Research Project in Mathematics

Analysis of Algorithms to Solve Matrix Equations

Gaussian Elimination

LU Decomposition

Cholesky Decomposition

Flop Counts and Complexities

Practical Application to Structural Engineering: Global Matrix Equations

Python Code for Algorithms

Bibliography

This research project assumes an understanding of undergraduate level linear algebra and summations.

For the sake of simplicity, this project will deal only with matrix equations that have square $n \times n$ coefficient matrices and exactly one solution.

Gaussian Elimination

Every sytem of linear equations can be rewritten as $Ax = b$, where A is a matrix and b is a vector. One algorithm to solve $Ax = b$ is a method that employs classical Gaussian elimination. It is a two step process: elimination and backward substitution.

The goal of the elmination step is to transform the coefficeint matrix into an upper triangular matrix by reducing the rest of the entries to 0. To accomplish this, there are three elementary row operations that can be performed on the rows of a matrix: swapping the positions of two rows, multiplying a row by a non-zero scalar, and adding a scalar multiple of one row to another row.

Let's demonstrate with an example. Suppose we have the following system of linear equations:

$$x_1 + 2x_2 - x_3 = 3$$

$$4x_1 + 2x_2 + 2x_3 = 6$$

$$-3x_1 + x_2 + x_3 = -7$$

This can be rewritten in matrix form $Ax = b$:

$$\begin{bmatrix} 1 & 2 & -1 \\ 4 & 2 & 2 \\ -3 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 3 \\ 6 \\ -7 \end{bmatrix}$$

where A is the coefficient matrix, b is the right-hand-side vector, and x is the vector that has our variables as its components.

To solve our matrix equation, create an augmented matrix by rewriting our equation in tableu form:

$$\left[\begin{array}{ccc|c} 1 & 2 & -1 & 3 \\ 4 & 2 & 2 & 6 \\ -3 & 1 & 1 & -7 \end{array}\right]$$

First, we need to do elimination. Our goal is to make A upper triangular; in other words, we want to produce 0's below all the pivots by using the elementary row operations. For each position that we are reducing to to 0, calculate a multiplier by dividing the target position by the pivot above it. This multiplier is then used to scale the pivot row, and the result is subtracted from the row containing the entry we intend to eliminate.

To eliminate column 1:

subtract 4 × row 1 from row 2

$$\left[\begin{array}{ccc|c} 1 & 2 & -1 & 3 \\ 0 & -6 & 6 & -6 \\ -3 & 1 & 1 & -7 \end{array}\right]$$

subtract $-3 \times$ row 1 from row 2

$$\left[\begin{array}{rrr|r} 1 & 2 & -1 & 3 \\ 0 & -6 & 6 & -6 \\ 0 & 7 & -2 & 2 \end{array}\right]$$

To eliminate column 2:

subtract $-\frac{7}{6} \times$ row 2 from row 3

$$\left[\begin{array}{rrr|r} 1 & 2 & -1 & 3 \\ 0 & -6 & 6 & -6 \\ 0 & 0 & 5 & -5 \end{array}\right]$$

Now, do backward substitution:

$$x_1 + 2x_2 - x_3 = 3$$

$$-6x_2 + 6x_3 = -6$$

$$5x_3 = -5$$

$$x_3 = -1$$

$$-6x_2 + 6(-1) = -6$$

$$-6x_2 - 6 = -6$$

$$-6x_2 = 0$$

$$x_2 = 0$$

$$x_1 + 2(0) - (-1) = 3$$

$$x_1 + 0 + 1 = 3$$

$$x_1 + 1 = 3$$

$$x_1 = 2$$

We now have our solution: $x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 2 \\ 0 \\ -1 \end{bmatrix}$

This process leaves potential for failure. In order to do the elimination step, we need to calcualte the multipliers by dividing by the pivots. If any of the pivots are 0, then we cannot do the elimination.

Not only are 0 pivots a problem, but so are very small pivots. If a pivot is very small, roundoff errors will occur, which can lead to instability.

Let's demonstrate with an example.

$$\left[\begin{array}{cc|c} 10^{-20} & 1 & 1 \\ 1 & 2 & 6 \end{array}\right]$$

First, do elimination. Our multiplier is $\frac{1}{10^{-20}} = 10^{20}$.

subtract $10^{20} \times$ row 1 from row 2

$$\left[\begin{array}{cc|c} 10^{-2} & 1 & 1 \\ 0 & 2 - 10^{20} & 6 - 10^{20} \end{array}\right]$$

Now, do backward substituion:

$$10^{-2}\ x_1 + x_2 = 1$$

$$(2 - 10^{20})x_2 = 6 - 10^{20}$$

$$x_2 = \frac{6 - 10^{20}}{2 - 10^{20}}$$

$$10^{-20}x_1 + \left(\frac{6 - 10^{20}}{2 - 10^{20}}\right) = 1$$

$$10^{-20}x_1 = 1 - \frac{6 - 10^{20}}{2 - 10^{20}}$$

$$x_1 = 10^{20}\left(1 - \frac{6 - 10^{20}}{2 - 10^{20}}\right)$$

$$x_1 = 10^{20}\left(\frac{2 - 10^{20}}{2 - 10^{20}} - \frac{6 - 10^{20}}{2 - 10^{20}}\right)$$

$$x_1 = 10^{20}\left(\frac{2 - 10^{20} - (6 - 10^{20})}{2 - 10^{20}}\right)$$

$$x_1 = 10^{20}\left(\frac{2 - 10^{20} - 6 + 10^{20})}{2 - 10^{20}}\right)$$

$$x_1 = 10^{20}\left(\frac{-4}{2 - 10^{20}}\right)$$

$$x_1 = \frac{-4 \times 10^{20}}{2 - 10^{20}}$$

Accordingly, the answer is: $\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} \frac{-4\times10^{20}}{2-10^{20}} \\ \\ \frac{6-10^{20}}{2-10^{20}} \end{bmatrix} \approx \begin{bmatrix} 4 \\ 1 \end{bmatrix}$

However, when a computer solves it using IEEE double-precison floating point arithmetic, we get a different result.

The elimination step is the same, so we have:

$$\left[\begin{array}{cc|c} 10^{-20} & 1 & 1 \\ 0 & 2-10^{20} & 6-10^{20} \end{array}\right]$$

However, $2-10^{20}$ and $6-10^{20}$ are both rounded to the nearest floating point number which is -10^{20}, so the following matrix is stored:

$$\left[\begin{array}{cc|c} 10^{-20} & 1 & 1 \\ 0 & -10^{20} & -10^{20} \end{array}\right]$$

Now, do backward substitution:

$$10^{-2}\ x_1 + x_2 = 1$$

$$-10^{20}x_2 = -10^{20}$$

$$x_2 = 1$$

$$10^{-20}x_1 + (1) = 1$$

$$10^{-2}\ x_1 = 0$$

$$x_1 = 0$$

We have an unexpected answer: $\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$. Our final result has large relative error compared with the exact solution; the x_1 component is 0 instead of being around 4.

This problem was caused by the large multiplier 10^{20}. The effect of subtracting the large number 10^{20} times the first equation from the second equation overpowered, or "swamped," the second equation; the original values in the second equation became so irrelevant after subtracting such large numbers that they were lost during rounding. (In fact, the second equation became essentially a multiple of the first.) Therefore, this is sometimes referred to as "swamping."

It is apparent that the way to avoid this is to keep the multipliers as small as possible. To do this, we have to ensure the numerators, which are the numbers we are reducing to 0, have small absolute value, and the denominators, which are the pivots, have large absolute value.

The solution is called partial pivoting. Before the elimination with each pivot, if the pivot does not have larger absolute value than all the entries below it, then we swap the pivot row with the row below that has the largest absolute value in the pivot column. As a result, the multipliers will always have an absolute value of at most 1.

