Strengths and Weaknesses for Methods of Solving Systems

Alina Enikeeva

March 2024

1 Purpose

1.1 Abstract

This third laboratory in the series explores various methods for finding the solution to systems of equations. This report compares the benefits and drawbacks of Naive Gaussian Elimination, PA = LU factorization, the Jacobi Method, and Multivariate Newton's Method.

1.2 Introduction

Naive Gaussian Elimination: performed on a matrix by a series of row operations (interchanging rows, adding or subtracting a multiple of one row from another, and multiplying rows by nonzero constants) (Sauer). This method gets the matrix in a triangular form and then uses back substitution to solve for the variables of the system. When performed on a computer, it requires less storage in memory compared to other methods for solving systems; however, the algorithm is not considered to be very stable, often compounding rounding errors. The decomposition and back substitution also have to be re-computed for every new right-hand-side vector.

PA = LU Factorization: finds the solution to a system of equations by decomposing a matrix into the product of a lower triangular matrix (L) and an upper triangular matrix (U) (Sauer). The permutation matrix (P) is the identity matrix or a variation of it that reflects any row changes that may have occurred. Numerical stability can be preserved for PA = LU factorization by implementing pivoting techniques (Mayorov). The benefits and drawbacks of this method are explored in Sections 2 and 3 of this laboratory.

Jacobi Method: uses an initial guess for the solution of the system that is "plugged in" to get a new vector guess. This process repeats until the numbers stabilize to yield a final solution. It has guaranteed convergence for diagonally dominant matrices (Sauer). While neither PA = LU factorization or Naive Gaussian Elimination are suited for sparse matrices, the Jacobi method is. The advantages and disadvantages of this method are further investigated in Section 3 of this laboratory.

Multivariate Newton's Method: similar to Newton's method (explored in the second laboratory of this series). This version uses a vectorized version of f(x) to solve a system of equations. This method requires knowledge of the derivative (calculated as the Jacobian of the vectorized f(x) and also requires that the derivative does not equal zero when used in the computations. This method is investigated in Section 4 of this laboratory.

2 PA = LU Factorization for Linear Systems

2.1 Why is PA = LU Factorization Needed?

PA = LU Factorization is generally more efficient than Gaussian Elimination (Sauer). Gaussian elimination requires $O(n^3)$ operations to get the matrix in a triangular form and an additional $O(n^2)$ operations for back substitution. Similarly, PA = LU Factorization uses $O(n^3)$ operations for the decomposition of the original matrix and $O(n^2)$ operations to solve the resulting linear systems; however, the expensive part of PA = LU factorization can be done without knowing the vector on the right-hand side—so once the decomposition of the A matrix is done, the system $Ax = b$ can be solved for various b vectors without requiring further decomposition.

Additionally, Naive Gaussian elimination is often known for being prone to large errors. Suppose we have:

$$A = \begin{bmatrix} 1.000 & 2.000 \\ 0.999 & 2.000 \end{bmatrix}, \quad b = \begin{bmatrix} 3.000 \\ 2.999 \end{bmatrix}$$

Alternatively, PA = LU factorization provides the following decomposition (which can be used for any other b vector):

$$L = \begin{bmatrix} 1.000 & 0.000 \\ 0.999 & 1.000 \end{bmatrix}, \; U = \begin{bmatrix} 1.000 & 2.000 \\ 0.000 & 0.002 \end{bmatrix}, \; P = \begin{bmatrix} 1.000 & 0.000 \\ 0.000 & 1.000 \end{bmatrix}$$

$$\xrightarrow{\text{Performing } y = L\backslash(P * b), \text{ and then } x = U\backslash y \text{ in Matlab yields:)}} \quad x = \begin{bmatrix} -5.9984 \\ 2.9994 \end{bmatrix}$$

We can see solving the same system in Matlab with Naive Guassian Elimination inaccurately yeilds:

$$x = \begin{bmatrix} 10.000 \\ -3.5000 \end{bmatrix}$$

2.2 How to identify systems Ax = b for which PA = LU is not suited

PA = LU factorization is not well-suited for sparse matrices (matrices with many zero entries), since resulting permutation matrices tend to be dense (Sauer). In these cases, the Jacobi method may be a better option (the Jacobi method is explored in the next section). Most importantly, this factorization does not work well for singular matrices, since there is often not a unique LU factorization in these cases. Take the following example of a singular matrix (see Figure 2.2 for code):

$$A = \begin{bmatrix} 2 & 8 \\ 1 & 4 \end{bmatrix} \xrightarrow{\text{PA = LU factorization in Matlab yields}} x = \begin{bmatrix} -inf \\ inf \end{bmatrix}$$

.

