Final Coding Project

AM213A UCSC

Kevin Silberberg

2025-03-13

SVD for image compression

Structure of code for SVD problem

```
svd/
 ├── Makefile
 ├── dog_bw_data.dat
 ├── plot_errors.py
 ├── plot_compressed_dog.py
 ├── requirements.txt
 ├── include/
 │   ├── svd.h
 │   └── utils.h
 └── src/
     ├── main.c
     ├── svd.c
     └── utils.c
```

Usage

Dependencies

Python

The Python script requires the following dependencies:

- `numpy`
- `matplotlib`
- `pandas`

before proceeding make sure to run the following to set up a python virtual environment.

```
cd svd/
python -m venv myenv
source myenv/bin/activate
pip install -r requirements.txt
```

LAPACKE

The program requires the LAPACKE C interface to LAPACK (available via the headers `lapacke.h` and `cblas.h`) installed on your system.

Compilation

Compile the program by executing:

```
make
```

This generates a `main` executable in the `svd/` directory.

Execution

Run the executable with the provided dataset as follows:

```
./main dog_bw_data.dat
```

The program will print singular values 1–9 to the console and generate corresponding files named `image_appn_xxxx.dat` for singular values:

$k = 10, 20, 40, 80, 160, 320, 640, 1279$ as well as their singular values.

Additionally a file `errors.csv` containing the errors of each corresponding k using the average Frobenius norm,

$$E_k = \frac{\|\mathbf{A} - \mathbf{A}_{\sigma_k}\|_F}{mn} \tag{1}$$

will be created.

Important Notes

- **Warning:** The program is specifically designed for the provided dataset. If using other datasets, ensure you modify the hard-coded `int kvals[]` array in `svd/src/main.c` so that $k \leq \min(m, n)$ for a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$.

Generating the Image

There are two `.py` python files responsible for generating both images, mainly, `plot_compressed_dog.py` and `plot_errors.py`.

After running the main program with `dog_bw_data.dat` and generating the `image_appn_####.dat` files, use the following commands to create the gray scale image for 8 images corresponding to each $k > 9$ value:

```
python plot_compressed_dog.py
```

This script reads the generated data files and saves the image `compressed_dog.png` to the `svd/` directory.

To generate the image of the plot for the E_k by k run the following command,

```
python plot_errors.py
```

This will produce a plot `errors.png` in the current directory.

Finally run,

```
deactive
```

Report Singular values

The *Singular Values* printed by the program are as follows:

Table 1: Singular values for increasing k

σ_k	Singular values
σ_1	281897.276065
σ_2	46561.709015
σ_3	31487.799647
σ_4	26436.718035
σ_5	19631.551107
σ_6	15569.226576
σ_7	14390.263395
σ_8	11254.046754
σ_9	9660.394021
σ_{10}	9411.738263
σ_{20}	4167.180601
σ_{40}	2035.985992
σ_{80}	1193.122010
σ_{160}	758.441870
σ_{320}	456.383015
σ_{640}	189.133997
σ_{1279}	4.760516

All eight compressed dog images

Each of the sub-figures in the following, are reconstructions $\mathbf{A}_{\sigma_k} = U\Sigma_k V^T$, where

$$\Sigma_k = \begin{bmatrix} \sigma_1 & 0 & \cdots & 0 \\ 0 & \sigma_2 & \cdots & 0 \\ \vdots & \vdots & \sigma_k & \vdots \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{2}$$