Let's demonstrate that partial pivoting resolves the problem in our example:

$$\left[\begin{array}{cc|c} 10^{-20} & 1 & 1 \\ 1 & 2 & 6 \end{array}\right]$$

Before starting elimination with the first pivot, swap the first row with the row that has the entry with the lowest absolute value in the first column:

$$\left[\begin{array}{cc|c} 1 & 2 & 6 \\ 10^{-20} & 1 & 1 \end{array}\right]$$

Now, proceed with elimination. Our multiplier is 10^{-2} .

subtract $10^{-20} \times$ row 1 from row 2

$$\left[\begin{array}{cc|c} 1 & 2 & 6 \\ 0 & 1-2\times10^{-20} & 1-6\times10^{-20} \end{array}\right]$$

$1-2\times10^{-20}$ and $1-6\times10^{-20}$ are both rounded to the nearest floating point number, which is 1, so the following matrix is stored:

$$\left[\begin{array}{cc|c} 1 & 2 & 6 \\ 0 & 1 & 1 \end{array}\right]$$

Do backward substitution:

$$x_1 + 2x_2 = 6$$

$$x_2 = 1$$

$$x_2 = 1$$

$$x_1 + 2(1) = 6$$

$$x_1 + 2 = 6$$

$$x_1 = 4$$

Our result is: $\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 4 \\ 1 \end{bmatrix}$. This is the result we were expecting.

LU Decomposition

Another algorithm for solving matrix system of linear equations utilizes LU decomposition. This method involves finding a factorization of A into two matrices represented by L and U, or $A = LU$, where L is a lower triangular matrix and U is an upper triangular matrix.

LU decomposition is three steps: elimination, forward substitution, and backward substitution.

The elimination step is the same as that of classical Gaussian elimination (without involving b), which will leave us with U. L is formed by putting 1's in the main diagonal and the multipliers in the lower triangle. Each multiplier is placed in the spot in L corresponding to the entry it eliminated in the original matrix.

Once we find L and U, our equation $Ax = b$ can be rewritten as $LUx = b$. Define $c = Ux$.

The next step is solving $Lc = b$ for c using forward substation.

The last step is solving $Ux = c$ for x using backward substitution.

Let's solve our original matrix equation using this method:

$$\begin{bmatrix} 1 & 2 & -1 \\ 4 & 2 & 2 \\ -3 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 3 \\ 6 \\ -7 \end{bmatrix}$$

First, form L and U using elimination:

Subtract $4 \times$ row 1 from row 2

$$\begin{bmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ & & 1 \end{bmatrix}\begin{bmatrix} 1 & 2 & -1 \\ 0 & -6 & 6 \\ -3 & 1 & 1 \end{bmatrix}$$

subtract $-3 \times$ row 1 from row 3

$$\begin{bmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ -3 & & 1 \end{bmatrix}\begin{bmatrix} 1 & 2 & -1 \\ 0 & -6 & 6 \\ 0 & 7 & -2 \end{bmatrix}$$

To eliminate column 2:

subtract $-\frac{7}{6} \times$ row 2 from row 3

$$\begin{bmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ -3 & -\frac{7}{6} & 1 \end{bmatrix}\begin{bmatrix} 1 & 2 & -1 \\ 0 & -6 & 6 \\ 0 & 0 & 5 \end{bmatrix} = LU$$

Next, define $c = Ux$ and solve $Lc = b$ for c using forward substation:

$$\begin{bmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ -3 & -\frac{7}{6} & 1 \end{bmatrix}\begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} 3 \\ 6 \\ -7 \end{bmatrix}$$

$$c_1 = 3$$

$$4c_1 + c_2 = 6$$

$$-3c_1 - \frac{7}{6}c_2 + c_3 = -7$$

$$c_1 = 3$$

$$4(3) + c_2 = 6$$

$$12 + c_2 = 6$$

$$c_2 = -6$$

$$-3c_1 - \frac{7}{6}c_2 + c_3 = -7$$

$$-3(3) - \frac{7}{6}(-6) + c_3 = -7$$

$$-9 + 7 + c_3 = -7$$

$$-2 + c_3 = -7$$

$$c_3 = -5$$

Therefore, $c = \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} 3 \\ -6 \\ -5 \end{bmatrix}$

Finally, solve $Ux = c$ for x using backward substitution:

$$\begin{bmatrix} 1 & 2 & -1 \\ 0 & -6 & 6 \\ 0 & 0 & 5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 3 \\ -6 \\ -5 \end{bmatrix}$$

$$x_1 + 2x_2 - 1x_3 = 3$$

$$-6x_2 + 6x_3 = -6$$

$$5x_3 = -5$$

$$x_3 = -1$$

$$-6x_2 + 6(-1) = -6$$

$$-6x_2 - 6 = -6$$

$$-6x_2 = 0$$

$$x_2 = 0$$

$$x_1 + 2(0) - (-1) = 3$$

$$x_1 + 0 + 1 = 3$$

$$x_1 + 1 = 3$$

$$x_1 = 2$$

The final answer is: $x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 2 \\ 0 \\ -1 \end{bmatrix}$. This is the same solution as the one we got using classical Guasain elimination.

Unsurprisingly, the problems of 0 pivots and roundoff errors due to swamping do not only exist in classical Gaussian elimination but also in LU decomposition.

Furthermore, for some matrices, an $A = LU$ factorization does not even exist. For example, $\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$ does not have an LU factorization. We can prove this by contradiction:

Suppose $\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$ has an LU factorization. This would mean:

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ a & 1 \end{bmatrix} \begin{bmatrix} b & c \\ 0 & d \end{bmatrix} = \begin{bmatrix} b & c \\ ab & ac + d \end{bmatrix}$$

However, if we equate coefficients, $b = 0$ and $ab = 1$ or $a(0) = 1$ or $0 = 1$, which is a contradiction. Therefore, $\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$ must not have an LU factorization.

The solution to all three of these problems is, once again, partial pivoting.

In order to implement partial pivoting in LU decomposition, we need to use permuation matrices.

A permutation matrix is a square matrix consisting of all 0's, except for a single 1 in every row and column. In other terms, a permutation matrix is a matrix created by applying arbitrary row or column exchanges to an identity matrix.

If P is a permutation matrix formed by a particular set of row exchanges applied to the identity matrix, then PA is the matrix obtained by applying the same row exchanges to A.

We can apply this idea to LU decomposition. To do the elimination step of LU decomposition with partial pivoting, we can store the row exchanges in P, such that $PA = LU$. We can then substitute this into $PAx = Pb$ to get $LUx = Pb$.

We do not store the multipliers in L right away. Rather, we use the positions we are reducing to 0 as storage spots for the corresponding multipliers. Only once we are done the elimination will we form L. Therefore, the multipliers will stay with their rows if future row exchanges are made.

The forward substitution step and backward substitution step are the same as earlier: solve $Lc = Pb$ (where P and b are multiplied so that b is permuted) for c, and then solve $Ux = c$ for x.

Let's solve our matrix equation using this method:

$$\begin{bmatrix} 1 & 2 & -1 \\ 4 & 2 & 2 \\ -3 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 3 \\ 6 \\ -7 \end{bmatrix}$$

Row 2 has the largest absolute value in the first column, so swap row 1 with row 2:

$$\begin{bmatrix} 4 & 2 & 2 \\ 1 & 2 & -1 \\ -3 & 1 & 1 \end{bmatrix}, \qquad P = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Eliminate column 1 and store the multipliers in the spots of the 0:

$$\text{subtract } \tfrac{1}{4} \times \text{ row 1 from row 2}$$

$$\begin{bmatrix} 4 & 2 & 2 \\ \frac{1}{4} & \frac{3}{2} & -\frac{3}{2} \\ -3 & 1 & 1 \end{bmatrix}$$

$$\text{subtract } -\frac{3}{4} \times \text{ row 1 from row 3}$$

$$\begin{bmatrix} 4 & 2 & 2 \\ \frac{1}{4} & \frac{3}{2} & -\frac{3}{2} \\ -\frac{3}{4} & \frac{5}{2} & \frac{5}{2} \end{bmatrix}$$

Row 3 has larger absolute value than row 3 in the second column, so swap row 2 with row 3:

$$\begin{bmatrix} 4 & 2 & 2 \\ -\frac{3}{4} & \frac{5}{2} & \frac{5}{2} \\ \frac{1}{4} & \frac{3}{2} & -\frac{3}{2} \end{bmatrix}, \qquad P = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

Notice how the multipliers in the first column switch spots, which is why we will only form L at the end.

