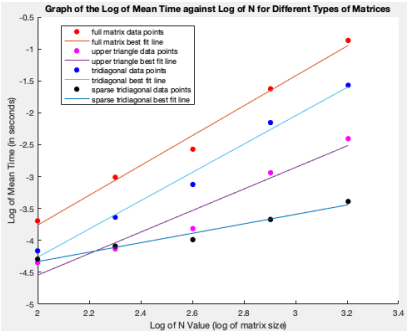
Computing Assignment #2 - Report

Part a)

N x N Matrix Size (N Value)	**T_{ave} for Full Matrix**(secs)	**T_{ave} for Upper Triangular Matrix**(secs)	**T_{ave} for Tridiagonal Matrix**(secs)	**T_{ave} for Sparse Tridiagonal Matrix**(secs)
100 x 100	0.00020253	0.00004766	0.000068619	0.000050877
200 x 200	0.00020253	0.000073463	0.00023014	0.000081169
400 x400	0.0026806	0.00015333	0.00075158	0.00010277
800 x 800	0.023764	0.0011492	0.0070136	0.00021308
1600 x 1600	0.13617	0.0039301	0.027048	0.00040721

Table 1: Data of the average time taken (T_{ave}) to solve a linear system using MATLAB's "\" command for different types of matrices and matrix sizes.

Since the matrix size and mean time taken to solve a linear system are related via some power law of the form y = a(x^p), we have log(y) = log(axp) = log(a) + p log(x), so log(y) depends linearly on log(x). With the matrix size N (x) and mean time T_{ave} (y) being related in this way, we can plot the power law relationship:



Graph 1: Log-log graph of the mean time against N.

Comments/Observations: This graph was made by taking the log of the data in Table 1 and using MATLAB's polyfit command to find a least squares best fit line to show that log(mean time) vs log(N) is approximately linear. From the graph, we can clearly see that a sparse tridiagonal matrix takes the shortest amount of time to solve, followed by an upper triangular, tridiagonal, and full matrix respectively. We can use the polyfit command to fit the data to give us c(1), or the value of the power law exponent p on y = a(x^p). This value can be used to approximate the Big O runtime for solving each type of matrix. After fitting the data, my results for c(1) for each matrix were as follows:
Full matrix: c(1) = 2.3390 ⇒ O(n^2.3390)
Upper triangular matrix: c(1) = 1.6879 ⇒ O(n^1.6879)
Tridiagonal matrix: c(1) = 2.2175 ⇒ O(n^2.2175)
Sparse tridiagonal matrix: c(1) = 0.7394 ⇒ O(n^0.7394)
We see that the polyfit line yields results that match the data in Graph 1, as the smaller the Big O, the faster the computation time.

Relation to Operation Counts: In lecture we learned that solving a full matrix requires gaussian elimination and back-substitution. We found that this is roughly equal to ⅔(n^3) for large n, meaning the Big O for solving a full matrix with gaussian elimination is O(n^3). This is rather inconsistent with my results from MATLAB, however this may be because I did not use a large enough N. I believe that as the N value approaches infinity, the Big O runtime taken to solve a full matrix should approach O(n^3). To support this claim, I found that when testing with larger N, my c(1) value would slightly increase.

We also learned that the total operation count for solving an upper triangular matrix is equal to the operation count of performing back-substitution, which is n^2. This means that the Big O is O(n^2), which is fairly consistent with my c(1) results (withthin ~0.3). In fact, it is faster, so it is possible that MATLAB could have a faster algorithm, or that once again as N approaches infinity, Big O approaches O(n^2).

In lecture, we discussed how the LU factorization algorithm can be applied to rapidly solve tridiagonal linear systems, with an operation count of O(n). At first I was confused as to why my results in MATLAB did not match the operation count we learned about in class. After some thinking, I realized that the only logical explanation for this is that when MATLAB tries to solve the tridiagonal matrix A_tri that I randomly generated through the diag command, MATLAB does not realize that matrix A_tri is actually a tridiagonal matrix. Instead, it treats matrix A_tri as a full matrix, and applies the same algorithm it uses to solve a full matrix's linear system, which is why the c(1) values for the full matrix and tridiagonal matrix are so similar (within ~0.1). However, when I randomly generated a sparse tridiagonal matrix A_sparse with the spdiags command, MATLAB is able to tell that matrix A_sparse is a (sparse) tridiagonal matrix, so it applies the LU factorization algorithm to solve it. Although my c(1) value for the sparse tridiagonal matrix is still ~0.3 away from O(n), I again noticed that when testing with larger N, my c(1) value would slightly increase, meaning I can assume that as N approaches infinity, the Big O runtime for solving a sparse tridiagonal matrix on MATLAB approaches O(n).

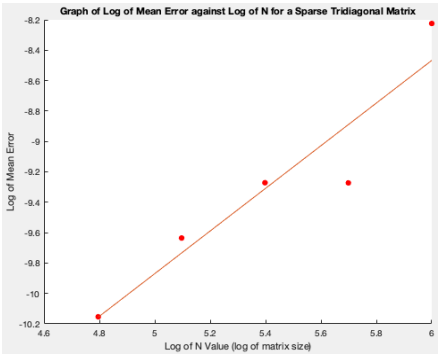
Part b)

N x N Matrix Size (N value)	**62500 x 62500**	**125000 x 125000**	**250000 x 250000**	**500000 x 500000**	**1000000 x 1000000**
Mean Error	7.0215e-11	2.3167e-10	5.3499e-10	5.3385e-10	5.9732e-09

Table 2: Data of the average round-off error in MATLAB's gaussian elimination "\" command for sparse tridiagonal matrices of different matrix sizes of large N.

To attain this data, I chose a range of 5 sufficiently large N values that doubled in size before maxing out at 1000000, and ran each matrix size over 100 trials. The error in gaussian elimination was computed in each of these trials, which was then used to find the mean error over 100 trials. I figured that with the exponential growth of these N values, they would be large enough to where I would be able to see the effect that round-off error has as N approaches infinity. I believe that 100 trials is a sufficient number of trials to where I could get an accurate representation of the error by minimizing the influence of outliers.

Again, by assuming the power law relationship between the mean error and matrix size (N), we can plot the log-log graph of mean error against N.



Graph 2: Log-log graph of the mean error against N.

Comments/Observations: From the graph, we can see that the error approximately grows linearly as N approaches infinity. However, there is also some variance in the error. This is evident when comparing the error on matrix sizes 250000 x 250000 and 500000 x 500000, as we can see there is a greater error on the 250000 x 250000 matrix. This is likely due to the way that the randomly generated sparse tridiagonal matrices are generated, as some may be well-conditioned, while others are ill-conditioned. Assuming a power law relationship, we can estimate the functional dependence on N Through the best fit line as y = 1.4026x - 16.8822, where y = log(mean error) and x = log(matrix size).

Number of Digits in the solution Reliable for N = 10^4: After running my part_b.m with matrix size N = 10^4, I went through each of the 100 trials' errors and found the largest error to be 9.412e-11. We can write this error out in standard form as 0.00000000009412. From this, we can count the number of decimal place zeros ahead of the first non-zero digit to find the number of reliable significant digits. We can see that a linear system with an N x N random tridiagonal coefficient matrix is approximately reliable to **10 significant digits** in the solution for N = 10^4.

Value of N for which the Computed Solution becomes Entirely Unreliable:
Since the x and y axes of Graph 2 are related via some power law of the form y = a(x^p), we can use MATLAB's polyfit command to find the values of p and a:
c(1) = **1.4026 = p**
c(2) = -16.8822 = log10(a) ⇒ **a = 1.3116e-17**
Let the mean error = E_{ave}. We need to solve for when $E_{ave}(N^*) = 1$, or equivalently, when our best fit function y = a(x^p) = 1 = a(N^p).
After plugging in our p and a values, we get:
1 = (1.3116e-17)(N^1.4026)
We can solve for N by applying some basic algebra to the above equation, which can be used to estimate our matrix size N*.
Thus we have **N = 1.0873 x 10^12**

Final Thoughts/Conclusion: Drawing on my experiments in part a), it is evident that as the N values approach infinity, the closer the Big O runtime of solving a linear system gets to the operation counts we discussed in class. It is clear that MATLAB is unable to identify matrices which are randomly generated using the diag command as tridiagonal matrices, so it treats them as a full matrix when solving. Although, when MATLAB is able to identity a matrix as tridiagonal, it can solve it very efficiently.

In part b), we see that there is some variance in the error, but the error still remains approximately linear to the matrix size. I would assume the variance decreases as we increase the number of trials, thus yielding a more accurate representation of the round-off error in gaussian elimination.