AM213A Final Project Deliverables

Owen Morehead

March 12, 2022

General Guidelines

Under the `project` directory inside `part1/code` contains all the `fortran` and `matlab` code associated with this assignment.

(a) First, under the `svd` subdirectory, you will find code associated to **SVD for image compression**. The associated code files include:

- `Driver_svd.f90` driver program which uses `LAPACK` to compute the svd of an image as well as low-rank approximated images using k singular values.
- `utility.f90` which holds the variable, $dp = kind(0.d0)$ to specify double or single precision floating point arithmetic.
- `Makefile` which is the makefile for this program.

The makefile is setup to compile the program such that the command `make LinAl` will make the executable that can then be run with the command, `./LinAl`. This will print the desired information to the screen, i.e. the first 10 singular values, and the rest singular values of σ_k for $k = 20, 40, 80, 160, 320, 640, 1280, 2560, 3355$.
Each of the compressed images for the above specified values of k are stored in an ascii format data file upon output, as well as a data file that contains the error for all compressed images. These data are further used in `Matlab` to visualize the compressed data and plot the errors of each image compression compared to the original image data.

(b) The next section explores three iterative methods, the **Gauss-Jacobi**, **Gauss-Seidel**, and **Conjugate Gradient** algorithms used to solve the equation $\mathbf{Ax} = \mathbf{b}$. Under the `iters` subdirectory one will find all the code related to these algorithms. This includes:

- `Driver_iter.f90` driver program which calls the appropriate subroutines and prints all desired results to the screen, i.e. the solution vector from all algorithms.
- `LinAl.f90` program containing all the relevant subroutines.
- `utility.f90` which holds the variable, $dp = kind(0.d0)$ to specify double or single precision floating point arithmetic.
- `Makefile` which is the makefile for this program.

Compiling and running the code is the exact same as for the svd program above. The makefile is setup to compile the program such that the command `make LinAl` will make the executable that can then be run with the command, `./LinAl`. This will print the desired information to the screen. In addition, data files are saved which contain the error at each iteration for a value of D specified in the driver program. More on that in the below section. `Matlab` is again used to plot and compare the errors for each iterative algorithm. This matlab file is also found in this `svd` subdirectory.

All code executions are done using double precision floating point arithmetic. If one wants to change this do single precision, the step to take is in the `utility.f90` file, which is to simply change $dp = kind(0.d0)$ to $dp = kind(0.e0)$.

1 SVD

The **Singular Value Decomposition** (**SVD**) of a matrix is one of the most powerful non-trivial tools of Numerical Linear Algebra. It is used for a vast range of applications from **image compression** to signal processing, matrix approximation, Least Square fitting, and much more. In general, many big data analysis methods utilize the SVD. The full SVD is a factorization of a real or complex matrix $\mathbf{A}$ (general size $m \times n$), of the form:

$$\mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^* \tag{1}$$

where $\mathbf{U}$ and $\mathbf{V}$ are respectively $m \times m$ and $n \times n$ unitary matrices, ($U^*U = UU^* = UU^{-1} = I$) (orthogonal if $\mathbf{A}$ is real), and Σ is an $m \times n$ rectangular diagonal matrix with non-negative real numbers on the diagonal, known as the singular values, σ_i, ordered from largest σ_1 to smallest σ_m. The concept behind the SVD is related the fact that it is possible, from any matrix $\mathbf{A}$ of size $m \times n$, to define singular values σ_i and left and right singular vectors $\mathbf{u}_i$ and $\mathbf{v}_i$ such that,

$$\mathbf{A}\mathbf{v}_i = \sigma_i\mathbf{u}_i. \tag{2}$$

Geometrically speaking, $\mathbf{u}_i$ are the normalized $m \times 1$ vectors parallel to the principal axes of the hyperellipse formed by the image of the unit ball via application of $\mathbf{A}$. $\sigma_i > 0$ are the lengths of these principal axes, and $\mathbf{v}_i$ are the pre-images of $\mathbf{u}_i$ and are of size $n \times 1$. If $r = \text{rank}(\mathbf{A})$, then there will be r such principal axes, and therefore r non-negative singular values.

The SVD has a number of important and useful properties. Some of them include:

- The SVD of a real matrix has real (and therefore orthogonal) $\mathbf{U}$ and $\mathbf{V}$.
- The number of non-zero singular values is equal to the rank($\mathbf{A}$).
- The vectors $\mathbf{u}_i, \ldots, \mathbf{u}_r$ span the range of $\mathbf{A}$, while the vectors $\mathbf{v}_{r+1}, \ldots, \mathbf{v}_n$ span the nullspace of $\mathbf{A}$.
- $||\mathbf{A}||_2 = \sigma_1$ and $||\mathbf{A}||_F = \sqrt{\sigma_1^2 + \cdots + \sigma_r^2}$. The second result stems from the fact that the Frobenius norm of the product of $\mathbf{A}$ with a unitary matrix is equal to the Frobenius norm of $\mathbf{A}$.
- If $\mathbf{A}$ is a square matrix $m \times m$, then $|\det\mathbf{A}| = \sigma_1\sigma_2 \cdots \sigma_m$. This result stems from the fact that the determinant of a unitary matrix is ± 1.