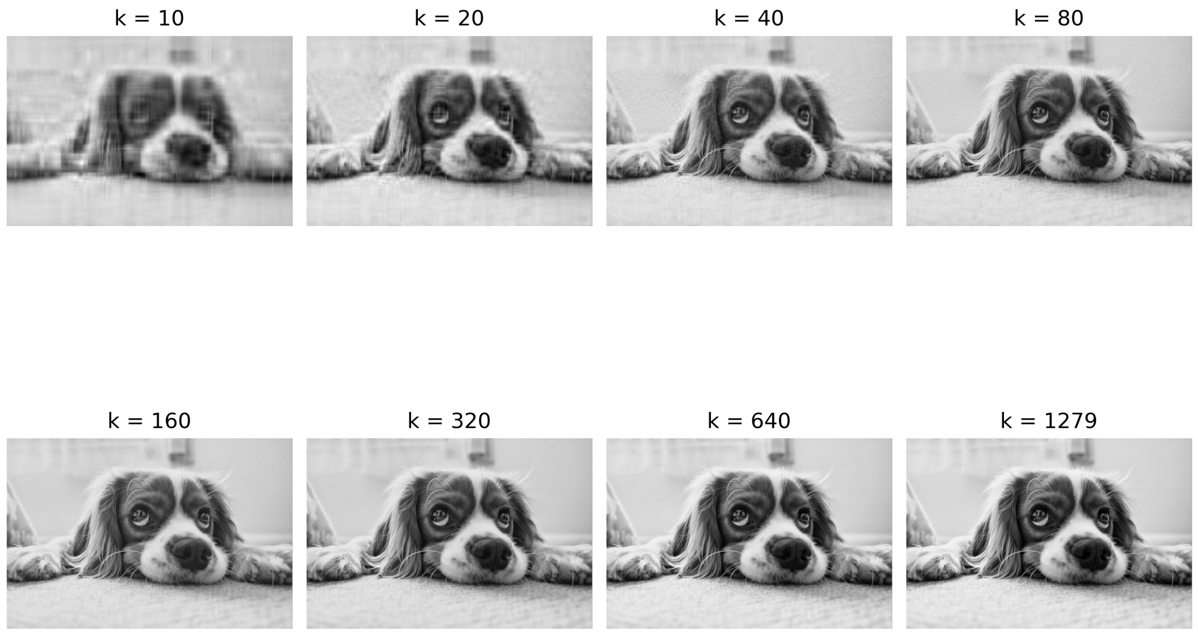


Figure 1: All eight reconstructed images after truncating the singular values from $1 \to k$ for $k = 10, 20, 40, 80, 160, 320, 640, 1279$

We can see that at the singular value $k = 10$ the image of the dog is still remarkably descernable and by reconstructing the image with only 80 singular values the image is almost identical to the full information image $k = 1279$.

Error plot

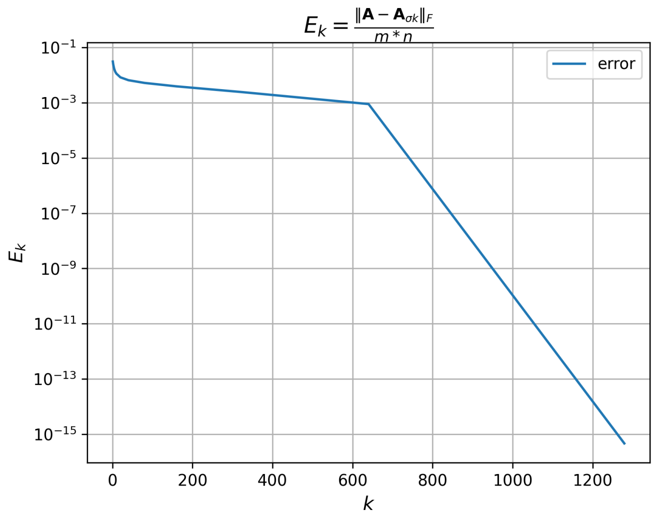


Figure 2: log(Y) by X plot of the averaged Frobenius norm error between the original image and the reconstructed images for increasing values of k.

The average Frobenius error decreases sharply from $k = 1 \to 10$ and doesn't go below 10^{-3} until $k = 640$ singular values are retained in the image reconstruction.

Table 2: Numerical values of the error plotted above for increasing values of k

k	Error
1	3.092049e-02
2	2.442466e-02
3	2.078820e-02
4	1.778348e-02
5	1.588530e-02
6	1.456523e-02
7	1.333441e-02
8	1.252213e-02
9	1.188816e-02
10	1.125341e-02
20	8.286540e-03
40	6.527427e-03
80	5.207148e-03
160	3.909973e-03
320	2.452157e-03
640	8.938753e-04
1279	4.672595e-16

Iterative Methods

Structure of the code for Iterative Methods problem

```
iterative/
 ├── Makefile
 ├── requirements.txt
 ├── plot_convergence.py
 ├── run10x10.sh
 ├── include/
 │    ├── iterative.h
 │    ├── mat.h
 │    └── utils.h
 └── src/
      ├── iterative.c
      ├── main.c
      ├── mat.c
      └── utils.c
```

Usage

Compilation

Compile the program by typing into the terminal:

```
cd iterative/
make
```

This will generate a `main` executable in the `iterative/` directory.

