

# Solving Systems of Equations, Errors and Explorations

David Tran and Spencer Kelly

March 10, 2024

## Abstract

This lab is the 3rd in a series of 4 labs exploring various numerical methods, implementing them, and examining their tradeoffs. In this lab, we explore the  $PA = LU$  factorization method for solving systems of linear equations, as well as the Jacobi fixed-point iteration method and multivariate Newton's method. We compare the convergence of the Jacobi FPI and the  $PA = LU$  factorization method, and we compare the convergence of Newton's method for different systems of equations. We find that the Jacobi FPI is a simple and easy-to-implement method for solving systems of linear equations, but it is not the most efficient method. We also find that the  $PA = LU$  factorization method is not suited for solving systems of linear equations when the matrix is large, and that the Jacobi FPI is much faster and more reliable for this problem. We also find that Newton's method is a very fast and reliable method for solving systems of equations, and that plotting the convergence of Newton's method allows one to understand the rate of convergence and the behavior of the system of equations.

## 1 Introduction

(The citations for the following are found in the references section at the end of the document.)

The problem of solving systems of linear equations forms the historical roots of linear algebra. It was deeply explored by Rene Descartes, who introduced the geometrical interpretation of solutions to systems as intersections in hyper-dimensional Cartesian space. Later, both Leibniz and Gauss independently developed methods for solving systems of linear equations, with the latter being the namesake for Gaussian elimination.

The methods explored in this lab, namely,  $PA = LU$  factorization, Jacobi fixed-point iteration, and Newton's method, are all later methods for solving systems of linear equations. The  $PA = LU$  factorization method is a direct substitution method, which means that it is a method that directly computes the solution to the system of linear equations. The Jacobi fixed-point iteration and Newton's method are iterative methods, which means that they are methods that use an initial guess to iteratively compute the solution to the system of linear equations. Both have their tradeoffs, and we will explore these tradeoffs in this lab. We will see why Newton's method is one of the more popular methods for solving systems of equations in modern times.

## 2 The $PA = LU$ factorization method for linear systems

### 2.1 Why is $PA = LU$ needed for solving linear systems approximately?

When solving linear systems of the form  $Ax = b$ , we begin by gaussian elimination of the matrix  $A$ , followed by back substitution, and ultimately arrive at our solution. However, when a particular

matrix  $A$  is being used for multiple iterations, the overhead involved can become quite an obstacle. This is because the process of Gaussian elimination is a computationally expensive process, with complexity on the order  $O(n^3)$ . But with  $PA = LU$  factorization, we essentially remove the overhead involved with Gaussian elimination, for all but the first iteration, by rewriting the matrix  $A$  in terms of the upper and lower matrices  $L$ , and  $U$ , respectively. Thus, for every subsequent iteration involving the same matrix, we need not perform gaussian elimination, since  $L$  and  $U$  allow us to immediately begin performing the second step of solving; back-substitution, which only has complexity  $O(n^2)$ .

However, when performing naive Gaussian elimination to form the matrices  $L$  and  $U$ , we are at risk of swamping, or the existence of a zero-pivot. With the help of a permutation matrix  $P$ , we can now swap rows and columns, to mitigate the propagation of errors due to multiplying rows by large values, and avoid zero-pivots. The permutation matrix  $P$  keeps track of the swapping of rows and columns, so that the linear system itself remains unperturbed (however it is now written  $PAx = Pb$ ).

A great example would be the system  $Ax = b$  with  $A = \begin{pmatrix} 1 & 2 & 4 \\ 3 & 8 & 14 \\ 2 & 6 & 13 \end{pmatrix}$ . To begin, we swap rows 1 and 2, since we want our multiplication to be by the smallest values possible during Gaussian elimination. When doing this, we update our permutation matrix:  $P = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$ , and we can subsequently perform gaussian elimination to yield the matrix  $\begin{pmatrix} 3 & 8 & 14 \\ (\frac{1}{3}) & \frac{-2}{3} & \frac{-2}{3} \\ (\frac{2}{3}) & \frac{2}{3} & \frac{11}{3} \end{pmatrix}$ , where the brackets around the values in the place of what should be 0 represents the multiplier for the elimination of the row, (important for bookkeeping when doing permutations).

Once again performing Gaussian elimination we obtain the matrix  $\begin{pmatrix} 3 & 8 & 14 \\ (\frac{1}{3}) & \frac{-2}{3} & \frac{-2}{3} \\ (\frac{2}{3}) & (-1) & 3 \end{pmatrix}$ . From this matrix, we can easily find that  $L = \begin{pmatrix} 1 & 0 & 0 \\ \frac{1}{3} & 1 & 0 \\ \frac{2}{3} & -1 & 1 \end{pmatrix}$  and  $U = \begin{pmatrix} 3 & 8 & 14 \\ 0 & \frac{-2}{3} & \frac{-2}{3} \\ 0 & 0 & 3 \end{pmatrix}$ . We can then use these to solve linear systems for  $b_k$ , with the increased efficiency of  $LU$  factorization, and the error mitigation of the pivots enabled by the inclusion of the permutation matrix.

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

Although it is a very effective direct method of giving (theoretically) the exact solution of the linear system,  $PA = LU$  factorization is not always the tool you want to employ for solving said linear system. For example, if the matrix involved is positive definite, and symmetric, we can employ methods such as Cholesky Factorization, which is more efficient. If the matrix is not only positive definite and symmetric, but also sparse, we can employ the Conjugate Gradient Method, which is even more efficient than the aforementioned method, and has even lower memory demand.

## 2.3 Larger applications of $PA = LU$ factorization

The applications of  $PA = LU$  factorization extend beyond just solving linear systems, and can be used to solve very common problems. One such problem is solving for the inverse of a matrix, which is made significantly easier when said matrix is decomposed into upper and lower matrices via  $PA = LU$  factorization. On top of this, we can easily compute the determinant of a matrix given its  $PA = LU$  factorization, when we recall that the determinant of a triangular matrix is just its diagonal entries, and that the determinant of a permutation matrix is just  $(-1)^n$ , where  $n$  is the number of rows swapped by the permutation.

## 3 Iterative solution of systems of linear equations

Jacobi FPI is a method for solving systems of linear equations of the form  $Ax = b$ . It is a multi-dimensional analogue of the one-dimensional fixed-point iteration we explored in Lab 2. We require that  $A$  is diagonally dominant, which means that the absolute value of the diagonal element of  $A$  is greater than the sum of the absolute values of the other elements in the row. This is a sufficient condition for the convergence of the Jacobi FPI. The Jacobi FPI is given by the following formula:

$$x^{(k+1)} = D^{-1}(L + U)x^{(k)} + D^{-1}b \quad (1)$$

where  $D$  is the diagonal of  $A$ ,  $L$  is the lower triangular part of  $A$ , and  $U$  is the upper triangular part of  $A$ . The Jacobi FPI is a simple and easy-to-implement method for solving systems of linear equations, but it is not the most efficient method. We will explore the convergence of the Jacobi FPI and compare it to the  $PA = LU$  factorization method.

### 3.1 Solving an equation for $n = 100,000$

We solve the following system of linear equations

$$\begin{bmatrix} 3 & -1 & 0 & 0 & 0 & \frac{1}{2} \\ -1 & 3 & -1 & 0 & \frac{1}{2} & 0 \\ 0 & -1 & 3 & -1 & 0 & 0 \\ 0 & 0 & -1 & 3 & 0 & 0 \\ 0 & \frac{1}{2} & 0 & 0 & 3 & -1 \\ \frac{1}{2} & 0 & 0 & 0 & -1 & 3 \end{bmatrix} x = b \quad (2)$$

where  $b = (5/2, 3/2, \dots, 3/2, 1, 1, 3/2, \dots, 5/2)^T$ , where there are  $n - 4$   $3/2$ 's in the middle of the vector. Using the code in Listing 1, we solve the system of linear equations for  $n = 100,000$ , with the following:

```
>> n = 100000;
>> [A, b] = sparsesetup(n);
>> x = jacobi(A, b, 50);
```