Another interesting property to note is that,

$$\mathbf{A}^{-1} = (\mathbf{U}\Sigma\mathbf{V}^*)^{-1} = \mathbf{V}\Sigma^{-1}\mathbf{U}^* \tag{3}$$

with Σ^{-1} now being the matrix of the reciprocals of the same singular values. Because these values are now ordered in increasing order rather than descending, we can switch this by doing a permutation of the rows and columns on either side:

$$\mathbf{A}^{-1} = (\mathbf{V}\mathbf{P})(\mathbf{P}\Sigma^{-1}\mathbf{P})(\mathbf{P}\mathbf{U}^*) \tag{4}$$

where $\mathbf{P}$ is the reverse identity matrix satisfying $\mathbf{P}\mathbf{P} = \mathbf{I}$. This leads to both $\mathbf{V}\mathbf{P}$ and $\mathbf{P}\mathbf{U}^*$ being unitary matrices since $\mathbf{V}$ and $\mathbf{U}$ are. This all leads to the ordered singular values of $\mathbf{A}^{-1}$ now being $(\sigma_m^{-1}, \ldots, \sigma_1^{-1})$. From this we can further deduce the property that is,

$$\text{cond}(\mathbf{A}) = ||\mathbf{A}||_2||\mathbf{A}^{-1}||_2 = \frac{\sigma_1}{\sigma_m}. \tag{5}$$

In terms of the SVD, we also see that,

$$\begin{aligned}\mathbf{A}^*\mathbf{A} &= (\mathbf{U}\Sigma\mathbf{V}^*)^*(\mathbf{U}\Sigma\mathbf{V}^*) = \mathbf{V}\Sigma^*\mathbf{U}^*(\mathbf{U}\Sigma\mathbf{V}^*) \\ &= \mathbf{V}\Sigma^*\Sigma\mathbf{V}^* = \mathbf{V}\begin{bmatrix}\sigma_1^2 & & & \\ & \sigma_2^2 & & \\ & & \ddots & \\ & & & \sigma_n^2\end{bmatrix}\mathbf{V}^*.\end{aligned} \tag{6}$$

This is effectively a diagonalization of the matrix $\mathbf{A}^*\mathbf{A}$, which shows that the non-zero eigenvalues of $\mathbf{A}^*\mathbf{A}$ are the square of the non-zero singular values of $\mathbf{A}$.

If $\mathbf{A}$ is hermitian ($\mathbf{A} = \mathbf{A}^*$), then we know its eigenvalue/eigenvector decomposition is,

$$\mathbf{A} = \mathbf{Q}^*\mathbf{D}_\lambda\mathbf{Q} \tag{7}$$

where $\mathbf{Q}$ is the matrix whose column vectors are the eigenvectors of $\mathbf{A}$. Similarly to Equation 2, this implies that vector by vector,

$$\mathbf{A}\mathbf{q}_i = \lambda_i\mathbf{q}_i. \tag{8}$$

The primary difference between this and a singular value decomposition is the fact that some λ_i can be negative while σ_i must be positive. However, note we can write that

$$\lambda_i = \text{sign}(\lambda_i)\sigma_i\mathbf{q}_i = \sigma_i\mathbf{u}_i.$$

We can therefore use the $\mathbf{q}_i$ to find the left and right singular vectors, $\mathbf{u}_i = \text{sign}(\lambda_i)\mathbf{q}_i$, and $\mathbf{v}_i = \mathbf{q}_i$, and then create the matrices $\mathbf{U}$ and $\mathbf{V}$ associated with the SVD of $\mathbf{A}$. This ultimately shows that for a Hermitian matrix, we have both $\sigma_i = |\lambda_i|$, and the corresponding eigenvectors $\mathbf{v}_i$ are the right singular vectors. And for any decomposition, Σ can be multiplied by a permutation matrix to order the singular values if they are not already.

There are many algorithms for forming the singular value decomposition of a matrix. The basic premise of what is not advised when computing it numerically is to *not* form the product $\mathbf{A}^*\mathbf{A}$. Even though forming this matrix suggests a simple technique for finding the SVD of $\mathbf{A}$ (if $\mathbf{A}$ is not ill-conditioned), we must recall that,

$$\text{cond}(\mathbf{A}^*\mathbf{A}) = \text{cond}(\mathbf{A})^2. \tag{9}$$

So if $\mathbf{A}$ is already poorly conditioned, then $\mathbf{A}^*\mathbf{A}$ will be even more poorly conditioned. This matrix will therefore be very sensitive to changes or errors in the input, and much more error in the output results from small error in the input. Any operation on such a matrix will be prone to truncation errors of order one.