Additionally, ill-conditioned matrices are sensitive to small changes in input and can cause errors during PA = LU factorization. Ill-conditioned matrices can be identified by calculating the condition number with $||A^{-1}||\,||A||$, where $||A||$ is the matrix norm.

2.3 Applications of PA = LU Factorization

Decomposition a matrix is the basis for inverting a matrix (Agarwal). The inverse of a matrix is applicable in many algorithms, including in some variations of Multivariate Newton's Method that use the inverse of the Jacobian matrix of a system. PA = LU factorization is especially applicable in situations where the right-hand side b vectors are not known in advanced (Mayorov).

3 Iterative solutions of systems of linear equations

3.1 Using Jacobi to solve the 100,000 equation version of Ex 2.24

Jacobi FPI continues iterating until a convergence criterion is met, such as reaching a specified tolerance level or a maximum number of iterations. The application of the method is illustrated in the code appendix (Figure 3.1) as well as in plot appendix (Plot 3.2.1 and Plot 3.2.2). Jacobi Fixed-Point Iteration (FPI) is an iterative method used to solve systems of linear equations. It is particularly useful for large sparse matrices where traditional methods like Gaussian elimination or PA = LU become computationally expensive due to the memory and processing requirements. The method starts with an initial guess for the solution vector and iteratively refines this guess until a satisfactory solution is obtained. At each iteration, the solution vector is updated using a formula derived from the original system of equations. In the case of Jacobi iteration for solving the linear system $Ax = b$, the update formula for the i-th component of the solution vector x is given by:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)} \right)$$

where:

- $x_i^{(k)}$ is the i-th component of the solution vector at the k-th iteration,
- a_{ii} is the diagonal element of matrix A,
- b_i is the i-th component of vector b, and
- a_{ij} are the off-diagonal elements of matrix A.

Jacobi FPI continues iterating until a convergence criterion is met, such as reaching a specified tolerance level or a maximum number of iterations. The application of the method is illustrated in the code appendix (Figure 3.1) as well as in plot appendix (Plot 3.2.1 and Plot 3.2.2).

3.2 Comparison of $PA = LU$ and Jacobi Iteration

In this section, we compare the performance of two methods, $PA = LU$ decomposition and Jacobi iteration, for solving the large linear system described in Example 2.25.

3.2.1 Maximum n for $PA = LU$ and Jacobi Iteration

The $PA = LU$ decomposition method requires the factorization of the matrix A into the product of a permutation matrix P, a lower triangular matrix L, and an upper triangular matrix U. The memory and computational requirements for this method increase significantly as n, the size of the matrix, grows. The largest n for which $PA = LU$ decomposition can solve this problem depends on the available memory and computational resources. However, for most practical applications, $PA = LU$ decomposition becomes infeasible for very large n due to memory limitations.

On the other hand, Jacobi iteration is an iterative method that updates the solution vector x until convergence is achieved. It has less stringent memory requirements compared to $PA = LU$ decomposition but can handle very large systems efficiently. The largest n that Jacobi iteration can handle depends on the convergence rate and computational resources available.

3.2.2 Comparison of $PA = LU$ and Jacobi Iteration

$PA = LU$ decomposition is a direct method that guarantees an exact solution to the linear system. However, it may not be practical for very large systems due to memory limitations and computational complexity. Additionally, it requires storing and manipulating the entire matrix A, which can be inefficient for sparse matrices. One of the biggest benefits of this factorization is that the decomposition of the A matrix into the product of the L and U matrices only needs to be done once for any b vector. This is the most expensive step, requiring $O(n^3)$ operations, but thereafter for any b vector the back substitution only requires $O(n^2)$ operations.

