LU Factorization of Banded Matrices

February 17, 2017

1 Statement of the problem

Linear systems are some of the most commonly encountered problems in applied math. They are represented in matrix form by $Ax = b$ where x and b are vectors. Solving these systems with traditional Gauss-Jordan elimination is $O(n^3)$, but if A is well structured it can be done much quicker. In this paper we examine the LU-factorization of a banded matrix and the speed of solving a system with the factorization.

2 Description of the Mathematics

We are told in the problem that the matrix we have is diagonally dominant and 0 except on a 5 specific diagonals, i.e. A is of the form

$$\begin{pmatrix}
b_1 & c_1 & 0^* & 0^* & d_1 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 & 0 \\
a_1 & b_2 & c_2 & 0^* & 0 & d_2 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 & 0 \\
0^* & a_2 & b_3 & c_3 & 0 & 0 & d_3 & 0 & \cdots & 0 & 0 & 0 & 0 & 0 \\
0^* & 0^* & a_3 & b_4 & c_4 & 0 & 0 & d_4 & \cdots & 0 & 0 & 0 & 0 & 0 \\
e_1 & 0 & 0 & a_3 & b_4 & c_4 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 & 0 \\
0 & e_2 & 0 & 0 & a_3 & b_4 & c_4 & 0 & \cdots & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & e_3 & 0 & 0 & a_3 & b_4 & c_4 & \cdots & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & e_4 & 0 & 0 & a_4 & b_5 & \cdots & 0 & 0 & 0 & 0 & 0 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & c_{n-5} & 0 & 0 & d_{n-5} & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & b_{n-4} & c_{n-4} & 0 & 0 & d_{n-4} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & a_{n-4} & b_{n-3} & c_{n-3} & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 & a_{n-3} & b_{n-2} & c_{n-2} & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & a_{n-2} & b_{n-1} & c_{n-1} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & e_{n-4} & 0 & 0 & a_{n-1} & b_n
\end{pmatrix}.$$

Because of the large number of 0s, A can be more easily represented as a vector of its 5 non-zero diagonals as shown below:

$$\begin{pmatrix}
d_1 & d_2 & d_3 & \cdots & d_{n-5} & d_{n-4} & 0 & 0 & 0 & 0 \\
c_1 & c_2 & c_3 & \cdots & c_{n-5} & c_{n-4} & c_{n-3} & c_{n-2} & c_{n-1} & 0 \\
b_1 & b_2 & b_3 & \cdots & b_{n-5} & b_{n-4} & b_{n-3} & b_{n-2} & b_{n-1} & b_n \\
a_1 & a_2 & a_3 & \cdots & a_{n-5} & a_{n-4} & a_{n-3} & a_{n-2} & a_{n-1} & 0 \\
e_1 & e_2 & e_3 & \cdots & e_{n-5} & e_{n-4} & 0 & 0 & 0 & 0
\end{pmatrix}.$$

Because A is banded, we know that L and U will also be banded. However they will not have the same structure of 0s between diagonals and will be densely banded.

They are each guaranteed only 3 0s near the beginning, in the entries marked * in the matrix above. This is simply because of how L and U are generated: There is nothing to eliminate in those entries on L, and there is nothing above them to fill them in in U.

Representing L and U as only their bands, we can represent them together in a matrix reminiscent of the one for A. In the following matrix, the first 5 rows correspond to the diagonals of U and the bottom 4 correspond to L. We only need 4 diagonals for L because its major diagonal will be all ones.

