MAT 5080 Project

Connor Adams

November 23, 2021

Abstract

In this project we discuss tridiagonal matrices and a couple methods for finding the solutions of a system with a tridiagonal matrix. from the start we verified a generic LU factorization for and arbitrary size tridiagonal matrix, T. Then, we discovered the closed form for computing the eigenvalues for a specific type of tridiagonal matrix, H. From here, we discuss a few different method for finding the solutions of $Hx = b$. A few important methods were: Gauss-Elimination w/o pivot, Gauss-Elimination w/ partial pivot, and LU factorization. Next, we explored the effects of the spectral radius when using the SOR method. Finally, we discovered that the best methods to use are LU factorization and Gauss-Elimination w/o pivot to solve for $Hx = b$.

1 Introduction

We are asked to consider a tridiagonal matrix, T:

$$T = \begin{bmatrix} b_1 & a_1 & 0 & 0 \\ c_1 & b_2 & a_2 & 0 \\ 0 & c_2 & b_3 & a_3 \\ 0 & 0 & c_3 & b_4 \end{bmatrix}$$

and the LU factorization of T:

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ \frac{c_1}{d_1} & 1 & 0 & 0 \\ 0 & \frac{c_2}{d_2} & 1 & 0 \\ 0 & 0 & \frac{c_3}{d_3} & 1 \end{bmatrix} \quad \text{and} \quad U = \begin{bmatrix} d_1 & a_1 & 0 & 0 \\ 0 & d_2 & a_2 & 0 \\ 0 & 0 & d_3 & a_3 \\ 0 & 0 & 0 & d_4 \end{bmatrix}$$

More specifically, we are asked to describe the above matrices as arbitrary sizes, nxn. Then, create an H matrix like below:

$$H = \underbrace{\begin{bmatrix} 2 & -1 & 0 & \dots & 0 & 0 \\ -1 & 2 & -1 & 0 & \dots & 0 \\ 0 & -1 & 2 & -1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & -1 & 2 & -1 \\ 0 & 0 & \dots & 0 & -1 & 2 \end{bmatrix}}_{nxn}$$

from here we will explore the closed form for computing the eigenvalues and how the spectral radius changes based on the size of matrix H. After that, we are given a b-vector:

$$b = \underbrace{\begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix}}_{nx1}$$

Finally, we will examine the accuracy and efficiency of the following methods for solving $Hx = b$: Gauss-Elimination w/o pivot, Gauss-Elimination w/ partial pivot, LU factorization, Jacobi, Gauss-Seidel, and SOR with $\omega = .5, 1, 1.5$.

2 Discussion

First, Let's begin by setting up our tridiagonal matrix T:

$$T = \begin{bmatrix} b_1 & a_1 & 0 & 0 \\ c_1 & b_2 & a_2 & 0 \\ 0 & c_2 & b_3 & a_3 \\ 0 & 0 & c_3 & b_4 \end{bmatrix}$$

In this project we are given that the LU factorization of T are:

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ \frac{c_1}{d_1} & 1 & 0 & 0 \\ 0 & \frac{c_2}{d_2} & 1 & 0 \\ 0 & 0 & \frac{c_3}{d_3} & 1 \end{bmatrix} \text{ and } U = \begin{bmatrix} d_1 & a_1 & 0 & 0 \\ 0 & d_2 & a_2 & 0 \\ 0 & 0 & d_3 & a_3 \\ 0 & 0 & 0 & d_4 \end{bmatrix}, \text{ with } d_1 = b_1 \text{ and } d_{i+1} = b_{i+1} - \frac{a_i c_i}{d_i}$$

2.1 Verifying the LU factorization

We can verify that L and U are indeed the LU factorization of T by multiplying L and U together which should give us back T. I used the symbolic feature in MATLAB, which gives us:

$$L * U = \begin{bmatrix} 1 & 0 & 0 & 0 \\ \frac{c_1}{d_1} & 1 & 0 & 0 \\ 0 & \frac{c_2}{d_2} & 1 & 0 \\ 0 & 0 & \frac{c_3}{d_3} & 1 \end{bmatrix} \begin{bmatrix} d_1 & a_1 & 0 & 0 \\ 0 & d_2 & a_2 & 0 \\ 0 & 0 & d_3 & a_3 \\ 0 & 0 & 0 & d_4 \end{bmatrix} = \begin{bmatrix} d_1 & a_1 & 0 & 0 \\ c_1 & d_2 + \frac{a_1 c_1}{d_1} & a_2 & 0 \\ 0 & c_2 & d_3 + \frac{a_2 c_2}{d_2} & a_3 \\ 0 & 0 & c_3 & d_4 + \frac{a_3 c_3}{d_3} \end{bmatrix}$$

If we recall above, $d_1 = b_1$ and $d_{i+1} = b_{i+1} - \frac{a_i c_i}{d_i}$ which implies $b_{i+1} = d_{i+1} + \frac{a_i c_i}{d_i}$. Now, we can replace the diagonal with the b-vector $\rightarrow \begin{bmatrix} b_1 & b_2 & b_3 & b_4 \end{bmatrix}$ which magically gives the beautiful tridiagonal matrix, T:

$$\begin{bmatrix} d_1 & a_1 & 0 & 0 \\ c_1 & d_2 + \frac{a_1 c_1}{d_1} & a_2 & 0 \\ 0 & c_2 & d_3 + \frac{a_2 c_2}{d_2} & a_3 \\ 0 & 0 & c_3 & d_4 + \frac{a_3 c_3}{d_3} \end{bmatrix} = \begin{bmatrix} b_1 & a_1 & 0 & 0 \\ c_1 & b_2 & a_2 & 0 \\ 0 & c_2 & b_3 & a_3 \\ 0 & 0 & c_3 & b_4 \end{bmatrix} = T$$

Just for fun, lets show that this is the Lu factorization by actually computing the LU factorization using MATLAB. This programming language has a built in LU factorization and since we have our T-matrix in symbolic form we can use:

```
[L,U] = lu(T)
```

which then gives us:

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ \frac{c_1}{b_1} & 1 & 0 & 0 \\ 0 & \frac{c_2}{b_2 - \frac{a_1 c_1}{b_1}} & 1 & 0 \\ 0 & 0 & \frac{c_3}{b_3 - \frac{a_2 c_2}{b_2 - \frac{a_1 c_1}{b_1}}} & 1 \end{bmatrix} \quad \text{and} \quad U = \begin{bmatrix} b_1 & a_1 & 0 & 0 \\ 0 & b_2 - \frac{a_1 c_1}{b_1} & a_2 & 0 \\ 0 & 0 & b_3 - \frac{a_2 c_2}{b_2 - \frac{a_1 c_1}{b_1}} & a_3 \\ 0 & 0 & 0 & b_4 - \frac{a_3 c_3}{b_3 - \frac{a_2 c_2}{b_2 - \frac{a_1 c_1}{b_1}}} \end{bmatrix}$$

This looks insane, but when we apply the variables $d_1 = b_1$ and $d_{i+1} = b_{i+1} - \frac{a_i c_i}{d_i}$ we get:

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ \frac{c_1}{d_1} & 1 & 0 & 0 \\ 0 & \frac{c_2}{d_2} & 1 & 0 \\ 0 & 0 & \frac{c_3}{d_3} & 1 \end{bmatrix} \quad \text{and} \quad U = \begin{bmatrix} d_1 & a_1 & 0 & 0 \\ 0 & d_2 & a_2 & 0 \\ 0 & 0 & d_3 & a_3 \\ 0 & 0 & 0 & d_4 \end{bmatrix}$$

2.2 Exploring an arbitrary size tridiagonal matrix

2.2.1 Generalized form of T

Now, Let us look at the T-matrix with an arbitrary size nxn:

$$T = \underset{nxn}{\begin{bmatrix} b_1 & a_1 & 0 & \dots & 0 & 0 \\ c_1 & b_2 & a_2 & 0 & \dots & 0 \\ 0 & c_2 & b_3 & a_3 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & c_{n-2} & b_{n-1} & a_{n-1} \\ 0 & 0 & \dots & 0 & c_{n-1} & b_n \end{bmatrix}}$$

Then, from an observation of our MATLAB LU factroization version we can produce an arbitrary size L and U:

$$L = \underset{nxn}{\begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ \frac{c_1}{b_1} & 1 & 0 & \ddots & 0 \\ 0 & \frac{c_2}{b_2 - \frac{a_1 c_1}{b_1}} & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & \frac{c_{n-1}}{b_{n-1} - \frac{a_{n-2}c_{n-2}}{\substack{\vdots \\ b_2 - \frac{a_1 c_1}{b_1}}}} & 1 \end{bmatrix}} \quad \text{and} \quad U = \underset{nxn}{\begin{bmatrix} b_1 & a_1 & 0 & \dots & 0 \\ 0 & b_2 - \frac{a_1 c_1}{b_1} & a_2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \ddots & \dots & \ddots & a_{n-1} \\ 0 & 0 & \dots & 0 & b_n - \frac{a_{n-1}c_{n-1}}{b_{n-1} - \frac{a_{n-2}c_{n-2}}{\substack{\vdots \\ b_2 - \frac{a_1 c_1}{b_1}}}} \end{bmatrix}}$$

and when we apply $d_1 = b_1$ and $d_{i+1} = b_{i+1} - \frac{a_i c_i}{d_i}$, for $i = 2 : n-1$, we get:

$$L = \begin{bmatrix} 1 & 0 & \dots & 0 & 0 \\ \frac{c_1}{d_1} & 1 & 0 & \dots & 0 \\ 0 & \frac{c_2}{d_2} & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & \frac{c_{n-1}}{d_{n-1}} & 1 \end{bmatrix} \quad \text{and} \quad U = \begin{bmatrix} d_1 & a_1 & 0 & \dots & 0 \\ 0 & d_2 & a_2 & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & d_{n-1} & a_{n-1} \\ 0 & \dots & 0 & 0 & d_n \end{bmatrix}$$

2.2.2 The birth of the H-matrix

Now, L and U seem less scary. The purpose for creating this general matrix form is to allow us to create a new matrix where $a_i = c_i = -1$ and $b_i = 2$ for all $i = 1 : n$. I shall dub the new tridiagonal matrix, H. Below is the our new born matrix, H:

$$H = \underset{nxn}{\begin{bmatrix} 2 & -1 & 0 & \dots & 0 & 0 \\ -1 & 2 & -1 & 0 & \dots & 0 \\ 0 & -1 & 2 & -1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & -1 & 2 & -1 \\ 0 & 0 & \dots & 0 & -1 & 2 \end{bmatrix}}$$

Next, is to generate our new L and U factorization by subbing in the appropriate values for a_i, c_i, b_i and d_i. The tricky one is d_i because when i is large d_i becomes quite long to calculate. Or.......does it?

I want to look at the *sequence* d_i for $i = 1 : 3$ then again for $i = 1 : 4$ and one more time for $i = 1 : 5$ to get a great picture of pattern forming. Let us recall the formula for generating d_i:

$$d_1 = b_1 \quad \text{and} \quad d_{i+1} = b_{i+1} - \frac{a_i c_i}{d_i}, \quad i = 1 : n-1$$

Below will be each vector that lists the each sequence we talked about above.

$$d_{i=1:3} = \begin{bmatrix} 2 \\ 1.5 \\ 1.3\bar{3} \end{bmatrix} \quad d_{i=1:4} = \begin{bmatrix} 2 \\ 1.5 \\ 1.3\bar{3} \\ 1.25 \end{bmatrix} \quad d_{i=1:5} = \begin{bmatrix} 2 \\ 1.5 \\ 1.3\bar{3} \\ 1.25 \\ 1.2 \end{bmatrix}$$

NOTICE A PATTERN HERE? We can write an easier/better form for our sequence d_i as:

$$d_1 = b_1 \quad \text{and} \quad d_{i+1} = b_{i+1} - \frac{i}{i+1}, \quad i = 1 : n-1$$

Since, every entry for $b_i = 2$ for $i = 1 : n$, then we could rewrite this as:

$$d_1 = 2 \quad \text{and} \quad d_{i+1} = 2 - \frac{i}{i+1}, \quad i = 1 : n-1$$

Now, let us revisit our L and U matrix and filling in the appropriate values for our variables.

$$L = \underbrace{\begin{bmatrix} 1 & 0 & \dots & 0 & 0 \\ \frac{-1}{2} & 1 & 0 & \dots & 0 \\ 0 & \frac{-1}{1.5} & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & \frac{-1}{2-\frac{n-2}{n-1}} & 1 \end{bmatrix}}_{nxn} \quad \text{and} \quad U = \underbrace{\begin{bmatrix} 2 & -1 & 0 & \dots & 0 \\ 0 & 1.5 & -1 & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & 2-\frac{n-2}{n-1} & -1 \\ 0 & \dots & 0 & 0 & 2-\frac{n-1}{n} \end{bmatrix}}_{nxn}$$

2.3 The eigenvalues and spectral radius of H

2.3.1 Generating a general form for the eigenvalues

Now, that we have this nice general format of our L, U, and H matrix we can then solve for our eigenvalues. Recall that the way of solving for eigenvalues is by solving for λ in the equation $(A - \lambda I) = \vec{0}$. We will replace the A-matrix with our H-matrix which will give us:

$$\det(H - \lambda I) = 0$$

Which is helpful if we illustrate this equation as:

$$P_n = \det\left(\underbrace{\begin{bmatrix} 2 & -1 & 0 & \dots & 0 & 0 \\ -1 & 2 & -1 & 0 & \dots & 0 \\ 0 & -1 & 2 & -1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & -1 & 2 & -1 \\ 0 & 0 & \dots & 0 & -1 & 2 \end{bmatrix}}_{nxn} - \underbrace{\begin{bmatrix} \lambda & 0 & 0 & \dots & 0 & 0 \\ 0 & \lambda & 0 & 0 & \dots & 0 \\ 0 & 0 & \lambda & 0 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & 0 & \lambda & 0 \\ 0 & 0 & \dots & 0 & 0 & \lambda \end{bmatrix}}_{nxn} \right) = 0$$

Now, lets solve for the λ.

$$P_n = \det\left(\underbrace{\begin{bmatrix} 2-\lambda & -1 & 0 & \dots & 0 & 0 \\ -1 & 2-\lambda & -1 & 0 & \dots & 0 \\ 0 & -1 & 2-\lambda & -1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & -1 & 2-\lambda & -1 \\ 0 & 0 & \dots & 0 & -1 & 2-\lambda \end{bmatrix}}_{nxn} \right) = 0$$

