Programming Assignment 1

Daniel Mazus

September 16, 2024

I Executive Summary

A set of subroutines to calculate Matrix-Vector and Matrix-Matrix products to see how different structures within Matrices can impact computing time, storage, and algorithms. Computed results are compared to NumPy's Matrix-Vector and Matrix-Matrix multiplication by using Euclidean Norms to show any differences between the custom algorithms and the true result.

II Statement of Problem

We are given four subroutines that start from a broader spectrum and narrowing down to different structures that Matrices can have. This involves starting with a Unit Lower Triangular Matrix-Vector multiplication going to combining a Unit Lower Triangular Matrix and an Upper Triangular Matrix to compute Matrix-Matrix product. This is important in computation due to the complexity that matrices can have if structure is not spotted and exploited during the algorithm. These routines will help further understand how structure can affect computations and storage while furthering knowledge from mathematical structure to code structure.

III Description of Mathematics

In this section, I will go deeper into the structure and mathematics behind the different Matrices given in the subroutines.

1. Subroutine 1: We are wanting to solve the problem $w = Lv$ where L is a square Unit Lower Triangular Matrix and v is a vector. To say L is a square Unit Lower Triangular Matrix means:

$$L = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ \lambda_{2,1} & 1 & 0 & \cdots & 0 \\ \lambda_{3,1} & \lambda_{3,2} & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \lambda_{n,1} & \lambda_{n,2} & \lambda_{n,3} & \cdots & 1 \end{pmatrix}$$

   This shows that we have 1's on the diagonal, values all below the diagonal and 0's above the diagonal. This structure is important as the following two subroutines will build off of this matrix. We can associate v as:

$$v = \begin{pmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \\ \vdots \\ \omega_n \end{pmatrix}$$

IV Description of Algorithm and Implementation

I will go over the four different subroutines and how I implemented my code to fit the structure of each problem given. For the Matrices I use a 2D array and vectors use 1D arrays for any Matrix-Vector computation to help save space and reduce computation times slightly. The different subroutines will talk a little more on the structure of each and how it saves computations, storage, and execution time.

IV.I Subroutine Description

IV.I.1 Subroutine 1

We start by generating a Unit Lower Triangular Matrix, $L \in \mathbb{R}^{nxn}$ by iterating over the rows and columns to fill in the diagonal with 1's, below the diagonal is a random integer bounded by a and b, and filling above the diagonal with zeros. We then generate a random row vector, v, which has n elements since our vector $v \in \mathbb{R}^n$. Now that we have L and v we can they compute w which is m in the custom algorithm. I initialize the vector m by filling in 0's for whatever n is equal to. Then using a nested for loop allows iterating over the matrix, and we take $L[i][k]$ and multiply it by $v[i]$ and add and equal into m. So our function takes in 3 parameters, L, v, and n.

Let us now look at the amount of computations and storage that were used within the subroutine and function. For storage purposes, we know that L is a nxn matrix, so, n^2 storage is used and for each vector, v and output vector m, they both have n storage. In total for this storage intake, we have $n^2 + 2n$ being allocated. For the amount of operations used, we would normally have $2n^2 - n$ operations used but since we are only multiplying elements below the diagonal, we now have $n(n-1)$ multiplications and $(n(n-1)/2)$ additions. This gives us a total count of operations of $n^2 - n + ((n^2 - n)/2)$ which is $(3n^2 - 3n)/2$ which is less operations than the standard Matrix-Vector Multiplication $2n^2 - n$. Thus we have reduced the operations and storage needed compared to Standard Matrix-Vector Multiplication

IV.I.2 Subroutine 2

Subroutine 2 is very similar to Subroutine 1. The algorithm for subroutine is as follows:

1. Generate Unit Lower Triangular Matrix, L
2. Define the function for using Compressed Row Vectors (CRV) which consists of taking in 3 parameters, L, n, and indexing at 0:
   (a) We first calculate the size of the CRV given by $S = \frac{n(n-1)}{2}$.
   (b) Initialize the vector, crv, by multiplying $[0] * S$.
   (c) We then loop over L by the rows and then columns, only storing elements below the diagonal because we do not need to store 1 and anything above.
3. After creating the CRV, we now have the inputs for the function that does the multiplication, crv, v, n.
   (a) I initialize m as a 1D array, vector.
   (b) Set index = 0 to properly iterate through.
   (c) Compute the Multiplication by using nested for loops just as the same as Subroutine 1, but have instead a 1D array multiplied by 1D array.
   (d) Add the diagonal for each i in the loop