$$\begin{pmatrix} d_1 & d_2 & d_3 & \cdots & d_{n-5} & d_{n-4} & 0 & 0 & 0 & 0 \\ 0 & z_2^* & z_3^* & \cdots & z_{n-5}^* & z_{n-4}^* & z_{n-3}^* & 0 & 0 & 0 \\ 0 & 0 & y_3^* & \cdots & y_{n-5}^* & y_{n-4}^* & y_{n-3}^* & y_{n-2}^* & 0 & 0 \\ c_1^* & c_2^* & c_3^* & \cdots & c_{n-5}^* & c_{n-4}^* & c_{n-3}^* & c_{n-2}^* & c_{n-1}^* & 0 \\ b_1^* & b_2^* & b_3^* & \cdots & b_{n-5}^* & b_{n-4}^* & b_{n-3}^* & b_{n-2}^* & b_{n-1}^* & b_n^* \\ a_1^* & a_2^* & a_3^* & \cdots & a_{n-5}^* & a_{n-4}^* & a_{n-3}^* & a_{n-2}^* & a_{n-1}^* & 0 \\ 0 & 0 & x_3^* & \cdots & x_{n-5}^* & x_{n-4}^* & x_{n-3}^* & x_{n-2}^* & 0 & 0 \\ 0 & w_2^* & w_3^* & \cdots & w_{n-5}^* & w_{n-4}^* & w_{n-3}^* & 0 & 0 & 0 \\ e_1^* & e_2^* & e_3^* & \cdots & e_{n-5}^* & e_{n-4}^* & 0 & 0 & 0 & 0 \end{pmatrix}.$$

In this representation, we have a constant 26 0s that would not need to be stored if the diagonal elements were stored in separate vectors. However, the computation convenience of the matrix form outweighs the storage space cost because the 0s will only make up a tiny fraction of the total entries in the matrix. It is a $nx9$ matrix, and n must be greater than or equal to 8 for A to have the full structure we expect so the zeros will make up at most $\frac{26}{72} \approx 36\%$ of the matrix. The fraction of the matrix that is zero is always $\frac{26}{9n} \approx \frac{3}{n}$, so if n is at least 30 the zeros will make up less than 10% of the storage space.

Finding L and U can also in linear time. Because there are at most only 4 nonzero elements below the diagonal, and at most 4 nonzero elements to the right of the diagonal, advancing to the next step only takes roughly 20 operations: at stage i only the $4x5$ section of the matrix starting directly below b_i needs to be updated. This must be repeated $n-1$ times; once for each row beyond the first.

This also allows for quickly solving systems of the form $Lv = f$ or $Uv = f$ with the L or U output from the system. Generally solving these systems is $O(n^2)$, but because of the banded structure in L and U they can be done in $O(4n)$ and $O(5n)$ computations respectively. They are each based off a simple 4-term linear recurrence. For $LV = f$, this recurrence is

$$v_i = f_i - a_{i-1}^* f_{i-1} - x_{i-2}^* f_{i-2} - w_{i-3}^* f_{i-3} - c_{i-4}^* f_{i-4},$$

where we are exploiting the fact that L is unit diagonal. The recurrence is started off with

$$\begin{aligned} v_1 &= f_1 \\ v_2 &= f_2 - a_1^* f_1 \\ v_3 &= f_3 - a_2^* f_2 \\ v_4 &= f_4 - a_3^* f_3 \end{aligned}$$

because of the three zeros from A that are guaranteed to not be filled in. The recurrence for $Uv = f$ is very similar but is started of at the end with $v_n = f_n / b_n^*$ and requires dividing by b_i^* because U is not unit diagonal. This is why our assumption

that A was diagonal dominant was important: If not, the b_i values can become very small and cause instability when dividing. If we do not have a diagonally dominant matrix, we can avoid some stability issues by implementing *partial pivoting* where we interchange rows to ensure sufficiently large diagonal values. Specifically, at each step i we search through the ith column to find the row with the largest value in that column, then interchange that row with the ith row before finding the Schur complement and proceeding. Searching the column requires only 4 comparisons, but by permuting the L and U matrices we lose their banded structure. There will still only be at most 4 or 5 non-zero elements in any column, but since we no longer know where they are we cannot use the same solving algorithm as before. The 6 zeros that remained from As 0-diagonals are also no longer preserved in the L and U factors. Additionally, we no longer have a guarantee that the updates will be limited to a $4x5$ section in the corner of the matrix because exchanging the row exchange introduces the non-zeros from the later diagonal elements to earlier rows so we lose most of the structure of A. This is illustrated below for an $8x8$ matrix. We assume $e_i > b_i$ and apply partial pivoting to demonstrate the structure lost in the first few steps applying immediate update.

