CSC 120: Computer Science for the Sciences (R section)

Radford M. Neal, University of Toronto, 2016

http://www.cs.utoronto.ca/∼radford/csc120/

Week 4

Making Vectors by Repetition

As you may recall from previous lab exercises, 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

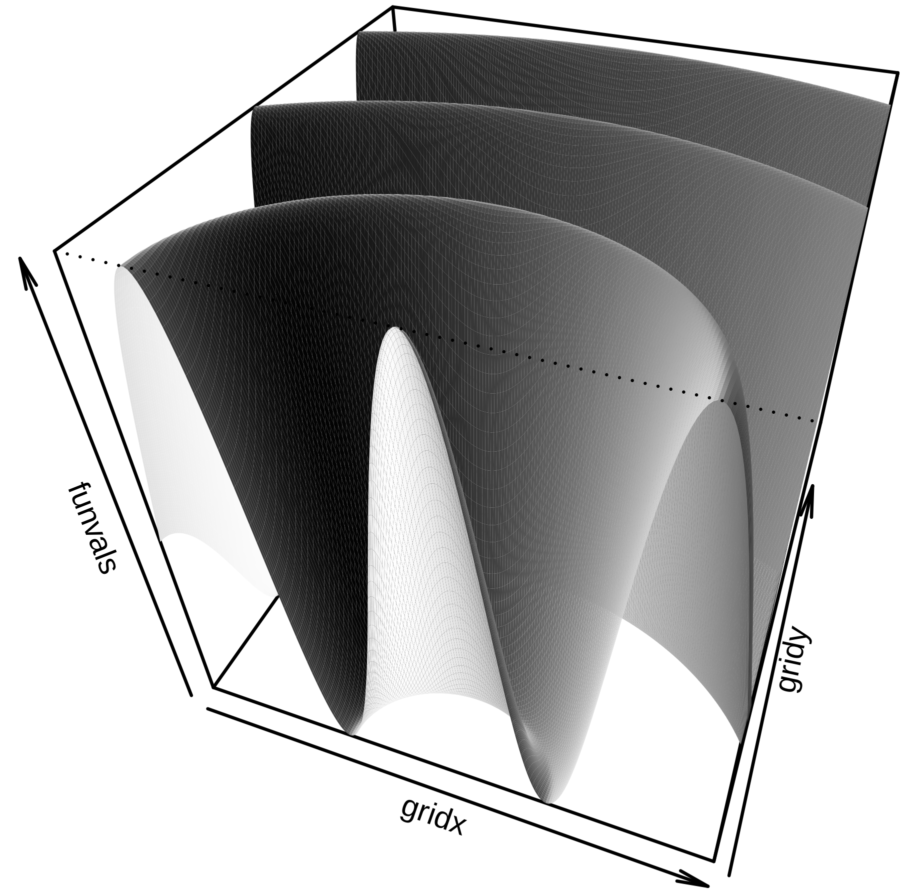
Here's a single column of the matrix of function values that we computed:

```
> 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):

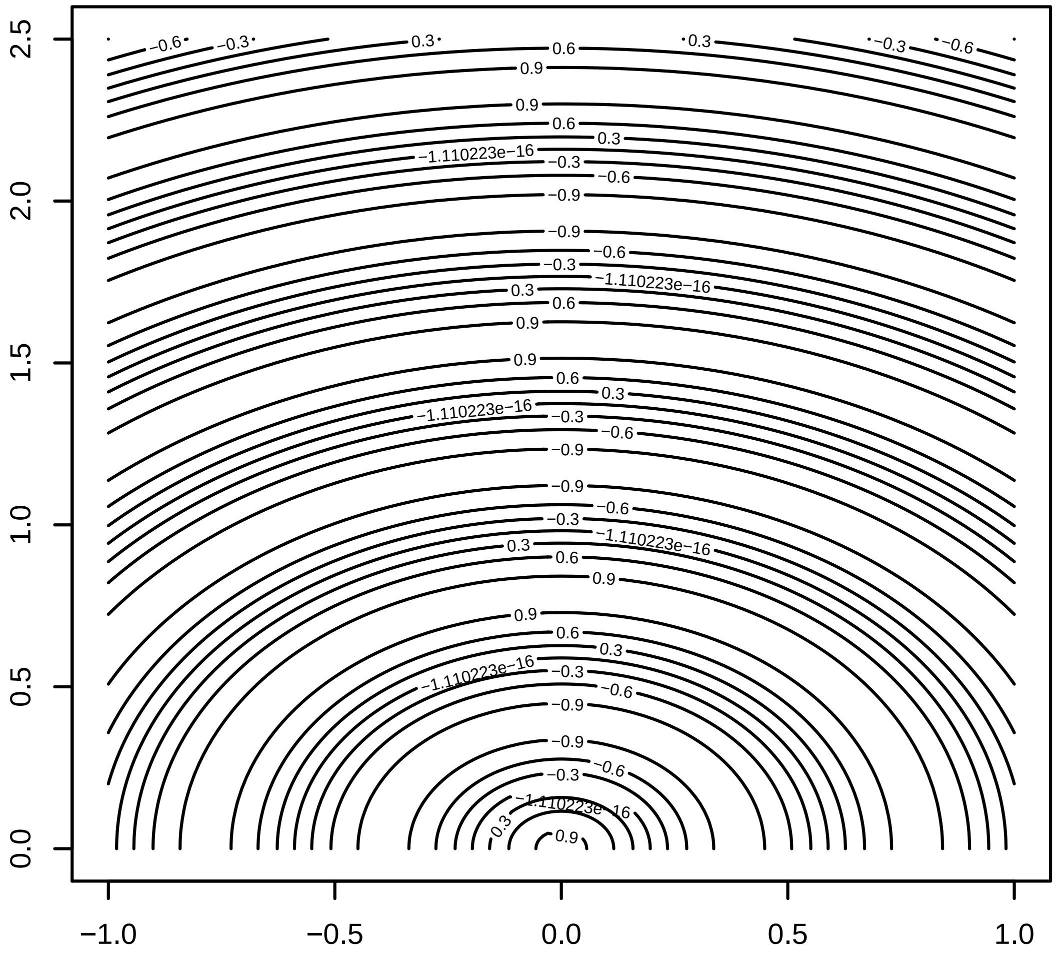
```
> persp(gridx,gridy,funvals,phi=40,theta=20,shade=0.75,border=NA)
```



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.

Creating a Plot in Stages

Many simple plots can be created with a single `plot` command — eg, `plot(x,y)` will plot points with coordinates given by the vectors `x` and `y`.

More complicated plots can be created in stages by adding more points, lines, and text to what has already been plotted.

The general approach:

- Create a new plot with `plot`. It might contains some points or lines, or might be completely empty. Features such as the axis scales and labels are determined at this stage.

- Then add more information, using functions such as `points`, `lines`, `abline`, and `text`. You can call these functions as many times as needed, perhaps with different options for things like colour and line width each time.

- You can also add a title above the plot with the `title` function.

Creating a New Plot

You create a new plot with the `plot` function. It takes one or two data vectors as its first arguments, but has many, many other possible arguments. You'll want to let most of these have their default values, and refer to any that you set by name.

Here are some of the possible arguments to `plot`:

`type`	Type of plotting — `"p"` for points (the default), `"l"` for lines, `"b"` for both points and lines, `"c"` for lines only but with space for points
`col`	Colour for points/lines plotted (default is `"black"`)
`xaxt`	Set to `"n"` to get rid of horizontal axis numbers