We can look at the storage and operations for this subroutine now that we have described the function for multiplication. For operations, we will have the same amount of operations as Subroutine 1 due to the multiplication being almost identical, the only difference is the way L is stored. The storage for Subroutine 2 will be smaller than subroutine 1 since we are not storing the full matrix, rather only the elements below the diagonal. Since L is now stored as a 1D array with elements only consisting below the diagonal, we now have storage of n elements for m and v and S for crv, so the total storage is now equivalent to $\frac{n^2+3n}{2}$ which is definitely less than $n^2 + 2n$. So this subroutine not only decreases storage from Subroutine 1 but also reduces operations again from a full Matrix-Vector Multiplication.

IV.I.3 Subroutine 3

Subroutine 3 now also builds off of Subroutine 1 and 2 by L not only being a unit lower triangular matrix but also being a 2-bandwidth banded matrix. The code for Subroutine 3 is a little different than the first 2 but takes similar ideas. We store L in a 2D array instead of a 1D array because of the bands. I will again layout the steps simply to help explain the routine.

1. First we generate a 2-bandwidth banded matrix, B that has the 2 bands on the first two sub-diagonals and everywhere else is 0 and the diagonal is 1.
2. Then take B and put the matrix into the function that stores B in W where W only consists of the 2 sub-diagonal elements. Within this I also shift the second sub-diagonal values where I match up the corresponding columns with rows, as given in the example.
3. This W is now inputted into the function created for multiplication.
   (a) Initialize the vector m
   (b) Setup our for loop to look over the rows of m
   (c) Since we did not store the diagonal, we take our vector, v and accumulate the values from v into m since v would be multiplied by 1 along the diagonal
   (d) Check to see if first sub-diagonal exists and if it does, the algorithm multiplies the corresponding element in $W[1]$ with v
   (e) Check to see if second sub-diagonal exists and if it does, the algorithm multiplies the corresponding element in $W[0]$ by v

The storage for Subroutine 3 is going to be even less than the previous two because of the structure of a 2-bandwidth banded matrix. If we look at W, W has two rows, where $W[0]$ for the second sub-diagonal and $W[1]$ for the first sub-diagonal. The amount of storage in $W[0]$ is $n-2$ elements and $W[1]$ has $n-1$ elements. So, W has storage of $2n-3$ elements. Vector v and m both have n elements since they are the random generated and resulting vectors, respectively. Total storage for Subroutine 3 would consist of $4n-3$ elements which is significantly less than both Subroutine 1 and 2. For the amount of operations, this Subroutine involve only accessing a linear amount of elements from W, which will make this Subroutine not have any n^2 operations. For additions, we have the diagonal additions, n, the first sub-diagonal, $n-1$, and the second sub-diagonal, $n-2$, which makes for a total of $3n-3$ additions. For multiplication, we have the first sub-diagonal multiplications $n-1$ and the second sub-diagonal multiplications $n-2$ for a total of $2n-3$ multiplications. So we now have a total of $5n-6$ operations for this Subroutine.

IV.I.4 Subroutine 4

Subroutine 4 is different than the other subroutines in the fact that instead of Matrix-Vector multiplication or something similar, we are now computing the Matrix-Matrix Multiplication where L is a Unit Lower Triangular Matrix and U is an Upper Triangular Matrix. We then combine L and U into one matrix, LU where we will iterate through and compute the Matrix Product inside LU.

1. First again, we generate L and we generate U.
2. Then define a function where we take all elements of L that are below the diagonal and all elements of U that are on and above the diagonal and store them in LU
3. We then take LU and put it into our function that will compute the Matrix-Matrix product inside of LU.
   (a) We first initialize our resulting matrix M
   (b) We then set up our nested for loop that will iterate over M and LU.
   (c) Inside the nested for loop we first handle the case of the diagonal where we initialize M with the elements of LU on the diagonal. We then iterate over the range of the rows and multiply the Lower Triangular part by the Upper Triangular Part.

(d) After handling the diagonal case we then go into the case of handling the Lower Triangular where we iterate over the columns, $k+1$ and multiply corresponding parts.