Using LAPACK's `dgesvd` for an Image Compression Application

In general, low-rank approximations are a very useful applicaitons of SVD. This allows one to approximate large matrices with smaller ones. Accurately being able to summarize much of the 'information' in a very large matrix into a smaller matrix is extremely helpful to any processing program, speeding up the algorithm immensely.
In essence, it can be proved that taking the $\nu < r$ largest singular values of $\mathbf{A}$ and replacing the rest with zero, and recomputing $\mathbf{A} = \mathbf{U}\Sigma'\mathbf{V}^*$ will give us the *best* $\nu-$ rank approximation to the matrix $\mathbf{A}$.
The idea stems from the following. By definition of the SVD, we know that

$$a_{ij} = \sum_{k,n} u_{ik}(\Sigma)_{kn}v^*_{jn} = \sum_{k=1}^{r} u_{ik}\sigma_k v^*_{jk} \tag{10}$$

assuming that $\text{rank}(\mathbf{A}) = r$. This can be rewritten in vector form as,

$$\mathbf{A} = \sum_{k=1}^{r} \sigma_k \mathbf{B}_k \tag{11}$$

where the matrix $\mathbf{B}_k$ has components $(\mathbf{B}_k)_{ij} = u_{ik}v^*_{jk}$. Thus, the matrix $\mathbf{B}_1$ is formed by the outer product of the first column vectors of $\mathbf{U}$ and $\mathbf{V}$,

$$\mathbf{B}_1 = \mathbf{u}_1\mathbf{v}_1^* \tag{12}$$

and similarly any other matrix $\mathbf{B}_k$.
Resultingly, since the singular values appearing in the sum from 1 to r gradually decrease with k, we find that it is very much possible to ignore the terms in the sum for which σ_k is very small. Meaning, we can truncate the sum at an order $\nu < r$, and still have a great approximation for $\mathbf{A}$.

Formally, we can define, for any $1 \leq \nu < r$, the approximate image defined by the matrix

$$\mathbf{A}_\nu = \sum_{k=1}^{\nu} \sigma_k \mathbf{u}_k \mathbf{v}_k^* \tag{13}$$

then

$$||\mathbf{A} - \mathbf{A}_\nu||_2 = \inf_{\mathbf{B},\text{rank}(\mathbf{B})\leq\nu} ||\mathbf{A} - \mathbf{B}||_2 = \sigma_{\nu+1} \tag{14}$$

and

$$||\mathbf{A} - \mathbf{A}_{nu}||_F = \inf_{\mathbf{B},\text{rank}(\mathbf{B})\leq\nu} ||\mathbf{A} - \mathbf{B}||_F = \sqrt{\sigma^2_{nu+1} + \cdots + \sigma^2_r}. \tag{15}$$

These equations imply that the norm of the residual $||\mathbf{A} - \mathbf{A}_\nu||$ has the smallest possible norm among all possible $||\mathbf{A} - \mathbf{B}||$ created using matrices $\mathbf{B}$ that have rank ν or less. In other words, $\mathbf{A}_\nu$ is the best possible matrix of rank ν that can be created to approximate $\mathbf{A}$ in both Euclidean and Frobenius norm sense. This is extremely useful in **image compression**.

Here we consider a black and white image which is really just an $m \times n$ matrix with entries a_{ij} measuring the grey-level (grey-scale) associated with the pixel location at position (x_i, y_i). m and n are usually in excess of a few thousand pixels in each direction, making the total amount of pixels extremely large. Doing an SVD of the image matrix $\mathbf{A}$ will allow us to reconstruct the image quite accurately using $\nu \approx \mathcal{O}(100)$. In other words, the image in matrix $\mathbf{A}$ can be compressed by saving *much* less than the total amount of vectors $\mathbf{u}_i$ and $\mathbf{v}_i$ associated with the original matrix.

The black and white image we want to compress is the file `dog_bw_data.dat` which is a cute and high-resolution picture of a dog. The data stored in this file is in double precision ascii format, with the values ranging from 0 (maps to black) to 255 (maps to white). The pixel size, measured as width $\times$ height of the original image is 5295×3355, which is transposed in linear algebra to conform the conventional row $\times$ column dimensions, giving a matrix that is $(m \times n) = (3355 \times 5295)$.

To perform the image compression via SVD, we call the `dgesvd` routine in the `LAPACK` (linear algebra package), written in `Fortran`. After obtaining the full singular value decomposition of $\mathbf{A}$ using `dgesvd` with all the necessary inputs to this function (documentation) we can perform a series of singular value approximations, $\mathbf{A}_{\sigma_k}$, of the original data $\mathbf{A}$.

We reconstruct eight separate compressed (low-rank approximated) images, namely:

$$\mathbf{A}_{\sigma_k} \quad \text{for} \quad k = 20, 40, 80, 160, 320, 640, 1280, 2560$$

as well as the full rank approximation with $k = \text{rank}(\mathbf{A}) = 3355$. Each k-SVD approximation is defined by,

$$\mathbf{A}_{\sigma_k} = \mathbf{U}\Sigma_{\sigma_k}\mathbf{V}^T \tag{16}$$

where each $\mathbf{A}_{\sigma_k} \in \mathbb{R}^{m\times n}$ is a newly reconstructed compressed data from the original data $\mathbf{A}$. Each singular value matrix $\Sigma_{\alpha_k} \in \mathbb{R}^{m\times n}$ is a reduced matrix of the full diagonal matrix Σ cntaining only up to the first k largest singular values with $k/leqr = \text{rank}(\mathbf{A}) = 3355$, given as:

$$\Sigma_{\alpha_k} = \begin{bmatrix} \sigma_1 & & & & & & \\ & \sigma_2 & & & & & \\ & & \ddots & & & & \\ & & & \sigma_k & & & \\ & & & & 0 & & \\ & & & & & \ddots & \\ & & & & & & 0 \end{bmatrix} \tag{17}$$

Although Equation 16 is one method of computing the compressed images, we can utilize a much more efficient formula. First, we note that `dgesvd` returns the SVD with $\mathbf{U}$ and $\mathbf{V}^T$ as full matrices, and the singular values gathered into a 1-d array. Utilizing Equation 16 would entail rearranging the singular values into a full matrix of the correct shape, and then performing matrix-matrix products. However, this is pretty expensive due to the size of the image.

We can use to our leverage the fact that the Σ_{α_k} matrices are diagonal to formulate a much cheaper process. By definition of the SVD, this leads us to what has already been noted above, Equation 13, where ν is the desired rank, σ_k is the kth singular value, and $\mathbf{u}$ and $\mathbf{v}$ are the kth singular vectors. Using this summation instead of 2 matrix products will be much cheaper overall. We can also jump start the summation for any higher rank approximation once we have calculated lower rank approximations (since the first part of the sum will be the same).

Each of the eight $\mathbf{A}_{\sigma_k}$ ascii data files are saved with the names, `Image_appn_1xxxxx.dat` where the file index "1xxxxx" reflects the number of singular values used in each calculation, e.g., `100320` for $\Sigma_{\alpha_{320}}$.

With these saved compressed image files, we implement a plotting routine to visualized the compressed data and how they compare to the original data. Double-precision data are converted to an unsigned 8-bit data type to properly display the results without any overflow errors. In addition, the corresponding errors are calculated using the Frobenius norm,

$$E_k = \frac{||\mathbf{A} - \mathbf{A}_{\sigma_k}||_F}{mn} \tag{18}$$

and are plotted as a function of the number of singular values k.

Numerical Results

Table 1 reports the first 10 largest singular values, from σ_1 to σ_{10}, as well as the singular values σ_k for k in $k_r = [20, 40, 80, 160, 320, 640, 1280, 2560, 3355]$:

k	σ_k	k	σ_k
1	663180.20	20	7528.02
2	85706.60	40	5489.12
3	62129.03	80	3948.78
4	34664.63	160	2668.22
5	31861.79	320	1515.87
6	21872.72	640	821.89
7	19628.44	1280	513.57
8	18434.94	2560	179.12
9	13693.82	3355	16.72
10	12815.21		

Table 1: Singular values for the given image data matrix $\mathbf{A}$.

As expected, $\sigma_1 > \sigma_2 > \cdots > \sigma_{m=3355}$. Next, Figure 1 shows what the compressed images look like using the k largest singular values for each k in the list of values in k_r, where $k = 3355$ corresponds to the original image. Alongside these results, Figure 2 shows the compressed images corresponding error as a function of how many k largest singular values were used, with the average Frobenius norm error being calculated from Equation 18. These results are to be expected. We see that as more singular values are used, the image becomes much more clear. This greyscale image can be reconstructed quite accurately using almost all these k largest singular values. The compressed image is of course the most blurry when less singular values are used, which corresponds to the compressed images using $k = 20$ largest singular values out of the values of k tested. We see that intermediate values of k seem to return quite similar looking images compared to the original. As k increases, more of the finer details in the image such as the tips of the dogs hairs become clear. Essentially more singular values are needed to make regions in the image more clear which are comprised from a broader range of pixel color values (greyscale values in this case). If a large region, e.g. the middle of the dogs face, is essentially comprised of the same color (white) with minimal texture and color changes, this region can most easily be described using a compressed image with a smaller amount of singular values.

The error results in Figure 2 agree with the notion that compressed images using an intermediate number of singular values return similar images that do not visually differ by much. For example, the images using $k = 320$ to $k = 2560$ singular values all look approximately the same, disregarding small details that indeed clear up as k continues to increase. We notice a steep drop off in the error between the original and compressed image for k greater than around 250. After this value, the error starts to slowly decrease towards zero as the number of singular values used tends towards the total amount in the original image, i.e. 3355. We can also confirm the full-rankness with $k = 3355$ from its error estimation value of 4.56×10^{-16}, which is indeed an order of machine accuracy for these double precision calculations. In addition, we also find that for $k \approx 900$, the error becomes lower than 1×10^{-3}. This is interpreted from Figure 2, since we only tested values $k = 640$ and $k = 1280$, at which the respective errors were 1.17×10^{-3}, and 7.11×10^{-4}. This tells us that somewhere between these values of k, the error decreases below 1×10^{-3}. And in regarding the values of k tested, we would say that at $k = 1280$, the error decreases below 1×10^{-3}.

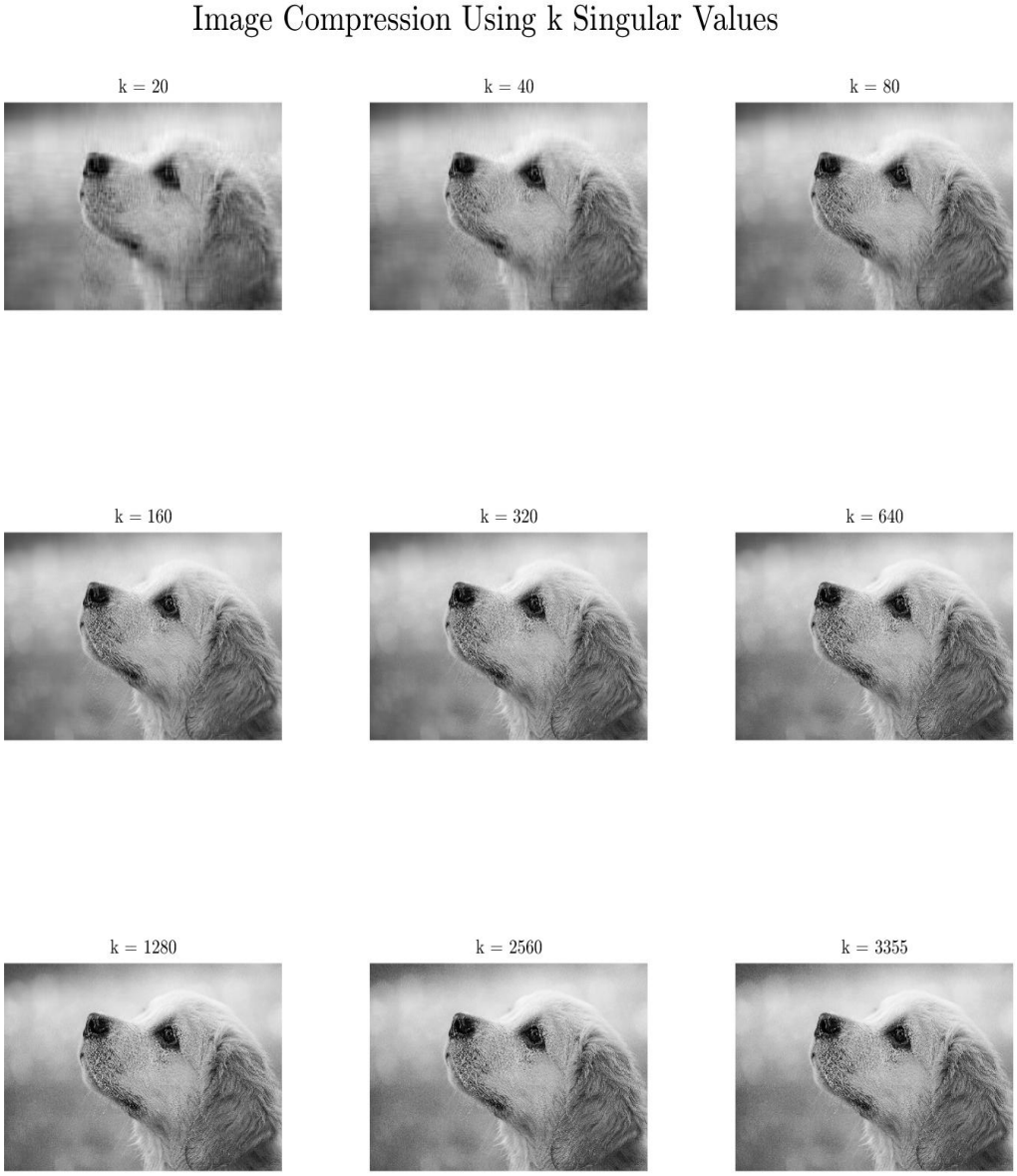

Figure 1: All eight compressed images along with the full-rank original image. Each subfigure caption shows how many k singular values were used to compute the compressed image, where $k = 3355$ corresponds to the full-rank original image.

