

# FCM Programming Assignment 1

Jonathan Cushman

September 2025

## 1 Executive Summary

In this report I will use 4 algorithms to compute matrix-matrix and matrix-vector products. I will use the structure of the mathematical objects for computational efficiency and evaluate their performance as the number of dimensions  $n$  grows

## 2 Statement of the Problem

In this assignment, we are supposed to solve the matrix-vector product  $Lv \rightarrow z$  along with matrix-matrix multiplication  $LU = M$  using a variety of inner, outer, or middle product techniques.

1. Subroutine 1 takes in a vector  $v$  and a matrix  $L$ , where  $L$  is a unit lower triangular matrix.
2. Subroutine 2 similarly takes in a vector  $v$  and a matrix  $L$ , where  $L$  is a unit lower triangular matrix, but differs in the fact that we are now compressing the nonzero elements below the unit diagonal of  $L$  into  $l$ . Re-framing the equation:  $lv \rightarrow z$
3. Subroutine 3 once again computes  $Lv \rightarrow z$  except in this routine  $L$  takes the form of a banded unit lower triangular matrix, with bandwidth 2
4. Subroutine 4 computes the product  $M = LU$  where we are given a matrix  $A$  and need to compute its product via  $L * U$  where  $L$  is a unit lower triangular matrix and  $U$  is an upper triangular matrix.

In the next section we will describe the structure of the mathematical objects and see how we can manipulate them for computational efficiency

## 3 Description of the Mathematics

### 3.1 Unit Lower Triangular Matrix Vector Multiplication

Consider a case with dimension  $n = 3$ : we compute each  $z_i$  by taking the dot product of  $L$  with the vector  $v_i$  since  $L$  is unit lower triangular, each row contains a 1 on the diagonal and possibly nonzero entries on the sub-diagonals.

$$\text{Let } z = Lv, \quad L \in R^{n \times n}, \quad v \in R^n$$
$$\text{where } L = \begin{bmatrix} 1 & 0 & 0 \\ \lambda_{21} & 1 & 0 \\ \lambda_{31} & \lambda_{32} & 1 \end{bmatrix} v = \begin{bmatrix} \gamma_1 \\ \gamma_2 \\ \gamma_3 \end{bmatrix} \rightarrow z = \begin{bmatrix} \gamma_1 \\ \gamma_1 \lambda_{21} + \gamma_2 \\ \lambda_{31} \gamma_1 + \lambda_{32} \gamma_2 + \gamma_3 \end{bmatrix}$$

In subroutine 2 we are doing the same thing mathematically, except this time we are only going to store the nonzero values below the unit diagonal.

$$\text{Let } z = lv, \quad l \in R^n, \quad v \in R^n$$
$$\text{where } l^T = [\lambda_{21} \quad \lambda_{31} \quad \lambda_{32}] v = \begin{bmatrix} \gamma_1 \\ \gamma_2 \\ \gamma_3 \end{bmatrix} \rightarrow z = \begin{bmatrix} \gamma_1 \\ \gamma_1 \lambda_{21} + \gamma_2 \\ \lambda_{31} \gamma_1 + \lambda_{32} \gamma_2 + \gamma_3 \end{bmatrix}$$

The advantage to this method is that we are only storing elements that will be necessary for computation, allowing us to save space on our machine.

### 3.2 Banded Unit Lower Triangular Matrix Vector Multiplication

In this case only the elements in the 2 sub-diagonals below the main unit diagonal contain the nonzero elements of matrix. These are also the only parts necessary for computation. Consider the case where  $n = 4$ :

$$\text{Let } z = vL_B, \quad L_B \in R^{n \times n}, \quad v \in R^n$$

$$\text{where } L_B = \begin{bmatrix} 1 & 0 & 0 & 0 \\ \lambda_{21} & 1 & 0 & 0 \\ \lambda_{31} & \lambda_{32} & 1 & 0 \\ 0 & \lambda_{42} & \lambda_{43} & 1 \end{bmatrix} v = \begin{bmatrix} \gamma_1 \\ \gamma_2 \\ \gamma_3 \\ \gamma_4 \end{bmatrix} \rightarrow z = \begin{bmatrix} \gamma_1 \\ \gamma_1 \lambda_{21} + \gamma_2 \\ \lambda_{31} \gamma_1 + \lambda_{32} \gamma_2 + \gamma_3 \\ \lambda_{42} \gamma_2 + \lambda_{43} \gamma_3 + \gamma_4 \end{bmatrix}$$