The Jacobi Method, on the other hand, is an iterative method that updates the solution vector x until convergence. It has lower memory requirements and can handle large sparse systems efficiently. This method, while having guaranteed convergence for strictly diagonal matrices, does not guarantee convergence for all matrices. It also may converge slowly for certain types of matrices, especially those with large condition numbers. Unlike PA = LU factorization, the Jacobi method is well-suited for sparse matrices. Additionally, it is efficient for systems where we already have a good approximation to the solution to use as our initial guess.

The choice between the two methods depends on the specific requirements of the problem and the available computational resources.

3.3 Importance of Solving Large Linear Systems in Applications

Solving large linear systems is crucial in various fields and applications, including:

- **Engineering and Science:** Large linear systems often arise in engineering simulations, scientific computing, and numerical modeling. For example, in computational fluid dynamics, solving systems of equations govern the flow of fluids around complex geometries.
- **Finance and Economics:** Linear systems are used in financial modeling, risk assessment, and portfolio optimization. Large systems of equations may represent interconnected financial markets, optimization problems, or forecasting models.
- **Machine Learning and Data Analysis:** Linear systems are fundamental in machine learning algorithms, such as linear regression, least squares, and principal component analysis. Solving large systems enables the analysis of vast datasets and the training of complex models.
- **Infrastructure and Planning:** Large linear systems are used in urban planning, transportation networks, and energy distribution. Solving these systems helps optimize resource allocation, improve infrastructure design, and enhance operational efficiency.

In summary, solving large linear systems is essential for addressing complex problems in diverse fields. Efficient algorithms and computational techniques play a crucial role in enabling advancements and innovation across various applications.

4 Implement Multivariate Newton's Method

4.1 Newton's Method using Vectorization

Newton's method for vectors and vector functions uses the following formula:

$$Df(x)\Delta x = -f(x), \quad \text{where} \quad Df(x) = \text{the Jacobian} = \begin{bmatrix} g_u & g_v \\ h_u & h_v \end{bmatrix}$$

For a system of functions $g(u, v)$ and $h(u, v)$, the `MyVectorNewton` function correctly vectorizes Newton's method, approximating the solution vector. The Matlab Code for this function is pictured in Figure 4.1. It uses the inputs 'f' (the vector function), 'Df' (the Jacobian of the function), '$xinitialvector$' (the initial guess for the method), 'TOL' (a tolerance to check for convergence), and '$MaxIters$' (a given maximum number of iterations) to calculate the ouput (the solution vector).

```
function [xsoln, fl] = MyVectorNewton(f, Df, xinitialvec, TOL, MaxIters)
    fl = 0; % initialize flag to zero, assume divergence until convergence
    x = xinitialvec; % initialize x

    for k = 1: MaxIters % iterate through the maximum number of iterations
        dx = -Df(x) \ f(x); % define dx
        x = x + dx; % define x

        if norm(dx, inf) < TOL % check for convergence
            fl = 1; % update flag if there is convergence
            xsoln = x; % update output to the convergent value of x
            return; % end loop if solution is within convergence tolerance
        end % end of if statement

    end % end for loop once maximum iteration is reached

    xsoln = x; % update output to the most recent value of x

end
```

Figure 4.1: Code for the `MyVectorNewton` function

4.2 Testing the code

The given code snippet utilizes the Newton method to solve a system of nonlinear equations. The system consists of three equations defined by the function f and its Jacobian matrix Df. The equations are as follows:

$$f_1(x) = 2x_1^2 - 4x_1 + x_2^2 + 3x_3^2 + 6x_3 + 2$$
$$f_2(x) = x_1^2 + x_2^2 - 2x_2 + 2x_3^2 - 5$$
$$f_3(x) = 3x_1^2 - 12x_1 + x_2^2 + 3x_3^2 + 8$$