Execution

The program can be run by the following usage:

```
./main <function> <a_11> <a_22> <a_33> ... <a_mm>
```

The first argument passed to `./main` is the name of the method you would like to use either `jacobi` or `seidel`, and the rest of the arguments to the program are the diagonal elements of the matrix $\mathbf{A}$ in order from top left to bottom right. The program will create a matrix of A of size $m \times m$ where m is the number of arguments passed after `<function>`.

example usage

```
./main jacobi 4 -7 3 0.6
```

this will create a matrix

$$\mathbf{A} = \begin{bmatrix} 4 & 1 & 1 & 1 \\ 1 & -7 & 1 & 1 \\ 1 & 1 & 3 & 1 \\ 1 & 1 & 1 & 0.6 \end{bmatrix} \tag{3}$$

and a corresponding vector

$$\mathbf{b} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} \tag{4}$$

and solve the system $\mathbf{Ax} = \mathbf{b}$ for $\mathbf{x}$ and print out the vector

$$\mathbf{x} = \begin{bmatrix} -3.2105 \\ 1.0789 \\ -3.8157 \\ 16.5789 \end{bmatrix} \tag{5}$$

run10x10.sh bash script

the script contains simply the following lines:

```
#!/usr/bin/bash
D=(2 5 10 100 1000)
for n in "${D[@]}"; do
    ./main jacobi $n $n $n $n $n $n $n $n $n $n
    ./main seidel $n $n $n $n $n $n $n $n $n $n
done
```

by makeing the program executable and running the script

```
chmod +x run10x10.sh
./run10x10.sh
```

this is an easy way to call the `./main` program and answer the part of the question "run the code for a 10x10 matrix $\mathbf{A}$ with $D = 2, 5, 10, 100, 1000$".

This will produce 10 `.csv` files corresponding to the error of each iterative jacobi and seidel main function call on each 10x10 matrix.

plot_convergence.py

all that is left to do is call the plotting routine for the `.csv` files containing the errors. You can produce the figures by first creating the python virtual environment

```
python -m venv myenv
source myenv/bin/activate
pip install -r requirements.txt
```

and then running the code

```
python plot_convergence.py
```

this will produce 5 `convergence_D####.png` files corresponding to each 10x10 matrix.

Gauss-Jacobi

The Gauss-Jacobi algorithm updates the column vector $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$ iteratively until the 2-norm of the vector $\|\mathbf{b} - \mathbf{A}\mathbf{x}^{(k+1)}\|_2 \to \epsilon$. Component-wise this can be written as

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{m} r_{ij}x_j^{(k)}\right) \tag{6}$$

where the term $r_{ij} \in \mathbf{R}$ is the original matrix $\mathbf{A}$ except with zeros along the diagonal.

The folloing algorithm is how it is implemented in the code

Algorithm 1 Gauss-Jacobi

```
Require: $\mathbf{A} \in \mathbb{R}^{m\times m}$ and $\mathbf{b} \in \mathbb{R}^{m\times 1}$
 1: $\mathbf{x} \sim \text{Uniform}(-1,1) \in \mathbb{R}^{m\times 1}$
 2: $\mathbf{y} \in \mathbb{R}^{m\times 1}$
 3: $r = \|\mathbf{b} - \mathbf{A}\mathbf{x}\|_2$
 4: while $r > \epsilon$ do
 5:     for $i = 1 : m$ do
 6:         $s = 0.0$
 7:         for $j = 1 : m$ do
 8:             if $i \neq j$ then
 9:                 $s = s + \mathbf{A}_{i,j} \cdot \mathbf{x}_j$
10:             end if
11:         end for
12:         $\mathbf{y}_i = \frac{1}{\mathbf{A}_{i,i}} \cdot (\mathbf{b}_i - s)$
13:     end for
14:     $\mathbf{x} = \mathbf{y}$
15:     $r = \|\mathbf{b} - \mathbf{A}\mathbf{x}\|_2$
16: end while
17: return $\mathbf{x}$
```

The function `jacobi` takes in as argument the matrix $\mathbf{A}$ and the column vector $\mathbf{b}$ and returns the solution column vector $\mathbf{x}$. First we ensure that the matrix $\mathbf{A}$ is square, then we initialize the solution vector $\mathbf{x}$ as a random vector sampled from the Uniform distribution in the range $-1, 1$. The function then proceeds to iteratively compute

$$\mathbf{x}^{(k+1)} = \mathbf{D}^{-1}\left(\mathbf{b} - \mathbf{R}\mathbf{x}^{(k)}\right) \tag{7}$$

where $\mathbf{A} = \mathbf{D} + \mathbf{R}$ and $\mathbf{D}$ is a diagonal matrix only containing the diagonal entries of the matrix $\mathbf{A}$ and the matrix $\mathbf{R}$ is the counter part to $\mathbf{D}$ containing all the off-diagonal entries of $\mathbf{A}$ with zeros along the diagonal.

the algorithm converges if the spectral radius of the matrix $\mathbf{D}^{-1}\mathbf{R}$ is less than 1, these matrices are called diagonally dominant.