From this example we are able to derive the formula  $z_i = \gamma_i + L_{i,i-1}\gamma_{i-1} + L_{i,i-2}\gamma_{i-2}$  with the condition that  $i - 1 \geq 1$  and  $i - 2 \geq 1$

### 3.3 Matrix-Matrix Multiplication using Middle Product

Assume we are given a matrix  $A$ , that we want to separate into a unit lower triangular matrix  $L$ , and an upper triangular matrix  $U$ . Consider the case where  $n = 3$

$$\text{where } A = \begin{bmatrix} \lambda_{11} & \lambda_{12} & \lambda_{13} \\ \lambda_{21} & \lambda_{22} & \lambda_{23} \\ \lambda_{31} & \lambda_{32} & \lambda_{33} \end{bmatrix} \text{ can be separated into } L = \begin{bmatrix} 1 & 0 & 0 \\ \lambda_{21} & 1 & 0 \\ \lambda_{31} & \lambda_{32} & 1 \end{bmatrix} \quad U = \begin{bmatrix} \lambda_{11} & \lambda_{12} & \lambda_{13} \\ 0 & \lambda_{22} & \lambda_{23} \\ 0 & 0 & \lambda_{33} \end{bmatrix}$$

Then use the middle product  $\sum_{k=1}^n L_{ik}U_{kj} = M_{ij}$  to compute our desired result. Given the structure of  $L$  and  $U$  we notice that for within our Matrices, only values where  $i, j \geq k$  have nonzero values for the respective matrices. thus we can take advantage of this fact by using taking  $n = \min(i, j)$ . We can do this because  $L_{ik} = 0$  for  $k > i$  and  $U_{kj} = 0$  for  $j < k$  causing the sum to reduce to  $k = 1, \dots, \min(i, j)$  This makes it so that we are not wasting time computing elements of  $M_{ij}$  that include zeros, making our algorithm run faster than if it would have included every single element.

## 4 Description of the Algorithm and Implementation

A note beforehand that each algorithm is column oriented rather than row. This means we will first loop over columns  $j$  rather than  $i$  rows.

### 4.1 Unit Lower Triangular Matrix-Vector Product

The first algorithm is computing the matrix vector product. Given that we know our first element of  $z$  is going to always be the first elements in our vector  $v$  we initialize the value to begin with and then set a conditional to move onto the next element of our desired output vector  $z$ . This incrementally fills the next element of  $z$  with the remaining computations necessary. This algorithms runs with  $O(n^2)$  complexity

Listing 1: Algorithm 1

```

1 %subroutine 1
2 function [z]=Lvmult_col(M,v,n)
3 % Initialize the output vector z
4 z=zeros(n,1);
5
6 for i = 1:n
7     z(i) = z(i) + v(i);

```

```

8
9     if i < n
10         z(i+1:n) = z(i+1:n) + M(i+1:n,i) * v(i);
11     end
12 end
13 return

```

Similar to algorithm 1, the second subroutine is computing the same result as earlier, but instead of indexing over elements of the Matrix, we instead store the nonzero elements in a single array and then compute the product similarly but this time need to account for the necessary elements being indexed differently. We set a variable  $p = 1$  to index between for loops to ensure indexing stays consistent for matrices of large sizes. This algorithm also runs with complexity  $O(n^2)$

Listing 2: Algorithm 2

```

1 %subroutine 2
2 function [z] = Lvmult_col_compressed(M,v,n)
3 z = zeros(n,1);
4 % p is used as counter variable that gets incremented
5 p = 1;
6 % l is the compressed elements of the L Unit Lower Triangular matrix
7 l=[];
8 %loop to store the elements - Made with Alex
9     for j = 1:n
10         for i = j+1:n
11             l = [l;M(i,j)];
12         end
13     end
14 %algorithm built with Jonathan to compute the product
15     for j = 1:n
16         z(j) = z(j) + v(j);
17
18         for i = j+1:n
19             z(i) = z(i) + l(p) * v(j);
20             p = p + 1;
21         end
22     end
23 end

```

## 4.2 Banded Unit Lower Triangular Matrix Vector Product

The third algorithm first requires we store the lower bands of the Matrix L into two 1-D arrays to store the elements necessary for computation. After storing the nonzero values we follow the derived formula, which accesses the bands with only elements necessary for computation stored. This algorithm runs with  $O(n)$  complexity

Listing 3: Algorithm 3

```

1 %subroutine 3 - helped build with vincent, and Alex
2 function [z] = Lvmult_col_banded(M,v,n)
3     z = zeros(n,1);
4     band1 = zeros(n,1);
5     band2 = zeros(n,1);
6
7 % Note we are only storing 2 Bands max
8 % If L is a unit lower triangular matrix with its other
9 % potentially nonzero locations restricted to its two subdiagonals
10
11 %Storage for first subdiagonal
12 for i =2:n
13     band1(i) = M(i,i-1);

```

```

14     end
15
16     %storage for second subdiagonal
17     for i =3:n
18         band2(i) = M(i,i-2);
19     end
20
21     %this for loops computes:
22     % z(i) = z(i) + band1(i)*v(i-1) + band2(i)*v(i-2)
23
24     for i=1:n
25         z(i) = v(i);
26         if i-1 >= 1
27             z(i) = z(i) + band1(i)*v(i-1);
28         end
29
30         if i-2 >=1
31             z(i) = z(i) + band2(i)*v(i-2);
32         end
33     end
34 end

```

### 4.3 Unit Lower Triangular and Upper Triangular Matrix product

For the final subroutine, first assume you are given a matrix A which is dense with values in every spot. We then take this matrix and decompose into U and L. We then compute the product using the middle product. The algorithm starts with the index k:n for the number of necessary computations but with the addition of checks within the program we are able to shrink k to the  $\min(i, j)$ . This way we don't compute unnecessary values that include elements equal to zero which will not have an affect on our final answer. The algorithm runs with complexity  $O(n^3)$

Listing 4: Algorithm 4

```

1 %subroutine number 4
2
3 function [A] = LUmult(M,n)
4
5     % Initialize result
6     A = zeros(n,n);
7
8     % Loop over the "middle index" k
9     for k = 1:n
10
11         % For rows i >= k
12         for i = k:n
13             if i == k
14                 L(i,k) = 1;
15                 % L(k,k) = 1 on the unit diagonal
16             else
17                 L(i,k) = M(i,k);
18                 % Nonzero elements of L stored below diagonal
19             end
20
21             % For cols j >= k
22             %index starts at K because this is inside another for loop,
23             %thus k will start at 1
24             for j = k:n
25                 %store the nonzero elements of U, aka when j >= k by math
26                 U(k,j) = M(k,j);
27

```

```

28         %Summation of L*U, this is included here because i and j
29         %are both following the condition i,j <= k
30         A(i,j) = A(i,j) + L(i,k) * U(k,j);
31     end
32 end
33 end
34 end

```

## 5 Description of the Experimental Design and Results

To ensure the algorithms work properly we need to test them on a large number of cases to ensure our operations are working as intended for sufficiently large  $n$ . In this project we were given the driver file which allowed for testing in exactly this situation. The design of the experiment is to compare the errors given by our algorithm tested against the prebuilt functions in Matlab for sufficiently large cases of  $n$  and compare the absolute and relative error between the computations. Each nonzero entry of the matrix and vector that get input into our functions is chosen from a normal distribution with mean 0 and standard deviation 500. The number of dimensions  $n$  runs through a loop from  $n = 30 : 100$ . In the plotted results below, the solid line represents the max relative error while the dotted line represents the mean relative error.

$$\text{Define Absolute error} := \|x_{\text{comp}} - x_{\text{true}}\| \quad \text{Relative error} := \frac{\|x_{\text{comp}} - x_{\text{true}}\|}{\|x_{\text{true}}\|}$$

Figure 1: Mean and Maximum Relative Error for Algorithm 1 with 100 Dimensions

```

*****
Tests for size n = 100
*****
For 5 problems with n = 100  Mean Relative Error = 2.07e-16 Maximum Relative Error = 2.50e-16

```

Figure 2: Mean and Maximum Relative Error for Algorithm 2 with 100 Dimensions

```

*****
Tests for size n = 100
*****
For 5 problems with n = 100  Mean Relative Error = 1.86e-16 Maximum Relative Error = 2.26e-16

```

Figure 3: Mean and Maximum Relative Error for Algorithm 3 with 100 Dimensions

```

*****
Tests for size n = 100
*****
For 5 problems with n = 100  Mean Relative Error = 8.90e-17 Maximum Relative Error = 1.12e-16

```

Figure 4: Mean and Maximum Relative Error for Algorithm 4 with 100 Dimensions

```

*****
Tests for size n = 100
*****
For 5 problems with n = 100  Mean Relative Error = 1.58e-16 Maximum Relative Error = 1.66e-16