The Jacobian matrix Df is also defined accordingly. The initial guess for the solution vector x is set to $[0; 0; 0]$. The code iterates using the `MyVectorNewton` function, aiming to find the solution vector $xsoln$ within a tolerance of 10^{-12} and a maximum of 10 iterations. Upon completion, the solution vector $xsoln$ and the exit flag fl are displayed, providing insights into the convergence behavior of the Newton method for this particular system of equations.

```
clc
f = @(x)[2*x(1)^2 - 4*x(1) + x(2)^2 + 3*x(3)^2 + 6*x(3) + 2;
         x(1)^2 + x(2)^2 - 2*x(2) + 2*x(3)^2 - 5;
         3*x(1)^2 - 12*x(1) + x(2)^2 + 3*x(3)^2 + 8];
Df = @(x)[4*x(1)-4, 2*x(2), 6*x(3)+6;
          2*x(1), 2*x(2)-2, 4*x(3);
          6*x(1)-12, 2*x(2), 6*x(3)];

[xsoln, fl] = MyVectorNewton(f, Df, [0; 0; 0], 1e-12, 10);

% Display the solution
disp('Solution for the set of equations:');
disp(xsoln);
disp('Exit flag:');
disp(fl);
```

```
Command Window
Solution for the set of equations:
   1.096017841000413
  -1.159247184210588
  -0.261147936702016

Exit flag:
     1
```

Figure 4.2: Code for the `MyVectorNewtonTest` function

4.3 Justification for Regularization in Newton Method

The inclusion of regularization in the Newton function enhances the stability and convergence of the Newton method by mitigating issues with singular or ill-conditioned Jacobian matrices. Refer to Figure 4.3 in the code appendix for implementation details.

- **Singularity Handling**:
  - When the Jacobian matrix becomes singular, the Newton-Raphson method fails because it involves matrix inversion.Singularity often occurs when the system of equations is ill-conditioned or when there's a dependency among equations.
  - Regularization introduces a small diagonal matrix (scaled by a regularization parameter) to the Jacobian, making it non-singular.
  - This regularization helps stabilize the inversion process, preventing numerical instability and allowing the method to continue iterations.
- **Convergence Improvement**:
  - Regularization can improve the convergence behavior of the Newton-Raphson method, especially when dealing with ill-conditioned systems.
  - By adding a small positive value to the diagonal elements of the Jacobian, regularization ensures that the matrix remains well-conditioned and facilitates smoother convergence towards the solution.
- **Robustness Enhancement**:
  - Regularization enhances the robustness of the Newton-Raphson method by making it less sensitive to small perturbations in the system of equations.
  - It helps prevent divergence or erratic behavior that may occur due to numerical inaccuracies or floating-point errors.
- **Controlled Modification**:
  - The regularization parameter allows for controlled adjustment of the regularization strength.
  - In the provided code, the regularization parameter λ can be adjusted based on the characteristics of the system of equations and the desired convergence behavior.

5 Conclusion

5.1 Results

In conclusion, we cannot really say one method for solving a system of equations is better than another.

Section 2: illustrated that PA = LU factorization tends to be fairly more accurate than Guassian Elimination and more efficient when solving systems for multiple right-hand-side b vectors.

Section 3: explored how PA = LU decomposition provides an exact solution but may not be feasible for very large systems. In contrast, the Jacobi iteration, while only approximate, can handle large systems efficiently with lower memory requirements; however, it does not guarantee convergence for non-diagonally dominant matrices.

Section 4: investigated the vectorization of Newton's method to solve systems of equations using a guess for an initial vector. We saw that it can accurately converge to solutions. There are many ways to improve the algorithm to make up for its shortcomings, including enhancements that allow the method to work better for singular and ill-conditioned matrices. Each algorithm has its own advantages and disadvantages, so deciding on which method to implement truly depends on the problem at hand.

5.2 Teamwork Statement

This laboratory continued to be a equally collaborative effort between both members of the team. Erina focused primarily on question two, and Alina worked more on question three. The work for question four and all the writing sections were split evenly down the middle.