The program checks for convergence by simply checking if the error

$$r = \|\mathbf{b} - \mathbf{A}\mathbf{x}\|_2 \tag{8}$$

is less than the value 10000 for every iteration of the while loop.

We can check how many allocations the program uses by running

```
valgrind ./main jacobi 4 -7 3 0.6
```

which shows that only 7 allocations were made.

Gauss-Seidel

The Gauss-Seidel algorithm is very similar to the Jacobi algorithm. However, instead of updating a separate vector $\mathbf{y}$ and then setting $\mathbf{x} = \mathbf{y}$ after each iteration, the algorithm directly updates $\mathbf{x}$ without requiring an additional vector. The algorithm iteratively updates the column vector $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$ by using the most recent values of $\mathbf{x}$ for each row until the 2-norm of the residual vector satisfies $\|\mathbf{b} - \mathbf{A}\mathbf{x}^{(k+1)}\|_2 \leq \epsilon$. Component-wise, this can be written as:

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{m} a_{ij}x_j^{(k)}\right) \tag{9}$$

To visually see the difference take the following 3×3 matrix

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \tag{10}$$

We update the state vector $\mathbf{x}^{(k+1)}$ for every k as follows:

Let $\mathbf{x}^{(0)} \sim \text{Uniform}(-1, 1)$

$$\mathbf{x}^{(k+1)} = \begin{bmatrix} \frac{1}{a_{11}}\left(b_1 - a_{12}x_2^{(k)} - a_{13}x_3^{(k)}\right) \\ \frac{1}{a_{22}}\left(b_2 - a_{21}x_1^{(k+1)} - a_{23}x_3^{(k)}\right) \\ \frac{1}{a_{33}}\left(b_3 - a_{31}x_1^{(k+1)} - a_{32}x_2^{(k+1)}\right) \end{bmatrix} \tag{11}$$

Notice that as we iterate through the rows of $\mathbf{A}$, we exclude the diagonal elements from the summation before subtracting from the corresponding b_i element. The terms included in the summation are the

kth iteration values corresponding to the upper diagonal entries of $\mathbf{A}$ and the $(k+1)$th iteration values corresponding to the lower diagonal entries of $\mathbf{A}$.

The folloing algorithm is how it is implemented in the code

Algorithm 2 Gauss-Seidel

```
Require: $\mathbf{A} \in \mathbb{R}^{m\times m}$ and $\mathbf{b} \in \mathbb{R}^{m\times 1}$
 1: $\mathbf{x} \sim \text{Uniform}(-1,1) \in \mathbb{R}^{m\times 1}$
 2: $r = \|\mathbf{b} - \mathbf{Ax}\|_2$
 3: while $r > \epsilon$ do
 4:     for $i = 1 : m$ do
 5:         $s = 0.0$
 6:         for $j = 1 : m$ do
 7:             if $i < j$ then
 8:                 $s = s + \mathbf{A}_{i,j} \cdot \mathbf{x}_j$
 9:             end if
10:             if $i > j$ then
11:                 $s = s + \mathbf{A}_{i,j} \cdot \mathbf{x}_j$
12:             end if
13:         end for
14:         $\mathbf{x}_i = \frac{1}{\mathbf{A}_{i,i}} \cdot (\mathbf{b}_i - s)$
15:     end for
16:     $r = \|\mathbf{b} - \mathbf{Ax}\|_2$
17: end while
18: return $\mathbf{x}$
```

The Gauss-Seidel algorithm generally has much better convergence than the Gauss-Jacobi algorithm. As we will see in figure 3, if the Gauss-Jacobi algorithm converges, then the Gauss-Seidel algorithm converges faster. Additionally, the Gauss-Seidel algorithm sometimes converges for matrices that do not converge with the Gauss-Jacobi algorithm.