So, when we take the determinate of this matrix we get:

$$P_n = (2-\lambda) * \det\left(\underset{(n-1)x(n-1)}{\begin{bmatrix} 2-\lambda & -1 & 0 & \dots & 0 & 0 \\ -1 & 2-\lambda & -1 & 0 & \dots & 0 \\ 0 & -1 & 2-\lambda & -1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & -1 & 2-\lambda & -1 \\ 0 & 0 & \dots & 0 & -1 & 2-\lambda \end{bmatrix}}\right) - (-1) * \det\left(\underset{(n-1)x(n-1)}{\begin{bmatrix} -1 & -1 & 0 & 0 & \dots & 0 \\ 0 & 2-\lambda & -1 & 0 & \ddots & \vdots \\ 0 & -1 & 2-\lambda & -1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & -1 & 2-\lambda & -1 \\ 0 & 0 & \dots & 0 & -1 & 2-\lambda \end{bmatrix}}\right)$$

$$= (2-\lambda) * \det\left(\underset{(n-1)x(n-1)}{\begin{bmatrix} 2-\lambda & -1 & 0 & \dots & 0 & 0 \\ -1 & 2-\lambda & -1 & 0 & \dots & 0 \\ 0 & -1 & 2-\lambda & -1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & -1 & 2-\lambda & -1 \\ 0 & 0 & \dots & 0 & -1 & 2-\lambda \end{bmatrix}}\right) - (-1) * (-1)\det\left(\underset{(n-2)x(n-2)}{\begin{bmatrix} 2-\lambda & -1 & 0 & \dots & 0 & 0 \\ -1 & 2-\lambda & -1 & 0 & \dots & 0 \\ 0 & -1 & 2-\lambda & -1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & -1 & 2-\lambda & -1 \\ 0 & 0 & \dots & 0 & -1 & 2-\lambda \end{bmatrix}}\right)$$

Then, with a little bit of simplification and using P_{n-1} for the determinant of our $(n-1)x(n-1)$ matrix and P_{n-2} for the determinant of our $(n-2)x(n-2)$ we get:

$$P_n = (2-\lambda)P_{n-1} - P_{n-2}$$

$$P_n - (2-\lambda)P_{n-1} + P_{n-2} = 0$$

with

$$P_0 = 1, \quad \text{and} \quad P_2 = 2-\lambda$$

From here we can find the characteristic equation:

$$r^2 - (2-\lambda)r + 1 = 0$$

When we solve for r we get:

$$r_1 = \frac{(2-\lambda) - \sqrt{\lambda^2 - 4\lambda}}{2}$$

$$r_2 = \frac{(2-\lambda) + \sqrt{\lambda^2 - 4\lambda}}{2}$$

First, let us look at the case where $r_1 \neq r_2$, so the general solution is:

$$P_n(\lambda) = \alpha r_1^n + \beta r_2^n$$

Then, plugging in our initial conditions to get:

$$1 = \alpha + \beta, \quad \text{and} \quad 2-\lambda = \alpha r_1 + \beta r_2$$

So,

$$\beta = 1 - \alpha$$

Then,

$$2-\lambda = \alpha r_1 + (1-\alpha) r_2$$

Therefore,

$$\alpha = \frac{(2-\lambda) - r_2}{r_1 - r_2}, \quad \beta = \frac{r_1 - (2-\lambda)}{r_2 - r_1}$$

Now, we can rewrite the solutions as:

$$P_n(\lambda) = \frac{(2-\lambda) - r_2}{r_1 - r_2} r_1^n + \frac{r_1 - (2-\lambda)}{r_2 - r_1} r_2^n$$

Notice how the sum of r_1 and r_2 is $2-\lambda$. This allows us to rewrite the slutions again as:

$$P_n(\lambda) = \frac{r_1 + r_2 - r_2}{r_1 - r_2} r_1^n + \frac{r_1 - r_1 + r_2}{r_2 - r_1} r_2^n = \frac{r_1}{r_1 - r_2} r_1^n + \frac{r_2}{r_2 - r_1} r_2^n = \frac{r_1^{n+1} - r_2^{n+1}}{r_1 - r_2}$$

Since, $P_n(\lambda) = 0$ then we get:

$$r_1^{n+1} = r_2^{n+1}$$

Now, let's look at:

$$\frac{r_1^{n+1}}{r_2^{n+1}} = \left(\frac{r_1}{r_2}\right)^{n+1} = 1$$

Now using the n^{th} root of unity we know that:

$$\frac{r_1}{r_2} = e^{i\frac{2k\pi}{n+1}} \quad \Longrightarrow \quad r_1 = r_2 e^{i\frac{2k\pi}{n+1}}, \quad \text{for } k = 1, 2, 3, ..., n$$

Remember $k = 0$, because as we said above, $r_1 \neq r_2$. For ease, I will be letting $z = e^{i\frac{2k\pi}{n+1}}$ and subbing the expression for our roots, r_1 and r_2:

$$\frac{(2-\lambda) - \sqrt{\lambda^2 - 4\lambda}}{2} = z\frac{(2-\lambda) + \sqrt{\lambda^2 - 4\lambda}}{2}$$

Then, simplify to get:

$$(2-\lambda) - \sqrt{\lambda^2 - 4\lambda} = z((2-\lambda) + \sqrt{\lambda^2 - 4\lambda})$$
$$(2-\lambda)(z-1) = (z+1)\sqrt{\lambda^2 - 4\lambda}$$

Now, we will square both sides to get rid of that nasty square root:

$$(2-\lambda)^2(z-1)^2 = (z+1)^2(\lambda^2 - 4\lambda) = (z+1)^2((2-\lambda)^2 - 4)$$
$$(2-\lambda)^2((z-1)^2 - (z+1)^2) = -4(z+1)^2$$
$$-4z(2-\lambda)^2 = -4(z+1)^2$$
$$(2-\lambda)^2 = \frac{(z+1)^2}{z}$$

Next, let's take the square root of both side to solve for λ.

$$2-\lambda = \frac{z+1}{\sqrt{z}}$$

Then, we will re-sub $z = e^{i\frac{2k\pi}{n+1}}$ into our equation:

$$2-\lambda = \frac{e^{i\frac{2k\pi}{n+1}} + 1}{\sqrt{e^{i\frac{2k\pi}{n+1}}}} = \frac{e^{i\frac{2k\pi}{n+1}} + 1}{e^{i\frac{k\pi}{n+1}}}$$

To simplify the equation even further, we multiply the top and bottom by $e^{-i\frac{k\pi}{n+1}}$:

$$2-\lambda = \frac{e^{i\frac{2k\pi}{n+1}} + 1}{e^{i\frac{2k\pi}{n+1}}} * \frac{e^{-i\frac{k\pi}{n+1}}}{e^{-i\frac{k\pi}{n+1}}}$$
$$2-\lambda = e^{i\frac{k\pi}{n+1}} + e^{-i\frac{k\pi}{n+1}}$$

Now, we can finish this up by using Euler's formula:

$$2-\lambda = \cos\left(\frac{k\pi}{n+1}\right) + i\sin\left(\frac{k\pi}{n+1}\right) + \cos\left(\frac{-k\pi}{n+1}\right) + i\sin\left(\frac{-k\pi}{n+1}\right)$$

