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

Radford M. Neal, University of Toronto, 2016

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

Week 5

Combining Data of Different Types in a List

We've seen how we can put several numbers into a vector of numbers. Or we can put several strings into a vector of strings. But what if we want to combine both types of data? Let's try...

```
> c(123,"fred",456)
[1] "123"  "fred" "456"
```

R converts the numbers to character strings, so that the elements of the vector will all be the same type (character).

But we *can* put together data of different types in a *list*:

```
> list(123,"fred",456)
[[1]]
[1] 123

[[2]]
[1] "fred"

[[3]]
[1] 456
```

Lists Can Contain Anything

Elements of a list can actually be anything, including vectors of different lengths:

```
> list (1:4, 3:10)
[[1]]
[1] 1 2 3 4


[[2]]
[1]  3  4  5  6  7  8  9 10
```

You can even put lists within lists (though these are hard to read when printed):

```
> list(4,list(5,6))
[[1]]
[1] 4


[[2]]
[[2]][[1]]
[1] 5


[[2]][[2]]
[1] 6
```

Extracting and Replacing Elements of a List

You can get a single element of a list by subscripting with the `[[ ...]]` operator:

```
> L <- list (c(3,1,7), c("red","green"), 1:4)
> L[[2]]
[1] "red"   "green"
> L[[3]]
[1] 1 2 3 4
```

You can replace elements the same way. Continuing from above...

```
> L[[3]] <- c("x","y","z")
> L
[[1]]
[1] 3 1 7


[[2]]
[1] "red"   "green"


[[3]]
[1] "x" "y" "z"
```

Notice that the new value can have a type different from that of the old value.

Making Lists Bigger

You can make list longer by assigning to an element that doesn't exist yet:

```
> L <- list(1,3)
> L
[[1]]
[1] 1

[[2]]
[1] 3
> L[[3]] <- "xx"
> L
[[1]]
[1] 1

[[2]]
[1] 3

[[3]]
[1] "xx"
```

You can create a list this way starting with an empty list made with `list()`.