Run code for 10×10 matrix

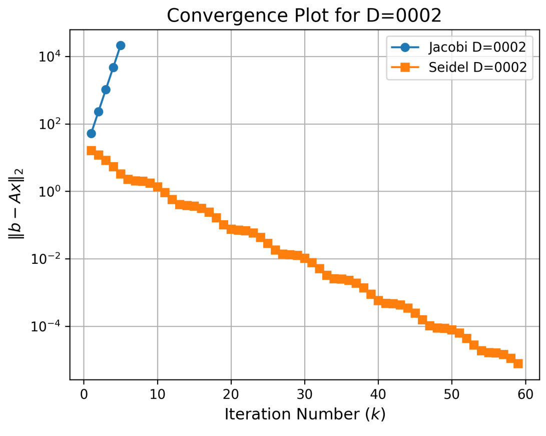
As mentioned previously we can run the code by calling the `run10x10.sh` script which calls both the `jacobi` and `seidel` algorithm on the 10×10 matrices with all $2, 5, 10, 100,$ and 1000 entries along the diagonal.

Recall that the convergence of the Gauss-Jacobi and Gauss-Seidel algorithms are tied to the diagonal dominance, or the the inverse of the spectral radius of the matrix $\mathbf{A}$ in the system $\mathbf{Ax} = \mathbf{b}$. As the diagonal elements are all equal in each of the matrices above as $D \to \infty$ the matrix becomes increasingly well-conditioned and the inverse of the spectral radius of the matrix $\mathbf{D}^{-1}\mathbf{R}$ goes to zero.

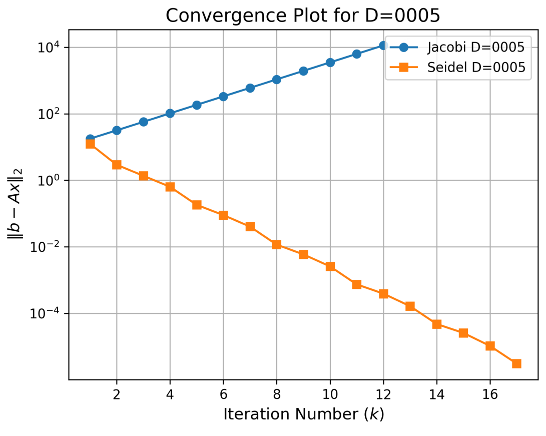
Table 3: Spectral radius for increasing values of the diagonal entries of $\mathbf{A}$

D	$\rho(\mathbf{D}^{-1}\mathbf{R})$
2	4.5
5	1.8
10	0.9
100	0.09
1000	0.009

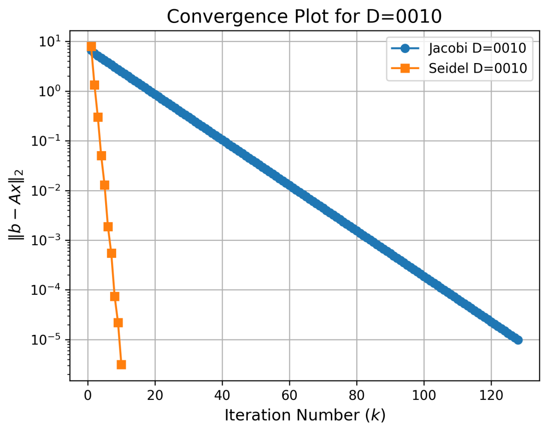
The table 3 shows that for increasingly larger values of D, the spectral radius of the matrix $\mathbf{D}^{-1}\mathbf{R}$ decreases. This explains why the Gauss-Jacobi algorithm for matrices with diagonal entries $D = 2, 5$ the



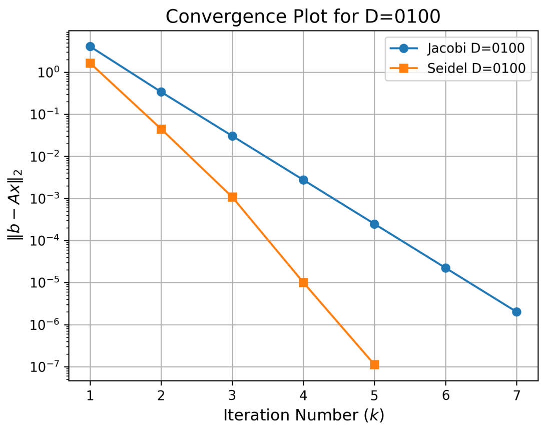
(a)