which converges to  $(1, \dots, 1)^T$  after just 50 iterations.

```
1 % Inputs: full or sparse matrix a, r.h.s. b,
2 % number of Jacobi iterations, k
3 % Output: solution x
4 function x = jacobi(a,b,k)
5     n=length(b); % find n
```

```

6      d=diag(a); % extract diagonal of a
7      r=a-diag(d); % r is the remainder
8      x=zeros(n,1); % initialize vector x
9      for j=1:k % loop for Jacobi iteration
10         x = (b-r*x)./d;
11     end % End of Jacobi iteration loop

```

Listing 1: The code for the Jacobi FPI

### 3.2 Comparison of $PA = LU$ and Jacobi Iteration

Theoretically, Jacobi is guaranteed to converge for all  $n$  if  $A$  is diagonally dominant, which  $A$  is in this case. If we try  $PA = LU$  decomposition for this problem, using

```

1 % Solve a system of equations with PA = LU
2 function x = solve_lu(A, b)
3     [L, U, P] = lu(A);
4     x = U \ (L \ (P * b));
5 end

```

Listing 2: The code for the LU decomposition

the function takes much longer due to its  $O(n^3)$  time complexity. Even for  $n = 10000$ , the function takes almost a minute on an Apple M1 Pro chip. Usually for  $PA = LU$ , if the matrix is rank-deficient (multiple rows/columns that are linearly dependent), the  $LU$  decomposition may not be unique and thus may fail to find the correct solution. However in this case,  $A$  is not rank-deficient, so the  $LU$  decomposition should work. But, the Jacobi method is much faster and more reliable for this problem.

### 3.3 Why is solving such large systems important in applications?

Solving large systems of linear equations is important in applications because it is a fundamental problem in many areas of science and engineering. For example, in statistics, a linear model may depend on millions or even billions of parameters which translates to a system of equations of the same size. As a concrete example, a dimensionality-reduction technique such as principal component analysis requires solving a system with as many equations as there are data points and/or desired dimensions. Or, for example, modelling fluid dynamics similarly requires very large systems of equations due to the complexity of the problem. Data is naturally highly multi-dimensional, which makes solving large systems important in practice.

## 4 Implement Newton's method for multiple variables

### 4.1 Implementation of Newton's method for systems using vectorization

```

1 function [x,flag] = vectornewton(f,Df,x0,tol,maxiter)
2 flag = -1;
3 x = x0;
4 for i = 1:maxiter
5     x_old = x;
6     x = x - Df(x)\f(x);
7     if norm(x - x_old) <= tol
8         flag = i;
9         break

```

```

10     end
11 end
12 end

```

Listing 3: The code for the Newton’s method

## 4.2 Testing

We test Newton’s method on the following system of equations in Listing 4: which yields the approximate solution of  $u = 1.0960, v = -1.1592, w = -0.2611$  in 7 iterations with a tolerance of  $10^{-12}$ .

$$\begin{aligned}
 2u^2 - 4u + v^2 + 3w^2 + 6w + 2 &= 0 \\
 u^2 + v^2 - 2v + 2w^2 - 5 &= 0 \\
 3u^2 - 12u + v^2 + 3w^2 + 8 &= 0
 \end{aligned} \tag{3}$$

In the same code, we solve

$$\begin{aligned}
 y^2 - x^3 &= 0 \\
 x^2 + y^2 - 1 &= 0
 \end{aligned} \tag{4}$$

which yields  $x = 0.8260, y = 0.5636$  in 7 iterations with a tolerance of  $10^{-12}$ .