5.3 Future Research

It would be interesting to see which methods for solving systems of equations are more robust overall. For example, if we get a computer to randomly generate a set of 1000 matrices, and have each method solve all of the matrices, we could compare a variety of factors. Not only could we look at the accuracy of the solutions, but we could also note the time it takes for each algorithm to generate a solution, if it is able to generate a solution at all for a given matrix. Ultimately, each algorithm has its own strengths and will perform better than other algorithms for specific systems.

6 References

Agarwal, Monica, and Rajesh Mehra. "Title of the Paper." Journal of Engineering Research and Applications (IJERA), vol. 4, no. 1, Jan. 2014, www.ijera.com/papers/Vol4_issue1/Version/202/G41064754.pdf.

Mayorov, Alexander. "Using the PA=LU Factorization to Solve Linear Systems of Equations for Many Right-Hand Sides Efficiently.", 27 Nov. 2020, zerobone.net/blog/cs/pa-lu-factorization/.

Sauer, Tim. *Numerical Analysis*. Pearson, 2018.

7 Code Appendix

```
>> A = [1, 2, 0.999, 2];
>> [L, U, P] = lu(A)

L =

     1


U =

    1.0000    2.0000    0.9990    2.0000


P =

     1
```

```
>> y = L \ (P*b)

y =

    3.0000
    1.4990

>> x = U \ y
Warning: Matrix is singular to working precision.

x =

  -Inf
   Inf
```

Figure 2.2: Code for the PA=LU factorization for the 2x2 Singular Matrix A

```
% Define the size of the system
n = 12;

% Initialize the sparse matrix A and the vector b

b = zeros(n, 1);

% Define the diagonals of A
main_diag = 3 * ones(n, 1);
sub_super_diag = -1 * ones(n-1, 1);

% Set the values of the main diagonal
A = diag(main_diag);
A = A + diag(sub_super_diag, 1) + diag(sub_super_diag, -1);

% Set the values of the off-diagonal elements (i, n + 1 - i)
for i = 1:n
    if i ~= n/2 && i ~= n/2 + 1
        A(i, n + 1 - i) = 0.5;
    end
end

% Define the vector b
b(1) = 2.5;
b(n) = 2.5;
b(2:n-1) = 1.5;
b(n/2) = 1.0;
b(n/2 + 1) = 1.0;

% Use Jacobi iteration to solve the system
x = jacobi(A, b, 100000);

% Display the solution vector x
disp('Solution vector x:');
fprintf('x = %.6f\n', x);
```

```
%2.5 Iterative Methods | 115
% Program 2.2 Jacobi Method
% Inputs: full or sparse matrix a, r.h.s. b,
% number of Jacobi iterations, k
% Output: solution x
function x = jacobi(a,b,k)
n=length(b); % find n
d=diag(a); % extract diagonal of a
r=a-diag(d); % r is the remainder
x=zeros(n,1); % initialize vector x
for j=1:k % loop for Jacobi iteration
    x = (b-r*x)./d;
end % End of Jacobi iteration loop
```

Figure 3.1: Code for the Jacobi FPI for the 12x12 Singular Matrix A with 100,000 iterations

```
Solution vector x:
x = 0.976852
x = 0.893519
x = 0.912037
x = 0.856481
x = 0.837963
x = 0.768519
x = 0.768519
x = 0.837963
x = 0.856481
x = 0.912037
x = 0.893519
x = 0.976852
```

```
>> lab3_jacobi
Solution vector x:
x = 1.000000
x = 1.000000
x = 1.000000
x = 1.000000
x = 1.000000
x = 1.000000
x = 1.000000
x = 1.000000
x = 1.000000
x = 1.000000
x = 1.000000
x = 1.000000
```