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"
```

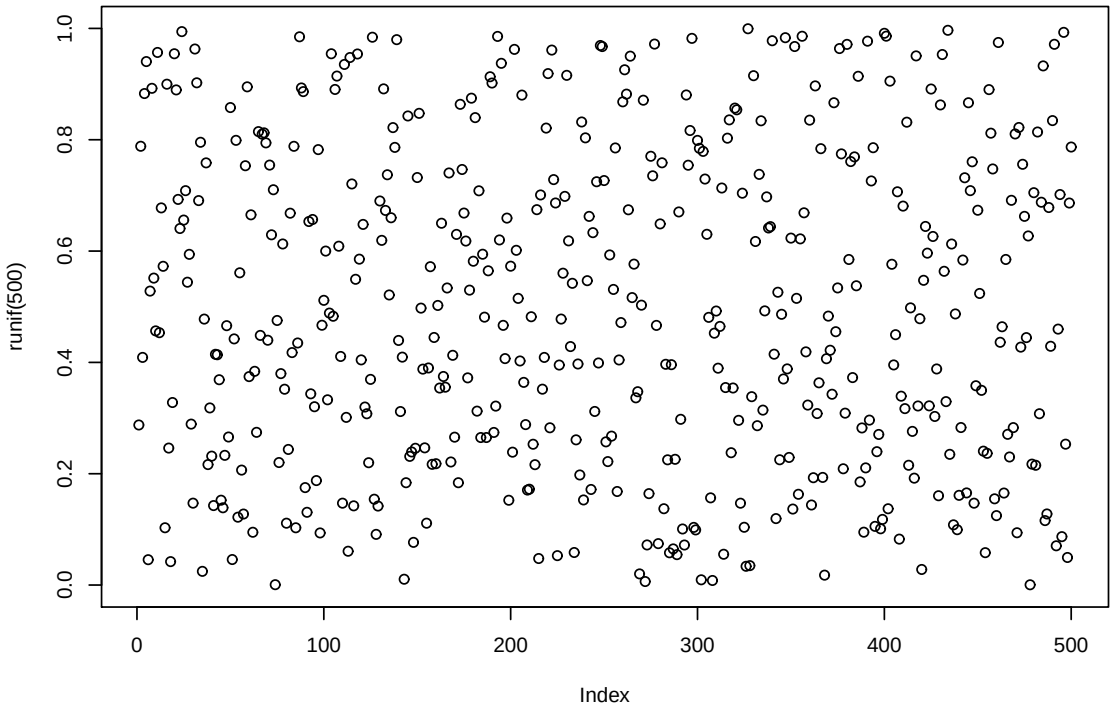
Generating Random Vectors

Recall that we can generate a single random number uniformly distributed between 0 and 1 with `runif(1)`.

We can instead ask for a whole vector of random numbers at once, by letting the (first) argument of `runif` be greater than one.

For instance, here we plot 500 random numbers uniformly distributed from 0 to 1, using the command

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

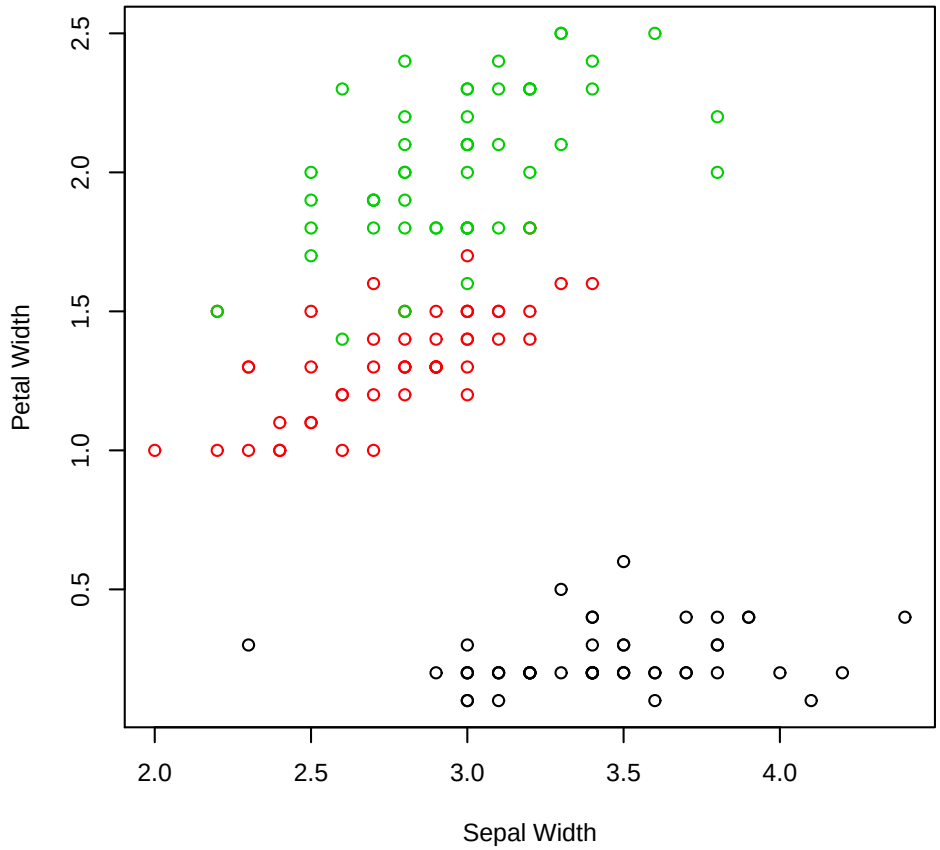


The Problem with Plotting Rounded Data Points