Use the even-odd property from trigonometry and we get:

$$2-\lambda = \cos\left(\frac{k\pi}{n+1}\right) + i\sin\left(\frac{k\pi}{n+1}\right) + \cos\left(\frac{k\pi}{n+1}\right) - i\sin\left(\frac{k\pi}{n+1}\right)$$
$$2-\lambda = 2\cos\left(\frac{k\pi}{n+1}\right)$$

Finally, we solve for λ:

$$\lambda = 2 - 2\cos\left(\frac{k\pi}{n+1}\right)$$

Let's explore cases when n = 2,3,5,6,7:

$$\underset{2x2}{\lambda} = \begin{bmatrix}1\\3\end{bmatrix},\quad \underset{3x3}{\lambda} = \begin{bmatrix}2\\2-\sqrt{2}\\2+\sqrt{2}\end{bmatrix},\quad \underset{5x5}{\lambda} = \begin{bmatrix}1\\2\\3\\2-\frac{\sqrt{3}}{2}\\2+\frac{\sqrt{3}}{2}\end{bmatrix},\quad \underset{7x7}{\lambda} = \begin{bmatrix}2\\2-\sqrt{2}\\2+\sqrt{2}\\2-\sqrt{2-\sqrt{2}}\\2+\sqrt{2-\sqrt{2}}\\2-\sqrt{2+\sqrt{2}}\\2+\sqrt{2+\sqrt{2}}\end{bmatrix}$$

The reason for picking these example specifically is because it is easy to see the pattern here. Notice, how all of these eigenvalues can be rewritten as:

$$\underset{2x2}{\lambda} = \begin{bmatrix}2-1\\2+1\end{bmatrix},\quad \underset{3x3}{\lambda} = \begin{bmatrix}2-0\\2-\sqrt{2}\\2+\sqrt{2}\end{bmatrix},\quad \underset{5x5}{\lambda} = \begin{bmatrix}2-1\\2-0\\2+1\\2-\frac{\sqrt{3}}{2}\\2+\frac{\sqrt{3}}{2}\end{bmatrix},\quad \underset{7x7}{\lambda} = \begin{bmatrix}2-0\\2-\sqrt{2}\\2+\sqrt{2}\\2-\sqrt{2-\sqrt{2}}\\2+\sqrt{2-\sqrt{2}}\\2-\sqrt{2+\sqrt{2}}\\2+\sqrt{2+\sqrt{2}}\end{bmatrix}$$

or even as:

$$\underset{2x2}{\lambda} = \begin{bmatrix}2\\2\end{bmatrix} - \begin{bmatrix}1\\-1\end{bmatrix},\quad \underset{3x3}{\lambda} = \begin{bmatrix}2\\2\\2\end{bmatrix} - \begin{bmatrix}0\\\sqrt{2}\\-\sqrt{2}\end{bmatrix},\quad \underset{5x5}{\lambda} = \begin{bmatrix}2\\2\\2\\2\\2\end{bmatrix} - \begin{bmatrix}1\\0\\-1\\\frac{\sqrt{3}}{2}\\-\frac{\sqrt{3}}{2}\end{bmatrix},\quad \underset{7x7}{\lambda} = \begin{bmatrix}2\\2\\2\\2\\2\\2\\2\end{bmatrix} - \begin{bmatrix}0\\\sqrt{2}\\-\sqrt{2}\\\sqrt{2-\sqrt{2}}\\-\sqrt{2-\sqrt{2}}\\\sqrt{2+\sqrt{2}}\\-\sqrt{2+\sqrt{2}}\end{bmatrix}$$

Now, when we focus on the right vector for each of the sets of eigenvalues, we see the equation we derived above start to form:

$$\underset{2x2}{\lambda} = \begin{bmatrix}2\\2\end{bmatrix} - \begin{bmatrix}2\cos(\frac{\pi}{3})\\2\cos(\frac{2\pi}{3})\end{bmatrix},\quad \underset{3x3}{\lambda} = \begin{bmatrix}2\\2\\2\end{bmatrix} - \begin{bmatrix}2\cos(\frac{2\pi}{4})\\2\cos(\frac{\pi}{4})\\2\cos(\frac{3\pi}{4})\end{bmatrix},\quad \underset{5x5}{\lambda} = \begin{bmatrix}2\\2\\2\\2\\2\end{bmatrix} - \begin{bmatrix}2\cos(\frac{2\pi}{6})\\2\cos(\frac{3\pi}{6})\\2\cos(\frac{4\pi}{6})\\2\cos(\frac{\pi}{6})\\2\cos(\frac{5\pi}{6})\end{bmatrix},\quad \underset{7x7}{\lambda} = \begin{bmatrix}2\\2\\2\\2\\2\\2\\2\end{bmatrix} - \begin{bmatrix}2\cos(\frac{4\pi}{8})\\2\cos(\frac{2\pi}{8})\\2\cos(\frac{6\pi}{8})\\2\cos(\frac{3\pi}{8})\\2\cos(\frac{5\pi}{8})\\2\cos(\frac{1\pi}{8})\\2\cos(\frac{7\pi}{8})\end{bmatrix}$$

Finally, we can write a general formula as:

$$\underset{nxn}{\lambda} = \begin{bmatrix}2-2\cos\left(\frac{\pi}{n+1}\right)\\2-2\cos\left(\frac{2\pi}{n+1}\right)\\2-2\cos\left(\frac{3\pi}{n+1}\right)\\\vdots\\2-2\cos\left(\frac{(n-1)\pi}{n+1}\right)\end{bmatrix}$$

2.3.2 Help me find my spectral radius

Given that the max any eigenvalue, λ, can be is 4, the spectral radius of the H-matrix is 4.

2.4 Solving the system $Hx = b$

Remember Our H-matrix that was nxn:

$$H = \underbrace{\begin{bmatrix} 2 & -1 & 0 & \dots & 0 & 0 \\ -1 & 2 & -1 & 0 & \dots & 0 \\ 0 & -1 & 2 & -1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & -1 & 2 & -1 \\ 0 & 0 & \dots & 0 & -1 & 2 \end{bmatrix}}_{nxn}$$

First, let's introduce our b-vector that will be $nx1$:

$$b = \underbrace{\begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix}}_{nx1}$$

We want explore different methods for solving for $Hx = b$:

$$Hx = b \quad \Longrightarrow \begin{bmatrix} 2 & -1 & 0 & \dots & 0 & 0 \\ -1 & 2 & -1 & 0 & \dots & 0 \\ 0 & -1 & 2 & -1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & -1 & 2 & -1 \\ 0 & 0 & \dots & 0 & -1 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix}$$

The list of different methods we will use to solve this system are: Gauss-Elimination w/o pivot, Gauss-Elimination w/ partial pivot, LU factorization, Jacobi, Gauss-Seidel, and SOR with $\omega = .5, 1, 1.5$. We will use different matrix sizes, $n = 1 : 500$ to see if this effects our results. We will also be using MATLAB to run these operations.

2.4.1 Analyzing the CPU time

When deciding which method to use, it is helpful to know the speed it takes for each method to calculate the solutions. That way you can get your solutions as quick as possible. Below is a plot that illustrates the time, in seconds, it takes for each method to compute the solutions at each matrix size, nxn for $n = 1 : 500$.

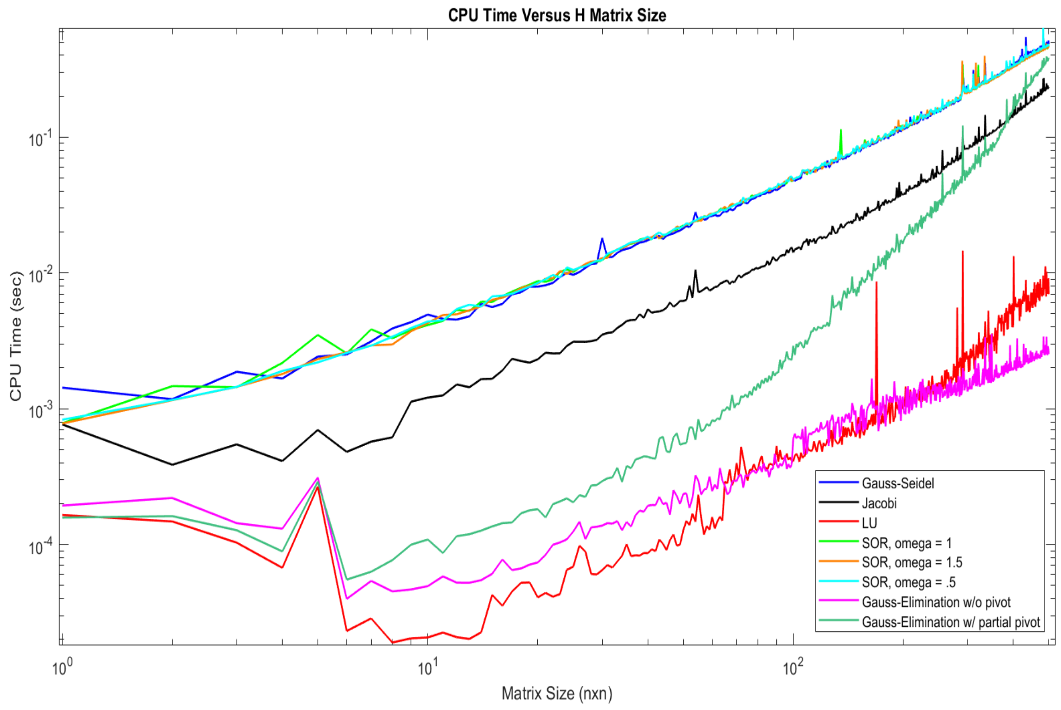

Figure 1: This plot illustrates the amount of time it takes each method to compute the solutions to $Hx = b$ at each nxn matrix size of H.

What's amazing about this plot is the difference in CPU time as the matrix size gets really big. If we look at the LU factorization method when $n < 100$ we would think that it's the fastest overall, but if we look at the end of it's journey we notice that it becomes slower than Gauss-Elimination w/o pivot. Gauss-Elimination w/ partial pivot also seems promising at the start but as n gets much bigger it's cpu time grows rapidly. Now, based on this plot it seems that if $n < 100$ then LU factorization will probably be the optimal option for computing the solution for $Hx = b$, but if $n \geq 100$ the optimal option seems to be Gauss-Elimination w/o pivot.

Let's see if we can judge from this graph the trends for each method as n gets larger. To do this we will plot the average CPU time.

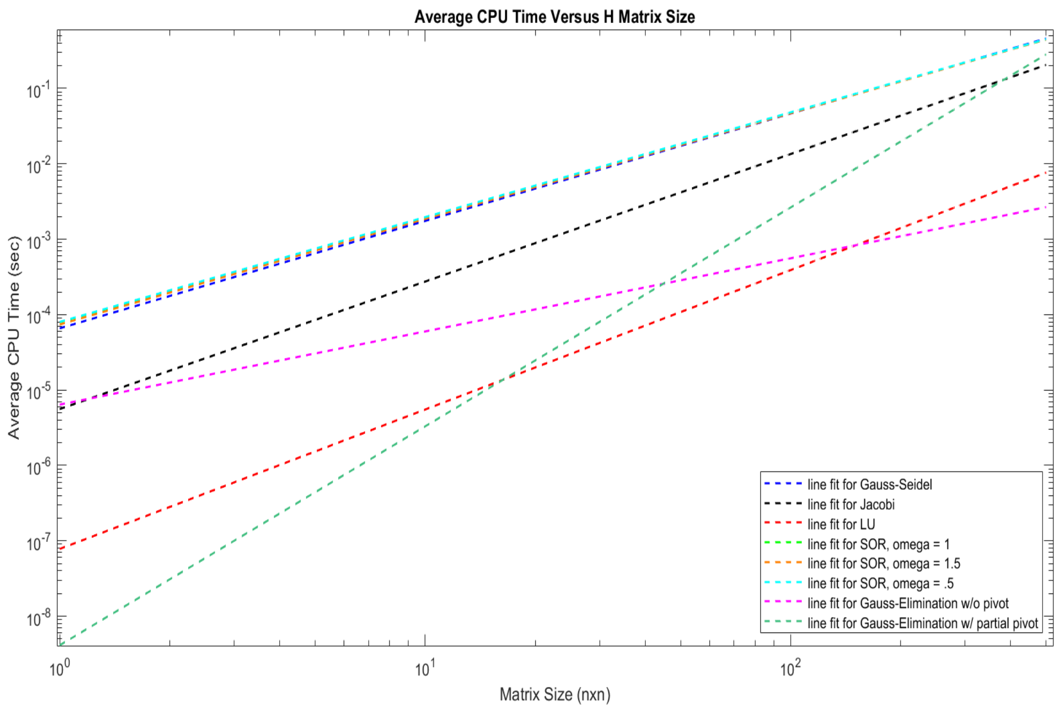

Figure 2: This plot illustrates the average amount of time it takes each method to compute the solutions to $Hx = b$ at each nxn matrix size of H.

This graph can seems much easier to read than the fig. 1. Based on this plot we can see that the slope for Gauss-Elimination w/o pivot has the flattest slope which tells us that as n increases than on average we should expect to see theGauss-Elimination w/o pivot continue being the fastest method for computing the solutions. The slope for Gauss-Elimination w/ partial pivot seems to be the largest, so we can expect on average as n gets larger this method will be the slowest.

