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

Radford M. Neal, University of Toronto, 2015

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

Week 9

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 first:

```
> 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
```

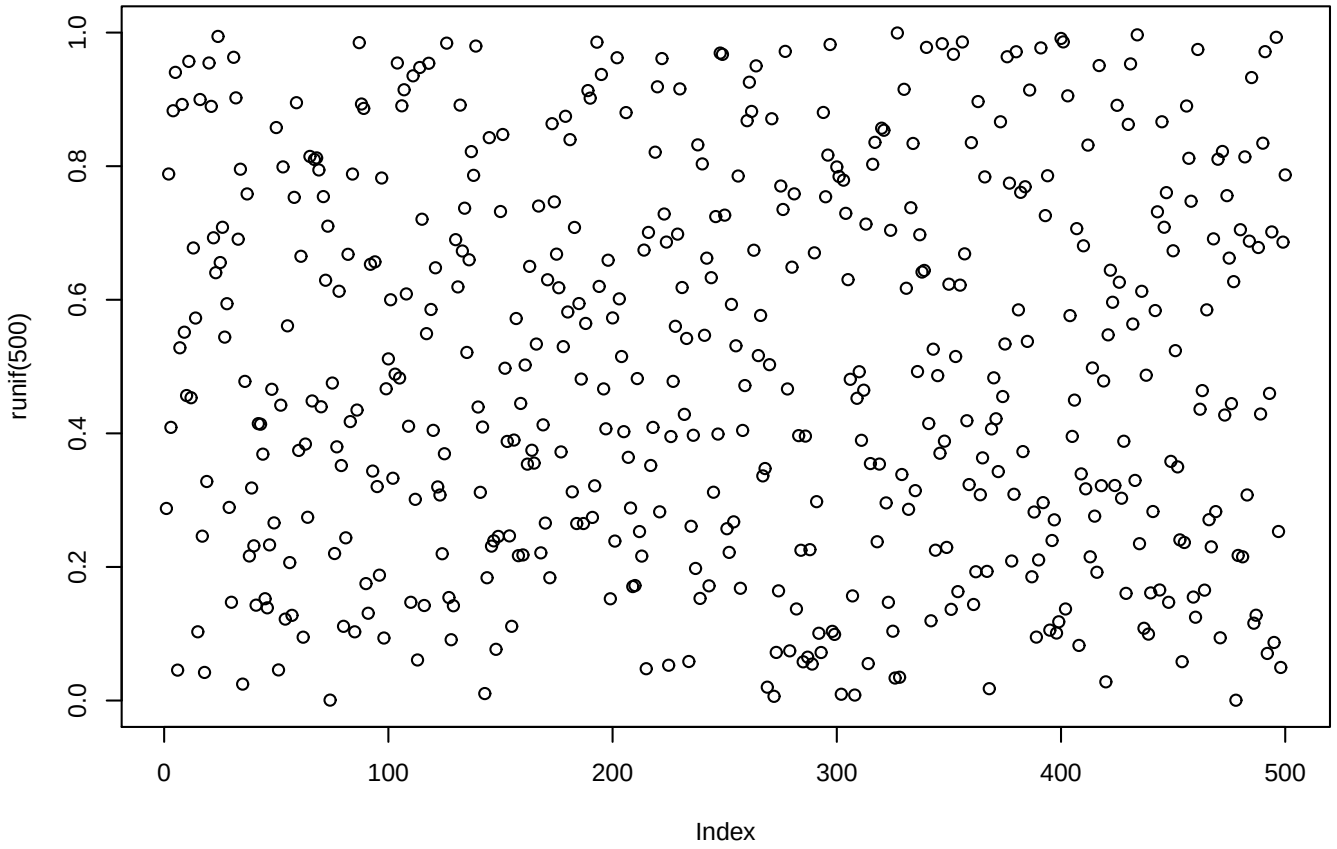
The random numbers generated are supposed to be *independent* — eg, which one we get the second time is unrelated to what the first one was.

Generating Random Vectors

We can ask for a whole vector of random numbers at once.