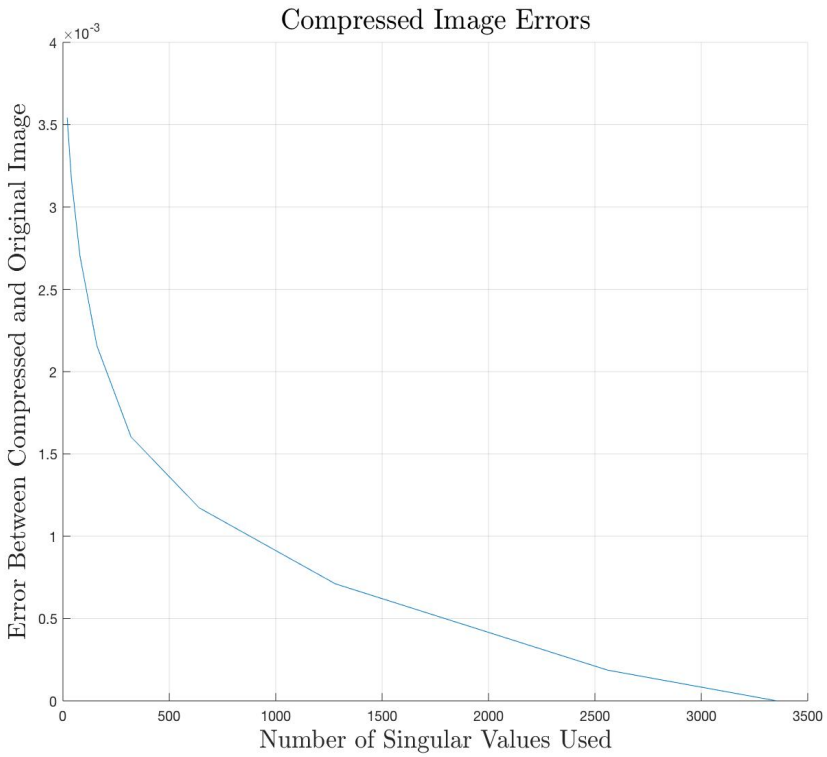

Figure 2: Errors between compressed images and the original image (calculated using Equation 18) as a function of the number of singular values used.

2 Iterative Methods for the Solution of Linear Systems

2.1 Gauss-Jacobi and Gauss-Seidel Algorithms

In computational mathematics, an iterative method is a mathematical procedure that uses an initial guess to generate a sequence of improving approximate solutions, which are derived from the solution at the previous iteration. The basic algorithms to solve a linear system take $\mathcal{O}(m^3)$ floating point operations to find $\mathbf{x}$. This becomes very expensive for large matrices. Iterative methods only use N iterations of an $\mathcal{O}(m^2)$ operation (e.g. a matrix multiplication), where $N \ll m$. Iterative methods are also much easier to operate in parallel compared to basic LU and Gaussian schemes.

The **Gauss-Jacobi** and **Gauss-Seidel** algorithms stem from the same idea. We begin first with a description of the simplest, Gauss-Jacobi.

2.1.1 Gauss-Jacobi

The general idea of the Gauss-Jacobi (GJ) algorithm is to write $\mathbf{A} = \mathbf{D} + \mathbf{R}$ where $\mathbf{D}$ is a diagonal matrix containing the diagonal elements of $\mathbf{A}$, and $\mathbf{R}$ is the rest, i.e. a matrix whose diagonal elements are 0. Then,

$$(\mathbf{D} + \mathbf{R})\mathbf{x} = \mathbf{b} \rightarrow \mathbf{D}\mathbf{x} = b - \mathbf{R}\mathbf{x} \rightarrow \mathbf{x} = \mathbf{D}^{-1}(\mathbf{b} - \mathbf{R}\mathbf{x}) \tag{19}$$

and since $\mathbf{D}$ is diagonal with elements a_{ii}, $D^{-1} = 1/a_{ii}$ is known and finding it entails no expensive computations. Viewing this as an iterative algorithm, we start with a guess, $\mathbf{x}^{(0)}$, and apply,

$$\mathbf{x}^{(k+1)} = \mathbf{D}^{-1}(\mathbf{b} - \mathbf{R}\mathbf{x}^{(k)}). \tag{20}$$

If this algorithm converges, then the limit satisfies $\mathbf{x} = \mathbf{D}^{-1}(\mathbf{b} - \mathbf{R}\mathbf{x})$, which is equivalent to $\mathbf{A}\mathbf{x} = \mathbf{b}$, and so $\mathbf{x}$ is precisely the solution we are looking for. The operation count for each step is very low, we can write it out as:

$$x_i^{(k+1)} = \frac{1}{a_{ii}}(b_i - \sum_{j=1}^{m} r_{ij} x_j^{(k)}) \tag{21}$$

which contains exactly m multiplications/divisions per value of i (noting that the diagonals r_{ij} with $i = j$ are equal to 0). Hence there are a total of m^2 multiplications/divisions per iteration. If the algorithm converges in a number of steps $N \ll m$, then we can use this method to solve $\mathbf{A}\mathbf{x} = \mathbf{b}$ in $\mathcal{O}(m^2)$ steps. Updating each component can be done independently of the others, so this is intrinsically parallelizable.