## 4.3 Adding Visualization to Newton’s Method

Plotting the convergence of Newton’s method allows one to understand the rate of convergence and the behavior of the system of equations, since we often don’t know how the system may converge or behave beforehand. For example, We can plot the convergence of Newton’s method for the system of equations in Listing 4 by using the code in Listing 5, which yields the plot in Figure 1. It shows that although we don’t reach the desired tolerance until iteration 7, the rate of convergence is very fast, and the solution is very close to the true solution after just 4 iterations.

# 5 Summary

## 5.1 Results

We explored 3 different methods of solving systems of equations across 2 different categories: direct substitution methods, and iterative methods. We found that the Jacobi FPI is a simple and easy-to-implement method for solving systems of linear equations, but it is not the most efficient method. We also found that the PA = LU factorization method is not suited for solving systems of linear equations when the matrix is large, and that the Jacobi FPI is much faster and more reliable for this problem. We also implemented Newton’s method for systems of equations and found that it is a very fast and reliable method for solving systems of equations, and that plotting the convergence of Newton’s method allows one to understand the rate of convergence and the behavior of the system of equations.

## 5.2 Team Work Problems and Ideas

We worked well together and were able to complete the lab in a timely manner. We were able to divide the work evenly and work on the lab together. We were able to communicate effectively and work together to solve the problems in the lab. A problem that was encountered was in the testing

of the PA = LU method, where the function took much longer than expected to run. We were able to solve this problem by starting with much smaller input sizes (e.g.  $n = 100$ ) and growing  $n$  much slower to see how the function scaled.

### 5.3 Future Explorations

In the future, we would like to explore other methods for solving systems of equations, such as the Gauss-Seidel method, the SOR method, and the conjugate gradient method. We would also like to explore the convergence of these methods and compare them to the methods we explored in this lab. We would also like to further explore how ill-conditioned matrices may behave with the discussed methods, and how to transform the matrices to be better conditioned.

## References

- [1] Christensen, R. H. (2012). *The History of Linear Algebra*. Retrieved from: <https://www.math.utah.edu/~gustafso/s2012/2270/web-projects/christensen-HistoryLinearAlgebra.pdf>
- [2] Fearnley-Sander, D. (1979). *Hermann Grassmann and the Creation of Linear Algebra*. *American Mathematical Monthly*, 86, 809–817.
- [3] Hart, R. (2010). *The Chinese Roots of Linear Algebra*. JHU Press. ISBN 9780801899584.
- [4] Sauer, T. (2012). *Numerical Analysis*. Pearson. ISBN 978-0-321-58746-2.

## 6 Appendices

### 6.1 Code

```

1 clear
2 clc
3
4 f1 = @(x) [
5     2*x(1)^2 - 4*x(1) + x(2)^2 + 3*x(3)^2 + 6*x(3) + 2;
6     x(1)^2 + x(2)^2 - 2*x(2) + 2*x(3)^2 - 5;
7     3*x(1)^2 - 12*x(1) + x(2)^2 + 3*x(3)^2 + 8
8 ];
9
10 Df1 = @(x) [
11     4*x(1) - 4, 2*x(2), 6*x(3) + 6;
12     2*x(1), 2*x(2) - 2, 4*x(3);
13     6*x(1) - 12, 2*x(2), 6*x(3)
14 ];
15
16 [x1_soln, iters1] = vectornewton(f1, Df1, [0; 0; 0], 10^(-12), 100)
17
18
19 f2 = @(x) [
20     x(2) - x(1)^3 ;
21     x(1)^2 + x(2)^2 - 1
22 ];
23 Df2 = @(x) [
24     -3*x(1)^2 , 1 ;
25     2*x(1) , 2*x(2)

```

```

26 ];
27 [x2_soln , iters1] = vectornewton(f2 , Df2 , [ 1 ; 2 ], 10^(-12) ,10)

