20,30))
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
[3,]   10   20   30
```

Example: Plotting a Function of Two Arguments

One use of matrices is in plotting functions or data in three dimensions.

Here, we compute values of the function $\cos(8\sqrt{x^2+y^2})$ for a grid of values for x from -1 to $+1$, and a grid of values for y from 0 to 2.5, storing these values in a matrix called `funvals`. The grid points are spaced apart by 0.01.

```
> gridx <- seq(-1,1,by=0.01)
> gridy <- seq(0,2.5,by=0.01)
>
> funvals <- matrix (0, nrow=length(gridx), ncol=length(gridy))
> for (i in 1:length(gridx))
+   for (j in 1:length(gridy))
+     funvals[i,j] <- cos (8*sqrt(gridx[i]^2 + gridy[j]^2))
```

One Column of the Computed Matrix

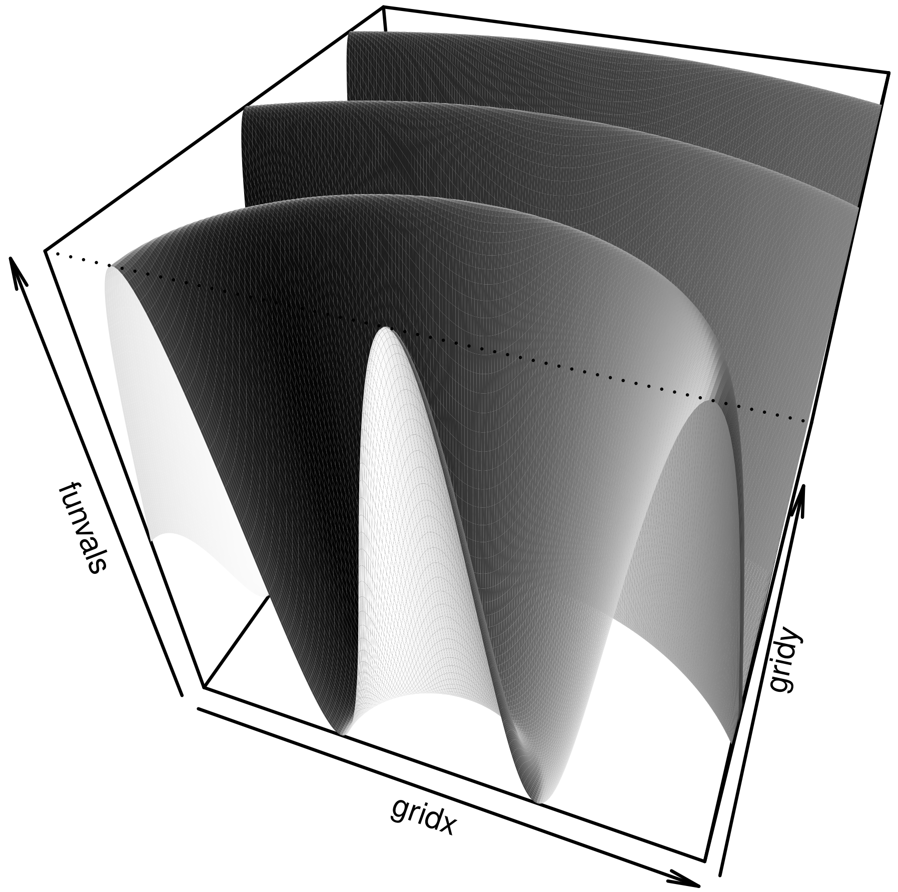
Here's a single column of the matrix of function values that we computed:

```
> round (funvals[,1], 2)
  [1] -0.15 -0.07  0.01  0.09  0.17  0.25  0.33  0.40  0.47  0.54  0.61  0.67
 [13]  0.73  0.78  0.83  0.87  0.91  0.94  0.96  0.98  0.99  1.00  1.00  0.99
 [25]  0.98  0.96  0.93  0.90  0.87  0.82  0.78  0.72  0.67  0.60  0.54  0.47
 [37]  0.40  0.32  0.25  0.17  0.09  0.01 -0.07 -0.15 -0.23 -0.31 -0.38 -0.46
 [49] -0.52 -0.59 -0.65 -0.71 -0.77 -0.81 -0.86 -0.90 -0.93 -0.96 -0.98 -0.99
 [61] -1.00 -1.00 -0.99 -0.98 -0.97 -0.94 -0.91 -0.88 -0.84 -0.79 -0.74 -0.68
 [73] -0.62 -0.56 -0.49 -0.42 -0.34 -0.27 -0.19 -0.11 -0.03  0.05  0.13  0.21
 [85]  0.29  0.36  0.44  0.51  0.57  0.64  0.70  0.75  0.80  0.85  0.89  0.92
 [97]  0.95  0.97  0.99  1.00  1.00  1.00  0.99  0.97  0.95  0.92  0.89  0.85
[109]  0.80  0.75  0.70  0.64  0.57  0.51  0.44  0.36  0.29  0.21  0.13  0.05
[121] -0.03 -0.11 -0.19 -0.27 -0.34 -0.42 -0.49 -0.56 -0.62 -0.68 -0.74 -0.79
[133] -0.84 -0.88 -0.91 -0.94 -0.97 -0.98 -0.99 -1.00 -1.00 -0.99 -0.98 -0.96
[145] -0.93 -0.90 -0.86 -0.81 -0.77 -0.71 -0.65 -0.59 -0.52 -0.46 -0.38 -0.31
[157] -0.23 -0.15 -0.07  0.01  0.09  0.17  0.25  0.32  0.40  0.47  0.54  0.60
[169]  0.67  0.72  0.78  0.82  0.87  0.90  0.93  0.96  0.98  0.99  1.00  1.00
[181]  0.99  0.98  0.96  0.94  0.91  0.87  0.83  0.78  0.73  0.67  0.61  0.54
[193]  0.47  0.40  0.33  0.25  0.17  0.09  0.01 -0.07 -0.15
```

A Perspective Plot of the Function

We can produce a three dimensional plot from the function values we computed using R's `persp` function (with options `phi` and `theta` to set the viewing angle):

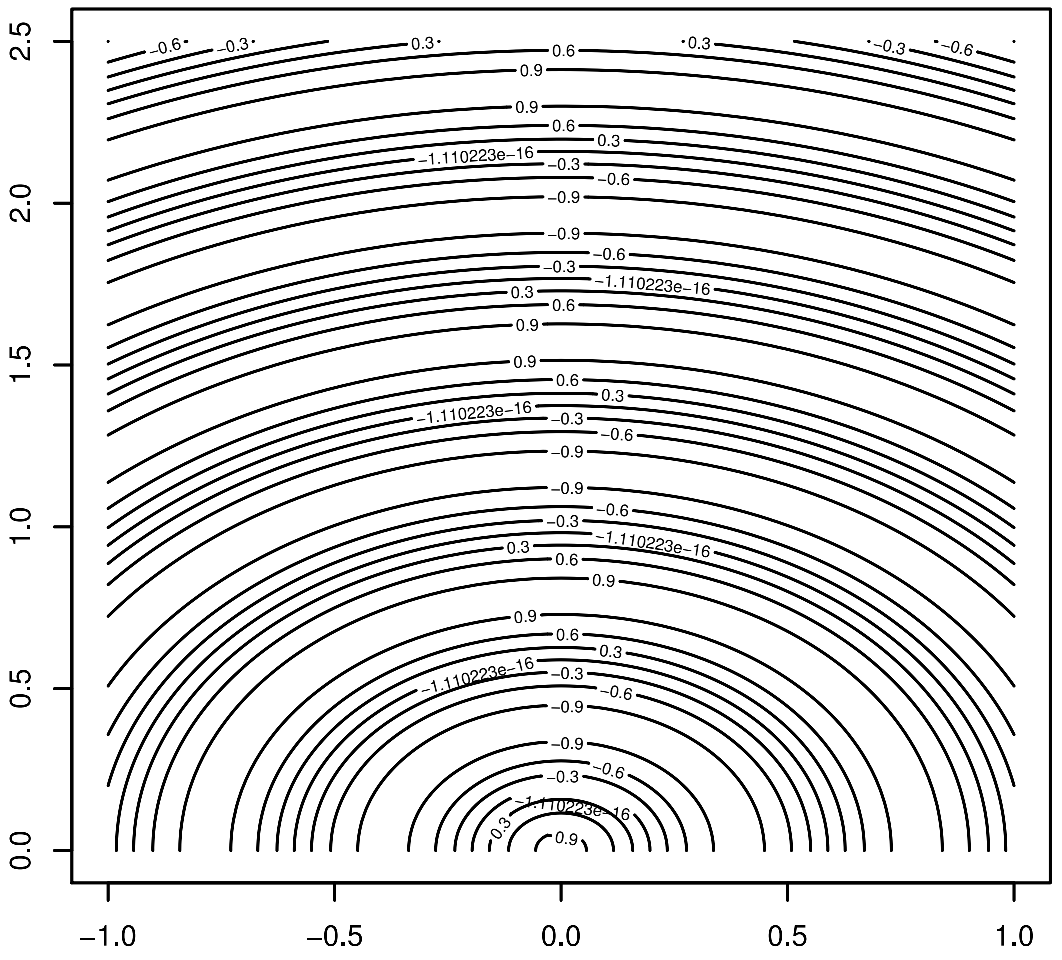
```
> persp(gridx,gridy,funvals,phi=40,theta=20,shade=0.75,border=NA)
```



A Contour Plot of the Function

Another way to display a function or data is with a contour plot, which we can produce as follows:

```
> contour (gridx, gridy, funvals, levels=seq(-0.9,0.9,by=0.3))
```



Specifying Function Arguments by Name

Suppose you define a function with several arguments, such as

```
hohoho <- function (times, what) {
    r <- what
    while (times > 1) { r <- paste(what,r); times <- times-1 }
    r
}
```

You can call the function by just giving values for the arguments, in the same order as in the function definition. For example:

```
> hohoho (3, "ho")
[1] "ho ho ho"
```

But you can instead specify arguments using their names, in any order:.

```
> hohoho (times=3, what="ho")
[1] "ho ho ho"
> hohoho (what="ho", times=3)
[1] "ho ho ho"
```

This is very useful if there are many arguments, whose order is hard to remember.

Default Values for Function Arguments

When you define a function, you can specify a *default* value for an argument, which is used if a value for the argument isn't specified when the function is called. For example, here is the **hohoho** function with defaults for both arguments:

```
  hohoho <- function (times=3, what="ho") {
      r <- what
      while (times > 1) { r <- paste(what,r); times <- times-1 }
      r
  }
```

Here are some calls of this function:

```
> hohoho(4)            # 'what' will default to "ho"
[1] "ho ho ho ho"
> hohoho(what="hee")  # 'times' will default to 3
[1] "hee hee hee"
> hohoho()             # uses defaults for both arguments
[1] "ho ho ho"
```

This is very useful for functions with many arguments that are often set to the same (default) value, as is the case for many of R's pre-defined functions.

Giving Names to List Elements

You can give names to elements of a list, and then refer to these elements by name with the `$` operator. For example:

```
> L <- list (a=c(3,1,7), bc=c("red","green"), q=1:4)
> L$a
[1] 3 1 7
> L$bc
[1] "red"   "green"
> L$q <- TRUE
> L
$a
[1] 3 1 7

$bc
[1] "red"   "green"

$q
[1] TRUE
```

If an element has a name, R uses it for printing, rather than the numerical index.

Using a List to Return Multiple Values from a Function

This function takes as input a vector of character strings, and returns a list of two vectors, with the first and the last characters of the input strings:

```
first_and_last_chars <- function (strings) {
    first <- character(length(strings)) # Create two string vectors for
    last <- character(length(strings))  # the results, initially all ""
    for (i in 1:length(strings)) {
        nc <- nchar(strings[i])
        first[i] <- substring(strings[i],1,1)  # Find first & last chars
        last[i] <- substring(strings[i],nc,nc) # of the i'th string
    }
    list (first=first, last=last)  # Return list of both result vectors
}
```

Here's an example of its use:

```
  > fl <- first_and_last_chars (c("abc","wxyz"))
  > fl$first
  [1] "a" "w"
  > fl$last
  [1] "c" "z"
```

Names for Vector Elements and Matrix Rows and Columns

You can also give names to elements of vectors, and use the names as indexes:

```
> x <- c (dog=5, cat=3)
> x
dog cat
  5   3
> x["cat"]
cat
  3
```

You can also give names to the rows and columns of matrices:

```
> M <- matrix(1:4,ncol=2,nrow=2,dimnames=list(c("cat","dog"),
+                                             c("big","small")))
> M
    big small
cat   1     3
dog   2     4
> M["dog","big"]
[1] 2
```

Scanning the Elements of a Matrix

Here's an example function that finds the largest negative element in a numeric matrix (ie, the negative element with smallest absolute value), returning this element's value, or minus infinity if there are no negative elements.

Note that you can find the number of rows and number of columns in a matrix with `nrow` and `ncol`.

```
largest_neg <- function (M) {
    result <- -Inf
    for (i in 1:nrow(M))
        for (j in 1:ncol(M))
            if (M[i,j] < 0 && M[i,j] > result)
                result <- M[i,j]
    result
}
```

Here's an example call:

```
> largest_neg (matrix (c(-6,3,-2,1), nrow=2, ncol=2))
[1] -2
```