2.4.2 Picking the optimal relaxation parameter ω

When using the SOR method, picking the optimal relaxation parameter, ω will improve the accuracy of your results. We can pick the optimal ω when looking at a graph of the spectral radius versus matrix size, n. Below is an example of this:

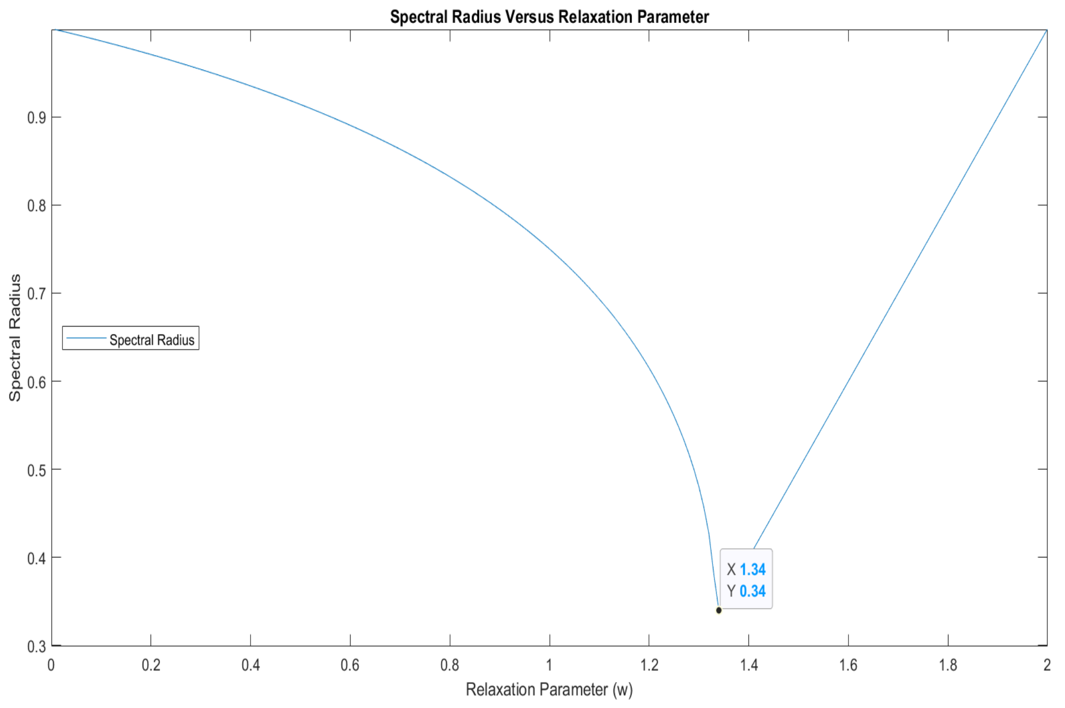

Figure 3: This plot helps us find the optimal spectral radius for H being $5x5$.

Notice, that the optimal ω is the minimum point of this line. So, if we find the minimum spectral radius over all possible ωs for each matrix size, then we can find the optimal ω for each H-matrix. the figure below clearly illustrates this for us.

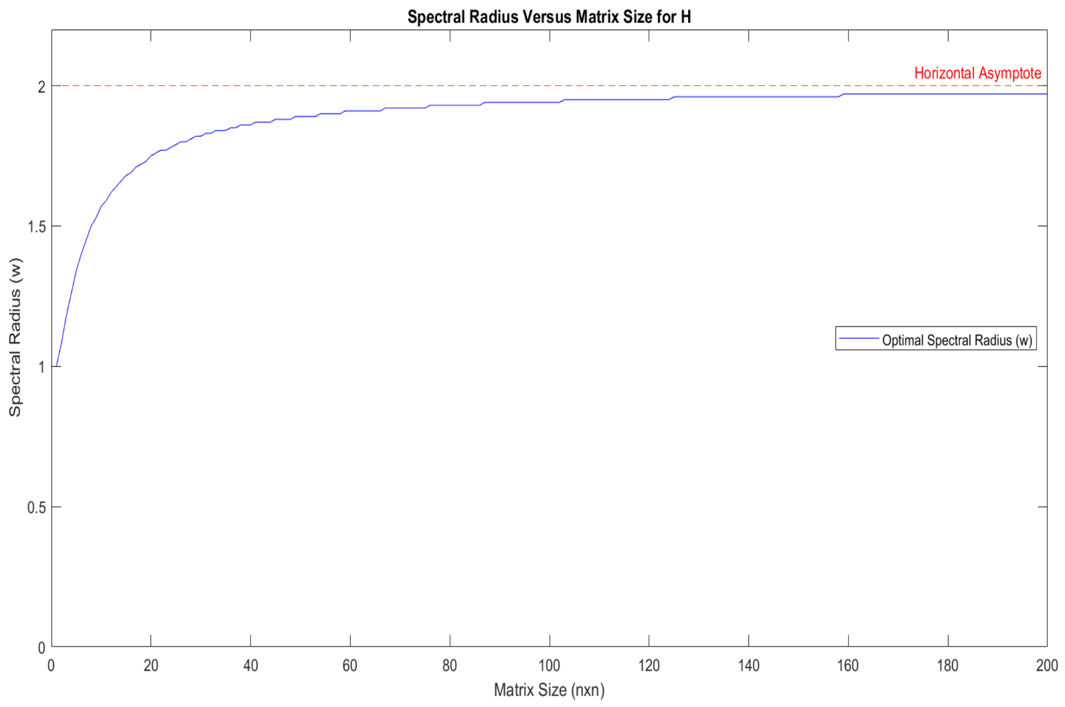

Figure 4: This plot allows us to see the optimal relaxation parameters with each matrix size, nxn

From fig. 4 we notice that as the H-matrix size gets bigger, so does the optimal relaxation parameter. Also, as $n \to \infty$, the optimal spectral radius, $\omega \to 2$.

2.4.3 How accurate are each of these methods?

Now, it is time to test how precise each method is at calculating the solutions to $Hx = b$. To do this we will find the norm of each solution vector for every $H - matrix$, $n = 1 : 500$. Then, we will compare it to the norm of the exact values to give us our errors.

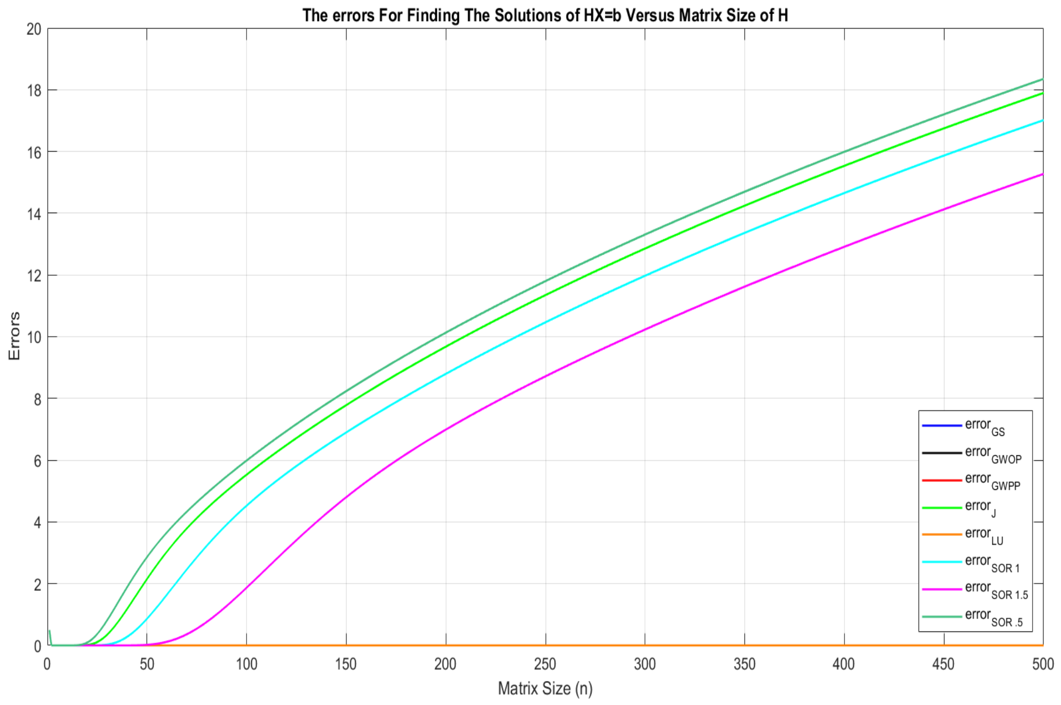

Figure 5: This plot shows how for most methods the errors increase as the matrix size for H increases.