Figure 3.1: 3 iterations vs. 100000 iterations results of convergence

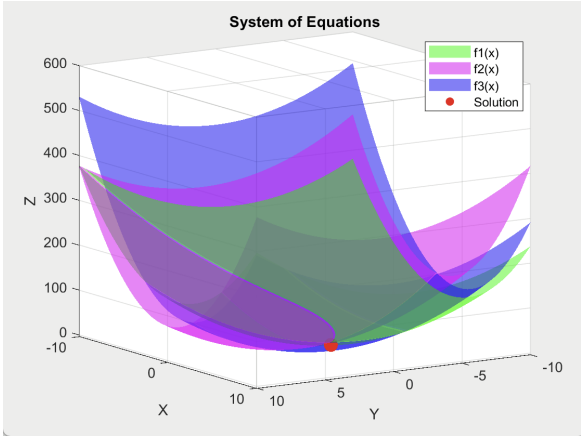
```
function [xsoln, fl] = MyVectorNewtonRegularized(f, Df, xinitialvec, TOL, MaxIters, lambda)
    fl = 0; % Initialize flag to zero, assume divergence until convergence
    x = xinitialvec; % Initialize x

    for k = 1:MaxIters % Iterate through the maximum number of iterations
        J = Df(x);
        J_reg = J + lambda * eye(size(J)); % Regularize the Jacobian matrix

        dx = -J_reg \ f(x); % Define dx
        x = x + dx; % Update x

        if norm(dx, inf) < TOL % Check for convergence
            fl = 1; % Update flag if there is convergence
            xsoln = x; % Update output to the convergent value of x
            return; % End loop if solution is within convergence tolerance
        end
    end

    xsoln = x; % Update output to the most recent value of x
end
```

```
Command Window

Regularized Method:
Solution:
   0.155112990339408
   0.537122180404068
   1.811283639943853
   1.462984535387763
   1.488745612864590
```

```
clc

% Define the system of equations and its Jacobian
f = @(x) [x(1)^2 - x(2) + 1;
          x(1) - x(2)^3 + x(3) - 2;
          x(1)^2 + x(2)^2 + x(3)^2 - 3;
          x(4)^2 + x(5)^2 - 2;
          x(4)*x(5) - 1];
Df = @(x) [2*x(1), -1, 0, 0, 0;
           1, -3*x(2)^2, 1, 0, 0;
           2*x(1), 2*x(2), 2*x(3), 0, 0;
           0, 0, 0, 2*x(4), 2*x(5);
           x(5), x(4), 0, x(5), x(4)];

% Call the MyVectorNewtonRegularized function to solve the system
[xsoln, fl] = MyVectorNewtonRegularized(f, Df, ones(5, 1), 1e-12, 50, 1e-6); % Regularization parameter (lambda) is set to 1e-6

% Display the solution
disp('Solution:');
disp(xsoln);
disp('Exit flag:');
disp(fl);
```

Figure 4.3: `MyVectorNewtonRegularized` function and solution for n=5 matrix

8 Plot Appendix



Plot 3.2.1: Plot for the system of equations from 3.2 example

```
% Define the functions for the system of equations
f1 = @(x, y) 2*x.^2 - 4*x + y.^2 + 3*y + 6 + 2;
f2 = @(x, y) x.^2 + y.^2 - 2*y + 2 + 2*x.^2 - 5;
f3 = @(x, y) 3*x.^2 - 12*x + y.^2 + 3 + 8;

% Generate data points for plotting
x = linspace(-10, 10, 100);
y = linspace(-10, 10, 100);
[X, Y] = meshgrid(x, y);

% Compute the values of each equation for the grid of points
F1 = f1(X, Y);
F2 = f2(X, Y);
F3 = f3(X, Y);

% Plot each equation as a surface
figure;
surf(X, Y, F1, 'FaceAlpha', 0.5, 'EdgeColor', 'none', 'FaceColor', 'g');
hold on;
surf(X, Y, F2, 'FaceAlpha', 0.5, 'EdgeColor', 'none', 'FaceColor', 'm');
surf(X, Y, F3, 'FaceAlpha', 0.5, 'EdgeColor', 'none', 'FaceColor', 'b');

% Define solution vector
solution = [ 1.1,-1.2,-0.3];
% Plot solution vector as scatter points
scatter3(solution(1), solution(2), solution(3), 100, 'r', 'filled');

xlabel('X');
ylabel('Y');
zlabel('Z');
title('System of Equations');
legend('f1(x)', 'f2(x)', 'f3(x)', 'Solution', 'Location', 'best');
view(3);
hold off;

```

Plot 3.2.1: Code for the plot of the 3.2 example