`yaxt`	Set to `"n"` to get rid of vertical axis numbers
`xlab`	Label for the horizontal axis
`ylab`	Label for the vertical axis
`xlim`	Horizontal range for plot (vector of length two)
`ylim`	Vertical range for plot (vector of length two)
`asp`	Aspect ratio, `asp=1` ensures one vertical unit looks the same length as one horizontal unit

For example, `plot (c(), xlim=c(0,2), ylim=c(1,5))` will plot an empty frame with horizonal axis labels from 0 to 2 and vertical axis labels from 1 to 5.

Adding Points to a Plot

We can add points to a plot with the `points` function. Like `plot`, it takes two vectors as its first two arguments, containing the x and y coordinates of the points. (Or just a single vector argument with the y coordinates, in which case the x coordinates are 1, 2, 3, . . .)

It can also take other arguments that set various options, such as

`type`	Set to `"b"` for lines as well as points
`col`	Colour for points plotted
`pch`	Character to plot points with — default is a circle, other possibilities are `pch="x"` for plotting with x symbols, or `pch=20` for solid dots

For example, `points (x, y, col="red", pch=20)` will add solid red dots to the plot, at the coordinates given by the vectors `x` and `y`.

Adding Lines to a Plot

We can add lines to a plot with the `lines` function.

In addition to one or two arguments giving the coordinates of the points to connect with lines, it can take other arguments such as those below (which can also be used for `plot`):

`type`	Set to `"b"` for points too, `"c"` for lines only but with space for points
`col`	Colour for lines plotted
`lty`	Line type — eg, `"dotted"`, `"dashed"`, or `"solid"` (the default)
`lwd`	Line width (default is 1)

For example, `lines (y, col="green", lty="dotted")` will add dotted green lines to the plot, at the x coordinates 1, 2, 3, $\ldots$ and y coordinates given by the vector `y`.

Adding Text to a Plot

We can add text to a plot with the `text` function.

Here's an example that adds "WOW" to the origin of the plot:

```
> text (0, 0, "WOW")
```

We can put many character strings on a plot with one call of `text`, since its arguments can be vectors of x coordinates, y coordinates, and character strings.

For example:

```
> x <- 1:10
> y <- x^2
> plot(x,y,xlim=c(0,11))
> text(x,y+2,paste("square of",x))
```

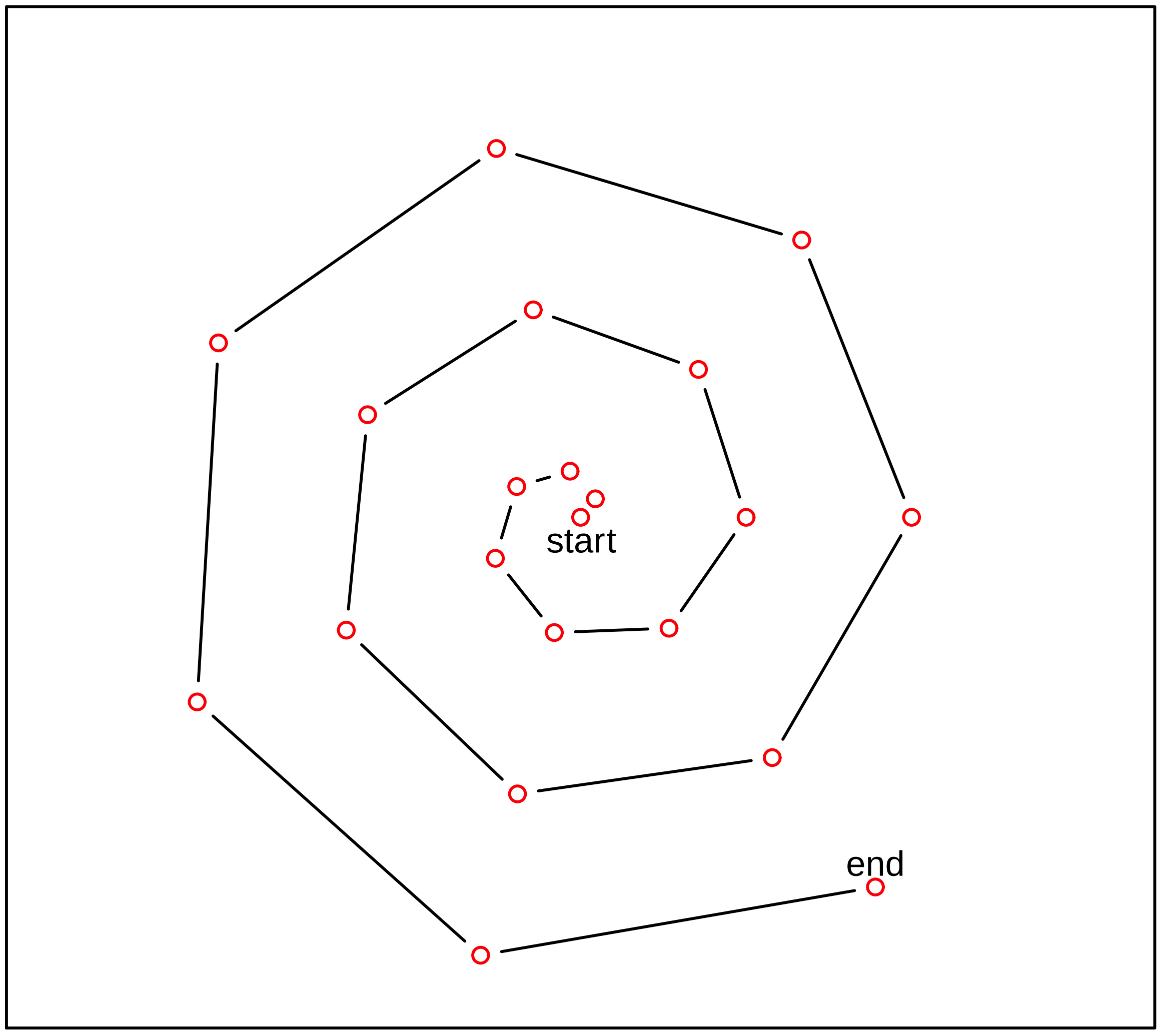
Example: Drawing a Spiral

Here's an example R script that draws a spiral in a plain box, using 7 segments each time it winds around, with red dots at the vertices. The start and end are labelled with "start" and "end".