In this plot we can see that the errors for Gauss-Seidel, Gauss-Elimination w/o pivot, Gauss-Elimination w/ partial pivot, and Lu factorization are really close to 0 for any matrix size which means that these methods are the best for finding the solutions. If we recall from above, Lu factorization, and Gauss-Elimination w/o pivot were the fastest. So, not only are they the fastest but they are also the best for calculating the solutions. Out of all 3 SOR methods, the one with $\omega = 1.5$ is the best for calculating the solutions. This also makes sense considering our discussion about fig. 4. $\omega = 1.5$ is closest to our optimal ω for any matrix size over $3x3$, which is reflected in fig. 5.

3 Source code and computer output

```
    % Project MAT 5080

clear
close all
clc

% Parameters
n = 4;
a = sym('a',[1,n-1]);
b = sym('b',[1,n]);
c = sym('c',[1,n-1]);
d = sym('d',[1,n]);
c_over_d = c(1:n-1)./d(1:n-1);

% Matrices setup
T = diag(b) + diag(a,1) + diag(c,-1);
[L,U] = LUFactorization(a,d,c_over_d,n);


% Showing L*U = T
LUverification(L,U,T,a,b,c,d,n)

%Notice that T_new is the exact same as T so this means L and U are the LU
%factorizations of T.


% Part 2 ii
n = 5;

H = diag(2*ones(1,n)) + diag(-1*ones(1,n-1),1) + diag(-1*ones(1,n-1),-1);


% Part 3
    % Solve for eigenvalues
eig_H = round(double((eig(H))),5);

    % Solve for the cosine equation
for k = 1:n
    eig_cosine(k) = round(2-2*cos(k*pi/(n+1)),5);
end
    % Checking to see if the 2 arrays are equivalent
eigCheck(eig_H,eig_cosine,n)

spectral_radius1 = max(abs(double(eig(H))));

    % Set up for problem 4
n = 500;
x_exact = zeros(n,n);
x_Gauss_Seidel = x_exact; x_Gauss_Wo_Pivot = x_exact;
x_Gauss_WP_Pivot = x_exact; x_Jacobi = x_exact; x_LUFactorization = x_exact;
x_SOR_1 = x_exact; x_SOR_1_half = x_exact; x_SOR_half = x_exact;
% New Parameters for 4
for n = 1:500
H = diag(2*ones(1,n)) + diag(-1*ones(1,n-1),1) + diag(-1*ones(1,n-1),-1);
```

```
clear a b c d
a = -1*ones(1,n-1);
c = a;
b = 2*ones(1,n);
d(1) = b(1);
for i = 2:n
    d(i) = b(i) - (a(i-1)*c(i-1)/d(i-1));
end
c_over_d = c(1:n-1)./d(1:n-1);

% Calculating a new L and U
[L,U] = LUFactorization(a,d,c_over_d,n);

format long

b_solve = zeros(n,1);
b_solve(1) = 1; b_solve(n) = 1;

    % Exact solutions
    x_exact(1:n,n) = ones(n,1);

    % Part 4a - Gaussian Elimination Without Pivoting
        % n comes form part 2ii so n=4

tic
x_Gauss_Wo_Pivot(1:n,n) = Gauss_WO_Pivot(H,b_solve,L,U);
comp_time_Wo(n) = toc;

tic
x_Gauss_WP_Pivot(1:n,n) = Gauss_WP_Pivot(H,b_solve,n);
comp_time_WP(n) = toc;
    % Part 4b LU Factorization method page 5 of week 4

tic
x_LUFactorization(1:n,n) = LUFactorization_Solution(L,U,b_solve);
comp_time_LU(n) = toc;
    % Part 4c - Jacobi
        % For fun make a while loop to see how long it takes to get to the
        % correct points.

tic
x_Jacobi(1:n,n) = Jacobi(H,b_solve,n);
comp_time_J(n) = toc;

    % Part 4d - Gauss Seidel

tic
x_Gauss_Seidel(1:n,n) = Gauss_Seidel(H,b_solve,n);
comp_time_GS(n) = toc;

    % Part 4e - SOR
     i = 1;
for omega = .5:.5:1.5
    [x_SOR(:,i), comp_time_SOR(i)] = SOR(H,b_solve,n,omega);
    i = i + 1;
```

```
end
x_SOR_half(1:n,n) = x_SOR(:,1);
x_SOR_1(1:n,n) = x_SOR(:,2);
x_SOR_1_half(1:n,n) = x_SOR(:,3);
comp_time_SOR_half(n) = comp_time_SOR(1);
comp_time_SOR_1(n) = comp_time_SOR(2);
comp_time_SOR_1_half(n) = comp_time_SOR(3);
clear x_SOR
end

% Plot the CPU time of each method
    iterations = 1:n;
    sample_size = round((n - 1)/2);

    % finding slope and intercepts of each method
cpu_times = [comp_time_GS; comp_time_J; comp_time_LU; comp_time_SOR_1;...
    comp_time_SOR_1_half; comp_time_SOR_half; comp_time_Wo; comp_time_WP];
for i = 1:8
[slope(i),intercept(i)] = slope_CPU(cpu_times(i,:),iterations,sample_size,n);
end
close all
color = [0,0,1; 0,0,0; 1,0,0; 0,1,0; 1,.5,0; 0,1,1; 1,0,1; .25,.75,.5];
for i = 1:8
    figure(1);
loglog(1:500,cpu_times(i,:),'color',color(i,:),'Linewidth',1.25)
hold on
    figure(2);
loglog(1:500,intercept(i)*iterations.^(slope(i)),'--','color',color(i,:),'Linewidth',1.25)
hold on
end
figure(1)
legend('Gauss-Seidel','Jacobi','LU','SOR, omega = 1','SOR, omega = 1.5',...
    'SOR, omega = .5','Gauss-Elimination w/o pivot',...
    'Gauss-Elimination w/ partial pivot','Location','SouthEast')
title('CPU Time Versus H Matrix Size')
xlabel('Matrix Size (nxn)')
ylabel('CPU Time (sec)')
figure(2)
legend('line fit for Gauss-Seidel','line fit for Jacobi','line fit for LU',...
    'line fit for SOR, omega = 1','line fit for SOR, omega = 1.5',...
    'line fit for SOR, omega = .5',...
    'line fit for Gauss-Elimination w/o pivot',...
    'line fit for Gauss-Elimination w/ partial pivot','Location',...
    'SouthEast')
xlabel('Matrix Size (nxn)')
ylabel('Average CPU Time (sec)')
title('Average CPU Time Versus H Matrix Size')
hold off


% plot the spectral radius for each matrix size.

% Setting up for B_SOR
for n = 1:200
H = diag(2*ones(1,n)) + diag(-1*ones(1,n-1),1) + diag(-1*ones(1,n-1),-1);
D_H = diag(diag(H));
```

```
I = eye(n);

Lower = tril(H,-1);
Upper = triu(H,1);

i = 1;

for omega = 0:.01:2
    B_SOR = inv(I-omega*inv(D_H)*Lower) * ((1-omega)*I + omega*inv(D_H)*Upper);