(e) Lastly, we handle the case of the Upper triangular part. We first again initialize M with the values of LU since the first row of M is the first row of LU. We then iterate over the range of the rows and multiply corresponding parts.

This function leads the correct multiplication of L and U when they are stored together in LU. The storage for this function is fairly easy to handle since LU is a dense matrix. This means that the storage required for LU is n^2. The storage required for M then is also n^2 as M is also a dense matrix. The operations for this case deal with Matrix-Matrix multiplication but will not be n^3 operations which is a reduction from general Matrix Multiplication. The amount of operations needed for the diagonal case is n^2. This is because we are taking a Matrix-Vector product in the diagonal and having n additions. So combined we have $n + (n-1)n$ operations for the diagonal. For the Lower Triangular Case, we have i which ranges from $k+1$ to $n-1$, then for each k we have $k+1$ operations for each i. This gives us a sum of sums of $2(k+1)$. This will eventually end up being $2n^2 - 2n$ operations. For the upper triangular part, we will again have a sum of sums giving us $\frac{n^2}{2}$ operations in total. Total operations for this routine therefore, will have $3n^2$ and all other n terms 0 out giving us $3n^2 + O(n)$.

V Description of Experimental Design and Results

V.I Tester for Subroutine 1, 2, 3

For the testers for the first 3 Subroutines, the motivation behind setting up the testers were computing the manual algorithms and comparing them to NumPy's dot product routine. To do this, I had to go through steps to start generating problems, matrices and vectors in this case, that were specific to the subroutine. I wanted to be able to test multiple runs of different dimensions in one sitting and then plot the calculated Euclidean norms and taking the mean for the given dimension over n times. I also thought, since the difference should theoretically always be 0 unless there is machine error due to rounding of floating points, that I could look at the execution times and compare the mean of the execution times against the computed and true given by NumPy. The steps I took in developing the tester were:

1. Initialize the different lists that will hold the overall values.
2. Created a for loop to run n times over each dimension.
3. I nested a for loop in the previous step to calculate for each run of the dimension, where most of the calculation occurs.
   (a) I take in the parameters needed for each Subroutine's Function.
   (b) I setup the timing algorithm using the *time* package in python that will calculate the start and end time and the difference to compute the computation time.
   (c) I did this as well for the NumPy algorithm function that takes in the general matrix used for each routine and calculated the time.
   (d) I then calculated the norms using the Euclidean Norm for vectors to measure the difference in size between the two computations.
   (e) I then appended the lists to add each norm and difference norm to their corresponding list.
   (f) After the nested for loop was done running for the number of tests given for the dimension, the mean would be calculated and stored into the general list.
4. After the runs were done, I would plot the norms on top of each other to show difference and the difference list graphed as well. Side by side with that would be the execution time graph that shows over dimensions, the time in milliseconds of the computation time and graphed together the Manual and NumPy algorithms.

These steps were generalized for the first 3 subroutines but a separate one was used for each. This was for ease of use until I could get 1 tester for all 4 subroutines. This general idea was also the motivation for Subroutine 4 which was slightly different due to being a Matrix-Matrix Multiplication.