Most importantly, to address when this method converges, we need to look at the **spectral radius** of a matrix $\mathbf{M}$. We recall that the spectral radius of any square matrix $\mathbf{M}$ is $\rho(\mathbf{M}) = \max|\lambda_i|$. Associated with the spectral radius are the following important properties:

- If $\mathbf{M}$ is diagonalizable, and $\rho(\mathbf{M}) < 1$, then $\lim_{k\to\infty} \mathbf{M}^k\mathbf{x} = 0$ for any vector $\mathbf{x}$. The proof stems from the fact that if $\mathbf{M}$ is diagonalizable, then we can write $\mathbf{x} = \sum_i \alpha_i \mathbf{v}_i$ where the $\mathbf{v}_i$ are the eigenvectors of $\mathbf{M}$ with eigenvalues λ_i.

- If $\rho(\mathbf{M}) < 1$, then the quantity $(\mathbf{I} - \mathbf{M})^{-1} = \sum_{k=0}^{\infty} \mathbf{M}^k$ exists. The proof stems from the fact that for any vector $\mathbf{x}$, we have that $(\mathbf{I} - \mathbf{M})\mathbf{B}_k\mathbf{x} = \mathbf{x} - \mathbf{M}^{k+1}\mathbf{x}$ where the partial sum $\mathbf{B}_k = 1 + \mathbf{M} + \cdots + \mathbf{M}^k$. Using the first property tells us that $(\mathbf{I} - \mathbf{M}) \lim_{k\to\infty} \mathbf{B}_k = \mathbf{I}$, which proves that the inverse, $(\mathbf{I} - \mathbf{M})^{-1} = \lim_{k\to\infty} \mathbf{B}_k$ as required.

Both these properties can be used to construct a necessary condition for convergence of the iterative algorithm:

$$\mathbf{x}^{(k+1)} = \mathbf{T}\mathbf{x}^{(k)} + \mathbf{c} \tag{22}$$

where $\mathbf{T}$ is a square matrix and $\mathbf{c}$ is a constant vector. This equation converges provided $\rho(\mathbf{T}) < 1$. With $\mathbf{T} = -\mathbf{D}^{-1}\mathbf{R}$ and $\mathbf{c} = \mathbf{D}^{-1}\mathbf{b}$, this theorem thus implies that

$$\mathbf{x}^{(k+1)} = \mathbf{D}^{-1}(\mathbf{b} - \mathbf{R}\mathbf{x}^{(k)}) \tag{23}$$

converges provided $\rho(\mathbf{D}^{-1}\mathbf{R}) < 1$. The limit of this quantity is the desired solution of the original equation, $(\mathbf{D} + \mathbf{R})\mathbf{x} = \mathbf{b}$, which is the **Gauss-Jacobi** algorithm in its entirety.
Proving that $\rho(\mathbf{D}^{-1}\mathbf{R}) < 1$ requires calculating the largest eigenvalue of $\mathbf{D}^{-1}\mathbf{R}$, which can be done for example using a basic power iteration on a single vector. However this can be an expensive task, so it is standard practice to simply apply the GJ algorithm and see whether or not it converges.

As a general principle, however, matrices whose diagonal coefficients are large compared with the non-diagonal ones are well-behaved from the viewpoint of Gauss-Jacobi convergence. If $\mathbf{D}$ has large eigenvalues, then $\mathbf{D}^{-1}$ will have small eigenvalues, and so one may hope that $\mathbf{D}^{-1}\mathbf{R}$ will also have small eigenvalues. Matrices that satisfy $\rho(\mathbf{D}^{-1}\mathbf{R}) < 1$ are therefore called **diagonally dominant**. They appear frequently in problems that arise from the solution of PDEs. In addition, the matrix $\mathbf{A}$ obtained in this case is usually banded (sparse matrix whose non-zero entries are confined to a diagonal band), so calculating $\mathbf{R}\mathbf{x}$ is $\mathcal{O}(m)$ instead of $\mathcal{O}(m^2)$, making the algorithm even more efficient.