    spectral_radius(i) = max(abs(eig(B_SOR)));
    i = 1+i;
end
omeg = find(spectral_radius==min(spectral_radius));
w(n) = (omeg-1)/100;
end

figure(3)
plot(1:n,w,'b')
hold on
yline(2,'--r','Horizontal Asymptote')
hold off
ylim([0,2.2])
xlabel('Matrix Size (nxn)')
ylabel('Relaxation Parameter (w)')
legend('Optimal Relaxation Parameter (w)','Location','East')
title('Relaxation Parameter Versus Matrix Size for H')

figure(4)
plot(0:.01:2,spectral_radius)
xlabel('Relaxation Parameter (w)')
ylabel('Spectral Radius')
title('Spectral Radius Versus Relaxation Parameter')
legend('Spectral Radius','Location','West')


% Time to compute the error.

for i = 1:500
   norm_exact(i) = norm(x_exact(:,i));
   norm_GS(i) = norm(x_Gauss_Seidel(:,i));
   norm_GWOP(i) = norm(x_Gauss_Wo_Pivot(:,i));
   norm_GWPP(i) = norm(x_Gauss_WP_Pivot(:,i));
   norm_J(i) = norm(x_Jacobi(:,i));
   norm_LU(i) = norm(x_LUFactorization(:,i));
   norm_SOR_1(i) = norm(x_SOR_1(:,i));
   norm_SOR_1_half(i) = norm(x_SOR_1_half(:,i));
   norm_SOR_half(i) = norm(x_SOR_half(:,i));
end
norms = [norm_exact;norm_GS;norm_GWOP;norm_GWPP;norm_J;norm_LU;norm_SOR_1;...
    norm_SOR_1_half;norm_SOR_half];

% Errors

for i = 1:9
    errors(i,:) = norms(1,:) - norms(i,:);
end
```

```
color = [0,0,1; 0,0,0; 1,0,0; 0,1,0; 1,.5,0; 0,1,1; 1,0,1; .25,.75,.5];
figure(5)
for i = 2:9
   plot(1:500,errors(i,:),'color',color(i-1,:),'LineWidth',1.25)
   hold on
end
xlabel('Matrix Size (n)')
ylabel('Errors')
legend('error_{GS}','error_{GWOP}','error_{GWPP}','error_{J}','error_{LU}','error_{SOR 1}',...
    'error_{SOR 1.5}','error_{SOR .5}','Location','SouthEast')
title('The errors For Finding The Solutions of HX=b Versus Matrix Size of H')
grid on

    % L and U factorization for Project 5080

function[L,U] = LUFactorization(a,d,c_over_d,n)

L = diag(ones(1,n)) + diag(c_over_d,-1);
U = diag(d) + diag(a,1);

end

    % Part 1 for the project 5080

function[] = LUverification(L,U,T,a,b,c,d,n)

T_new = L*U;

if T_new(1,1) == d(1)
    T_new(1,1) = b(1);
end

for i = 2:n
    if T_new(i,i) == d(i) + (a(i-1)*c(i-1))/d(i-1)
    T_new(i,i) = b(i);
    end
end

if T == T_new
    fprintf("It is offically confirmed via MATLAB that L*U = T\n")
end

end

    % eigenvalue check for project 5080

function[] = eigCheck(eig_H,eig_cosine,n)
true = 0;
for i = 1:n
   for j = 1:n
      if eig_H(i) == eig_cosine(j)
         true = 1;
         break
      end
   end
   if true ~= 1
```

```
        tru(i) = 0;
   else
        tru(i) = 1;
   end
   tru = tru';
end

if sum(tru) == n
   fprintf("It is confirmed that our eigenvalues are equal to the equation.\n")
end

end

    % eigenvalue check for project 5080

function[] = eigCheck(eig_H,eig_cosine,n)
true = 0;
for i = 1:n
   for j = 1:n
      if eig_H(i) == eig_cosine(j)
         true = 1;
         break
      end
   end
   if true ~= 1
        tru(i) = 0;
   else
        tru(i) = 1;
   end
   tru = tru';
end

if sum(tru) == n
   fprintf("It is confirmed that our eigenvalues are equal to the equation.\n")
end

end

    % Gauss with partial pivoting

function[x_Gauss_WP_Pivot] = Gauss_WP_Pivot(H,b,n)

if n==1
        x_Gauss_WP_Pivot = .5;
else
    % Creating the augmented matrix
Augmen = [H,b];
    % Partial Pivoting has Begun
for i = 1:n-1
    [Max, Row] = max(abs(Augmen(i:n,i)));
    store = Augmen(i,:);
    Augmen(i,:) = Augmen(Row+i-1,:);
    Augmen(Row+i-1,:) = store;
        % Time for echlon form
    for j = i+1:n
       multiplier = Augmen(j,i)/Augmen(i,i);
       Augmen(j,:) = Augmen(j,:) - multiplier * Augmen(i,:);
    end

end
for i = 2:n
    multiplier = Augmen(1:i-1,i)/Augmen(i,i);
    Augmen(1:i-1,:) = Augmen(1:i-1,:) - multiplier * Augmen(i,:);
end
x_Gauss_WP_Pivot = Augmen(:,n+1)./diag(Augmen);
end
end

    % Finding solutions using LU Factorizations

function[x_LUFactorization] = LUFactorization_Solution(L,U,b)

y = inv(L)*b;

x_LUFactorization = inv(U)*y;


end

    % Jacobi Method

function[x_Jacobi] = Jacobi(A,b,n)

x0 = zeros(1,n);
for t = 1:500
for i = 1:n
    xt = x0;
    xt(i) = 0;
    x(i) = (b(i)-sum(A(i,:)*xt'))/A(i,i);
end
    x0 = x;
end
x_Jacobi = x';
end

    % Gauss-Seidel Method

function[x_Gauss_Seidel] = Gauss_Seidel(A,b,n)

x0 = zeros(1,n);
for j = 1:500
for i = 1:n
    xt(i) = 0;
    x(i) = (b(i)-sum(A(i,:)*xt'))/A(i,i);
    xt = x0;
    xt(1:i) = x(1:i);
end
x0 = x;
end
x_Gauss_Seidel = x';

end

    % SOR Method
```

```
function[x_SOR, comp_time_SOR] = SOR(A,b,n,omega)

x0 = zeros(1,n);
xt = x0;
tic
for t = 1:500
for i = 1:n
    xt(i) = 0;
    x(i) = (1-omega)*x0(i) + omega*(b(i) - sum(A(i,:)*xt'))/A(i,i);
    xt = x0;
    xt(1:i) = x(1:i);
end
comp_time = toc;
    x0 = x;
end
comp_time_SOR = comp_time;
x_SOR = x';
end

    % Calculating slope and intercept for project MAT 5080

function[slope,intercept] = slope_CPU(y,x,sample_size,n)


random = randi([150,n],[sample_size,1]);

slope = sum((log(y(random)) - log(y(random-50))) /...
    (log(x(random)) - log(x(random-50))));


% y = ax^b   a = y/x^b    log(y) = b*log(ax)      log(y) = b*log(x) + b*log(a)
for i = 1:sample_size
a(i) = y(random(i))/(x(random(i))^slope);
end

intercept = sum(a)/sample_size;


end
```