```

Figure 5: Max/Mean Relative Error Norms Plot for Algorithm 1  
normal element distribution with mean/deviation 0 and 500

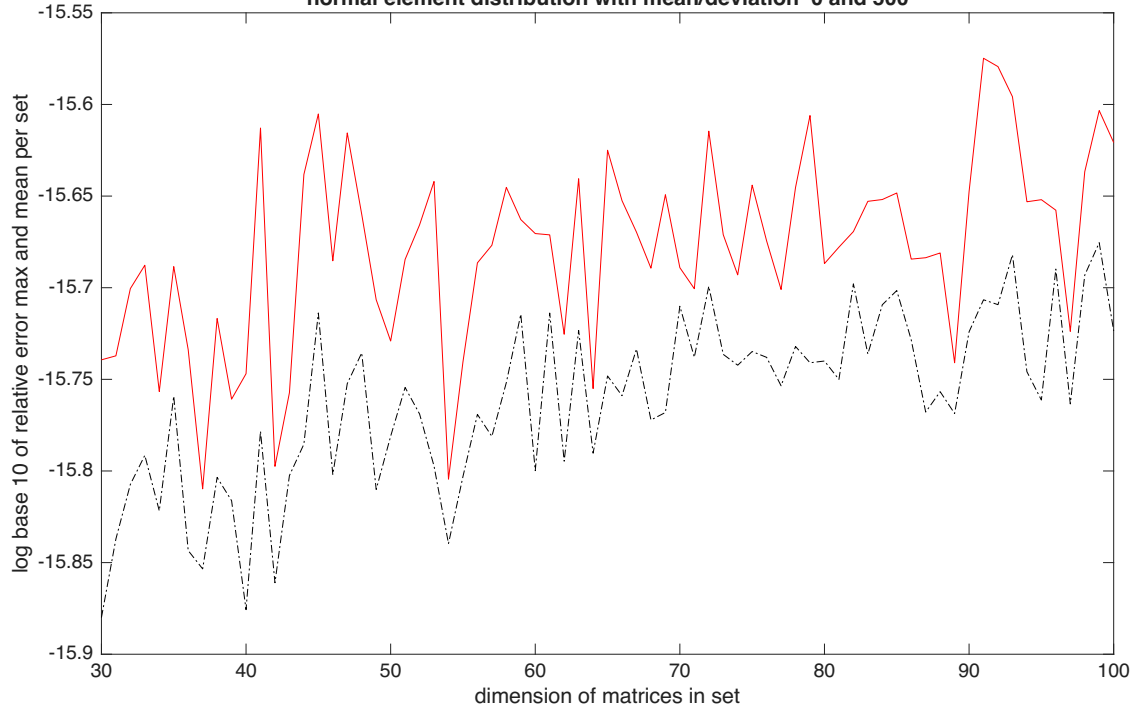


Figure 6: Max/Mean Relative Error Norms Plot for Algorithm 2  
normal element distribution with mean/deviation 0 and 500