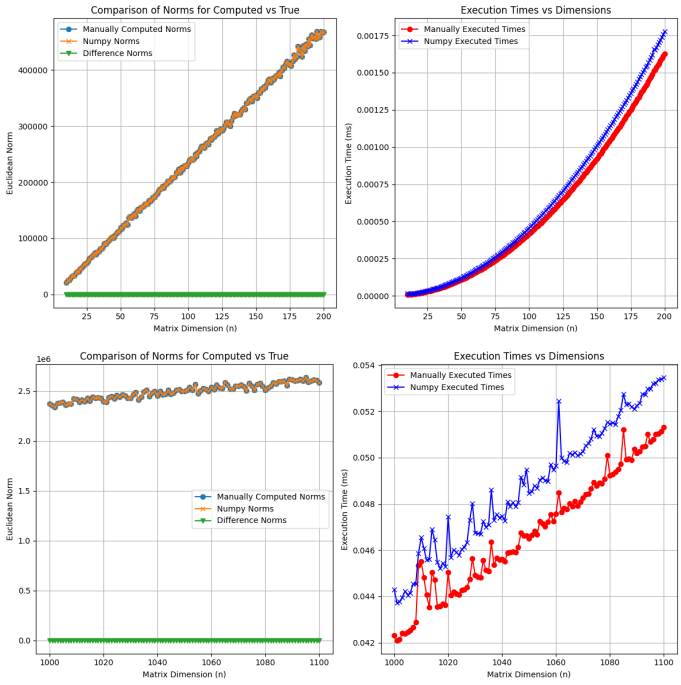
V.II Tester for Subroutine 4

The difference between this tester and the other 3 is that this tester had to generate two different matrices, L and U with their respective structure. Other than this, the same general structure follows as before. I also had to change the NumPy algorithm to now take the inputs of 2 matrices instead of a matrix and a vector.

V.III Results

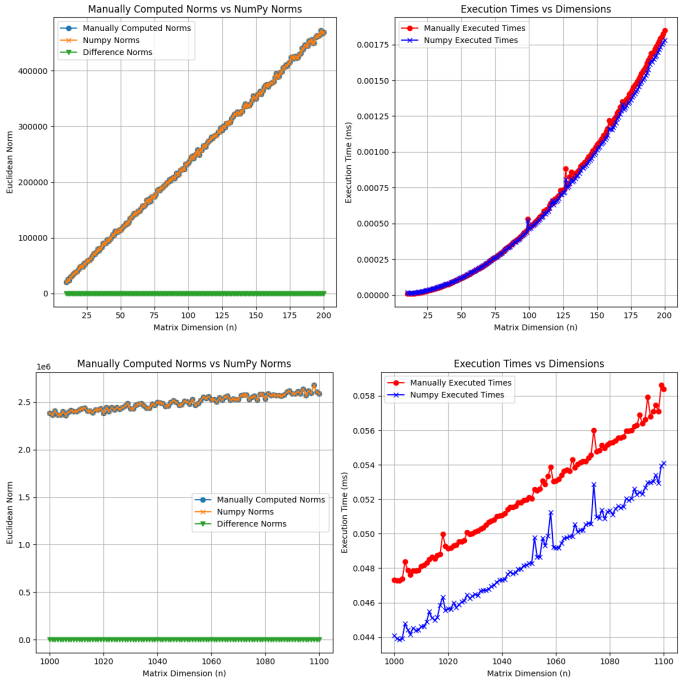
To design the results and testers for each subroutine, there were a variety of matrices of different dimensions taken along with random inputs from -100 to 100 to show negative and positive values. Each run usually consists of hundreds of matrices being tested for each dimension given the setup of the tester. The first of two plots for each subroutine calculates dimensions from 10 to 200 with 50 runs per dimension. The second two figures compute dimensions from 1000 to 1100 with 10 runs per dimension. This way, I have a range of dimensions to comapre results to. All code, testers, and graphs have come from PyCharm using Python with associated libraries where applicable.

V.III.1 Results and Figures for Subroutine 1



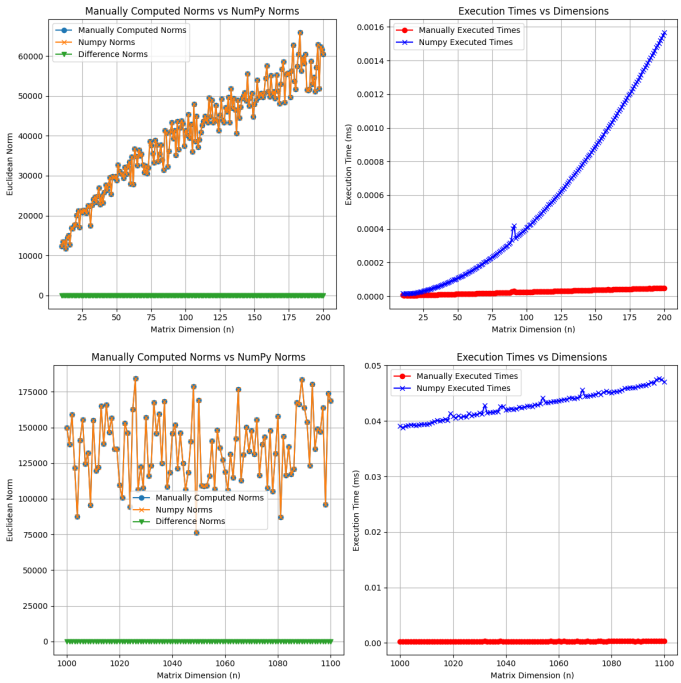
The two Figures above, compare the Euclidean Norms for the Computed Algorithm vs the NumPy Algorithm (True) and show the differences in the Norms. The right two figures show Execution Times vs the Dimension for NumPy's algorithm and the Manual Algorithm with time measured in milliseconds. The norms are what we expect to see which is zero given the inputs as integers. If we were to do floating points, we would see that there would be some error due to machine error and only being able to store 16 bits. The interesting find here is the Execution times. We see that the storage and operation reduction actually does contribute to execution times, more so for larger dimensional matrices. Towards the lower end we see that the two algorithms are very similar.