Eliminate column 2 and store the multiplier in the spot of the 0:

$$\text{subtract } \frac{3}{5} \times \text{ row 2 from row 3}$$

$$\begin{bmatrix} 4 & 2 & 2 \\ -\frac{3}{4} & \frac{5}{2} & \frac{5}{2} \\ \frac{1}{4} & \frac{3}{5} & -3 \end{bmatrix}$$

We can now write L and U:

$$PA = LU = \begin{bmatrix} 1 & 0 & 0 \\ -\frac{3}{4} & 1 & 0 \\ \frac{1}{4} & \frac{3}{5} & 1 \end{bmatrix} \begin{bmatrix} 4 & 2 & 2 \\ 0 & \frac{5}{2} & \frac{5}{2} \\ 0 & 0 & -3 \end{bmatrix}$$

Next, define $c = Ux$ and solve $Lc = Pb$ for c:

$$\begin{bmatrix} 1 & 0 & 0 \\ -\frac{3}{4} & 1 & 0 \\ \frac{1}{4} & \frac{3}{5} & 1 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 3 \\ 6 \\ -7 \end{bmatrix}$$

First, permutate b by multiplying by P:

$$\begin{bmatrix} 1 & 0 & 0 \\ -\frac{3}{4} & 1 & 0 \\ \frac{1}{4} & \frac{3}{5} & 1 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} 6 \\ -7 \\ 3 \end{bmatrix}$$

Now, do forward substitution:

$$c_1 = 6$$

$$-\frac{3}{4}c_1 + c_2 = -7$$

$$\frac{1}{4}c_1 + \frac{3}{5}c_2 + c_3 = 3$$

$$c_1 = 6$$

$$-\frac{3}{4}(6) + c_2 = -7$$

$$-\frac{18}{4} + c_2 = -7$$

$$c_2 = -\frac{5}{2}$$

$$\frac{1}{4}(6) + \frac{3}{5}\left(-\frac{5}{2}\right) + c_3 = 3$$

$$\frac{3}{2} - \frac{3}{2} + c_3 = 3$$

$$c_3 = 3$$

Therefore, $c = \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} 6 \\ -\frac{5}{2} \\ 3 \end{bmatrix}$

Finally, solve $Ux = c$ for x using backward substitution:

$$\begin{bmatrix} 4 & 2 & 2 \\ 0 & \frac{5}{2} & \frac{5}{2} \\ 0 & 0 & -3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 6 \\ -\frac{5}{2} \\ 3 \end{bmatrix}$$

$$4x_1 + 2x_2 + 2x_3 = 3$$

$$\frac{5}{2}x_2 + \frac{5}{2}x_3 = -\frac{5}{2}$$

$$-3x_3 = 3$$

$$x_3 = -1$$

$$\frac{5}{2}x_2 + \frac{5}{2}(-1) = -\frac{5}{2}$$

$$\frac{5}{2}x_2 - \frac{5}{2} = -\frac{5}{2}$$

$$\frac{5}{2}x_2 = 0$$

$$x_2 = 0$$

$$x_1 + 2(0) - (-1) = 3$$

$$x_1 + 0 + 1 = 3$$

$$x_1 + 1 = 3$$

$$x_1 = 2$$

The final answer is: $x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 2 \\ 0 \\ -1 \end{bmatrix}$. This is the solution we got earlier.

Cholesky Decomposition

There is a special case of matrices called positive-definite. A matrix is positive-definite if $x^T Ax > 0$ for all vectors $x \neq 0$. (Some prominent characteristics of positive-definite matrices are that they are symmetric and have positive entries on their main diagonals.)

Let's give an example of a positive-definite matrix:

$$A = \begin{bmatrix} 2 & 2 \\ 2 & 7 \end{bmatrix}$$

$$x^T Ax = \begin{bmatrix} x_1 & x_2 \end{bmatrix} \begin{bmatrix} 2 & 2 \\ 2 & 7 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

$$= 2{x_1}^2 + 4x_1x_2 + 7{x_2}^2$$

$$= 2{x_1}^2 + 4x_1x_2 + 2{x_2}^2 + 5{x_2}^2$$

$$= 2({x_1}^2 + 2x_1x_2 + {x_2}^2) + 5x_2$$

$$= 2(x_1+x_2)^2 + 5x_2$$

which is always non-negative and cannot be 0 unless both x_1 and x_2 are 0, which would imply $x = 0$, so A is positive-definite.

(There are numerous algorithms to test whether a matrix is positive-definite or not, but they are beyond the scope of this project.)

If a matrix is positive-definite, we can carry out Cholesky Decomposition. It involves finding a factorization of A into two matrices R^T and R, or $A = R^T R$, where R is an upper triangular matrix.

Let's demonstrate the algorithm. First, partition the matrix as follows:

$$A = \left[\begin{array}{c|c} a & b^T \\ \hline b & C \end{array}\right]$$

where b is an $(n - 1)$ vector and C is an $(n - 1) \times (n - 1)$ submatrix.

Using this matrix, we produce the following matrix:

$$\left[\begin{array}{c|c} \sqrt{a} & \frac{b^T}{\sqrt{a}} \\ \hline 0 & C - \frac{b}{\sqrt{a}}\frac{b^T}{\sqrt{a}} \end{array}\right]$$

This matrix is obtained by performing four operations in this order: taking the square root of a, dividing vector b by this result, multiplying this new vector by its transpose, and subtracting this product from C.

We repeat these steps recursively on the submatrix $C - \frac{b}{\sqrt{a}}\frac{b^T}{\sqrt{a}}$, which is one size smaller in each iteration, until we attain a singleton matrix. At this point, we simply take the square root of the value. At the end of this process, we are left with R.

Now, $Ax = b$ can be rewritten as $R^T Rx = b$. Define $c = Rx$.

The next step is solving $R^T c = b$ for c using forward substitution.

The last step is solving $Rx = c$ for x using backward substitution.

Let's do an example with the following positive-definite matrix:

$$A = \begin{bmatrix} 4 & -2 & 0 \\ -2 & 2 & -3 \\ 0 & -3 & 10 \end{bmatrix}, b = \begin{bmatrix} 0 \\ 3 \\ -7 \end{bmatrix}$$

First, partition the matrix:

$$\left[\begin{array}{c|cc} 4 & -2 & 0 \\ \hline -2 & 2 & -3 \\ 0 & -3 & 10 \end{array}\right]$$

$\sqrt{a} = \sqrt{4} = 2$, $\frac{b}{\sqrt{a}} = \frac{\begin{bmatrix} -2 \\ 0 \end{bmatrix}}{2} = \begin{bmatrix} -1 \\ 0 \end{bmatrix}$,

and $C - \frac{b}{\sqrt{a}}\frac{b^T}{\sqrt{a}} = \begin{bmatrix} 2 & -3 \\ -3 & 10 \end{bmatrix} - 1 \quad 0[-1 \quad 0] = \begin{bmatrix} 2 & -3 \\ -3 & 10 \end{bmatrix} - \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & -3 \\ -3 & 10 \end{bmatrix}$, so we produce the following matrix:

$$\left[\begin{array}{c|cc} 4 & -1 & 0 \\ \hline 0 & 1 & -3 \\ 0 & -3 & 10 \end{array}\right]$$

Now, we repeat these steps on the 2×2 submatrix. Start by partitioning:

$$\left[\begin{array}{c|cc} 4 & -1 & 0 \\ \hline 0 & 1 & -3 \\ 0 & -3 & 10 \end{array}\right]$$