```
n <- 20
angle <- 2*pi*(0:n)/7
dist <- 0:n
x <- dist * cos(angle)
y <- dist * sin(angle)

plot (x, y, type="c", xaxt="n", yaxt="n", xlab="", ylab="",
            xlim=c(-n,n), ylim=c(-n,n), asp=1)

points (x, y, col="red")

text (x[1], y[1]-1, "start")
text (x[n+1], y[n+1]+1, "end")
```

The Spiral Plot

```
> source("http://www.cs.utoronto.ca/~radford/csc120/spiral-script.r")
```



Random Numbers and Their Uses

Random variation is a big part of what statistics is about. So it's natural that R has facilities to create its own random variation — to generate *random numbers*.

Random numbers have many uses (and not just in statistics):

- Simulate random processes, such as how a disease epidemic might spread between people.
- See how the results of some statistical method vary when the data it is applied to vary randomly.
- Compute things using "Monte Carlo" methods.
- Make interactions with a user have a random aspect — we don't want a video game to behave the same way every time we play!

Generating Random Numbers with Uniform Distribution

One simple kind of random number is one that takes on a real value that is *uniformly distributed* within some bounds.

You can get such numbers in R using the `runif` function. It takes as arguments the number of random numbers to generate, the low bound, and the high bound. We'll try generating one at a time here:

```
> runif(1,0,10)     # one random number in (0,10)
[1] 3.195956
> runif(1,0,10)     # another one, not the same
[1] 5.551191
> runif(1,0,10)     # ... and another
[1] 1.165307
> runif(1,100,200)  # one from a different range
[1] 182.0236
```

The random numbers generated are supposed to be *independent* — eg, which one we get the second time is unrelated to what the first one was.

R’s Random Numbers Aren’t Really Random

Computers are carefully designed to *not* behave randomly.

Some computers have special devices for producing random numbers that are really random. This is useful for cryptography (you want a really random key for your code, so nobody else can guess it).

But for most purposes we don’t actually want real random numbers. They’re too hard to generate, and if we use them, we can’t reproduce our results another day.

For example: Imagine that after running your program for a long time, it stops with an error message, indicating it has a bug. You think you’ve now fixed the bug. But how do you verify that you’ve really fixed it if you can’t reproduce the run that led to the error?

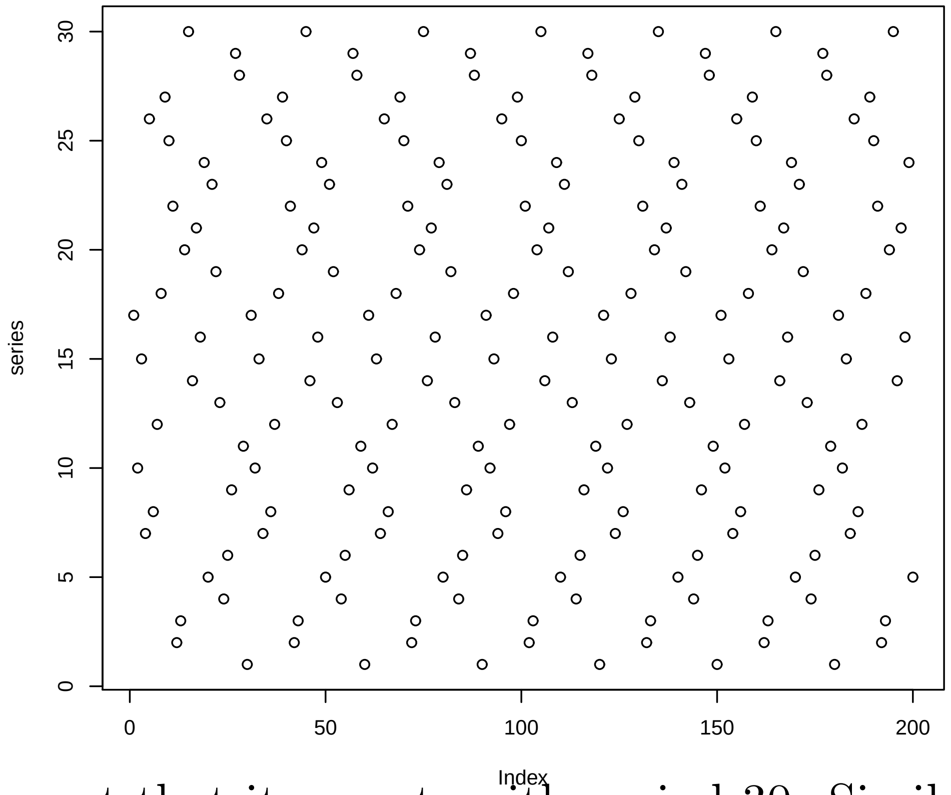
So most computer “random” numbers are really “pseudo-random” — numbers that *look random* for most purposes, but are actually generated by an algorithm that isn’t random at all, so if it is run again, it will generate exactly the same numbers.

An Example of a Pseudo-Random Generator

Here's one simple way to generate a series of pseudo-random numbers, uniformly distributed over the integers 1, 2, . . . , 30.

```
> nxt <- 1; series <- c()
> for (i in 1:200) { nxt <- (nxt * 17) %% 31; series <- c(series,nxt) }
```

Here's a plot of the resulting series:



It looks random, except that it repeats with period 30. Similar generators can have much longer periods, however.

Setting the Random Seed

R uses a more sophisticated pseudo-random generator, but it also is deterministic, and will reproduce the same sequence if restarted with the same “seed”.

For example:

```
> set.seed(123)
> runif(1)
[1] 0.2875775
> runif(1)
[1] 0.7883051
> runif(1)
[1] 0.4089769
> set.seed(123)
> runif(1)
[1] 0.2875775
> runif(1)
[1] 0.7883051
> runif(1)
[1] 0.4089769
```

For serious work, you should set the seed, so you’ll be able to reproduce your results.

The `sample` function

The call `sample(n)` will generate a random permutation of the integers from 1 to `n`, as illustrated below:

```
> set.seed(1)
> sample(10)
 [1]  3  4  5  7  2  8  9  6 10  1
> sample(10)
 [1]  3  2  6 10  5  7  8  4  1  9
> sample(10)
 [1] 10  2  6  1  9  8  7  5  3  4
```

With other kinds of arguments, `sample` can do other things as well, including sampling with replacement.