For instance, here we plot 500 random numbers uniformly distributed from 0 to 1 (which is the default range), using the command

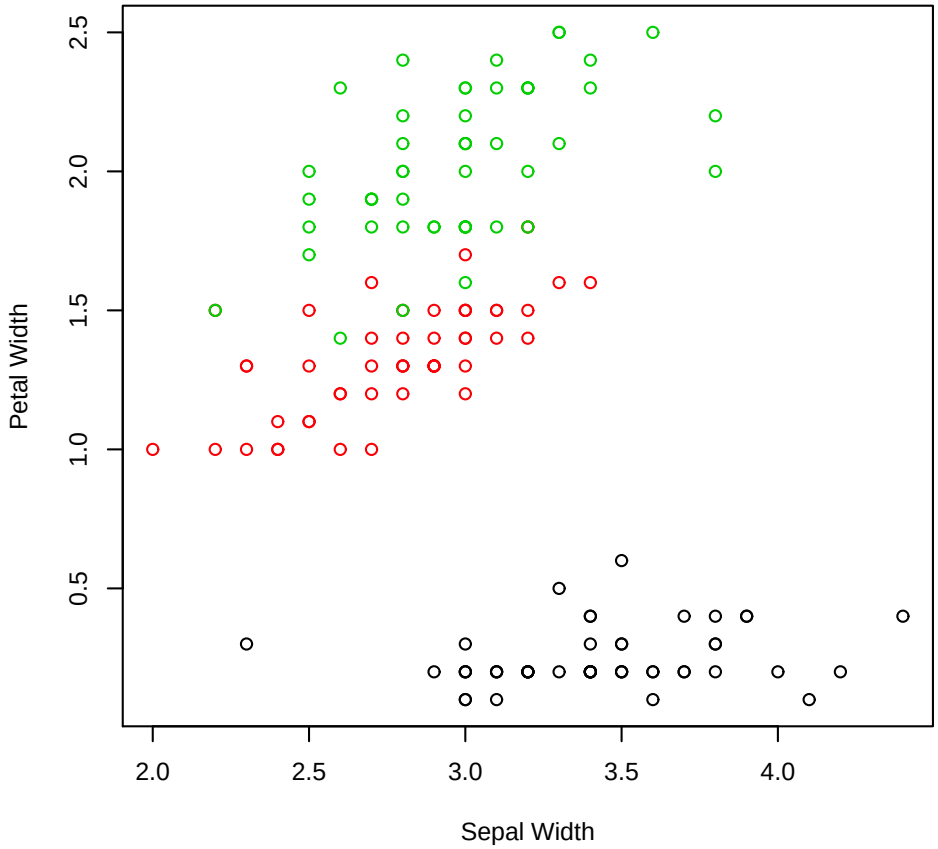
```
> plot(runif(500))
```



The Problem with Plotting Rounded Data Points

Recall the "iris" data set of width and length of petals and sepals in three species of Iris. Here's a scatterplot of two of the variables (species marked by colour):

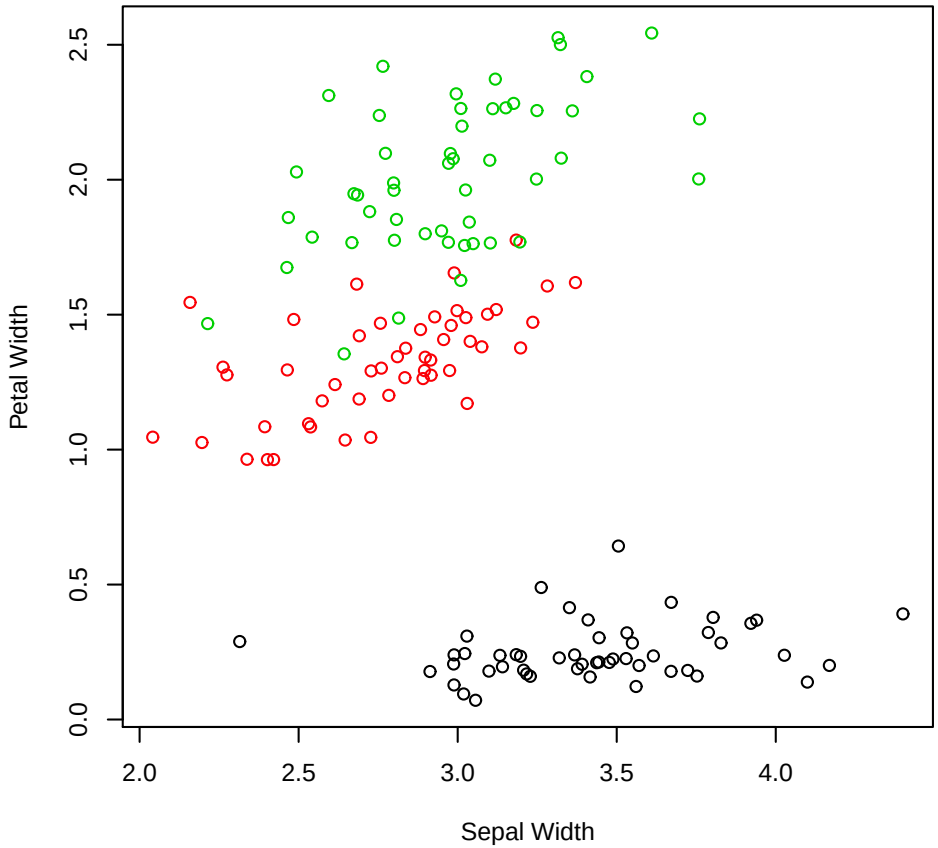
```
plot (iris$Sepal.Width, iris$Petal.Width, col=iris$Species,
      xlab="Sepal Width", ylab="Petal Width")
```