```

Listing 4: The code for the Newton's method test

```

1 function [x, flag] = vectornewtonvisualized(f, Df, x0, tol, maxiter)
2     flag = -1;
3     x = x0;
4
5     % Initialize vectors to store iteration information
6     iteration_numbers = zeros(maxiter, 1);
7     norms_of_steps = zeros(maxiter, 1);
8
9     for i = 1:maxiter
10         x_old = x;
11
12         % Compute the Newton step
13         step = Df(x)\f(x);
14
15         % Update the solution
16         x = x - step;
17
18         % Store iteration information
19         iteration_numbers(i) = i;
20         norms_of_steps(i) = norm(step);
21
22         % Check for convergence based on the norm of the step
23         if norms_of_steps(i) <= tol * (1 + norm(x))
24             flag = i;
25             break
26         end
27     end
28
29     % Plot the convergence
30     figure;
31     plot(iteration_numbers(1:flag), norms_of_steps(1:flag), '-o');
32     title('Convergence of Newton''s Method');
33     xlabel('Iteration');
34     ylabel('Norm of Newton Step');
35     grid on;
36 end

```

Listing 5: The code for the Newton's method with visualization

## 6.2 Plots

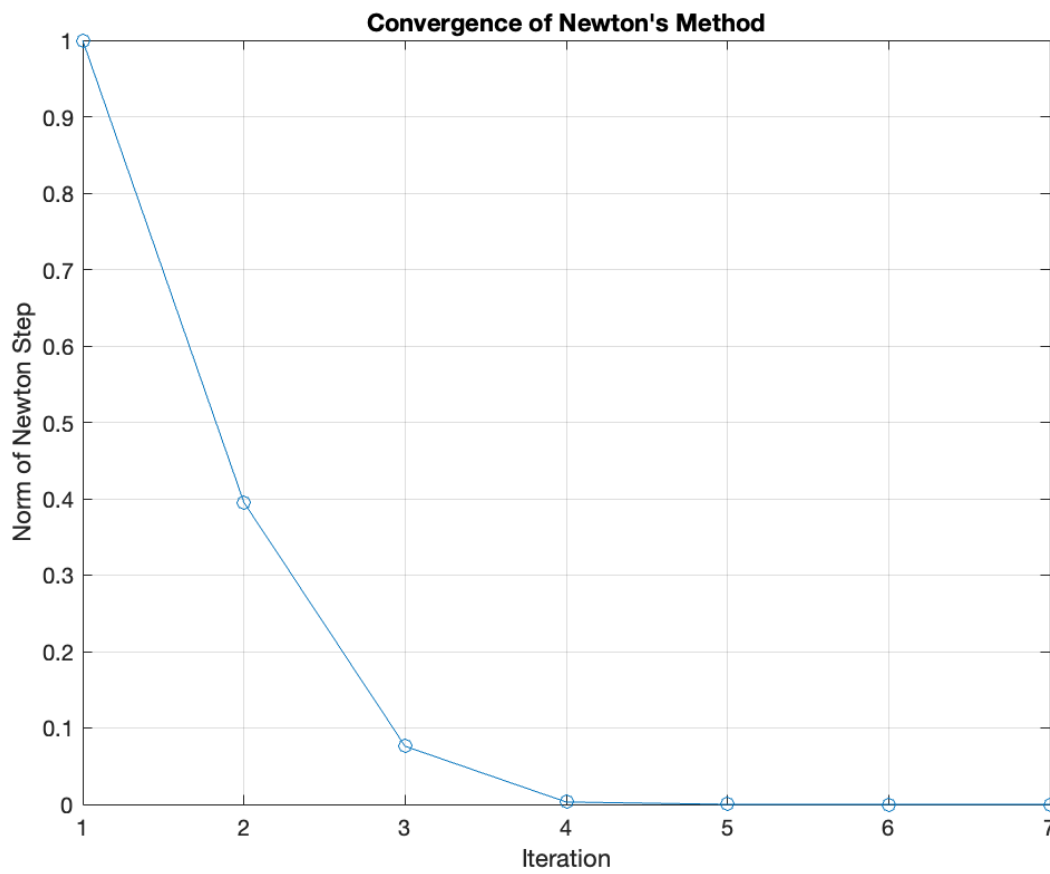


Figure 1: The convergence of Newton's method for the second system of equations in Listing 4