$\sqrt{a} = \sqrt{1} = 1$, $\frac{b}{\sqrt{a}} = \frac{[-3]}{1} = [-3]$,

and $C - \frac{b}{\sqrt{a}}\frac{b^T}{\sqrt{a}} = [10] - 3[-3] = [10] - [9] = [1]$, so we produce the following matrix:

$$\left[\begin{array}{c|cc} 4 & -1 & 0 \\ \hline 0 & 1 & -3 \\ 0 & 0 & 1 \end{array}\right]$$

Our submatrix is now a singleton matrix, so we take the square root of the single value: $\sqrt{1} = 1$. We now have the following matrix:

$$\left[\begin{array}{c|cc} 4 & -1 & 0 \\ \hline 0 & 1 & -3 \\ 0 & 0 & 1 \end{array}\right]$$

We now have R:

$$R = \begin{bmatrix} 4 & -1 & 0 \\ 0 & 1 & -3 \\ 0 & 0 & 1 \end{bmatrix}$$

Next, define $c = Rx$ and solve $R^T c = b$ for c using forward substitution:

$$\begin{bmatrix} 4 & 0 & 0 \\ -1 & 1 & 0 \\ 0 & -3 & 1 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 3 \\ -7 \end{bmatrix}$$

$$4c_1 = 0$$

$$-c_1 + c_2 = 3$$

$$-3c_2 + c_3 = -7$$

$$c_1 = 0$$

$$-(0) + c_2 = 3$$

$$c_2 = 3$$

$$-3(3) + c_3 = -7$$

$$-9 + c_3 = -7$$

$$c_3 = 2$$

Therefore, $c = \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 3 \\ 2 \end{bmatrix}$

Finally, solve $Rx = c$ for x using backward substitution:

$$\begin{bmatrix} 4 & -1 & 0 \\ 0 & 1 & -3 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 3 \\ 2 \end{bmatrix}$$

$$4x_1 - x_2 = 0$$

$$x_2 - 3x_3 = 3$$

$$x_3 = 2$$

$$x_3 = 2$$

$$x_2 - 3(2) = 3$$

$$x_2 - 6 = 3$$

$$x_2 = 9$$

$$4x_1 - (9) = 0$$

$$4x_1 - 9 = 0$$

$$4x_1 = 9$$

$$x_1 = \frac{9}{4}$$

The answer is: $x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} \frac{9}{4} \\ 9 \\ 2 \end{bmatrix}$

Flop Counts and Complexities

A floating-point operation, or flop, is an arithmetic operation performed on floating point numbers. For the purposes of this project, each addition, subtraction, multiplication, division, and square root will be counted as one flop.

The flop count of an algorithm is the exact number of flops required to carry it out, as a function of the size of the input or inputs. Counting the number of flops performed by an algorithm is the classical method to determine its cost of computation and efficiency.

The complexity of an algorithm is the approximate number of flops required to carry it out, when the size or sizes of the input or inputs are large. Essentially, the complexity consists of the dominant terms of the flop count. The rest of the terms are usually negligible since the flop count is, anyway, only a rough measure of an algorithm's cost of computation and efficiency.

We will now proceed to calculate the flop counts and complexities of our algorithms.

Let's calculate the flop count for solving a system of linear equations using classical Gaussian elimination. First, let's calculate the flop count for the elimination step. Suppose we have a matrix equation $Ax = b$ written in tableau form where a is an arbitrary pivot:

$$\left[\begin{array}{cccccc|c} 0 & 0 & \boldsymbol{a} & b & \cdots & c & d \\ \vdots & \vdots & \vdots & \vdots & \cdots & \vdots & \vdots \\ 0 & 0 & e & f & \cdots & g & h \end{array}\right]$$

To eliminate e, we do the following:

$$\left[\begin{array}{cccccc|c} 0 & 0 & \boldsymbol{a} & b & \cdots & c & d \\ \vdots & \vdots & \vdots & \vdots & \cdots & \vdots & \vdots \\ 0 & 0 & 0 & f-\frac{e}{\boldsymbol{a}}b & \cdots & g-\frac{e}{\boldsymbol{a}}c & h-\frac{e}{\boldsymbol{a}}d \end{array}\right]$$

For each entry we want to eliminate, we divide to find a multiplier and use it to multiply each entry in the pivot row after the pivot and subtract this product from the entry below it in the row of the entry we are eliminating. To calculate the total number of flops for the elimination step, we need to calculate the total number of divisions, subtractions, and multiplications.

To start, let's calculate the total number of flops needed for each entry being eliminated. There is one division for the multiplier. Where j is the column number of the entry we are eliminating, there are $n-j$ multiplications for A and 1 multiplication for b, or $n-j+1$ multiplications. Similarly, there are $n-j+1$ subtractions. (The number of multiplications and subtractions for A is $n-j$ and not $n-(j-1)$, or $n-j+1$, because, practically, we do not have to do the multiplication and subtraction for the entry we are actually eliminating because we already know that it is going to be 0.) Therefore, the total number of divisions, multiplications and subtractions for each entry is $(1)+(n-j+1)+(n-j+1)$, or $2(n-j+1)+1$.

To find the total number of flops for the elimination step, construct a matrix in which each entry represents the total number of flops needed to eliminate that entry:

$$\begin{bmatrix} \blacksquare & & & & & & \\ 2(n-j+1)+1 & \blacksquare & & & & & \\ 2(n-j+1)+1 & 2(n-j+1)+1 & & & & & \\ \vdots & \vdots & \ddots & & & & \\ \vdots & \vdots & & \ddots & \blacksquare & & \\ 2(n-j+1)+1 & 2(n-j+1)+1 & \cdots & \cdots & 2(n-j+1)+1 & \blacksquare & \\ 2(n-j+1)+1 & 2(n-j+1)+1 & \cdots & \cdots & 2(n-j+1)+1 & 2(n-j+1)+1 & \blacksquare \end{bmatrix}$$

Fill in the values for j:

$$\begin{bmatrix} \blacksquare & & & & & & \\ 2n+1 & \blacksquare & & & & & \\ 2n+1 & 2(n-1)+1 & & & & & \\ \vdots & \vdots & \ddots & & & & \\ \vdots & \vdots & & \ddots & \blacksquare & & \\ 2n+1 & 2(n-1)+1 & \cdots & \cdots & 2(3)+1 & \blacksquare & \\ 2n+1 & 2(n-1)+1 & \cdots & \cdots & 2(3)+1 & 2(2)+1 & \blacksquare \end{bmatrix}$$

Adding up the counts from right to left, (where j is a dummy variable and does not represent the column number as it did earlier,) we get:

$$\sum_{j=1}^{n-1}\sum_{i=1}^{j} 2(j+1)+1 = \sum_{j=1}^{n-1}\sum_{i=1}^{j} 2j+3 = \sum_{j=1}^{n-1} 2j^2+3j$$

$$= 2\left[\frac{(n-1)[(n-1)+1][2(n-1)+1]}{6}\right] + 3\left[\frac{(n-1)[(n-1)+1]}{2}\right]$$

$$= \frac{2(n-1)n(2n-1)}{6} + \frac{3(n-1)n}{2}$$

$$= \frac{2(n-1)n(2n-1)+9(n-1)n}{6}$$

$$= \frac{4n^3+3n^2-7n}{6}$$

$$= \frac{2}{3}n^3 + \frac{1}{2}n^2 - \frac{7}{6}n$$

Therefore, the total number of flops needed for the elimination step is $\frac{2}{3}n^3 + \frac{1}{2}n^2 - \frac{7}{6}n$.