Solving the Problem with Random Jitter

Because the data is rounded to one decimal place, many of the dots in the scatterplot are on top of each other. To see all the data points, we can add random "jitter" to each data point before plotting:

```
plot (iris$Sepal.Width + runif(nrow(iris),-0.05,+0.05),
      iris$Petal.Width + runif(nrow(iris),-0.05,+0.05),
      col=iris$Species, xlab="Sepal Width", ylab="Petal Width")
```



Making Random Choices

Often, we want to make a random choice, with certain probabilities for doing certain things.

If we have a binary choice (to do or not do something), we can compare a random number that's uniform over $(0, 1)$ to the desired probability.

For example, at some point in a computer game, we might want to kill the player and end the game with probability 0.15. We can do it as follows:

```
if (runif(1) < 0.15) stop("You're dead. Game over!")
```

Why does this work?

Suppose we have a three-way choice – do A with probability 0.15, do B with probability 0.4, or do C with probability 0.45. (Note that these three probabilities add to one.)

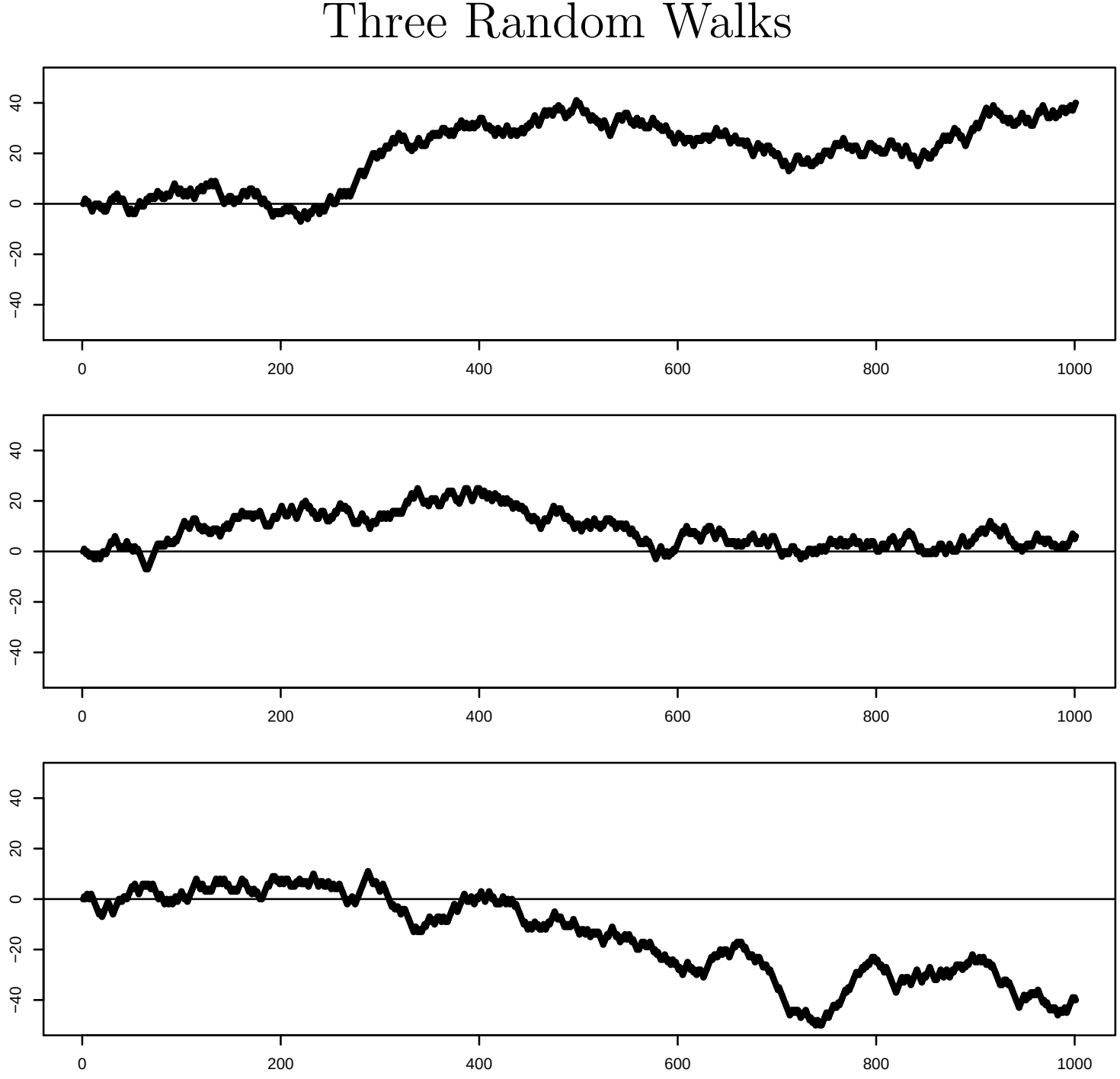
Could we generate one random number uniform over $(0, 1)$ and use it to make this choice?

Simulating a Random Walk

One well known "stochastic process" is a *random walk* on the integers, in which we start at 0, and at each time step thereafter we randomly go to the position one above or one below our current position, with probability 0.5 for either direction.

Here's an R function to simulate a random walk:

```
random_walk <- function (steps) {
    position <- numeric(steps+1)
    for (i in 1:steps) {
        if (runif(1) < 0.5)
            position[i+1] <- position[i] + 1
        else
            position[i+1] <- position[i] - 1
    }
    position
}
```


Three Random Walks
40
20
0
−20
−40
0
200
400
600
800
1000

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?

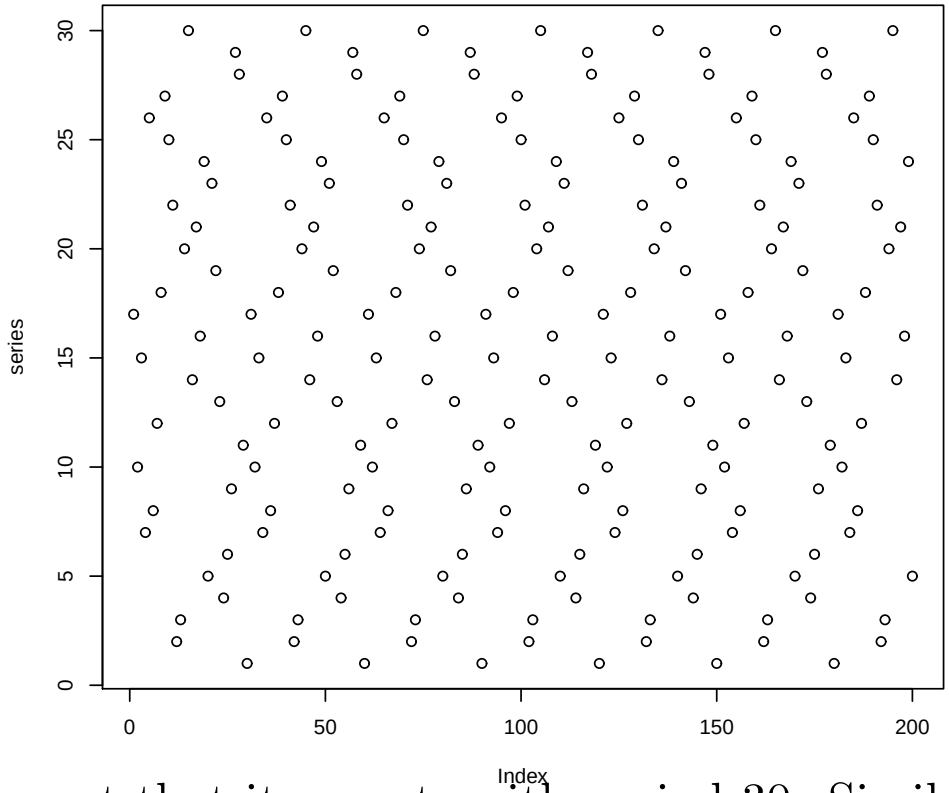
So most computer "random" numbers are really "pseudo-random" — numbers that *look random* for most purposes, but are actually generated by an algorithm that isn't random at all, so if it is run again, it will generate exactly the same numbers.

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.