V.III.2 Results and Figures for Subroutine 2



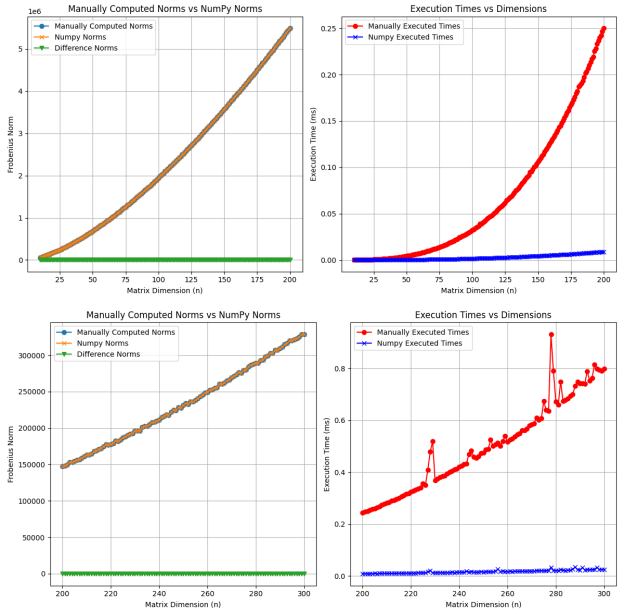
As we see similar results for Subroutine 2 as we did Subroutine 1 for the computed norms and differences to be zero. For execution tmies though, we see our Algorithm actually have longer Execution than NumPy's Dot Product. This one our algorithm seems to do better than NumPy for low dimensional matrices as since in the top right figure.

V.III.3 Results and Figures for Subroutine 3



This is what I expected would happen with the Banded Matrix. The computing time is almost nothing for both low and high dimensional matrices for our algorithm due to the operations being linear and not quadratic as we can see in the NumPy algorithm. We still see that our algorithm holds true for computation purposes as well in seeing no difference in norms.

V.III.4 Results and Figures for Subroutine 4



This is not what I expected to see when looking at Subroutine 4. We see that our algorithm was much slower than NumPy by a significant amount. We still obtain the same values and we may use less storage but computational time was very much affected by the way I set up the algorithm. The second plot here only goes from Matrix dimensions of 200 to 300 due to the computation tmie. When trying to use larger and larger dimensions, the amount of time to run per 1 run was too long. For time purposes, the dimensions were shrunk but we see the same results from dimension 10 to 200.

VI Conclusion

Within this report, we have seen how structure of matrices and exploitation of data structures within code can impact storage space for variables, the amount of operations and the reduction of this, and the computational time to complete each. Within each subroutine, we have shown that there are smart ways to handle matrices given the structure is spotted beforehand, or given. Specifically we see how to take a 2D array and push it into a 1D array to take less storage and save some computational time for lower dimensional matrices. There may be some flaws within the LU computation given the immense difference in computed times between the manual algorithm and NumPy's algorithm. This report has also concluded that given a banded matrix, the computation can be almost nothing, going from non-linear to linear operations. This was also the fastest out of the different Subroutines, but that was expected by the structure of a 2-bandwidth banded matrix.

VII Program Listing

All files can be compiled in any IDE that can read python. Each subroutine is designated by the file name with instructions once run for user inputs for Minimum Dimensions, Maximum Dimensions, Number of Runs per Dimension, Lower Bound for Random Integers, and Upper Bound for Random Integers. Each file also contains short description of the function per subroutine to describe the parameters and the output of the given function.