$$A_0 = \begin{pmatrix} b_1 & c_1 & 0^* & 0^* & d_1 & 0 & 0 & 0 \\ a_1 & b_2 & c_2 & 0^* & 0 & d_2 & 0 & 0 \\ 0^* & a_2 & b_3 & c_3 & 0 & 0 & d_3 & 0 \\ 0^* & 0^* & a_3 & b_4 & c_4 & 0 & 0 & d_4 \\ e_1 & 0 & 0 & a_3 & b_4 & c_4 & 0 & 0 \\ 0 & e_2 & 0 & 0 & a_3 & b_4 & c_4 & 0 \\ 0 & 0 & e_3 & 0 & 0 & a_3 & b_4 & c_4 \\ 0 & 0 & 0 & e_4 & 0 & 0 & a_4 & b_5 \end{pmatrix} \qquad P_1 A_0 = \begin{pmatrix} e_1 & 0 & 0 & a_3 & b_4 & c_4 & 0 & 0 \\ a_1 & b_2 & c_2 & 0^* & 0 & d_2 & 0 & 0 \\ 0^* & a_2 & b_3 & c_3 & 0 & 0 & d_3 & 0 \\ 0^* & 0^* & a_3 & b_4 & c_4 & 0 & 0 & d_4 \\ b_1 & c_1 & 0^* & 0^* & d_1 & 0 & 0 & 0 \\ 0 & e_2 & 0 & 0 & a_3 & b_4 & c_4 & 0 \\ 0 & 0 & e_3 & 0 & 0 & a_3 & b_4 & c_4 \\ 0 & 0 & 0 & e_4 & 0 & 0 & a_4 & b_5 \end{pmatrix}$$

$$M_1^{-1} P_1 A_0 = \begin{pmatrix} e_1 & 0 & 0 & a_3 & b_4 & c_4 & 0 & 0 \\ a_1^* & b_2 & c_2 & z_2^* & w_2^* & d_2^* & 0 & 0 \\ 0 & a_2 & b_3 & c_3 & 0 & 0 & d_3 & 0 \\ 0 & 0 & a_3 & b_4 & c_4 & 0 & 0 & d_4 \\ b_1^* & c_1 & 0 & a_4^* & b_5^* & c_5^* & 0 & 0 \\ 0 & e_2 & 0 & 0 & a_3 & b_4 & c_4 & 0 \\ 0 & 0 & e_3 & 0 & 0 & a_3 & b_4 & c_4 \\ 0 & 0 & 0 & e_4 & 0 & 0 & a_4 & b_5 \end{pmatrix} \qquad P_2 M_1^{-1} P_1 A_0 = \begin{pmatrix} e_1 & 0 & 0 & a_3 & b_4 & c_4 & 0 & 0 \\ 0 & e_2 & 0 & 0 & a_3 & b_4 & c_4 & 0 \\ 0 & a_2 & b_3 & c_3 & 0 & 0 & d_3 & 0 \\ 0 & 0 & a_3 & b_4 & c_4 & 0 & 0 & d_4 \\ b_1^* & c_1 & 0 & a_4^* & b_5^* & c_5^* & 0 & 0 \\ 0 & e_2 & 0 & 0 & a_3 & b_4 & c_4 & 0 \\ a_1^* & b_2 & c_2 & z_2^* & w_2^* & d_2^* & 0 & 0 \\ 0 & 0 & e_3 & 0 & 0 & a_3 & b_4 & c_4 \\ 0 & 0 & 0 & e_4 & 0 & 0 & a_4 & b_5 \end{pmatrix}$$

At this point, we can already clearly see that the useful structure of L and U is lost outside of the limit of the number of non-zero entries per column. The recurrence will not be as basic as before, and so solving the Lv or Uv systems will be less efficient.