Now, we will calculate the number of flops for backward substitution. Backward substitution works because the elimination step leaves us with an upper triangular matrix:

$$\left[\begin{array}{ccccccc|c} \boldsymbol{a_{11}} & a_{12} & a_{13} & \cdots & a_{1,n-2} & a_{1,n-1} & a_{1n} & b_1 \\ 0 & \boldsymbol{a_{22}} & a_{23} & \cdots & a_{2,n-2} & a_{2,n-1} & a_{2n} & b_2 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & \boldsymbol{a_{n-2,n-2}} & a_{n-2,n-1} & a_{n-2,n} & b_{n-2} \\ 0 & 0 & 0 & \cdots & 0 & \boldsymbol{a_{n-1,n-1}} & a_{n-1,n} & b_{n-1} \\ 0 & 0 & 0 & \cdots & 0 & 0 & \boldsymbol{a_{nn}} & b_n \end{array}\right]$$

Which is equivalent to:

$$\begin{aligned} \boldsymbol{a_{11}}x_1 + a_{12}x_2 + a_{13}x_3 + \cdots + a_{1,n-2}x_{n-2} + a_{1,n-1}x_{n-1} + a_{1n}x_n &= b_1 \\ \boldsymbol{a_{22}}x_2 + a_{23}x_3 + \cdots + a_{2,n-2}x_{n-2} + a_{2,n-1}x_{n-1} + a_{2n}x_n &= b_2 \\ \boldsymbol{a_{n-2,n-2}}x_{n-2} + a_{n-2,n-1}x_{n-1} + a_{n-2,n}x_n &= b_{n-2} \\ \vdots & \\ \boldsymbol{a_{n-1,n-1}}x_{n-1} + a_{n-1,n}x_n &= b_{n-1} \\ \boldsymbol{a_{nn}}x_n &= b_n \end{aligned}$$

Therefore, backward substitution requires us do the following steps, starting from the bottom and working up:

$$x_n = \frac{b_n}{\boldsymbol{a_{nn}}}$$

$$x_{n-1} = \frac{b_{n-1} - a_{n-1,n}x_n}{\boldsymbol{a_{n-1,n-1}}}$$

$$x_{n-2} = \frac{b_{n-2} - a_{n-2,n}x_n - a_{n-2,n-1}x_{n-1}}{\boldsymbol{a_{n-2,n-2}}}$$

$$\vdots$$

$$x_2 = \frac{b_2 - a_{2n}x_n - a_{2,n-1}x_{n-1} - a_{2,n-2}x_{n-2} - \cdots - a_{23}x_3}{\boldsymbol{a_{22}}}$$

$$x_1 = \frac{b_1 - a_{1n}x_n - a_{1,n-1}x_{n-1} - a_{1,n-2}x_{n-2} - \cdots - a_{13}x_3 - a_{12}x_2}{\boldsymbol{a_{11}}}$$

Clearly, adding up the flops for all the steps of substitution is:

$$1 + 3 + 5 + \cdots + (2n-1) = \sum_{i=1}^{n} 2i - 1 = 2\left[\frac{n(n+1)}{2}\right] - n = n^2$$

Therefore, the total number of flops needed for the backward substitution step is n^2.

Therefore, the total flop count for solving a system of linear equations with Gaussian elimination is:

$$\left(\frac{2}{3}n^3 + \frac{1}{2}n^2 - \frac{7}{6}n\right) + (n^2) = \frac{2}{3}n^3 + \frac{3}{2}n^2 - \frac{7}{6}n$$

The complexity is $\frac{2}{3}n^3$.

Let's repeat this process to find the flop count for solving a system of linear equations with LU decomposition.

The elimination step is the same, besides that we do not apply the row operations to *b*. In other words, the total number of divisions, multiplications and subtractions for

each entry being eliminated is $(1) + (\,n - j) + (n - j)$, or $2(n - j) + 1$. Therefore, the matrix representing the number of flops needed for each entry being eliminated will be the following:

$$\begin{bmatrix} \blacksquare & & & & & & \\ 2(n-j)+1 & \blacksquare & & & & & \\ 2(n-j)+1 & 2(n-j)+1 & & & & & \\ \vdots & \vdots & \ddots & & & & \\ \vdots & \vdots & & \ddots & \blacksquare & & \\ 2(n-j)+1 & 2(n-j)+1 & \cdots & \cdots & 2(n-j)+1 & \blacksquare & \\ 2(n-j)+1 & 2(n-j)+1 & \cdots & \cdots & 2(n-j)+1 & 2(n-j)+1 & \blacksquare \end{bmatrix}$$

Filling in the values for j gives us:

$$\begin{bmatrix} \blacksquare & & & & & & \\ 2(n-1) & \blacksquare & & & & & \\ 2(n-1) & 2(n-2)+1 & & & & & \\ \vdots & \vdots & \ddots & & & & \\ \vdots & \vdots & & \ddots & \blacksquare & & \\ 2(n-1) & 2(n-2)+1 & \cdots & \cdots & 2(2)+1 & \blacksquare & \\ 2(n-1) & 2(n-2)+1 & \cdots & \cdots & 2(2)+1 & 2(1)+1 & \blacksquare \end{bmatrix}$$

Adding up the counts from right to left we get:

$$\sum_{j=1}^{n-1}\sum_{i=1}^{j} 2j+1 = \sum_{j=1}^{n-1} 2j^2 + j$$

$$= 2\left[\frac{(n-1)[(n-1)+1][2(n-1)+1]}{6}\right] + \frac{(n-1)[(n-1)+1]}{2}$$

$$= \frac{2(n-1)n(2n-1)}{6} + \frac{(n-1)n}{2}$$

$$= \frac{2(n-1)n(2n-1) + 3(n-1)n}{6}$$

$$= \frac{4n^3 - 3n^2 - n}{6}$$

$$= \frac{2}{3}n^3 - \frac{1}{2}n^2 - \frac{1}{6}n$$

Therefore, the total number of flops needed for the elimination step is $\frac{2}{3}n^3 - \frac{1}{2}n^2 - \frac{1}{6}n$.

Forward substitution will obviously have the same flop count as backward substitution. Therefore, the flop count for the forward substitution and backward substitution together is $(n^2) + (n^2)$, or $2n^2$. (Permuting b does not practically require multiplication because we are just reordering the rows of b.)

Therefore, the total flop count for solving a system of linear equations with LU decomposition is:

$$\left(\frac{2}{3}n^3 - \frac{1}{2}n^2 - \frac{1}{6}n\right) + (2n^2) = \frac{2}{3}n^3 + \frac{3}{2}n^2 - \frac{1}{6}n$$

Like the Gaussian elimination algorithm, the complexity is $\frac{2}{3}n^3$

The flop count for solving a matrix equation with LU decomposition: $\frac{2}{3}n^3 + \frac{3}{2}n^2 - \frac{1}{6}n$, is greater than[1] the flop count of Gaussian elimination: $\frac{2}{3}n^3 + \frac{3}{2}n^2 - \frac{7}{6}n$. Additionally, LU requires additional memory to store L containing the multipliers. Furthermore, Gaussian is just a simpler algorithm; it does not have the extra step of forward substitution. The obvious question is: Why bother with LU decomposition?

[1] Sauer asserts that the flop count for LU decomposition is equal to the flop count for classical Gaussian elimination, but according to my calculation, it is greater. The texts of Burden and Faires and Lu arrived at the same LU decomposition flop count that I arrived at, albeit with a different method.

The answer is that it greatly reduces the flop count in a case where we are dealing with multiple b's. With classical Gaussian elimination, we need to redo the elimination step with every b. However, for LU decomposition, we only need to do the elimination step once; b is not involved until the forward substitution step. (Of course, if we know all the b's at the outset, we can solve all the problems simultaneously using one elimination, but this is often not the case.) This is a big deal because the expensive part of solving matrix equations is the elimination step. The elimination step has complexity n^3, while forward and backward substitution have complexities of only n^2.

Let k be the number of b's, then the flop count for Gaussian elimination is:

$$(flop\ count\ for\ elimination\ +\ flop\ count\ for\ substitution) \times k$$

$$= \left(\frac{2}{3}n^3 + \frac{1}{2}n^2 - \frac{7}{6}n + n^2\right)k = \left(\frac{2}{3}n^3 + \frac{3}{2}n^2 - \frac{7}{6}n\right)k = \frac{2}{3}kn^3 + \frac{3}{2}kn^2 - \frac{7}{6}kn$$

which has a complexity of $\frac{2}{3}kn^3$.