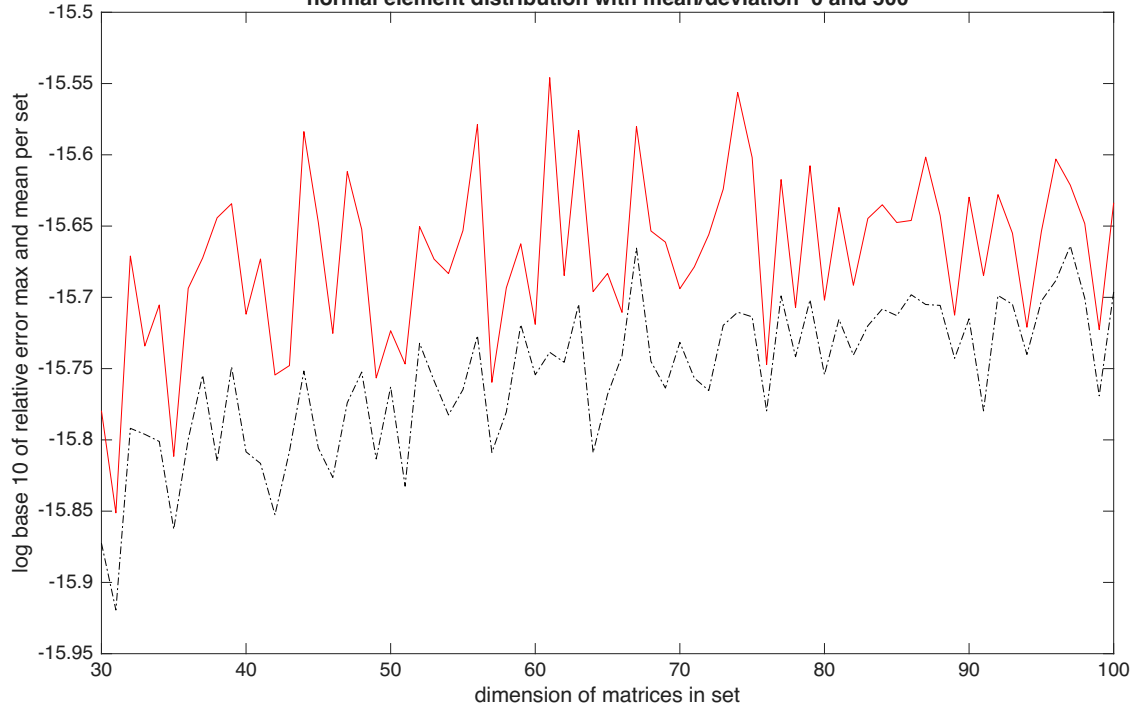


Figure 7: Max/Mean Relative Error Norms Plot for Algorithm 3  
normal element distribution with mean/deviation 0 and 500

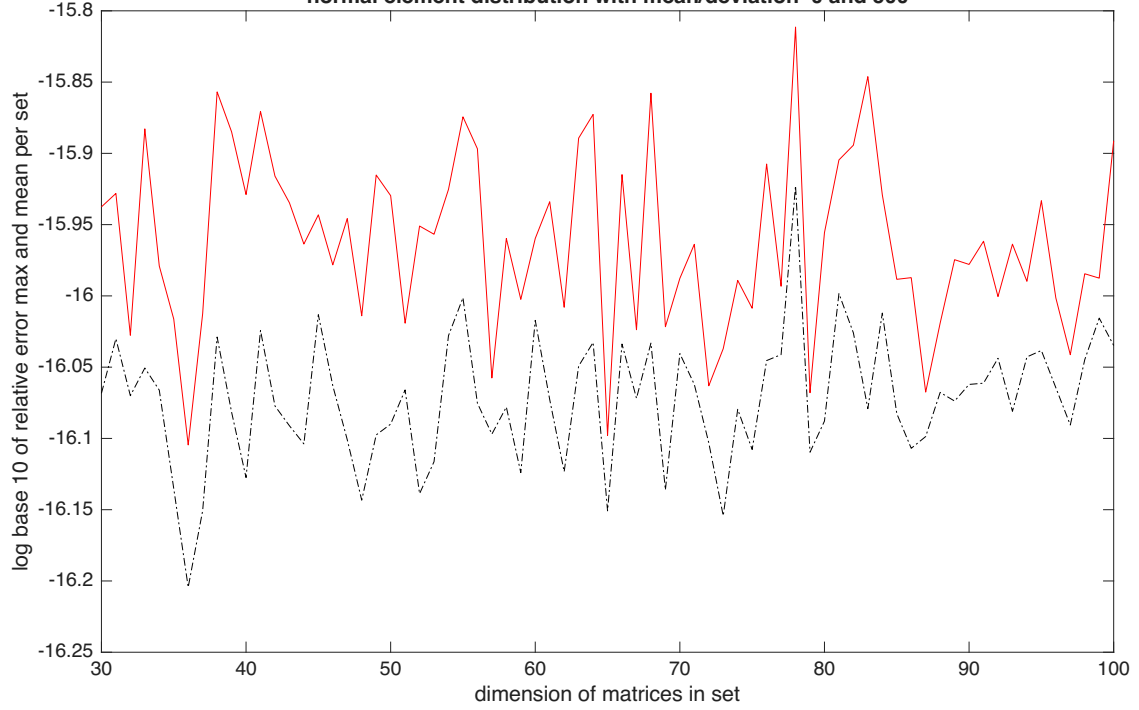
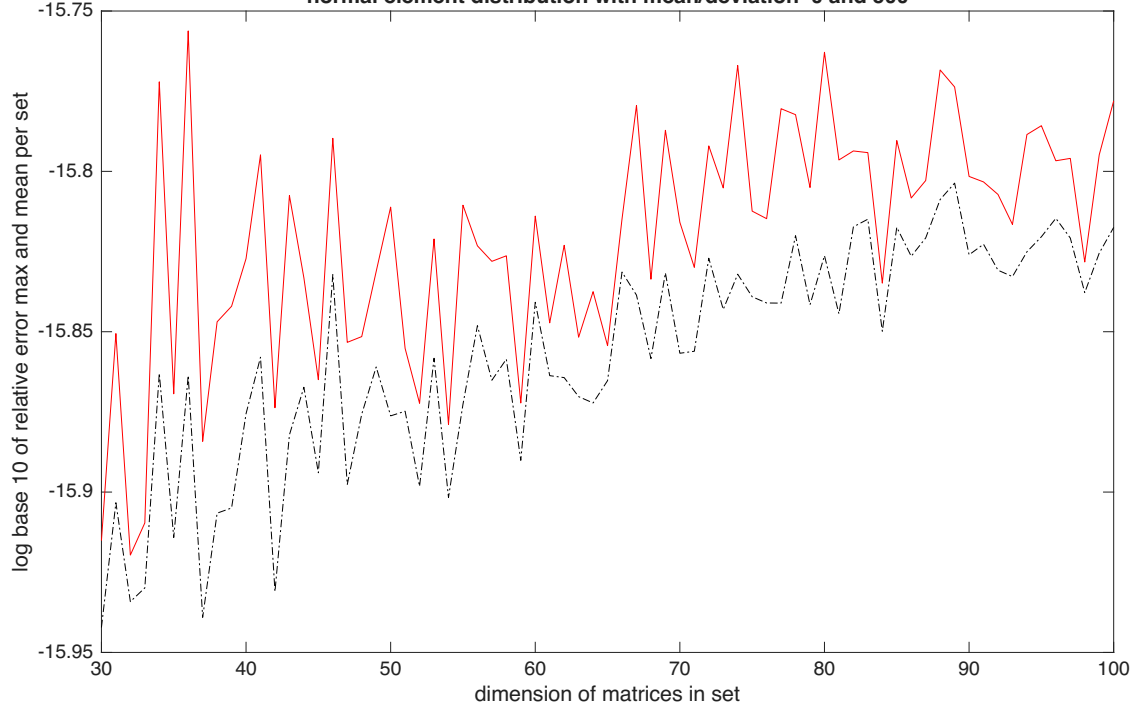


Figure 8: Max/Mean Relative Error Norms Plot for Algorithm 4  
normal element distribution with mean/deviation 0 and 500



As seen by the results, our error value is very small for each algorithm when tested for sufficiently large  $n$ , ie)  $n = 100$ . Thus we can claim that our algorithms is working as intended. Additionally, as seen in the graphed results, visually the trend is going upward. This is to be expected due to the fact that the number of computations needed to calculate the product for each algorithm increases as the number of dimensions

grow. This can be attributed to the floating point arithmetic associated with the stored values. When doing computations, machines are limited to what can be stored, thus there is a stopping point or rounding error that can be accounted for when producing numerical results. This is why our results are not perfect and the reasoning for the relationship between error and dimension size. A good way to think about this is in terms of complexity. For example, we know subroutine 3 has complexity  $O(n)$  which grows much slower with respect to subroutine 1 and 2 with complexity  $O(n^2)$ . Helping us gain some insight as to why the growth rate for the error increases faster for the later. We also notice that subroutine 3 returns the lowest error, which makes sense because it has the lowest number of operations needed to compute it product, thus less computations with floating point arithmetic leading to less change in error. This can also be seen in algorithm 4 which has the fastest and most noticeable increase in error growth rate, with complexity  $O(n^3)$ . This is somewhat obvious, as computing a product of matrices will require more operations necessary than a vector-matrix product due to the size of the mathematical objects.

## 6 Conclusions

The results from the testing indicate that our algorithms are correctly producing the product for both the matrix-vector and matrix-matrix multiplication with very low relative error. From the testing we notice from each graph that the mean/max error norm increases as our number of dimensions  $n$  increases. We can argue this is valid due to the fact that Matlab is restricted by floating point arithmetic. Thus each approximation will deviate further from the true value for each computation that is executed. The slight deviations up and down are caused from the random values generated inside each matrix, leaving us with some level of randomness when evaluating error. this is why if we were to run a linear regression through each resulting graph, we would notice a positive slope within each one, indicating that error is associated with the number of iterations being performed.

## 7 Program Files

In total there should be five Matlab(.m) files upload alongside this document. Each function is contained within it's own file. The fifth file is the driver file. At the top of the driver file is instructions on how to run each subroutine.