3 Description of the Algorithm and Implementation

The algorithms are implemented directly into Scala methods. The main factorization method takes the diagonals of A as arguments and outputs the matrix of diagonals of in the structure described above as a 2-dimensional array. Arrays are used for all data structures because they are mutable and so allow for immediate update to the parameters without needing to store a separate output structure. This means the original data of the matrix will be overwritten. The same implementation is used

for the L and U system solver: the input array is overwritten as the system is solved to minimize storage requirements.

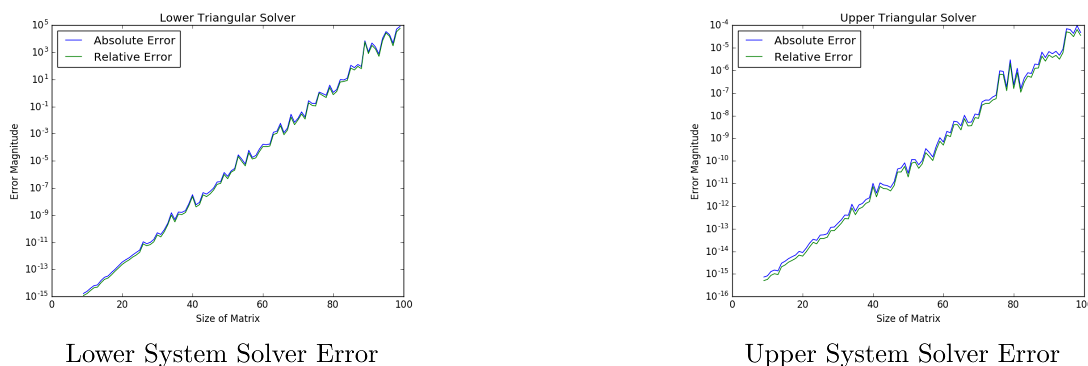
4 Description of the Experimental Design and Results

The triangular system solvers are test by randomly generating vectors to serve as their diagonals along with another randomly generated vector x. The matrix product $Lx = v$ or $Ux = v$ is computed, then the system solver is used to compute $\tilde{x}$ and compare it to x. To test the factorization method 5 vectors are randomly generated of specified lengths to represent the diagonals. To ensure diagonal dominance the entries for the main diagonal are randomly generated between the sum of the other entries of the column and three times their sum. Similarly to the method for testing the system solvers, a randomly generated vector x is multiplied by A to get $Ax = v$. Then A is factorized into L_A and U_A which are used to solve the system $L_A b = v$ and $U_A \tilde{x} = b$ to compare the computed $\tilde{x}$ with the original x. Because of limitations in the method for generating the diagonals, only matrices larger than $8x8$ are considered in the error analysis. Accuracy on these systems is tested for both The accuracy for smaller systems was verified with simple matrices, but on a small sample space. After accuracy was verified, the methods were timed to verify the expected execution time and complexity order.

5 Results

The results of the accuracy test for solving the $Lv = f$ and $Uv = f$ systems with matrices of size $9x9$ to $100x100$ are plotted below in Figure 1. Note that a logarithmic scale is required for this plot because of massive stability issues.

Figure 1

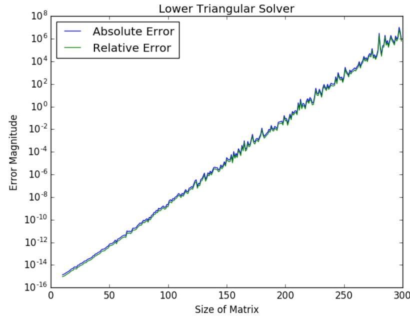


Lower System Solver Error

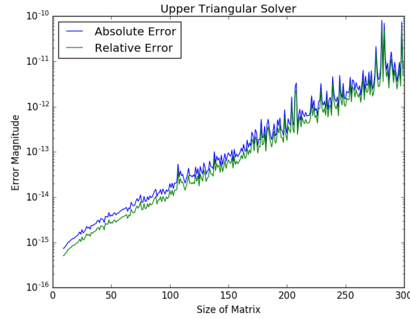
Upper System Solver Error

These stability issues are likely the result of catastrophic cancellation from adding numbers that are close in magnitude but opposite in sign. The matrices used to test the system solvers also did not have the same diagonally dominant structure as the matrices that would be factorized. The problem gets exponentially worse as n increases To help alleviate some of these issues, the accuracy was tested again with matrices that had all elements guaranteed to be positive. This time, matrices with sizes up to $300x300$ were plotted.

Figure 2



Positive Lower System Solver Error

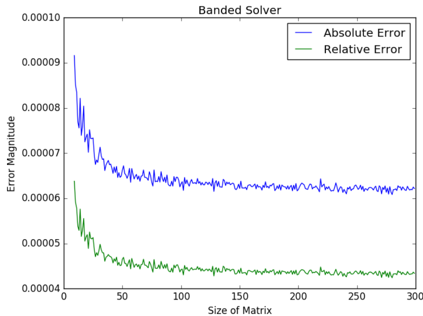


Positive Upper System Solver Error

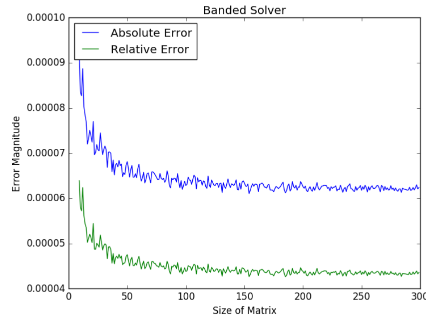
We observe that the Upper triangular system is much more stable than the lower triangular, but both are accurate within 10^{-10} for matrices smaller than $100x100$. The additional stability for the upper triangular system likely comes from the diagonal elements being non-unit: needing to divide by an element scales everything to be smaller, including the error. Repeated over 300 times causes the significant improvement in terms of error we observe.

In solving the system with a banded matrix as plotted in Figure 3, we have a very unexpected result: Error seems to decrease with matrix size and then level off. Additionally, no logarithmic scale is needed to view the error terms. However, they are also unfortunately very high, on the order of 10^{-5} even for small matrices. Compared with the lower triangular system solver, the order is very small. This is likely attributed to the structure of A providing additional structure to L and U that makes them more accurate. Additionally we note that adding a requirement that all entries of A be positive does not improve accuracy. This means that the errors in the general case are significantly lower than solving $Lv = f$ or $Uv = f$ with randomly generated triangular matrices, providing further evidence that the diagonal dominance of A is responsible for additional that adds stability to those computations

Figure 3



Banded Factorization System Solver Error

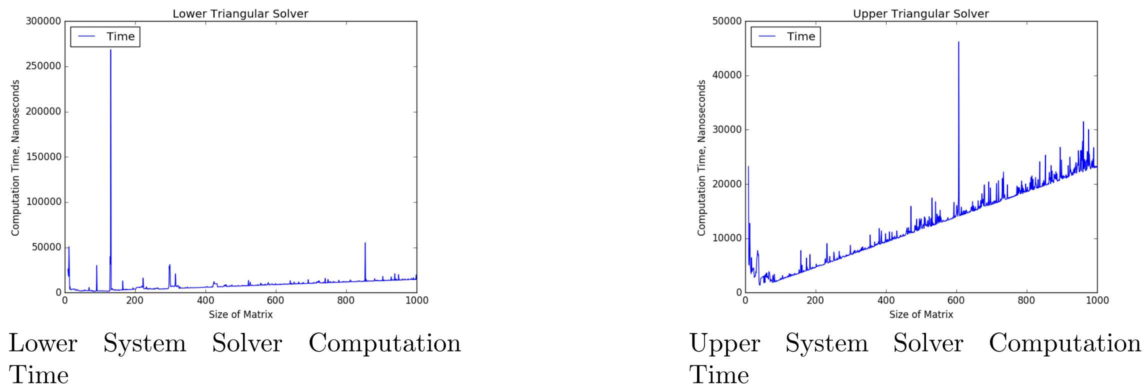


Positive Banded Factorization System Solver Error

In all cases we expect a linear relation between matrix size and computation time. With a few outliers, that is what we observe for the triangular system solvers in Figure 4.

The outliers in the lower system solver around $135x135$ systems make it appear

Figure 4



Lower System Solver Computation Time

Upper System Solver Computation Time

to almost be a constant time, but the computation time does increase gradually with system size. The trend is more clear in the upper system solver, which has a less significant outlier problem.

When computing the computation time for the factorization speed, we once again have a surprising result: Time is constant across matrix size.

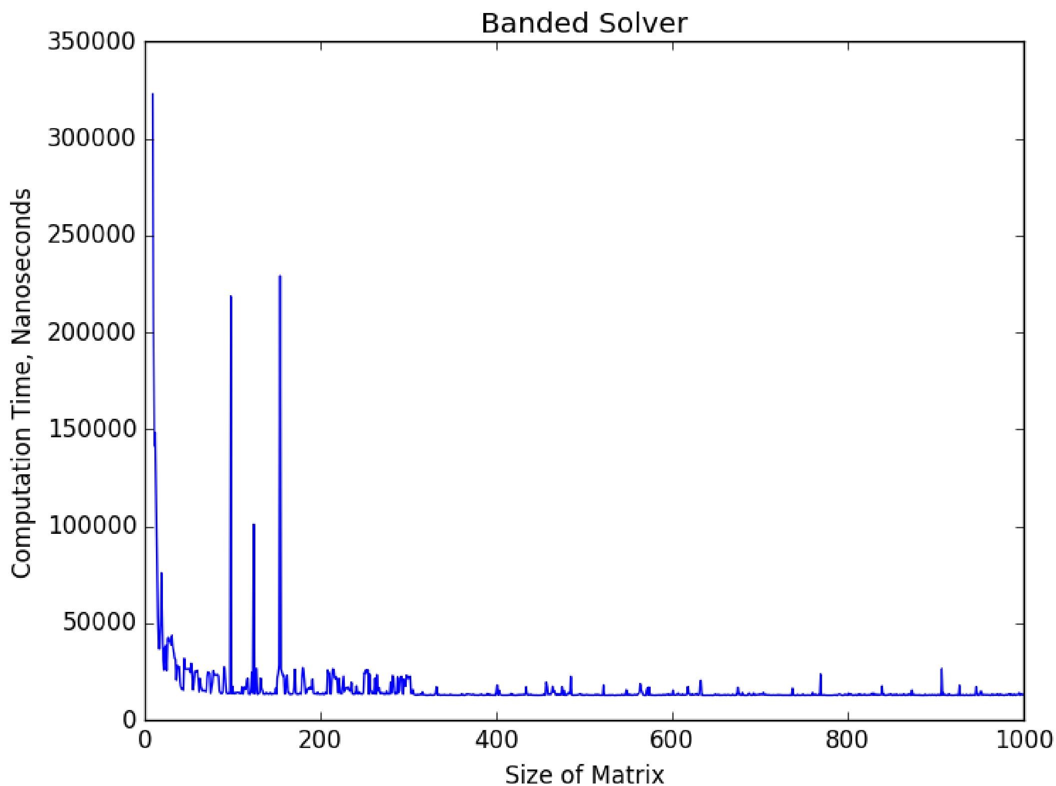


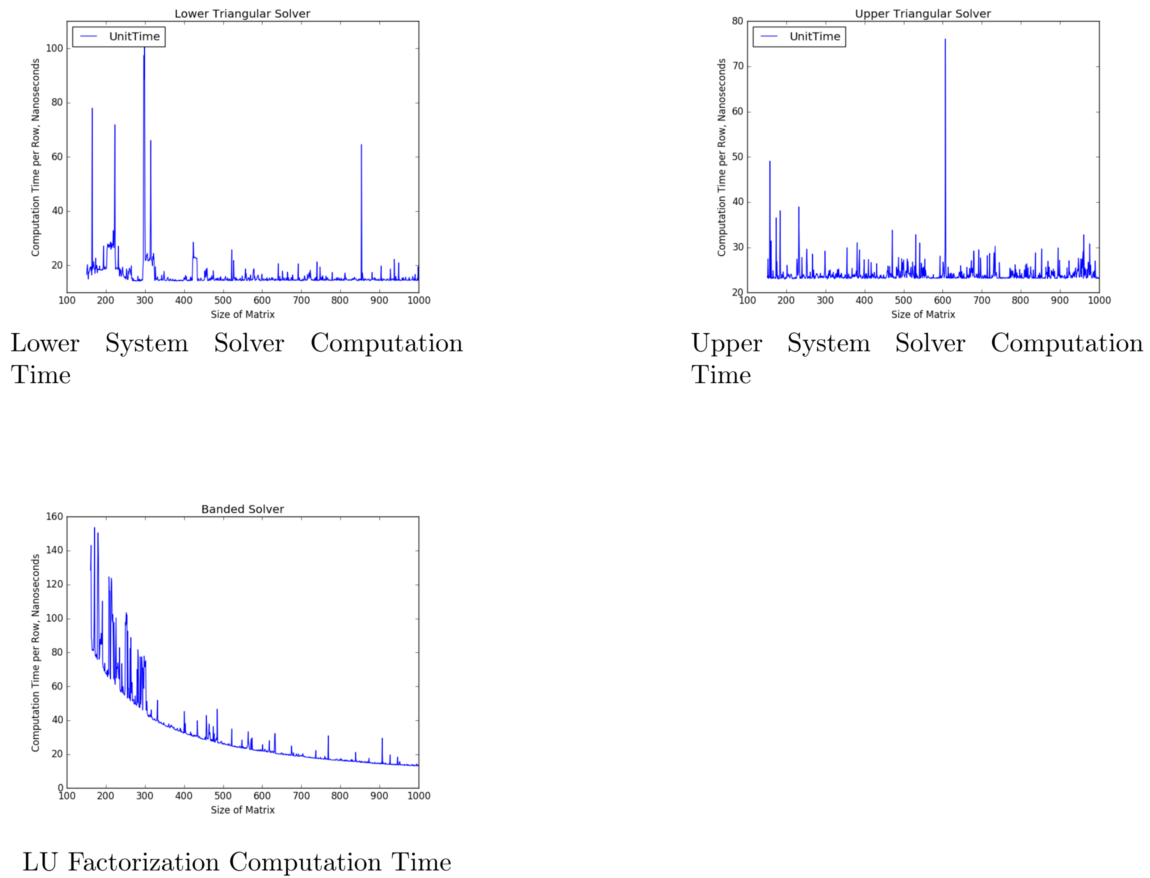
Figure 5: LU Factorization Computation Time

This is not simply the result of the scale making a slight linear trend appear constant due to scaling on the graph from outliers: outside of small amounts noise, there is no increase in computation time between matrices with sizes that are roughly $300x300$ and size $1000x1000$. The only possible explanation for this is optimizer improvement, but even that is unlikely to show that significant of a performance

improvement.

To verify that all of these are indeed linear relations, we plot the time scaled by the size of the matrix. If the relationships are linear, this will be a constant. Additionally, we consider only matrices that are larger than $150x150$ to minimize effects from compiler performance. These are plotted in Figure 6

Figure 6



Lower System Solver Computation Time

Upper System Solver Computation Time

LU Factorization Computation Time

As expected, we see that the system solvers are basically constant with some noise. Meanwhile, the factorization method is steadily decreasing and much less noisy than the system solvers.