While the flop count for LU decomposition is:

$$flop\ count\ for\ elimination$$

$$+ (flop\ count\ for\ forward\ substitution\ +\ flop\ count\ for\ substitution) \times k$$

$$= \frac{2}{3}n^3 - \frac{1}{2}n^2 - \frac{1}{6}n + (n^2 + n^2)k = \frac{2}{3}n^3 - \frac{1}{2}n^2 - \frac{1}{6}n + (2n^2)k = \frac{2}{3}n^3 + \left(2k - \frac{1}{2}\right)n^2 - \frac{1}{6}n$$

which is $\frac{2}{3}n^3 + \left(2k - \frac{1}{2}\right)n^2$.

When n^3 is considered large compared to n^2, LU decomposition is much more efficient.

Lastly, Let's calculate the flop count for solving a system of linear equations using Cholesky decomposition.

Let j be the size of the current matrix/submatrix. Each matrix/submatrix (besides when the submatrix is a singleton matrix) that we are performing the flops on has the following form:

$$\left[\begin{array}{c|c} a & b^T \\ \hline b & C \end{array}\right]$$

where the matrix/submatrix is size $j \times j$, b is a $(j-1)$ vector and C is a $(j-1) \times (j-1)$ submatrix.

The square root of a is 1 flop. The division of b by the found $\sqrt{a}$ is $j-1$ flops. One might expect that the flop count for the multiplication of the found quotient $\frac{b}{\sqrt{a}}$ by $\frac{b^T}{\sqrt{a}}$ is n^2 because since it is an outer product, each component of the vector is multiplied by each component in the transpose. However, since the result is always symmetric, we only need to find the upper or lower triangle, and then we know the rest. Therefore, the flop count is $1 + 2 + 3 + \cdots + (j-1) = \sum_{i=1}^{j-1} i$. For the same reason, the flop count for the subtraction of C by the found product $\frac{b}{\sqrt{a}}\frac{b^T}{\sqrt{a}}$ is $\sum_{i=1}^{j-1} i$.

Therefore, the flops needed for each matrix/submatrix is:

$$(1) + (j-1) + \left(\sum_{i=1}^{j-1} i\right) + \left(\sum_{i=1}^{j-1} i\right) = j + 2\sum_{i=1}^{j-1} i$$

$$= j + 2\left(\frac{(j-1)j}{2}\right)$$

$$= j + j^2 - j$$

$$= j^2$$

We perform these j^2 flops on each matrix starting from size $n \times n$ down to size 2×2. When we reach a singleton matrix, we take the square root of the value, which is 1 flop. Therefore, the total flop count is:

$$\left(\sum_{j=2}^{n} j^2\right) + 1 = \left[\left(\sum_{j=1}^{n} j^2\right) - \left(\sum_{j=1}^{1} j^2\right)\right] + 1$$

$$= \left[\left(\sum_{j=1}^{n} j^2\right) - (1)\right] + 1$$

$$= \sum_{j=1}^{n} j^2$$

$$= \frac{n(n+1)(2n+1)}{6}$$

$$= \frac{2n^2 + 3n^2 + n}{6}$$

$$= \frac{1}{3}n^3 + \frac{1}{2}n^2 + \frac{1}{6}n$$

As we did by LU decomposition, we will add $2n^2$ to account for the forward substitution and backward substitution. Therefore, the total flop count for solving a system of linear equations using Cholesky decomposition is:

$$\left(\frac{1}{3}n^3 + \frac{1}{2}n^2 + \frac{1}{6}n\right) + (2n^2)$$

$$= \frac{1}{3}n^3 + \frac{5}{2}n^2 + \frac{1}{6}n$$

The complexity is $\frac{1}{3}n^3$. Due to the symmetry of positive-definite matrices, the complexity of the method using Cholesky decomposition is half the complexity of the methods using classical Gaussian elimination and LU decomposition. Furthermore, like LU decomposition, when we have multiple b's, we only need to perform the expensive Cholesky decomposition step once.

To summarize, here are the flop counts and complexities for solving systems of equations with each of the three algorithms:

	Flop count for one *b*	**Complexity for one *b***	**Flop count for many *b*'s**	**Complexity for many *b*'s**
Gaussian elimination	$\frac{2}{3}n^3 + \frac{3}{2}n^2 - \frac{7}{6}n$	$\frac{2}{3}n^3$	$\frac{2}{3}kn^3 + \frac{3}{2}kn^2 - \frac{7}{6}kn$	$\frac{2}{3}kn^3$
LU decomposition	$\frac{2}{3}n^3 + \frac{3}{2}n^2 - \frac{1}{6}n$	$\frac{2}{3}n^3$	$\frac{2}{3}n^3 + \left(2k - \frac{1}{2}\right)n^2 - \frac{1}{6}n$	$\frac{2}{3}n^3 + \left(2k - \frac{1}{2}\right)n^2$
Cholesky decomposition	$\frac{1}{3}n^3 + \frac{5}{2}n^2 + \frac{1}{6}n$	$\frac{1}{3}n^3$	$\frac{1}{3}n^3 + \left(2k + \frac{1}{2}\right)n^2 + \frac{1}{6}n$	$\frac{1}{3}n^3 + \left(2k + \frac{1}{2}\right)n^2$

As mentioned earlier, the flop count is a crude measurement of how expensive an algorithm is. There are circumstances where algorithms with the same flop counts

are executed with extremely different amounts of time. Many more aspects need to be considered to accurately estimate the practical runtime of modern computers. Although counting flops is the traditional way to measure efficiency, memory access often takes much more time than floating point operations. Modern computers employ complex programming strategies to minimize the effect of memory access time. Considering this, let's test out the algorithms to see if their speeds are consistent with their flop counts and complexities.

Below is the Python code for a Monte Carlo simulation that runs 10,000 times. The method takes in three parameters: an algorithm, a matrix size, and a number of b's; it returns the average time, in seconds, that it takes to solve for x in the input situation. (For the sake of brevity, the code that I wrote for the Gaussian elimination with partial pivoting algorithm, the LU decomposition with partial pivoting algorithm, and the Cholesky decomposition algorithm is not included below but rather at the end.)

```
def monte_carlo_simulation(matrix_size, algorithm, num_of_bs):
	# Number of simulations to perform
	num_simulations = 10_000

	# Total time taken across all simulations
	total_time = 0

	# Initialize matrix A with a default zero matrix
	a = np.zeros((matrix_size, matrix_size))

	# Perform the simulation for the specified number of times
	for _ in range(num_simulations):
		# For Gaussian elimination or LU decomposition, generate a random matrix
		if algorithm == 'gaussian' or algorithm == 'lu':
			a = np.random.rand(matrix_size, matrix_size)
		# For Cholesky decomposition, generate a random positive definite matrix
		elif algorithm == 'cholesky':
			a = make_spd_matrix(matrix_size)
```

```
            # Generate a list with the specified number of random vector b's
            bs = [np.random.rand(matrix_size) for _ in range(num_of_bs)]

            # Record the start time for the algorithm
            start_time = time.time()

            # Perform specified algorithm for A and the list of b's
            if algorithm == 'gaussian':
                  gaussian_elimination_partial_pivoting(a, bs)
            elif algorithm == 'lu':
                  lu_decomposition_partial_pivoting(a, bs)
            elif algorithm == 'cholesky':
                  cholesky_decomposition(a, bs)

            # Record the end time for the algorithm
            end_time = time.time()

            # Update total_time with the current simulation duration
            total_time += end_time - start_time

      # Calculate the average time taken for the specified number of simulations
      average_time = total_time / num_simulations

      # Return the average time
      return average_time
```

Here are the time results, in seconds, for the Montecarlo simulation attained by running the code on my computer:

Size 5×5

	1 *b*	2 *b*'s	5 *b*'s	10 *b*'s	50 b's	100 b's
Gaussian	.00007434	.00014843	.00036508	.00073694	.00366659	.00748220
LU	.00013183	.00016111	.00025200	.00040987	.00165320	.00319800
Cholesky	.00008733	.00008966	.00013246	.00018647	.00069768	.00132936