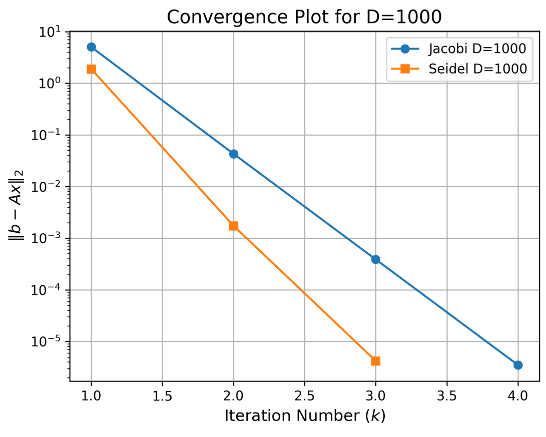
(b)



(c)



(d)



(e)

Figure 3: Convergence plots for the Gauss-Jacobi and Gauss-Seidel algorithms applied to solve the system $\mathbf{Ax} = \mathbf{b}$ where $\mathbf{A}$ is a matrix with ones in the off-diagonal entries and D along the diagonal, for $D = 2, 5, 10, 100, 1000$. $\mathbf{b}$ is a column vector such that $b_{ii} = i$. (3a) $D = 2$ Jacobi fails to converge, while Seidel converges. (3b) $D = 5$ Jacobi fails to converge, while Seidel converges. (3c) $D = 10$ Both converge, but Seidel is significantly faster. (3d & 3e) $D = 100$ & $D = 1000$ Both methods converge successfully. As D increases, both algorithms exhibit faster and more similar convergence rates.

spectral radius of the matrix $\mathbf{D}^{-1}\mathbf{R}$ are greater than one, for values $D \geq 10$ the spectral radius drops below 1 and both algorithms converge.

Run code for 10x10 with $a_{ii} = i$

We can run the code for this matrix by running the following two commands to the `main` executable

```
./main jacobi 1 2 3 4 5 6 7 8 9 10
./main seidel 1 2 3 4 5 6 7 8 9 10
```

which will produce a `jacobi_D0001.csv` and `seidel_D0001.csv` file containing the errors associated with solve each system.

to plot the error we use

```
source myenv/bin/activate
python plot_convergence.py
```

which will automatically read both files `*D0001.csv` and produce the plot `convergence_D0001.png`.

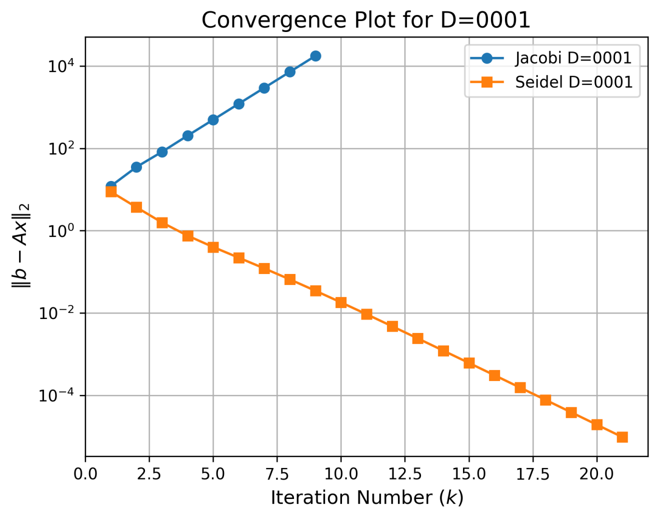


Figure 4: Convergence plot for the Gauss-Jacobi and Gauss-Seidel algorithm applied to the system $\mathbf{Ax} = \mathbf{b}$ where $\mathbf{A}$ is a matrix with all ones in the off-diagonal elements and $a_{ii} = i$ in the diagonal, the column vector $\mathbf{b}$ is such that each element $b_{ii} = i$. The algorithm converges for the seidel algorithm, but not for the jacobi algorithm

In Figure 4 we can see that the solving the system $\mathbf{Ax} = \mathbf{b}$ does not converge for the Gauss-Jacobi algorithm, but does converge for the Gauss-Seidel algorithm. If we look at the spectral radius of the matrix $\mathbf{D}^{-1}\mathbf{R}$ where $\mathbf{D}$ is strictly a diagonal matrix where the diagonal entries are $d_{ii} = i$.

this result makes sense because the spectral radius of $\mathbf{D}^{-1}\mathbf{R}$ is equal to 2.44, which is greater than 1.

found by computing in julia