CS1112-202/206/207–Spring 2010

Lab 6 Solution

Friday, March 5

1 Different Ways to Create Vectors

```
a= zeros(1,4) % 0 0 0 0

b= zeros(4,1) % 0
              % 0
              % 0
              % 0
              % What do the arguments specify? the number of rows and columns, respectively

c= ones(1,3)  % 1 1 1

d= 10:2:17    % 10 12 14 16

f= 10:-1:17   % Empty matrix: 1-by-0

g= [10 20 40] % 10 20 40
              % What does the space separator do? go to the next column

h= [10,20,40] % 10 20 40
              % What does the comma separator do? go to the next column

k= [10;20;40] % 10
              % 20
              % 40
              % What does the semi-colon separator do? go to the next row

m= [a g]      % 0 0 0 0 10 20 40

n= [b; k]     % 0
              % 0
              % 0
              % 0
              % 10
              % 20
              % 40

p= [a k]      % ERROR--mismatched dimensions! (Attempt to concatenate a column to a row)
              % Error because a is of size 1-by-4 and k is of size 3-by-1
              % Otherwise, the resulting p would look like this:
              % 0 0 0 0 10
              %         20
              %         40

q= b'         % 0 0 0 0
              % This operation is called "transpose"

r= [a b']     % 0 0 0 0 0 0 0 0
```

2 Rolling Multiple Dice

It is important that we really roll d dice for n times. That is, for each time we roll the dice, we need to roll each die one by one and sum the faces that show up. So, we have to pick d random numbers between 1 and 6 for each time we roll the dice. This is not the same as picking a random number between d and $6d$. The reason is as follows:

Suppose we pick a random number between d and $6d$. The probability (chance) that d is chosen is as likely as the probability that $3d$ is chosen. But if we pick d random numbers between 1 and 6, then the chance that the sum of the d faces is d is exactly the probability that *all* the d dice come up 1. Now, the chance that the sum of the d faces is $3d$ is the probability that all the d dice come up 3, or that half of the dice come up 1 and the other half come up 5. It is now easy to see that the chance of getting $3d$ is definitely more than the chance of getting d as the sum. Therefore, we need to be aware of this difference.

Now, let us reiterate what we already know. For each time we roll the dice, we need to roll each die one by one and sum the faces that show up. This suggests a nested-loop structure for the program. That is, the outer loop loops for n times for the number of times we roll the dice. For each of that, the inner loop loops for d times for the number of dice we are rolling. Then, we need to keep track of the sum of the faces. Since the sum is independent for each time we roll the dice (that is, the sum of this round does not affect the sum of any other round), we can reset (initialize) the sum before the inner loop. Now, it is easy to just add the outcome of each roll in that round to the sum.

Once we are done for each round, we need to increment the frequency of the sum we have. For this, the frequency is global for the whole script (that is, initializing it inside a loop will not represent the actual counts), so we have to initialize this frequency array outside and before the loops. How long should this vector be? It should be at least the maximum possible sum of the dice in each round. Since there are d dice, each having maximum face of 6, the maximum possible sum is $6d$.

Now that we have the frequency, we just have to plot the histogram. The built-in function `bar` does the job.

3 Examining a Subarray

Just an aside, this kind of array search is called *linear search* because we start searching from the first element of the array and look for the *key* element by element in the array until we find it. Here, r is the key. We are also limited with the number of elements we have to search before we report the results. Again, some care must be taken here, because n can be any integer, including nonpositive ones and ones that exceed the length of the vector. Therefore, before we index any element of the array, we need to make sure that the index is not out of bound. For nonpositive n's, this can be interpreted as, "Determine whether r appears in the first $[-1, 0, -50$, etc.$]$ components of v," but since no elements are in these first components of v, we can infer that r is not found and report this accordingly.

A first approach is to use a `for` loop. We keep searching until we have seen n elements or until we run out of elements (that is, we have examined the whole array). If r is found, then we set the return value to 1. What if r is not found? Then the return value should be 0 in the end. We cannot set it to 0 if the element we are examining is not r because we might have seen r earlier, and this will overwrite the result we should return. Hence, we have to initialize it to 0 prior to looking into the array. Then, if r is found, the return value will be set to 1 correctly, and otherwise it remains 0.

Nonetheless, it seems silly that we need to keep looking if we already find r. To contemplate this, suppose that the vector is really long, $n = 10000$, and that r is the first element of this vector.

Then we should be able to report that r is found once we have examined the first element. Using `for` loop as above, we need to unnecessarily go through 10000 elements. We can fix this by using a `while` loop and terminate the loop as soon as we find r. This is the solution posted on the course webpage.

4 Concatenating Arrays

Once we have `vectorQuery`, this task is easy. We keep generating a random number and adding it to v until what we generate is already in v. How do we know that an element is already in v? Well, this is what `vectorQuery` is for. Now, what do we pass as the arguments for `vectorQuery`? Certainly, we need to pass v and the number we have just generated. How about the length limit (n)? We can simply pass `length(v)`, which returns the length of v so that `vectorQuery` knows when to stop.