Damian Franco
dfranco24@unm.edu
101789677
CS-575

Homework 1

1) For the first question, we were asked to create a function that takes two inputs. The first is a matrix *A* and the next is a vector *b*. Our goal is to create a function that will multiply the matrix *A* and vector *b*, hence the name of the function *mat_vec*. You will see a screenshot of the function itself below. The function first creates a local vector *x* with the same shape of vector *b* for the outcome vector since matrix-vector multiplication has an output of a vector. Next, I put a local *n* variable that will have the length of the rows and columns of the matrix/vector. The function then has two for loops. The first for loop is a row iterator, while the nested/inner for loop is a column iterator. Within the for loops, the computation takes place by multiplying the current element of matrix *A* with the current element of vector *b*. When I state "current element", I am referring to the current *i* and *j* element of the matrix multiplied by the current *j* index of the *b* vector. After the row is completely summed up (dot product) the row will be iterated to the next element of the output vector *x*. This is the most naive approach to matrix-vector multiplication. You can view the code for my function below and in the Jupyter notebook.

Function mat_vec:

```
# Function to perform matrix-vector multiplication
def mat_vec(local_A, local_b):
    local_x = np.zeros(local_b.shape)
    local_n = len(local_b)

    for i in range(local_n):
      for j in range(local_n):
        local_x[i] += A[i][j] * b[j]

    return local_x
```

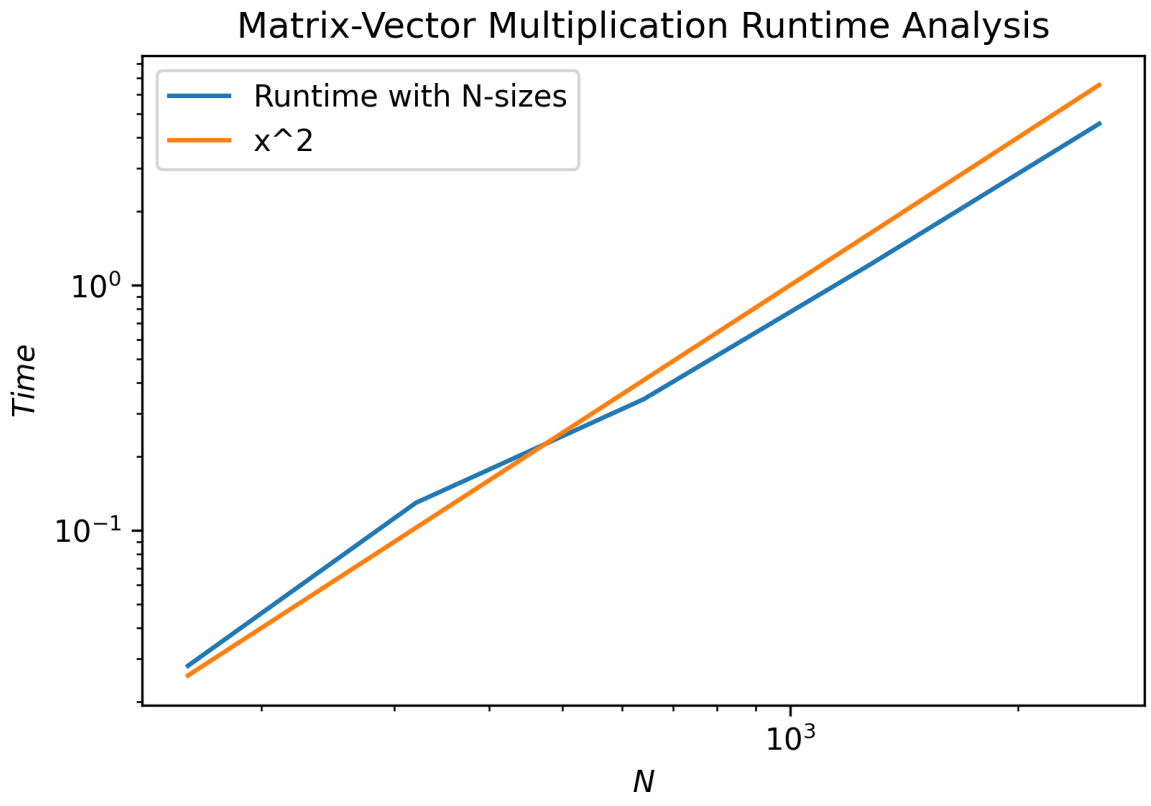
My analysis of the computation and runtime consisted of making input matrices *A* of various *n x n* sizes and vectors *b* of *n x 1* sizes. These dense matrices and vectors had random values generated at each position to check if the computation of the function is correct. The error was calculated by comparing the "exact" value generated with the % operator in the Numpy library and subtracting my approximation of the output with the exact value. I was expecting errors to be generally small with an increase in error as the size of *n* increases which is what I witnessed in my tests. I believe the increase in error is due to round-off errors with the increasing dimensions of the matrices and vectors. Runtime speed and error both increase as the matrices and vectors get larger Below you may see a table with the dimensions I used to test out this function. I used the same dimensions as Professor Zeb suggested in the example code.