Size 10 × 10

	1 *b*	2 *b*'s	5 *b*'s	10 *b*'s	50 b's	100 b's
Gaussian	.00024067	.00045850	.00114503	.00275699	.01150188	.02630285
LU	.00035559	.00043577	.00065333	.00113542	.00394715	.00865760
Cholesky	.00037932	.00042670	.00055092	.00074582	.00233612	.00504442

Size 50 × 50

	1 *b*	2 *b*'s	5 *b*'s	10 *b*'s	50 b's	100 b's
Gaussian	.00453289	.00946803	.04395937	.04522581	.25176332	.48880097
LU	.00584012	.00646787	.01290474	.01420678	.06483618	.11439137
Cholesky	.03881199	.04011419	.04184195	.04763783	.09026921	.13075287

Size 100 × 100

	1 *b*	2 *b*'s	5 *b*'s	10 *b*'s	50 b's	100 b's
Gaussian	.01706709	.03441374	.08761905	.19382505	.92928213	12.043899
LU	.02024834	.02475101	.03645452	.05772059	.21416254	.39270290
Cholesky	.34266684	.52231497	.31067891	.35240135	.87163078	.63579966

As expected, when *b* is greater than 1, the LU decomposition and Cholesky decomposition algorithms were much more efficient than the Gaussian elimination algorithm, especially when dealing with a lot of *b*'s or with a large matrix size. However, the speeds of the algorithms relative to each other did not match our

expectations. When dealing with 1 b, we expect the speed of Cholesky decomposition to be half the speed of the Gaussian elimination and LU decomposition algorithms. Yet, this was not the case. In fact, when the matrix size is 10 or higher, it was slower!

As stated earlier, it is exceedingly difficult to account for what is causing an algorithm to perform contrary to expectations because there are many aspects that affect the practical runtime of computers.

Python has the SciPy library which includes the scipy.linalg module which provides functionality for linear algebra calculations. Code utilizing SciPy is much faster than code written by even the skilled programmer because SciPy implements operations using compiled languages like C and Fortran and employs highly optimized, memory-efficient algorithms and data structures. SciPy provides methods to do LU decomposition and Cholesky decomposition but not classical Gaussian elimination. I did not use the available SciPy methods because, since they are much faster than the Gaussian elimination method that I coded, it would not be a fair comparison.

Nevertheless, it is worthwhile to examine the speed of LU decomposition versus the speed of Cholesky decomposition when they are optimized using SciPy. Let's see if Cholesky decomposition will now be double as fast as LU decomposition in the case of 1 b. Here are the results for different size matrices:

	LU	Cholesky
5 × 5	.000048573231697083	.000000200343132019
10 × 10	.000053895163536072	.000000399374961853
50 × 50	.000134746432304382	.000000399374961853
100 × 100	.004512627959251404	.000000696110725403
500 × 500	.020258081126213075	.000001306128501892
1000 × 1000	.058511302137374877	.000001292610168457

When using SciPy, Cholesky decomposition is several hundred times as fast as LU decomposition in the matrix size 5×5 case, and becomes increasingly more efficient relative to LU decomposition as the matrix size increases, until it reaches tens of thousands of times as fast as LU decomposition in the matrix size 1000×1000 case. The potential benefit of using Cholesky decomposition is clear.

Practical Application to Structural Engineering: Global Matrix Equations

A practical application of this discussion is to an area of structural engineering called Finite Element Analysis (FEA). FEA is a numerical method used for solving engineering problems related to the behavior of structures and components. Specifically, it is applicable to the global matrix equations:

$$Kq = f$$

where K is the global stiffness matrix, q is the vector of nodal displacements, and f is the vector of applied nodal forces.

The global stiffness matrix K is the matrix that is assembled by combining the stiffness matrices of individual elements that make up the structure. The stiffness matrix characterizes the stiffness of each element and how it relates to the overall structure.

The vector of nodal displacements q contains the displacement values for each degree of freedom at each node in the structure.

The vector of applied nodal forces f contains the force values applied at each degree of freedom at each node in the structure. These forces can come from external loads or other applied conditions.

Often, we want to analyze a structure under various loading conditions. To do this, we can define multiple loading vectors, each loading vector f corresponding to the applied forces for that unique case. Then, we can solve $Kq = f$ for q many times to find the nodal displacements for each condition. This parallels our discussion about

solving $Ax = b$ with multiple b's. If we want to solve $Kq = f$ with multiple f's, Cholesky decomposition and LU decomposition will be much more efficient. (K is always positive-definite, so we can use Cholesky decomposition. The reason for its positive-definiteness is beyond the scope of this project.)

Python Code for Algorithms

The following methods solve $Ax = b$ for x for all b's in a list of b's. They take in two parameters: a matrix and a list of b's; they return a list of the found x's.

Gaussian Elimination with Partial Pivoting

```
def gaussian_elimination_partial_pivoting(a, bs):
	# Initialize an empty list to store solutions
	x_list = []

	# Convert matrix A to float type
	a = a.astype(float)

	# Get the size of matrix A
	n = len(a)

	# Initialize solution vector x with a default 0 vector
	x = np.zeros(n)

	# Iterate over each vector b
	for i in range(len(bs)):
		# Get the current vector b
		b = bs[i]

		# Convert vector b to float type
		b = b.astype(float)

		# Create the augmented matrix [A|b]
		augmented_matrix = np.hstack((a, b.reshape(-1, 1)))

		# Iterate over the matrix rows
		for j in range(n - 1):
			# Find the row with the maximum absolute value in the current
			# column
			max_absolute_value_row =
				np.argmax(np.abs(augmented_matrix[j:, j])) + j

			# Swap the current row with the row with the maximum absolute
			# value in the current column if necessary
			if max_absolute_value_row != j:
				augmented_matrix[[j, max_absolute_value_row]] =
					augmented_matrix[[max_absolute_value_row, j]]

			# Iterate over the matrix rows
			for k in range(j + 1, n):
				# Calculate the multiplier
				multiplier =
					augmented_matrix[k, j] / augmented_matrix[j, j]

			# Update the elements in all necessary columns in the current row
			augmented_matrix[k, j + 1:] -=
				multiplier * augmented_matrix[j, j + 1:]
```

```
            # Backward substitution to find the solution vector x
            x[n - 1] = augmented_matrix[n - 1, n] / augmented_matrix[n - 1, n - 1]

            for j in range(n - 2, -1, -1):
                  temp = augmented_matrix[j, -1]

            for k in range(j + 1, n):
                  temp -= augmented_matrix[j, k] * x[k]
                  x[j] = temp / augmented_matrix[j, j]

      # Append x to the solution list
      x_list.append(x.copy())

      # Return the list of solutions
      return x_list
```

