CSC 120 (R Section)— Lab Exercise 5

This is a non-credit exercise, which you do not hand in.
You may work on your own or together with another student, as you please.

Recall that in the lecture slides for this week there is a function that simulates a random walk, which you can get from the course web page, and which is also shown here:

```
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
}
```

In this lab, you will modify this function in various ways, and write other funcitions that do things with random walks. The purpose is to provide you with general practice writing R functions, and practice with random number generation in particular. You'll also learn a bit about random walks.

You should create a file that contains the functions you write, and another script file (or several files) that starts by reading in these function definitions with `source`, and then uses the functions to do things described below. You might also try out the functions just by typing in the R console window.

Biased random walks. First, you should modify the `random_walk` function to create a `biased_random_walk` function, for which the probability of going up by 1 and going down by 1 need not be the same. It should take an additional argument that is the probability of going up. (Passing 0.5 for this argument should produce the same behaviour as the `random_walk` function.)

Try this function out with various values for the probability of going up, and see how the walks it generates differ from a random walk with equal probabilities of going up or down. You can plot a random walk with a command like `plot(biased_random_walk(1000,0.7))`.

Multiple random walks. To see how random walks vary, we would like to simulate many of them. This can be done in several ways. You should write functions called `random_walks_1`, `random_walks_2`, and `random_walks_3` that all take two arguments — how many steps to take for each of the random walks, and the number of random walks to simulate — and all return a matrix with number of rows equal to the number of steps plus 1 and number of columns equal to the number of random walks. Each column of this matrix contains the positions for one of the random walks. The first row of this matrix will have all 0s, since all the walks start at 0.

Make the probability of going up and going down both be 0.5 for these functions. Note that the different random walks should be independent (ie, they don't interact in any way).

The three functions will generate the random walks in different ways.

For `random_walks_1`, you should generate all of the first walk, then all of the second walk, etc. This will require two loops, one inside the other, with the outer loop being over the different walks, and the inner loop being over the steps for a walk. For `random_walk_2`, you should reverse which is the inner loop and which is the outer loop, so that you generate all walks at once, one step at a time. Try out both of these functions, for a fairly small number of steps and small number of walks, after setting the random seed to some value with `set.seed`. Do you get the same results? Why?

For the `random_walks_3` function, you should also generate all walks at once, but with only one loop (which goes over the steps of the walks). Instead of an inner loop, you should use random generation of vectors and arithmetic on vectors to figure out what every walk is going to do with one R statement. Recall that you can refer to a whole row of a matrix by just leaving out the second subscript (eg, `A[i,]`). This works both for getting a row, and changing a row. Note that you can compare a vector to a single value or to another vector and get a vector of FALSE/TRUE values. You can convert such a vector of FALSE/TRUE values to 0/1 values with `as.numeric`.

Try out `random_walks_3` after setting the random number seed. Is the result the same as either `random_walks_1` or `random_walks_2`?

Plotting multiple random walks. Now that you can generate multiple random walks, it would be nice to be able to plot them all on a single plot. Write a function called `plot_walks` that takes a matrix produced by one of the three random walks functions and plots all the walks, in different colours.

There are several ways this might be done. One way is to first create an empty plot with `plot(c(),...)` and then use `points` to add points for each walk, in different colours. (You can use the colours identified by the integers 1, 2, etc.) You'll need to set the axis ranges with the `xlim` and `ylim` arguments to `plot` so that all the walks will fit on the plot. If `walks` is the matrix with all the walks, `min(walks)` and `max(walks)` will give you the minimum and maximum value in any of the walks.

Seeing how the standard deviation over walks changes. All the walks will be in the same position at the beginning, since they all start at 0. After that, they will tend to get further and further from each other. We can measure how spread out they are after some number of steps by the standard deviation of their positions. Write a function called `walk_sd` that takes as its argument a matrix of random walks, and returns a vector giving the standard deviation of positions for these walks at each time. The R function `sd` finds the standard deviation of a vector of numbers.

Try your `walk_sd` function out with 50 walks that each go for 1000 steps. How does the standard deviation seem to change over time? Probability theory says it should increase according to the square root of the number of steps. Try plotting that function on top of the plot of standard deviations at each time (using `lines`), and see how well it matches the standard deviations. Then try it all again with a different random number seed.

Seeing how long it takes for a walk to reach some value. If we start our random walk at 0, we might wonder how many steps it will take before it first reaches some other value, say 4. Write a function called `steps_until_value` that returns how many steps it takes a simulated random walk to reach some value that is given as the argument to `steps_until_value`. You shouldn't try to save the whole random walk in this function, just find out how many steps it takes before the specified value is reached.

Write another function called `steps_distribution` that calls `steps_until_value` many times, and returns a vector of its values. The arguments of this function should be the value to pass on to `steps_until_value` and the number of times to call `steps_until_value`. Try this function out with the target value being 4 and with 500 walks. What sort of distribution do you see for the numbers you get? The distribution can be seen more easily if you sort the numbers with `sort`.