CS1112-202/206/207–Spring 2010

Section 7 Solution

Friday, March 12

At the time of this writing, no solutions have been posted on the course webpage. As a result, solutions to this exercise are posted on the section page. Please follow the styles used in these scripts (e.g., comment header and function description) as they represent a good style practice that would earn a full style credit on the projects.

1 Determinant of a 3x3 Matrix

The only complication of this problem is creating three new, smaller matrices from the matrix we are given so that we can use them for the built-in `det` function. Otherwise, the remaining of the function is simple since we need not use any loops or conditional statements.

2 Find a Value in a Matrix

Note that we need to find *all* the occurrences of the given value in the given matrix. The only way to do this correctly is to examine *all* the elements of the matrix. Since matrices come in rows and columns, one way to achieve this goal is to go through each row and each column. That is, for each row, go through each column and determine whether that element is equal to the given value. This immediately suggests a nested-loop structure for our solution. The built-in function `size` gives us the dimension of the matrix.

Now that we can access each element of the matrix, what should we do if the value is equal to the element we are considering? We should add the location (row and column) to the return variables. Because we are *adding* the location, the return variables should be initialized first. Where should we do it? Since they are global to the whole function (that is, initializing anywhere inside the loop might not give a correct solution), we initialize them before the loops.

Another, final concern is that the function requires r and c to be *column* vectors. This can be done by using `;` operator when we append the location to r and c.

Those are essentially what we have to do. Following this idea, we arrive at a solution similar to the one posted on the section page.

3 Random Walk

First, observe that a random walk on the 21×21 grid centered at the origin is on the square $[-10, 10] \times [-10, 10]$. In order to determine the most frequently visited square efficiently, a frequency matrix must be used. We first obtain random walk information from the function we are provided with. We can treat it as black boxes: We do not have to care what it actually does, and, more importantly, we cannot modify it. All we care is that the function returns the locations of the

walk. So, we just have to go through the locations one by one and update the frequencies accordingly. Once we are done with this, we can go through all the elements in the frequency matrix and find the location of the maximum frequency.

There are a few catches to this. Since the coordinates a location can be negative, some care must be taken when updating the frequency matrix, or we will index it with an out-of-bound value. Hence, we have to make sure that all the coordinates we might have correspond to positive indices for the frequency matrix. Since we know that the minimum value of any coordinate is -10, we just have to add 11 to each coordinate so the indices are between 1 and 21. Now, when we report the location of the most visited square, we also have to adjust this accordingly—by subtracting the indices we found by 11.

Another, more subtle catch is how we specify the dimension of the frequency matrix. Since the grid we are dealing with in this problem is square-shaped, this problem does not come up. What if we have a line (e.g., 1×100) grid? Here, there is only one row in the grid, but there are 100 columns. Hence, we should initialize a 1×100 matrix to keep track of frequency. We also need to index into the matrix correctly, as an x-coordinate corresponds to a column and a y-coordinate corresponds to a row. This is why the indices are swapped in the solution posted on the section page.

Finally, we can observe that some squares might not be visited at all, so why bother examining all the elements in the frequency matrix? A better way to do this, as an improvement, is to update the maximum as we go through the locations of the walk. Since frequencies only increases, we can do this. That is, when the frequency we are dealing with is greater than the maximum we have found so far, we can record that the current frequency is a new maximum and also record the location of this square. Then we will have the location of the most visited square immediately when we are done with the locations of the walk. This approach becomes the solution posted on the section page.

4 Bounded Random Walk

In a bounded random walk, we simulate for exactly the given number of steps. Call this value n. That is, if $n = 100$, we move to an adjacent square 100 times. If the current square is not at a boundary, then we simply pick a random direction just like a regular random walk. If the current square is at a boundary, however, we need to avoid going across the boundary. If the current square is at a corner, then there are only two directions that we can pick. This immediately suggests a conditional block for each of the cases. Now, what order should we consider arranging these cases? Suppose we put the unrestricted case as the first `if` condition, then it is likely that we will need to repeat the logical expressions in this condition when we go down the `if` block. Hence, it is better to put the most restricted case as the first condition. If the most restricted condition is false, then we can make the condition less restricted down the line. In the final case, where nothing is restricted, we can then only use `else` as no restrictions hold, so every square that did not fall into any restrictions are handled by the `else` branch.