Recall the "iris" data set of width and length of petals and sepals in three species of Iris. It is stored in a special kind of list called a "data frame", which we'll talk more about later.

Here's a scatterplot of two of the variables (species marked by colour):

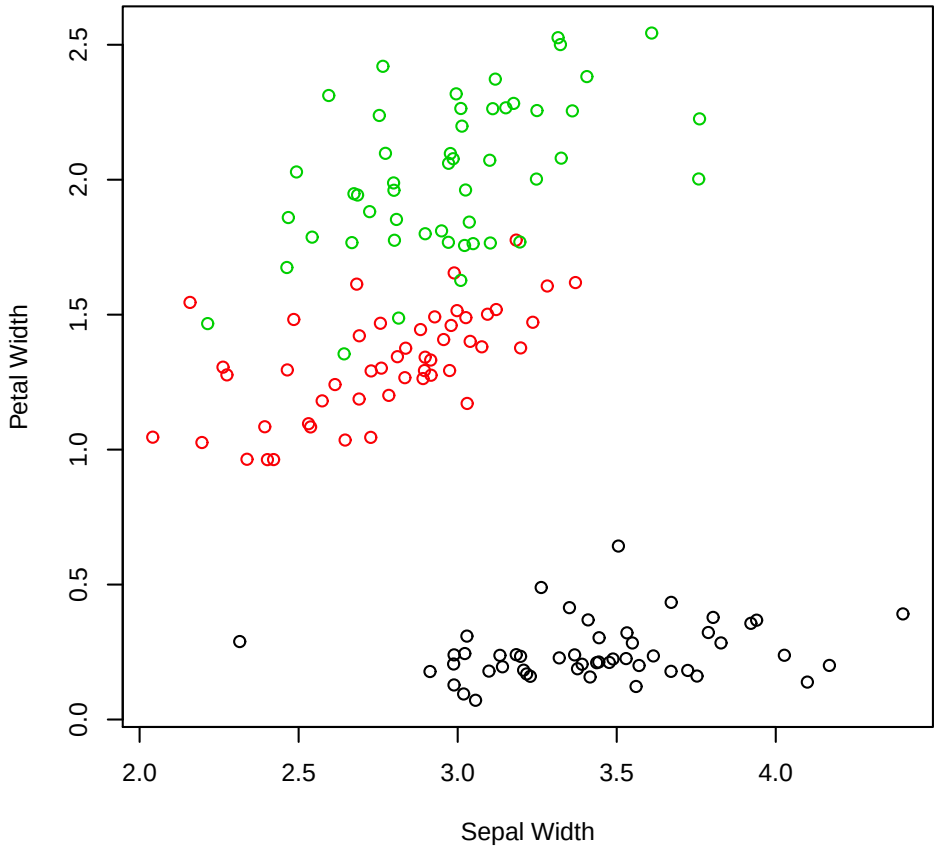
```
plot (iris$Sepal.Width, iris$Petal.Width, col=iris$Species,
      xlab="Sepal Width", ylab="Petal Width")
```



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 instead 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.)

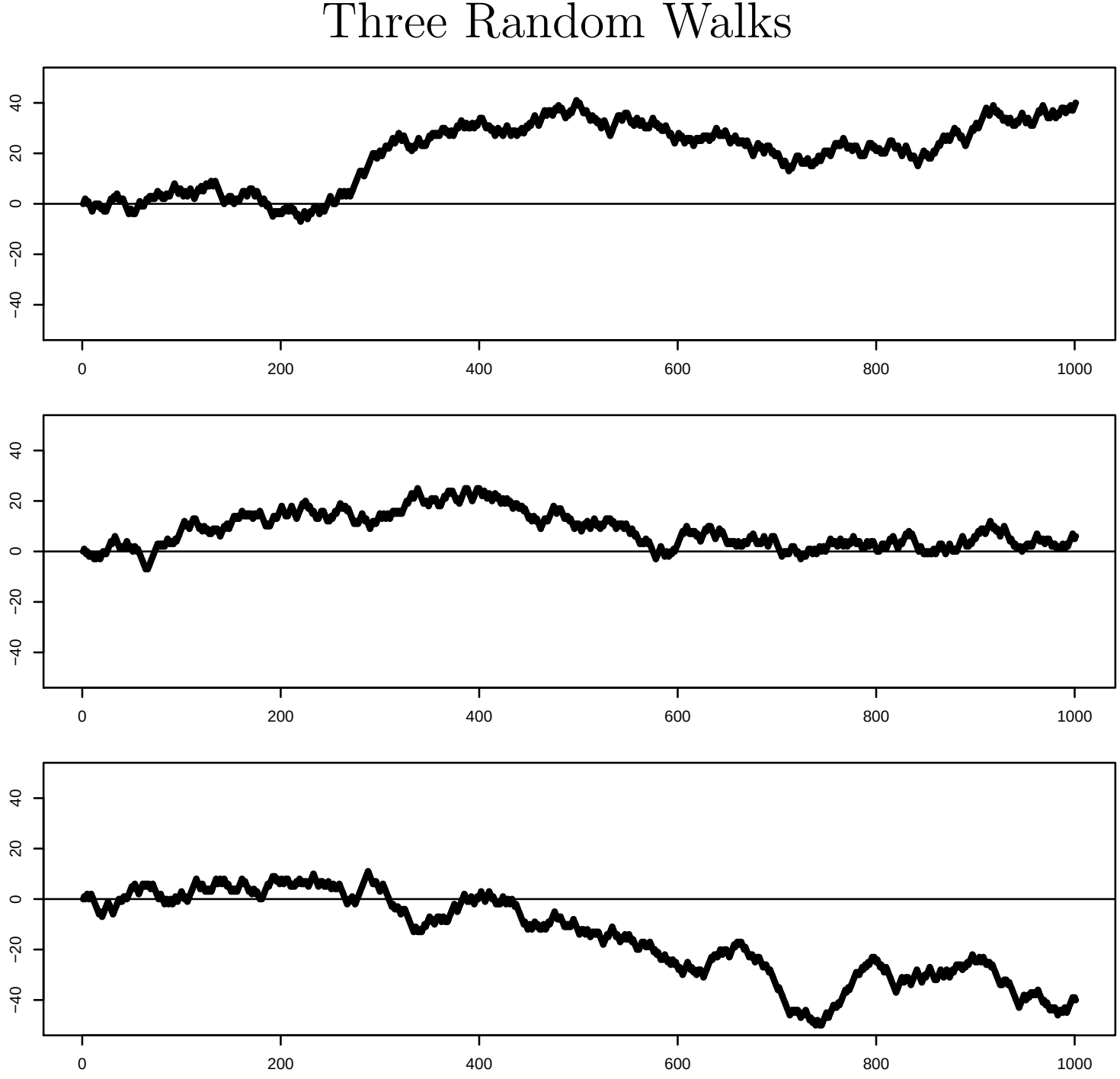
Could we generate one random number uniform over $(0, 1)$ and use it to make this choice?

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