LU Decomposition with Partial Pivoting

```
def lu_decomposition_partial_pivoting(a, bs):
      # Initialize an empty list to store solutions
      x_list = []

      # Convert matrix A to float type
      a = a.astype(float)

      # Get the size of matrix A
      n = len(a)

      # Initialize matrix L with a default 0 matrix
      lower = np.zeros((n, n))

      # Initialize matrix U with a copy of matrix A
      u = a.copy()

      # Initialize permutation matrix with an identity matrix
      p = np.eye(n)

      # Iterate over the matrix rows
      for k in range(n - 1):
            # Find the row with the maximum absolute value in the current column
            pivot_index = np.argmax(np.abs(u[k:, k])) + k

            # Swap the current row with the row with the maximum absolute value in
            # the current column if necessary
            if pivot_index != k:
                  u[[k, pivot_index], :] = u[[pivot_index, k], :]

                  # Perform corresponding swaps in matrix P
                  p[[k, pivot_index], :] = p[[pivot_index, k], :]

                  # Perform corresponding swaps in matrix L
                  lower[[k, pivot_index], :] = lower[[pivot_index, k], :]

            # Iterate over the matrix rows
            for i in range(k + 1, n):
```

```
                # Calculate the multiplier
                multiplier = u[i, k] / u[k, k]

                # Store multiplier in matrix L
                lower[i, k] = multiplier

                # Update the elements in all necessary columns in the current row
                u[i, k:] -= multiplier * u[k, k:]

    # Fill the diagonal of matrix L with 1's
    np.fill_diagonal(lower, 1)

    # Initialize vector c and solution vector x with default 0 vectors
    c = np.zeros(n)
    x = np.zeros(n)

    # Iterate over each vector b
    for i in range(len(bs)):
            # Get the current vector b
            b = bs[i]

            # Convert vector b to float type
            b = b.astype(float)

            # Initialize permuted vector b with a default 0 vector
            pb = np.zeros_like(b)

            # Permute vector b
            for j, row in enumerate(p):
                    one_index = np.where(row == 1)[0][0]
                    pb[j] = b[one_index]

            # Forward substitution to solve lower triangular system Lc = permuted b
            # for vector c
            c[0] = pb[0] / lower[0, 0]
            for j in range(1, n):
                    temp = pb[j]

                    for k in range(j):
                            temp -= lower[j, k] * c[k]

                    c[j] = temp / lower[j, j]

            # Backward substitution to solve upper triangular system Ux = c for
            # solution vector x
            x[n - 1] = c[n - 1] / u[n - 1, n - 1]

            for j in range(n - 2, -1, -1):
                    temp = c[j]

                    for k in range(j + 1, n):
                            temp -= u[j, k] * x[k]

                    x[j] = temp / u[j, j]

            # Append the solution to the list
            x_list.append(x.copy())

    # Return the list of solutions
    return x_list
```

Cholesky Decomposition

```
def cholesky_decomposition(a, bs):
	# Initialize an empty list to store solution vectors
	x_list = []

	# Get the size of matrix A
	n = a.shape[0]

	# Initialize upper matrix R with a copy of matrix A
	r = np.copy(a)
	r = r.astype(float)

	# Perform Cholesky decomposition
	for i in range(n - 2):
		r[i, i] = np.sqrt(r[i, i])
		r[i, i + 1:] = r[i + 1:, 0] / r[i, i]
		r[i + 1:, i] = 0
		submatrix_c = r[i + 1:, i + 1:]
		outer_product_b_div_sqrt_a = np.zeros((n - 1 - i, n - 1 - i))

		for j in range(n - 1 - i):
			for k in range(j, n - 1 - i):
				outer_product_b_div_sqrt_a[j, k] =
					r[i, i + 1:][j] * r[i, i + 1:][k]
				outer_product_b_div_sqrt_a[k, j] =
					outer_product_b_div_sqrt_a[j, k]

		for j in range(n - 1 - i):
			for k in range(j, n - 1 - i):
				r[i + 1 + j, i + 1 + k] =
					submatrix_c[j, k] - outer_product_b_div_sqrt_a[j, k]
				r[i + 1 + k, i + 1 + j] = r[i + 1 + j, i + 1 + k]

	r[n - 2, n - 2] = np.sqrt(r[n - 2, n - 2])
	r[n - 2, n - 1] = r[n - 2, n - 1] / r[n - 2, n - 2]
	r[n - 1, n - 2] = 0
	r[n - 1, n - 1] -= r[n - 2, n - 1] * r[n - 2, n - 1]
	r[n - 1, n - 1] = np.sqrt(r[n - 1, n - 1])

	# Transpose R to get R^T
	r_transpose = r.T

	# Initialize vector c and solution vector x with default 0 vectors
	c = np.zeros(n)
	x = np.zeros(n)

	# Iterate over each vector b
	for i in range(len(bs)):
		# Get the current vector b
		b = bs[i]

		# Convert vector b to float type
		b = b.astype(float)

		# Forward substitution to solve lower triangular system R^Tc = b for
		# vector c
		c[0] = b[0] / r_transpose[0, 0]

		for j in range(1, n):
```

```
                temp = b[j]

                for k in range(j):
                        temp -= r_transpose[j, k] * c[k]

                c[j] = temp / r_transpose[j, j]

        # Backward substitution to solve upper triangular system Rx = c for
        # solution vector x
        x[n - 1] = c[n - 1] / r[n - 1, n - 1]

        for j in range(n - 2, -1, -1):
                temp = c[j]

                for k in range(j + 1, n):
                        temp -= r[j, k] * x[k]

                x[j] = temp / r[j, j]

        # Append the solution to the list
        x_list.append(x.copy())

# Return the list of solutions
return x_list
```

SciPy Lu Decomposition with Partial Pivoting

```
def scipy_lu_decomposition_partial_pivoting(a, bs):
        # Initialize list to store solution vectors
        x_list = []

        # Perform LU decomposition with partial pivoting using SciPy's method
        P_indices, l, u = lu(a, p_indices=True)

        # Iterate over each vector b
        for i in range(len(bs)):
                # Get the current vector b
                b = bs[i]

                # Convert vector b to float type
                b = b.astype(float)

                # Permute vector b
                pb = b[p]

                # Forward substitution to solve lower triangular system Lc = permuted b
                # for vector c
                c = solve_triangular(l, pb, lower=True)

                # Backward substitution to solve upper triangular system Ux = c for
                solution vector x
                x = solve_triangular(u, c)

                # Append the solution to the list
                x_list.append(x.copy())
```

```
        # Return the list of solutions
        return x_list
```

SciPy Cholesky Decomposition

```
def scipy_cholesky_decomposition(a, bs):
        # Initialize an empty list to store solution vectors
        x_list = []

        # Perform Cholesky decomposition using SciPy's method
        r = cholesky(a)

        # Iterate over each vector b
        for i in range(len(bs)):
                # Get the current b
                b = bs[i]

                # Convert b to float type
                b = b.astype(float)

                # Forward substitution to solve lower triangular system R^Tc = b for
                # vector c
                c = solve_triangular(r.T, b, lower=True)

                # Backward substitution to solve upper triangular system Rx = c for
                # solution vector x
                x = solve_triangular(r, c)

                # Append the solution to the list
                x_list.append(x.copy())

        # Return the list of solutions
        return x_list
```

Bibliography

Boyd, Stephen, and Lieven Vandenberghe. *Introduction to applied linear algebra: vectors, matrices, and least squares*. Cambridge university press, 2018.

Burden, Richard L., and J. Douglas Faires. *Numerical analysis*. Boston, MA: Brooks/Cole, Cengage Learning, 2011.

Chernov, Nikolai. Numerical Linear Algebra. 2014. https://people.cas.uab.edu/~mosya/teaching/660new3.pdf.

Eldén, Lars. *Matrix methods in data mining and pattern recognition*. Philadelphia: SIAM, Society for Industrial and Applied Mathematics, 2019.

Hunger, Raphael. *Floating point operations in matrix-vector calculus*. Version 1.3. Munich, Germany: Munich University of Technology, Inst. for Circuit Theory and Signal Processing, 2007.

Kim, Nam-Ho, Bhavani V. Sankar, and Ashok V. Kumar. *Introduction to finite element analysis and design*. John Wiley & Sons, 2018.

Layton, William, and Myron Sussman. "Numerical linear algebra." University of Pittsburgh, Pittsburgh (2014): 28-39.

Lu, Jun. "Numerical matrix decomposition and its modern applications: A rigorous first course." arXiv preprint arXiv:2107.02579 (2021).

Sauer, Tim. *Numerical analysis*. 3rd ed. Hoboken: Pearson, 2019.

Tibshirani, Ryan. “Convex Optimization Lecture 19.” 2015. https://www.stat.cmu.edu/ ~ryantibs/convexopt-F18/scribes/Lecture_19.pdf.

Trefethen, Lloyd N., and David Bau. *Numerical linear algebra*. Philadelphia: Society for Industrial and Applied Mathematics, 2022.

Watkins, David S. *Fundamentals of Matrix Computations*. Oxford: Wiley, 2010.

Wong, Jeffrey. *Linear systems*. Edited by Holden Lee. 2020. https://services.math.duke.edu/~holee/math361-2020/lectures/Lec3-LinearSystems.pdf.