Table from five test runs of mat_vec:

N size	Runtime Speed (sec)	Error
160	0.038434505462646484	2.0122289269162218e-13
320	0.2090287208557129	7.830168052553239e-13
640	0.7512457370758057	3.0733581272450618e-12
1280	2.8239729404449463	1.2709685093242425e-11
2560	6.068006992340088	4.794161949873006e-11

The time complexity or upper asymptotic bound that I derived from the matrix-vector multiplication was *O(n^2)* because of the two for loops of *n* size that we use to compute the multiplication. I plotted the bound against the runtime of my five sessions and found that it does bound the algorithm. The runtime polynomial seems to be somewhat parallel with the bound polynomial. I was expecting the lower dimensions runtimes to be over the upper bound, but I did not experience it in my runs. If that did occur, it may be due to memory and cache sizes since if a matrix/vector is small enough to fit in cache with those dimensions. Other than that, the runtimes that I experienced aligned with what I was predicting. Below you can view my plot of the runtimes.

Plot of runtime and dimension size:



2) Our second question of this homework was very similar to the first, but instead of doing matrix-vector multiplication, we wanted to make a function that did matrix-matrix multiplication. The approach here was very similar to the approach in the first question. I first set up the function *mat_mat* in the same way as the previous matrix-vector multiplication function. This function will consist of three for loops. The *i* variable here is used to iterate to the correct row, the *j* variable is used to iterate to the correct column and the *k* variable is used to iterate through both row and column since matrix-matrix multiplication is multiplying the current row to the current column and sum the values. This function does that with the computation within the three for loops. Lastly it returns the output matrix *x*. Below you can see the code I wrote to perform this operation.

Function mat_mat:

```
# Function to perform matrix-matrix multiplication
def mat_mat(local_A, local_b):

    local_x = np.zeros(local_b.shape)
    local_n = len(local_A)

    for i in range(local_n):
      for j in range(local_n):
        for k in range(local_n):
          local_x[i][j] += A[i][k] * b[k][j]

    return local_x
```

My analysis of the computation and runtime consisted of the same testing as in the first question. I first made input matrices *A* and *b* of various *n x n* sizes. These dense matrices had random values generated at each position to check if the computation of the function is correct. The error was calculated by comparing the “exact” value generated with the % and dot operator in the Numpy library and subtracting my approximation of the output with the exact value. I was again expecting errors to be generally small with an increase in error as the size of *n* increases due to round-off errors. I noticed that this function had a much higher runtime then the matrix-vector multiplication function. This is due to more computation that matrix-matrix multiplication has compared to matrix-vector multiplication. Although the computation took longer than the first function, this function had similar error sizes. I did experience much larger runtimes with this implementation and reduced the dimensions quite a bit. Below you will be able to see that in the table I generated in Jupyter.

Table from five test runs of mat_mat:

N size	Runtime Speed (sec)	Error
20	0.01804351806640625	1.8336959187428107e-13
40	0.07774543762207031	8.098940525528068e-13
80	0.512141227722168	3.3092684066257312e-12
160	4.118232488632202	1.2828112654792042e-11
320	32.84832692146301	4.867509624235047e-11

The time complexity or upper asymptotic bound that I derived from the matrix-matrix multiplication was *O(n^3)* because of the three for loops of *n* size that we use to compute the multiplication. I plotted the bound against the runtime of my five sessions and found that it does bound the algorithm. I noticed that this bound was also fully parallel. You can see that the lines are practically over each other to the point where they seem to be separated by very small values. Overall, it is very interesting to note that you can somewhat understand how algorithms increase in runtime complexity when compared to a bound that is derived from the algorithm. I believe this is a great skill to have for correctness of algorithms alongside computational correctness.

Plot of runtime and dimension size: