Abstract

The project considers an n by n symmetric square matrix A and seeks an eigenvalue $\lambda \in R$ and eigenvector $v \in R^n$ pair which satisfy Av = λv. This is only n nonlinear equations for n+1 unknown v where λ and all n components in v. To make eigenvector unique, we will add another equation, $1 = \frac{1}{2}||v||_2^2$ into the system A. The given equation helps to construct the Jacobian matrix during the computation. Also in order to convert these equations into root finding problem we will define a new column vector X which has an n+1 dimensional vector with all v components and last entry is λ. It rewrites the equations in the form f(x) = 0 by moving all terms.

The project investigates comparing root-finding numerical methods we discussed in class to find the eigenvalue pair. The project implements three different methods for finding eigenvalue and eigenvector pair: 1. Newton's method, 2. Broyden method, and 3. "Good" Broyden method and compare average execution time and error reduction per iteration. In general MatLab and Python are common choices for implementing numerical analysis due to their rich linear algebra libraries. However, I chose Java because it is my strongest language, and I found an interesting library that is the Apache Commons Math library for matrix computations. I would love to explore it and allows me to create a real matrix class for its calculation. This experience will be beneficial to unitize other programming languages for my future numerical analysis projects.

Moreover, the class discussed a variety of algorithms for solving Ax = b to find a solution vector. This project also implements three algorithms to find solution vector x: 1. Steepest Descent, 2. Conjugate Gradient, and 3. QR Factorization algorithms and compare average execution time and error reduction per iteration.

The project verifies what I understand in this class and becomes a transition to handle a large-scale matrix. I am passionate about linear algebra application to computer science, such as machine learning and artificial intelligence, especially natural language processing and neural network training. This project helped me to understand each algorithm clearly by implementing them and getting used to large-size matrix calculation and optimization problems.

Methods

The project starts define matrix A for Av = λv and adding another equation $1 = \frac{1}{2}||v||_2^2$ makes the eigenvector almost unique. The equation notates as $1 = \frac{1}{2}[v1, v2 \dots vn]^T \cdot [v1, v2 \dots vn]$. To convert this equation into the form of a root-finding problem, we define a new variable x. It needs to add unknown value λ to v, and v has all n components such that v = [v1, v2…vn, λ]. Then rewrite the equations in the form f(x) = 0 and move all terms to the left. Let me define the full definition F as follow.

$$\begin{bmatrix} a11 & a12 & \dots & a1n \\ a21 & a22 & \dots & a2n \\ \dots & \dots & \dots & \dots \\ an1 & an2 & \dots & ann \end{bmatrix} \cdot [v1, v2 \dots vn]^T = \lambda \cdot [v1, v2 \dots vn]$$

$$1 = \frac{1}{2}[v1, v2 \dots vn]^T \cdot [v1, v2 \dots vn]$$

Let define a new variable x and it has an n+1 dimensional vector whose first n entries are all v components from v1 to vn and its last entry is λ. Merging these 2 forms in 1 block form and moving all the right-hand side term to left hand size. It allows to obtain F(x) as

$$\begin{bmatrix} a11 \cdot v1 + a12 \cdot v2 + & \dots & a1n \cdot vn - \lambda \cdot v1 & \\ a21 \cdot v1 + a22 \cdot v2 + & \dots & a2n \cdot vn - \lambda \cdot v2 & \\ \dots & \dots & \dots & \dots \\ an1 \cdot v1 + an2 \cdot v2 + & \dots & ann \cdot vn - \lambda \cdot vn & \\ -\frac{1}{2} \cdot v1^2 - \frac{1}{2} \cdot v2^2 - & \dots & -\frac{1}{2} \cdot vn^2 + 1 & \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \dots \\ 0 \\ 0 \end{bmatrix}$$

It constructs a root finding form f(x) = 0 in system.

1. Newton's method

Recalling newton's method for root finding problem.

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

To compute next x_{n+1} it requires to find $f'(x_n)$. Since our $f(x_n)$ is a matrix form, it needs to construct Jacobian matrix to obtain a tangent plain. We will obtain a Jacobian matrix as follow.

$$J(x) = \begin{bmatrix} \frac{\partial f1}{\partial v1} & \frac{\partial f2}{\partial v2} & \cdots & \frac{\partial f1}{\partial \lambda} \\ \frac{\partial f1}{\partial v2} & \frac{\partial f2}{\partial v2} & \cdots & \frac{\partial f2}{\partial \lambda} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial f1}{\partial v1} & \frac{\partial f2}{\partial v2} & \cdots & \frac{\partial f1}{\partial \lambda} \end{bmatrix}$$

$$= \begin{bmatrix} a11 - \lambda & a12 & \ldots & -v1 \\ a21 & a12 - \lambda & \ldots & -v2 \\ \ldots & & & \\ an1 & an2 & \ldots ann - \lambda & -vn \\ -v1 & -v2 & \ldots - vn & 0 \end{bmatrix}$$

The Jacobian matrix is not scalar and it can not divide F(Xn). Instead of performing divide operator, it multiplies inversion Jacobian matrix.

$$x_{n+1} = x_n - \mathrm{F}(\mathrm{x_n}) \cdot \mathrm{J}^{-1}(\mathrm{x_n})$$

Based on the class discussion it can derive as follow.

$$x_{n+1} - x_n = -F(x_n) \cdot \mathrm{J}^{-1}(x_n)$$
$$J(x_n) \cdot (x_{n+1} - x_n) = -F(x_n)$$
$$J(x_n) \cdot \Delta = -F(x_n)$$

Recall Ax = b form for solving x. One of the methods is LU factorization to solve x. Fortunately the Apache Commons Math library in Java has this method and solve x. We will solve Δ and it updates x_n for the next iteration as

$$x_{n+1} = \Delta + x_n$$

It keeps computing $J(x_{n+1}) \cdot \Delta = -F(\mathrm{X_{n+1}})$ until Δ is less than epsilon. This becomes the newton's method in system as our first algorithm for the experiment…①

2. Broyden's method

The class discussed how we approximate Jacobian matrix along with a secant method like manner. Recalling mean value theorem.

$$f(c)' = \frac{f(b) - f(a)}{b - a}$$

Substituting $x_n$to apply our context as follow.

$$f'(x_{(n)}) \approx \frac{f(x_{(n)}) - f(x_{(n-1)})}{x_{(n)} - x_{(n-1)}}$$
$$f'(x_{(n)}) \cdot (x_{(n)} - x_{(n-1)}) \approx f(x_{(n)}) - f(x_{(n-1)})$$

In the system scenario, it becomes

$$J(x_{(n)}) \cdot (x_{(n)} - x_{(n-1)}) \approx f(x_{(n)}) - f(x_{(n-1)})$$

We will replace Jacobian matrix with unknow matrix $A_{(n)}$ and we will solve for matrix $A_{(n)}$ with updating a new matrix $A_{(n+1)}$ satisfying the secant equation such that $||A_{(n)} - A_{(n-1)}||_F$ is minimized. We can follow the Broyden's method algorithm to update unknow matrix A each iteration as

$$A_{(n)} = A_{(n-1)} + \frac{(y - A_{(n-1)} \cdot \Delta_{(n)})}{\Delta_{(n)}^T \cdot \Delta_{(n)}} \cdot \Delta_{(n)}^T$$
$$where\ y = F(x_{(n)}) - F(x_{(n-1)})$$
$$\Delta = x_{(n)} - x_{(n-1)}$$

We can set initial matrix $A_{(0)}$ with Identity matrix or Jacobian matrix and compare the execution time and error reduction in the execution phase… ②, ③.

Speaking about Broyden's method, the class explore another variation of Broyden's method which is called "Good" Broyden's method. It updates the Jacobian approximation in a way that satisfies the secant condition compared to the standard Broyden't method. The update formula for the Jacobian A in the "Good" Broyden's method is:

$$A_{(n)}^{-1} = A_{(n-1)}^{-1} + \frac{(\Delta - A_{(n-1)} \cdot y)}{\Delta_{(n)}^T \cdot A_{(n-1)}^{-1} \cdot y} \cdot \Delta_{(n)}^T \cdot A_{(n-1)}^{-1}$$
$$where\ y = F(x_{(n)}) - F(x_{(n-1)})$$
$$\Delta = x_{(n)} - x_{(n-1)}$$

I will also add this algorithm to compare the execution time and error reduction analysis… ④

Algorithms ① through ④ are finding eigenvalue and eigenvector pair. I would like to walk through other algorithms first to solve Ax = b to find a minimum solution of f(x).

3. Steepest Decent

The Steepest decent method is a first-order iterative optimization algorithm for finding a local minimum of a function. It solves Ax = b where A is Symmetrix and Positive Definitive(SPD) matrix. The quadratic function f(x), to be minimized is typically defined as

$$f(x) = \frac{1}{2}x^T Ax - b^T x$$

We want to find minimum of f(x) is a solution to Ax = b. Take steps int the negative gradient direction such that $\bigtriangledown$f = Ax -b. Each iteration minimizes each step with computing step size $\alpha_{(n)}$ and $r_{(n)}$ is the residual at the nth iteration as follows.

$$r_{(n)} = b - Ax_{(n)}$$
$$\alpha_{(n)} = \frac{r_{(n)}^T r_{(n)}}{r_n^T A r_n}$$
$$x_{(n+1)} = x_{(n)} + \alpha_{(n)} r_{(n)}$$
$$r_{(n+1)} = b - Ax_{(n+1)}$$

Through the iteration process residual r is getting smaller and it stops when the r is less than epsilon…⑤

4. Conjugate gradient

There is another advanced iterative algorithm for a solving system of linear equations, particularly those where the matrix is an SPD matrix. We can add an extra step above the Steepest Decent algorithm, which is updated direction $\beta_{(n)}$. Let me rewrite again the above the algorithm with $\beta_{(n)}$.

$$d_{(0)} = r_{(n)} = b - Ax_{(n)}$$
$$\alpha_{(n)} = \frac{r_{(n)}^T r_{(n)}}{d_n^T A r_n}$$
$$x_{(n+1)} = x_{(n)} + \alpha_{(n)} d_{(n)}$$
$$r_{(n+1)} = b - Ax_{(n+1)}$$

If r is small, stop.

$$\beta_{(n+1)} = \frac{r_{(n+1)}^T r_{(n+1)}}{r_{(n)}^T r_{(n)}}$$
$$d_{(n+1)} = r_{(n+1)} + \beta_{(n+1)}\, d(n)$$

The conjugate gradient method improves steepest decent by avoiding repetitious steps. Each step in the Conjugate Gradient method is taken in a direction that is conjugate to the

previous steps, which is key to its efficiency and effectiveness, especially for large systems. It uses prior step to avoid redundancy. …⑥

5. QR Factorization

The QR factorization method is a technique used to solve linear systems of equations Ax=b by decomposing the matrix A into a product of two matrices, Q and R, where Q is an orthogonal matrix and R is an upper triangular matrix. I would like to briefly go over this algorithm to solve Ax = b.

1. Factorization A into QR
   where Q is orthogonal, and R is an upper triangle.
   Using Gram-Schmidt or Householder method.
2. Compute $Q^T b$
   Ax = b
   (QR)x = b (∵ A = QR)
   $Rx = Q^{-1}b$
   $Rx = Q^T b$
   $\because Q\ is\ orthgonal, Q^{-1} = Q^T$
3. Solve $Rx = Q^T b$
   Solve this system for xx using back substitution, starting from the last row and moving upwards.

This method is particularly useful when A is a square or a rectangular (non-square) matrix. I would like to investigate whether QR factorization is still an efficient method for large-size SPD matrices compared to the Decent Steepest method and Conjugate Gradient method. … ⑦

Procedure

As we discussed in class, there are a variety of methods are available to find eigenvalues and eigenvalue pairs. I decided to compare above algorithms' execution time and error reduction in the experiment 1 and 2. Comparing execution time and error reduction for large size SPD matrix to solve Ax = b in the experiment 3.

1. Experiment 1
    - Newton's method, Broyden's method with Identity Matrix, and Jacobian matrix (①, ②, and ③).
2. Experiment 2
    - Newton's method, Broyden's method with Jacobian Matrix, and "Good" Broyden's method (①, ③, and ④).
3. Experiment 3
    - Steepest Decent, Conjugate Gradient, and QR factorization (⑤, ⑥ and ⑦)

Recently, python and MATLAB have become very popular computer languages for numerical analysis and computation. However, I picked up Java. The primary reason is Java is my strong language for implementing algorithms. Also, I found an interesting math library in Java, which is an Apache Commons Mathematics Library, and I am curious how I can use this library for linear algebra computation. I have spent a decent amount of time implementing these algorithms, which lets me understand each detailed algorithm. It makes me easily switch to another language for my future numerical computation projects.

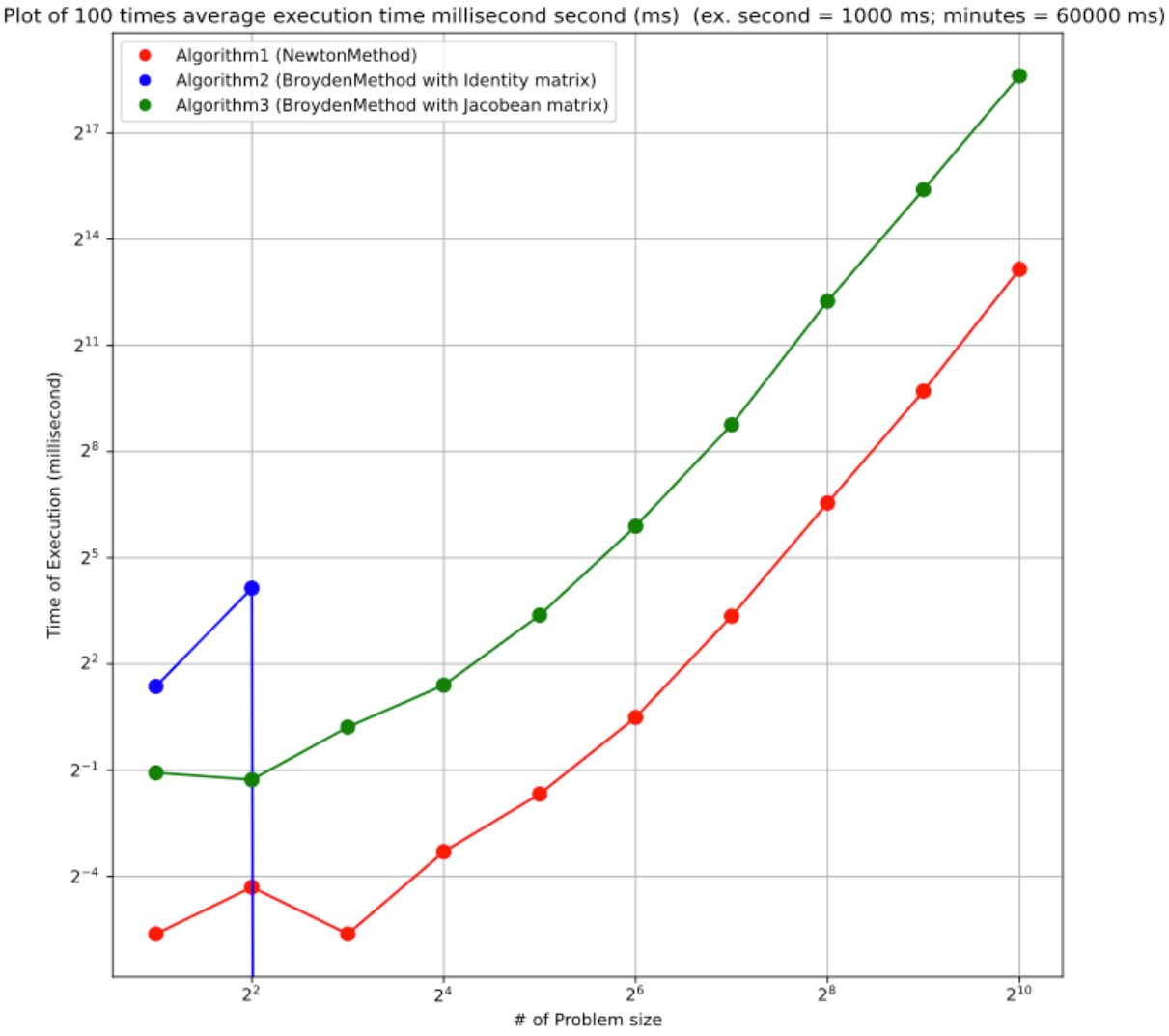
Each experiment starts with a matrix size of 2 until the maximum size of the matrix, which my device can handle. The maximum size is 1024. Each method's number of maximum iteration is 100,000 times. It executes the randomly generated SPD matrix 50 times with the initial guess as a positive column vector such that v = [1,1…1, λ] and negative column vector such that v = [-1, -1… -1, λ] and its total 100 times execution of each matrix size. The program calculates the average execution time and makes a plot.

For error analysis, it calculates the absolute value between the current and previous iteration, which shows how fast each method converges. Each comparison sets two different initial guesses with the experiment 1 and 2 as v0 = [1,1…1,] or v0 = [-1, -1…-1,]. The experiment 3 starts the initial guess with v0 = [1,1…1] or v0 = [-1, -1…-1]. It helps to consider how the initial guess affects several iterations to converge. For this analysis, it generates the random SPD matrix up to the size of 1024. Most of them took a long time to get execution results, and I think a measure of 2^10 is reasonable to consider as a large-size matrix.

Result

1. Experiment 1: Newton's method, Broyden's method with Identity Matrix, and Jacobian matrix (①, ②, and ③).

Average time



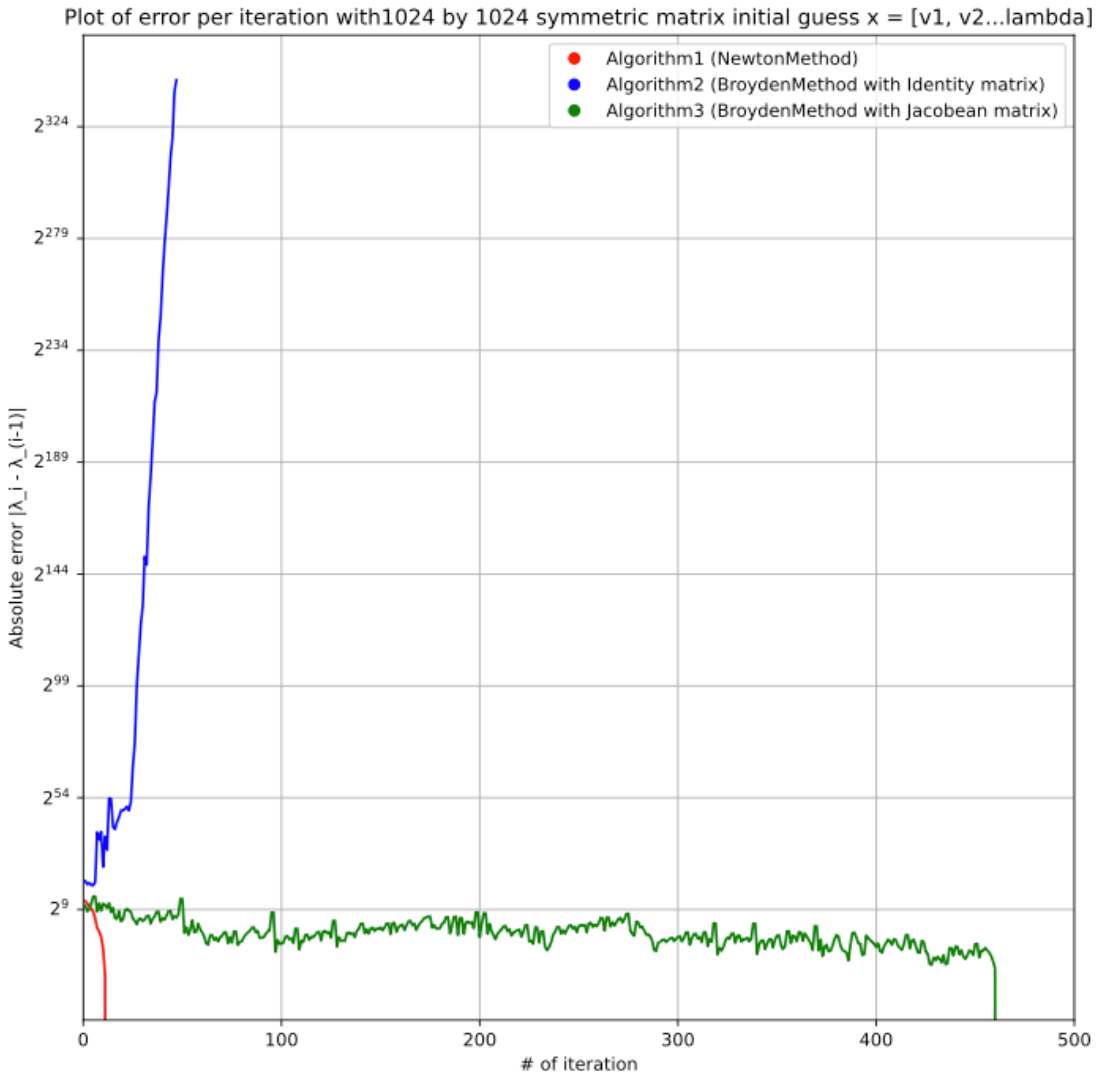
Problem size	Algorithm1(Newton)	Algorithm2(Broyden Jacobi with Identity Matrix)	Algorithm3(Good Broyden with Jacobi Matrix)
2	0.020202020202020204	2.5757575757575757	0.47474747474747475
4	0.050505050505050504	17.636363636363637	0.41414141414141414
8	0.020202020202020204	-0.0	1.1616161616161615
16	0.10101010101010101	-0.0	2.6363636363636362
32	0.31313131313131315	-0.0	10.391752577319588
64	1.404040404040404	-0.0	59.34736842105263
128	10.191919191919192	-0.0	432.7578947368421
256	93.4949494949495	-0.0	4865.0114942528735
512	836.7171717171717	-0.0	43285.02352941177
1024	9094.424242424242	-0.0	401510.37704918033

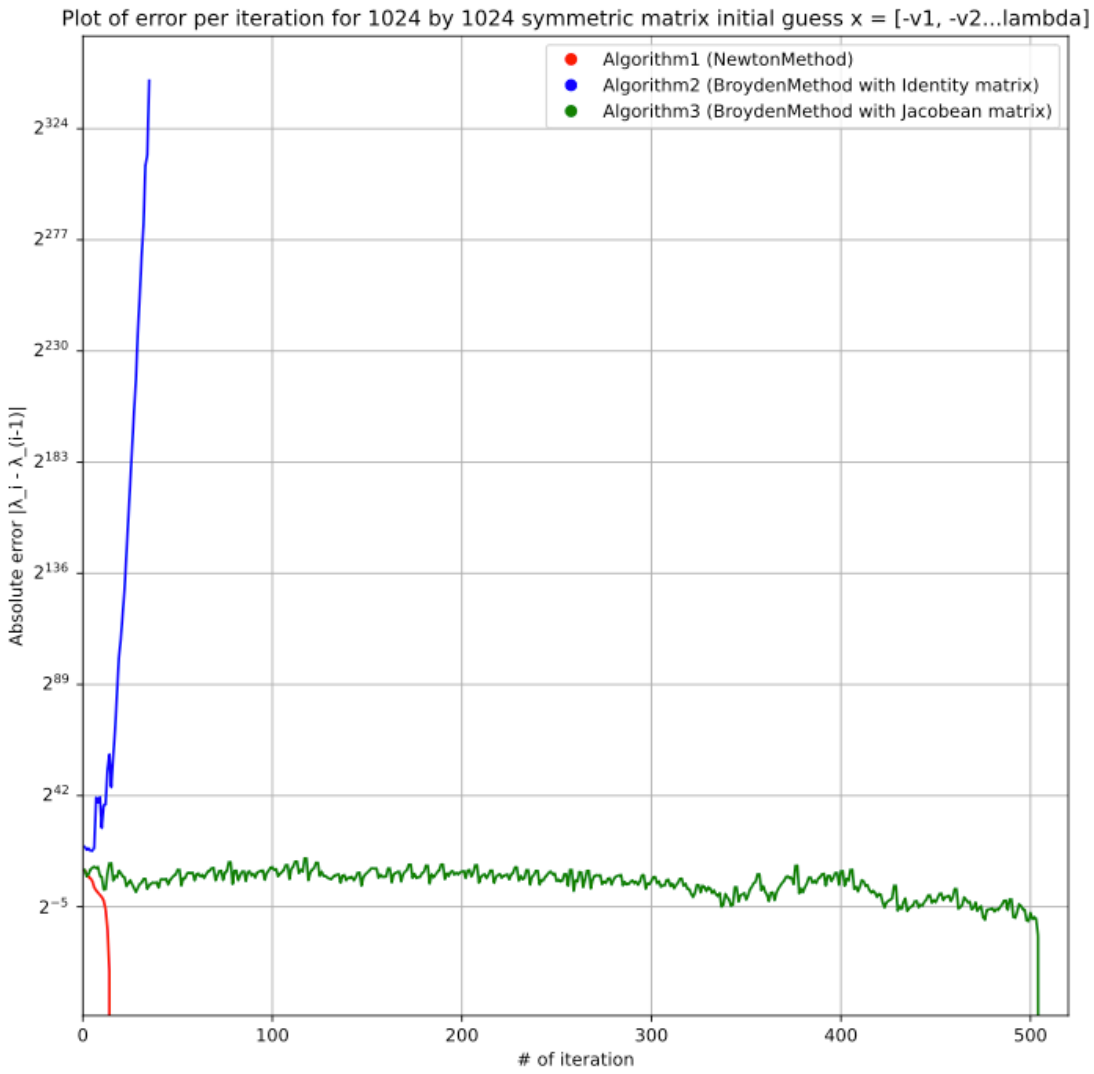
Newton's method has the fastest average execution time among the three algorithms. It has a superlinear convergence rate. Comparing Broyden's method starting with the Identity matrix and Jacobian matrix, the same method with different initial guesses makes different results. The initial guess with the identity matrix did not converge even though it is a small-size matrix. Broyden's method with a Jacobian matrix has better performance in terms of convergence success rate. It illustrates how sensitive some numerical methods can be to the choice of initial conditions. The failure to converge with an identity matrix as the initial guess suggests that for some problems, a closer approximation is necessary for convergence. It is important to note that the specific problem characteristics and the choice of initial conditions can significantly influence the performance of these methods.

The Broyden't method has a linear convergence rate, while Newton's method has a superlinear one. Theatrically, Newton's method converges faster. However, it may be a better execution time than Newton's method in different situations. It depends on how matrix A is complex and the calculation time to obtain the Jacobian matrix. It is important to note that the specific problem characteristics and the choice of initial conditions can significantly influence the performance of these methods.

Additionally, the data table shows notable results after a 64 by 64 matrix size; the average exertion time takes ten times more. It implies that the bigger the matrix size, the more computation time it takes. Even though an efficient computer is available nowadays, choosing the proper method is crucial for faster computation time. Compared to Newton's method and Broyden with Jacobian Matrix in 1024 by 1024, it is 40 times different for execution time. We can quickly expect the gap to increase to a more significant, larger data size.

Error analysis plots

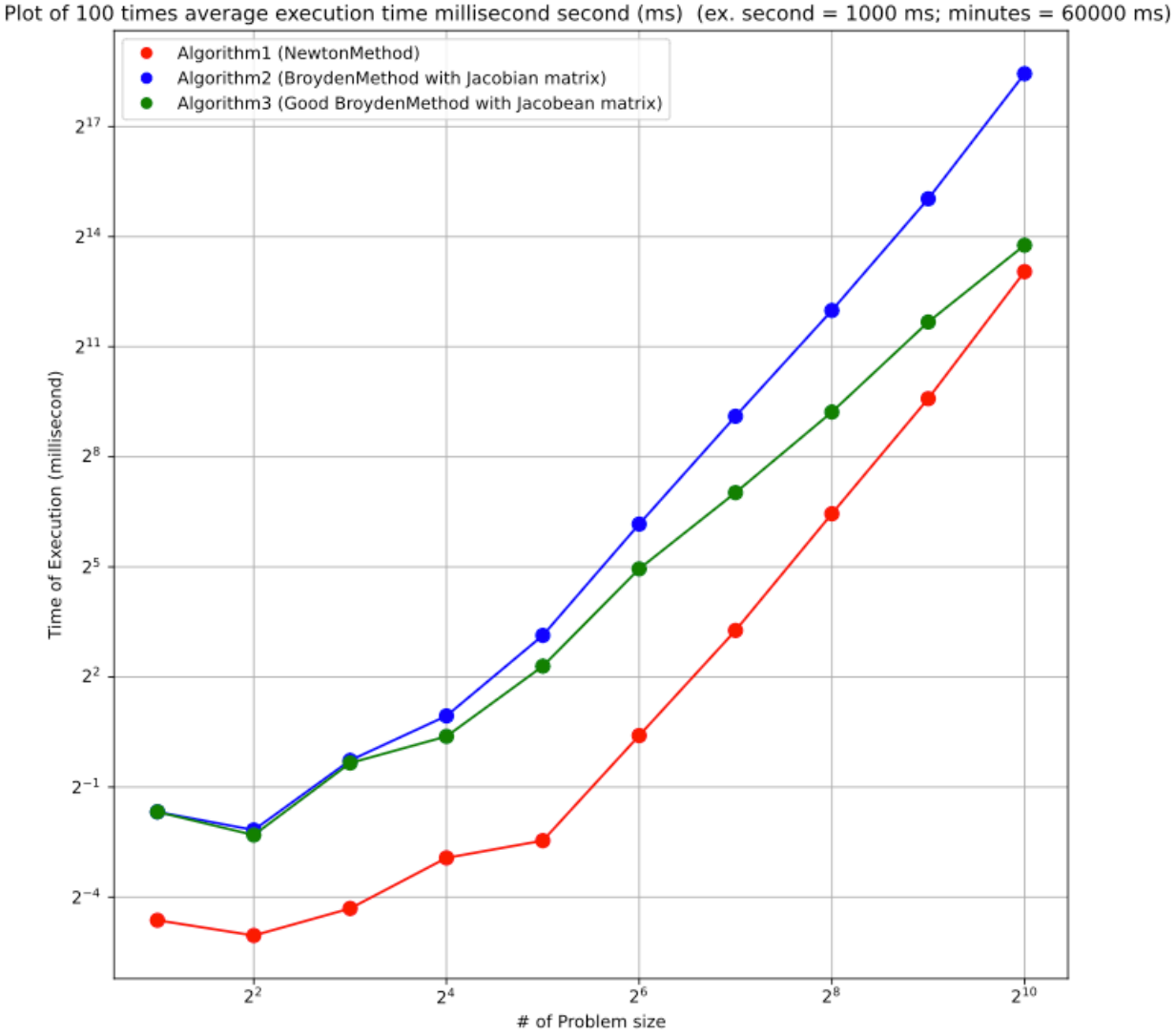

Plot of error per iteration with1024 by 1024 symmetric matrix initial guess x = [v1, v2...lambda]
Algorithm1 (NewtonMethod)
Algorithm2 (BroydenMethod with Identity matrix)
Algorithm3 (BroydenMethod with Jacobean matrix)
Absolute error |λ_i - λ_(i-1)|
2^{324}
2^{279}
2^{234}
2^{189}
2^{144}
2^{99}
2^{54}
2^{9}
0
100
200
300
400
500
# of iteration



Two plots show different initial guess results. One is all the positive components in the initial column vector such that x0 =[1, 1, 1…1, λ], and the other is all the negative components such that x0 = [-1, -1…-1, λ]. As we can expect by the average time result, Broyden's method with the Jacobian matrix needs more iteration until it gets converted. The Newton't method reduces residuals with a smooth, steep shape, while Broyden's method has a wave shape. Also, Brayden's method changes residual almost the same range, but after a particular iteration, the residual drops suddenly. This tendency appears to be positive and negative. It may imply that Broyden's method consistently makes incremental improvements until it achieves a breakthrough in the approximation of the Jacobian or aligns with a more direct descent path.

2. Experiment 2: Newton's method, Broyden's method with Jacobian Matrix, and "Good" Broyden's method (①, ③, and ④).

Average time

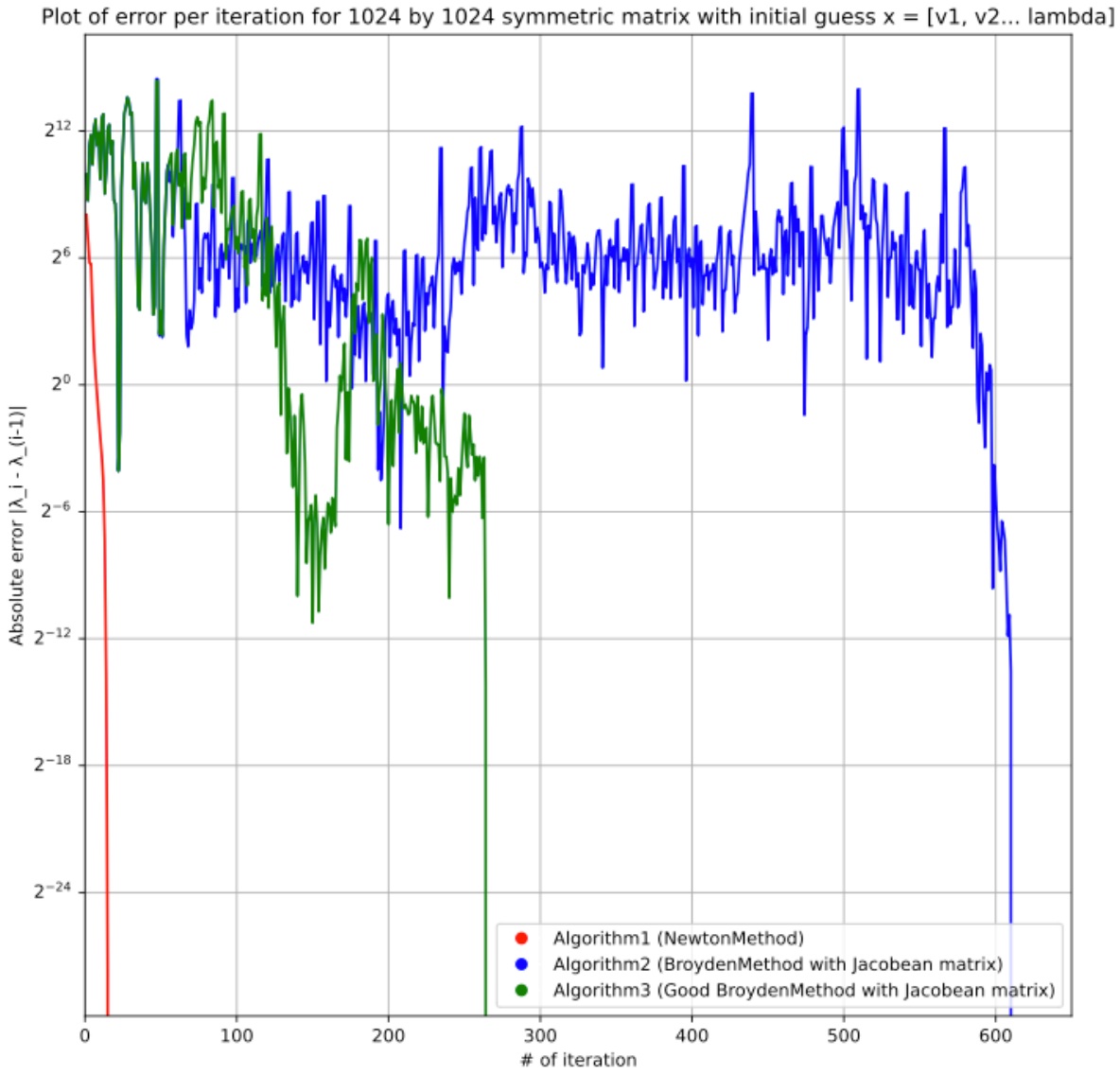

Plot of 100 times average execution time millisecond second (ms) (ex. second = 1000 ms; minutes = 60000 ms)
Algorithm1 (NewtonMethod)
Algorithm2 (BroydenMethod with Jacobian matrix)
Algorithm3 (Good BroydenMethod with Jacobean matrix)
Time of Execution (millisecond)
# of Problem size

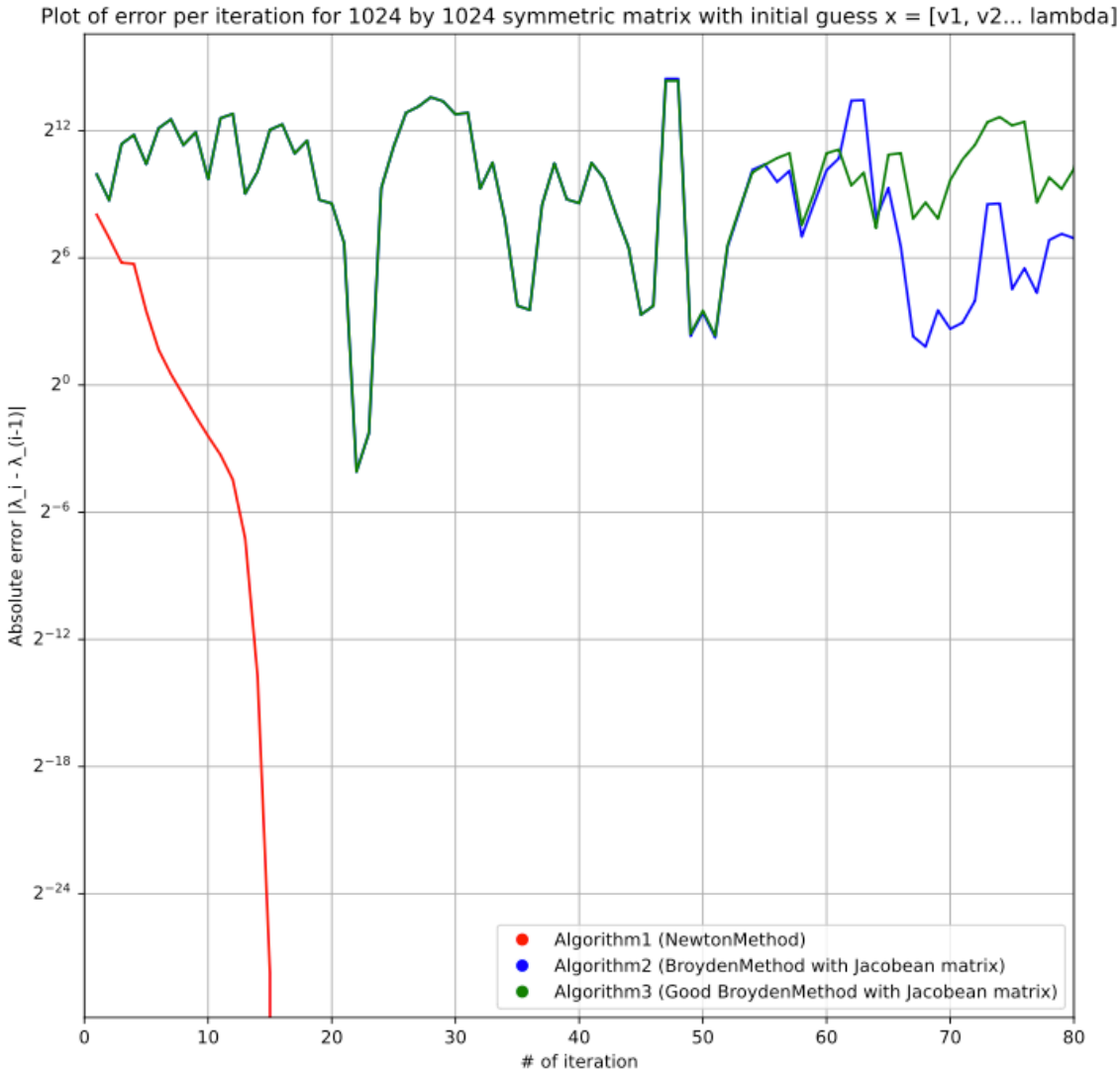
Problem size	Algorithm1(Newton)	Algorithm2(Broyden Jacobi)	Algorithm3(Good Broyden Jacobi)
2	0.04040404040404041	0.31313131313131315	0.31313131313131315
4	0.030303030303030304	0.2222222222222222	0.20202020202020202
8	0.050505050505050504	0.8282828282828283	0.7878787878787878
16	0.13131313131313133	1.9191919191919191	1.303030303030303
32	0.18181818181818182	8.777777777777779	4.909090909090909
64	1.3232323232323233	71.83838383838383	30.838383838383837
128	9.636363636363637	550.989247311828	130.20618556701032
256	87.4040404040404	4065.2597402597403	596.8045977011494
512	768.8383838383838	33508.126760563384	3267.3291139240505
1024	8463.858585858587	356078.0338983051	13950.014084507042

Newton's method has the fastest average execution time among the three algorithms, as we saw in the previous experiment. It has a superlinear convergence rate. Compared to the standard Broyden method and good Broyde's method, these two are similar results up to the size of a 2^3 matrix. However, Good Broyden's method executes better performance after the size of the 2^4 matrix. This could indicate that "Good" Broyden's approach scales better with increased problem size, possibly due to more effective updates that become significant in larger matrices.

Also, the average time difference between Newton's method and Good Broyden's method gets closer as the size of the matrix grows. Good Broyden's method can record faster execution time and the size of the matrix growth. I realized that matrix size is a crucial factor in considering algorithm efficiency. It will depend on the specifics of the problem, its size, and the implementation details of each algorithm.

Error analysis plots



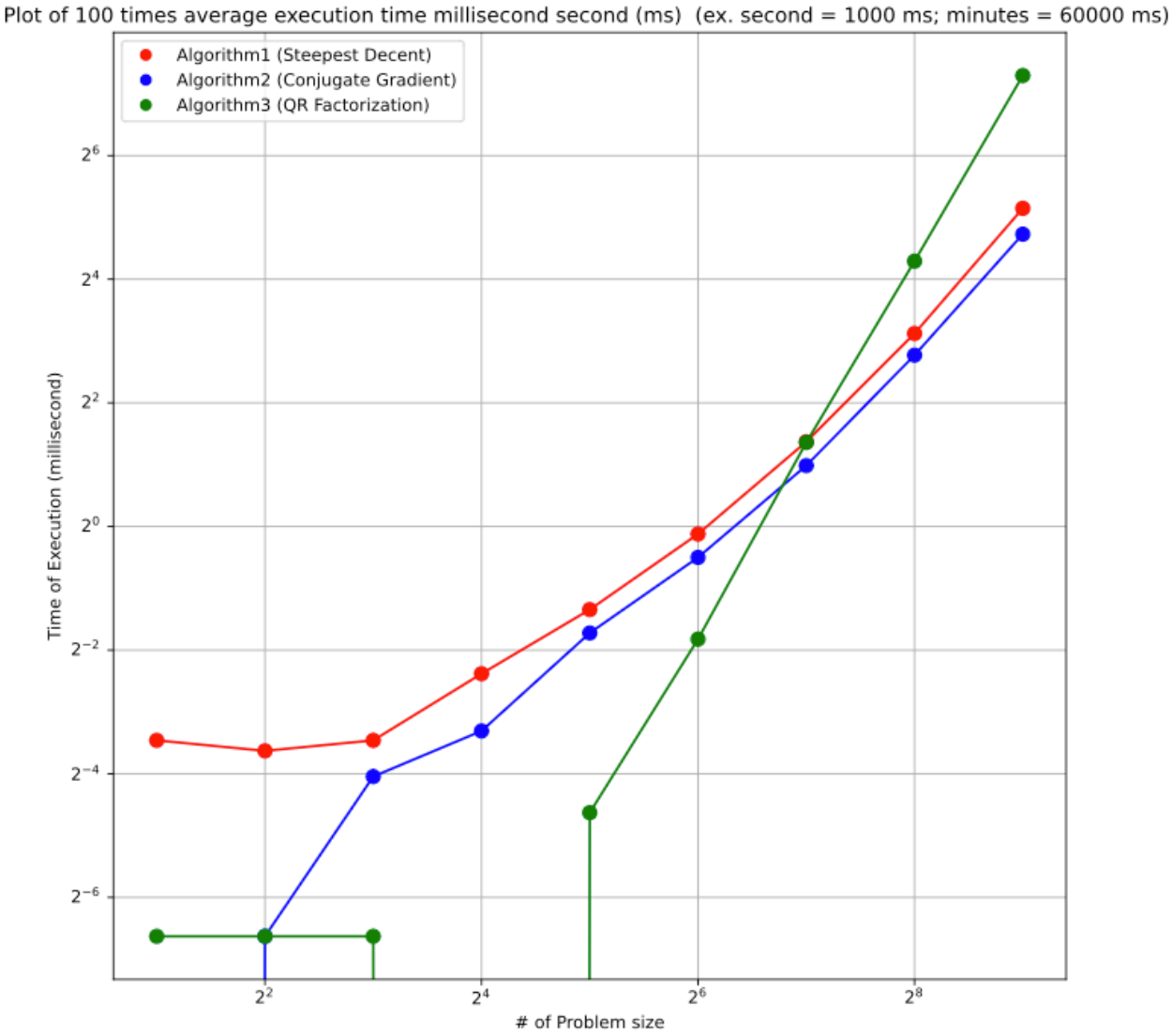


The first plot shows the iteration time until each algorithm converges. As we can see in the plot, “Good” Broyden’s method has fewer iterations than the standard Broyden’s method. These two methods have waveform-changing residuals, which suddenly drop after a particular iteration time, as we observed in the previous experiment.

However, the second plot shows a different and interesting manner compared to experiment 1. the first 80 iterations for the three methods. The standard Broyden’s method and “Good” Broyden’t method’s residual are almost identical around the 60th time iteration. However, the standard Broyden’t process needs a much bigger number of iterations until it gets converged. This might suggest that while both methods progress similarly in the initial phase, the "Good" Broyden's method eventually finds a more efficient route.

3. Experiment 3: Steepest Decent, Conjugate Gradient, and QR factorization (⑤, ⑥ and ⑦)

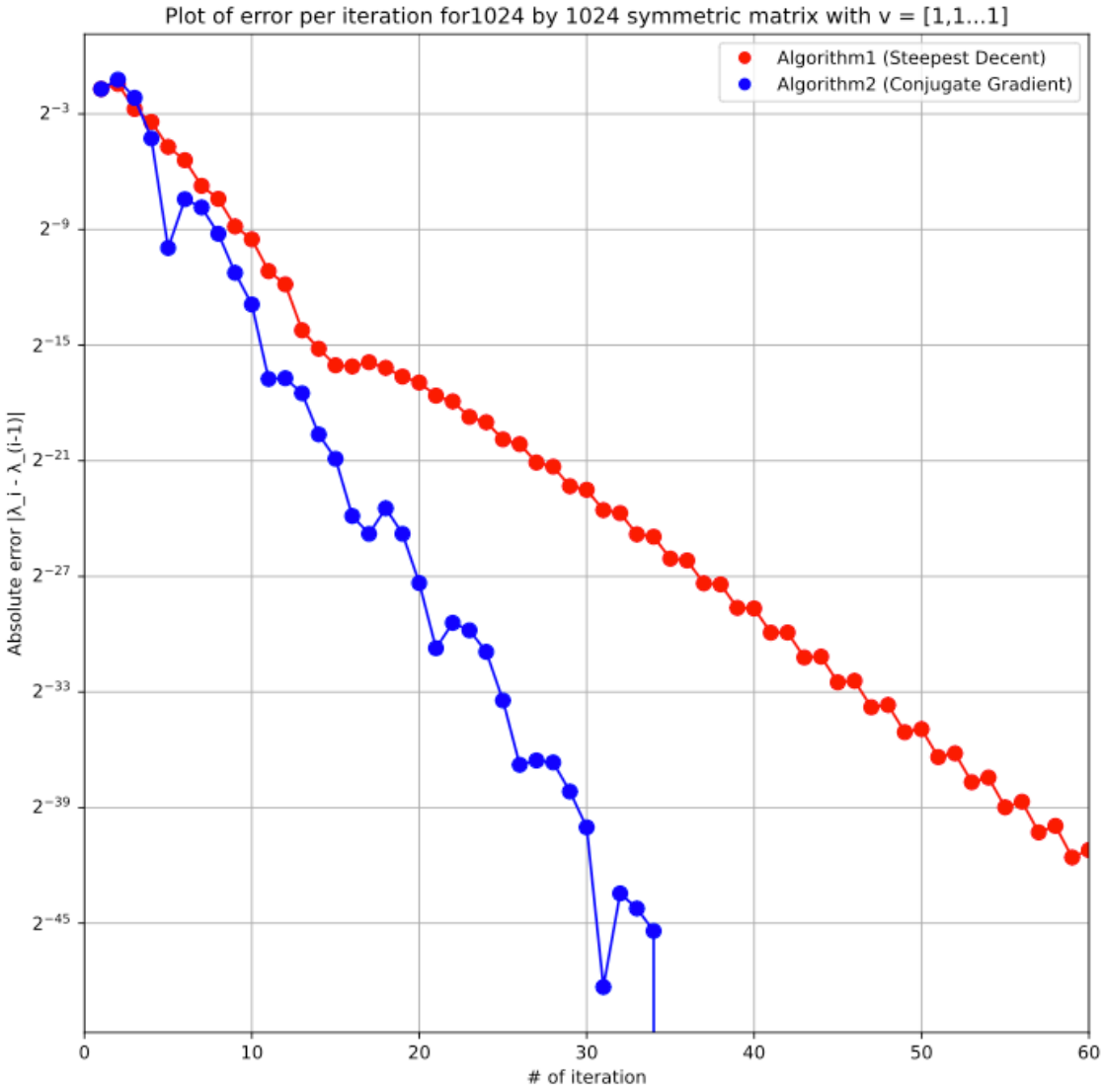
Average time

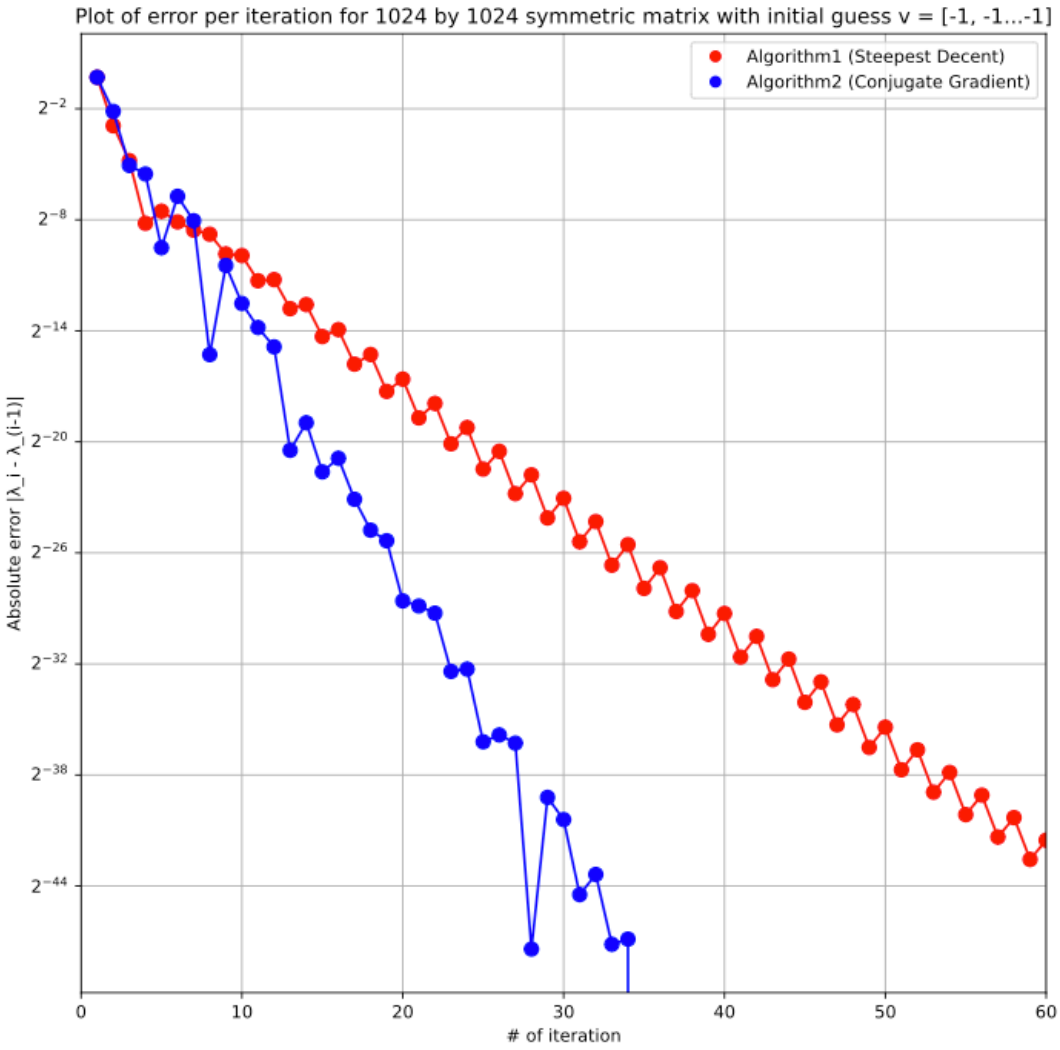


This experiment investigates how QR factorization is an efficient large-size matrix to solve Ax = b to find minimum f'(x). It impressed me that QR factorization performs better until the matrix size is 2^6. However, as the size of the matrix grows, the Steepest Decent and Conjugate Gradient have a faster average execution time. I expected QR factorization to be the quickest performance because it is a direct method, while the other two algorithms are iterative to solve Ax = b.

But, as we can see on the plot, the QR factorization takes more time after the size of the 2^7 matrix. The decompose QR process would take longer once the matrix becomes a specific number. This aligns with the expectation that direct methods like QR factorization can be efficient for smaller matrices. I learned that QR factorization is typically more computationally intensive than iterative methods like Steepest Descent and Conjugate Gradient, especially for large systems. The experiment effectively demonstrates the trade-offs between direct and iterative methods in numerical linear algebra.

Error analysis plot





These two plots compare how the Conjugate Gradient converges faster than the Steepest Gradient with two different initial guesses: all the components in the x0 are 1, or all the values in the x0 are -1. The plots show that the conjugate gradient residual reduces widely at some point. This is a crucial feature of the method, as it optimizes each iteration to make substantial progress towards convergence. It verifies that the Conjugate Gradient method improves the steepest descent by updating the direction and avoiding repetitious steps. It uses the prior step to avoid redundancy.

The plot shows that in the Conjugate Gradient method, the directions are conjugated to each other, and this conjugacy is crucial for the method's efficiency. It ensures that each step is optimally aligned with the contours of the problem space. The Conjugate Gradient method uses the history of all previous search directions to inform the next step. This cumulative knowledge allows it to avoid areas of the problem space it has already explored, thereby reducing redundancy.

Conclusion

After finishing this project, I reflected on how I went through; my experimental trials were minimal. I only tried limited initial guess and SPD matrix. When I was in a classroom studying these algorithms, I thought it would be crucial to construct efficient algorithms to save execution time and computation power. The class discussed a variety of derivations to improve algorithms. I agree with finding efficient algorithms to save execution time and reduce the number of iterations.

However, through this project I realized how the initial starting points affect the convergence result. Looking back at Broyden's method, we set up initial matrix A, the identity or Jacobian matrix. Then, the Jacobian matrix has a higher convergence rate and reduces error. However, starting the Identity matrix did not converge well in my experimental environment. Through this project, I learned how important it is to choose an initial guess wisely before executing algorithms.

Nowadays, people can use a laptop with a much higher CPU and memory than decades ago. It is easy to execute complex algorithms quickly. But I assume people spent a lot of time to iterate numerical computation before. I heard a story from my statistics professor. He waited, repeating the loop 20 times over for two weeks. I imagined people at that time tended to use multiple devices to set up a good initial guess before executing algorithms. Choosing a good initial guess is one of devices and preprocessing preparation, but people who are getting used to computers nowadays don't know these devices and may not appreciate spending time to prepare it. Those including me tend to underestimate importance of this preprocessing preparation. I want to explore how to choose initial guesses wisely and construct efficient algorithms for my following projects.

Moreover, I realized how disorganized my project structure was. Probably, this is a lack of experience, and I can improve the project structure to be more organized. This project experience was undoubtedly meaningful for understanding and practicing iterative method algorithms. It was my first experience utilizing mathematical concepts in computer science. I would love to enhance my mathematical knowledge and design proper procedures to apply computer science fields for handling large-size matrices. I am passionate about utilizing and applying the methods and algorithms I learned in this project to investigate optimization problems in various machine learning and Artificial Intelligence fields.