In terms of implementing this algorithm, it is basically as simple as Equation 23 shows. We first create D and R from A and set x equal to an initial guess (commonly $\mathbf{x} = 0$). Then, for each iteration, we calculate $\mathbf{D}^{-1}(\mathbf{b} - \mathbf{R}\mathbf{x}^{(k)})$, and replace the old value of x with this new value. This routine returns the solution x once the error computed as $||\mathbf{b} - \mathbf{A}\mathbf{x}||_2$ is smaller than the required accuracy, in this case we choose 1×10^{-5}. This error measures how well the current solution iterate satisfies the linear system of interest. Calculating the error as say, the difference between iterates, $\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}$, is also interesting but more-so for detecting stagnation than testing for convergence. Although this algorithm can be easily parallelizable, one of its main disadvantages is that its convergence is generally quite slow. A 'modification' of this algorithm to enhance the convergence rate is what we call the Gauss-Seidel algorithm.

2.1.2 Gauss-Seidel

The Gauss-Seidel (GS) algorithm is very similar to the Gauss-Jacobi algorithm but directly over-writes the vector $\mathbf{x}$ at each iteration. The advantage is that it effectively uses the updated values of the coefficients of $\mathbf{x}$ as they come to calculate the next coefficients. Consequently, this algorithm does not require storing 2 vectors ($\mathbf{x}$ and $\mathbf{x} = \mathbf{D}^{-1}(\mathbf{b} - \mathbf{R}\mathbf{x}^{(k)})$ at the new iteration). In essence, instead of using Equation 21 to compute $x_i^{(k+1)}$ as for the GJ algorithm, this algorithm now uses:

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{m} a_{ij}x_j^{(k)}\right) \tag{24}$$

noting that similarly the diagonals of r, $r_{ii} = 0$. In a compact matrix form, the GS algorithm is written as:

$$\mathbf{x}^{(k+1)} = \mathbf{D}^{-1}\big(\mathbf{b} - \mathbf{L}\mathbf{x}^{(k+1)} - \mathbf{U}\mathbf{x}^{(k)}\big) = (\mathbf{D} + \mathbf{L})^{-1}(\mathbf{b} - \mathbf{U}\mathbf{x}^{(k)}) \tag{25}$$

where $\mathbf{L}$ and $\mathbf{U}$ are respectively the lower and upper triangular matrices excluding the diagonal entries, $\mathbf{D}$, resulting in $\mathbf{A} = \mathbf{L} + \mathbf{D} + \mathbf{U}$. Thus the algorithm goes as follows: We first create $\mathbf{D}$, $\mathbf{L}$ and

$\mathbf{U}$ from $\mathbf{A}$, and set $\mathbf{x}$ equal to our initial guess (again, commonly $\mathbf{x} = 0$). Then we simply calculate and update $\mathbf{x}$ using Equation 25 at each iteration. An important task here is now in calculating the inverse of $(\mathbf{D} + \mathbf{L})$. Noting that it is a lower triangular matrix, we can simply utilize a forward substitution step as used to solve systems through LU or Cholesky decomposition for example. Adapting a forward substitution step as such will be ideal in this algorithm. Recall, the forward substitution step is essentially the process to solve any general lower triangular system, $\mathbf{Ly} = \mathbf{b}$, which gives the vector $\mathbf{y}$. The lower triangular system in this algorithm is, $(\mathbf{D} + \mathbf{L})\mathbf{y} = (\mathbf{b} - \mathbf{Ux})$, which will give the vector $\mathbf{y} = (\mathbf{D} + \mathbf{L})^{-1}(\mathbf{b} - \mathbf{Ux})$ as desired.

As with the GJ algorithm, we also terminate the algorithm once the error, $||\mathbf{b} - \mathbf{Ax}||_2$ is smaller than our desired tolerance, ultimately returning our solution vector $\mathbf{x}$. the GS algorithm generally has much better convergence properties than the GJ algorithm. It has been shown that *if the Gauss-Jacobi algorithm converges*, then the Gauss-Seidel algorithm converges *faster*. Furthermore, the GS algorithm sometimes converges for matrices for which GJ does not. For example, GS has been shown to converge for any symmetric, positive definite matrix, which is not the case for GJ. For sequential algorithms, one should therefore always prefer the GS algorithm. On the other hand, the GS algorithm is now no longer perfectly parallelizable, and so the gain in the convergence rate may not out weight the loss in scalability on parallel architectures. Nevertheless, the fact that it always converges faster than GJ for sequential algorithms makes it useful in practice.

2.2 Conjugate Gradient Algorithm and Minimizing Quadratic Forms

In the ideal case, i.e. when the known $m \times m$ matrix $\mathbf{A}$ is:

- real ($\mathbf{A} \in \mathbb{R}^{m \times m}$)
- positive definite ($\mathbf{x}^T\mathbf{Ax} > 0 \quad \forall$ non-zero vectors $\mathbf{x} \in \mathbb{R}^m$)
- and symmetric ($\mathbf{A}^T = \mathbf{A}$)

there exists a very powerful iterative method for solving $\mathbf{Ax} = \mathbf{b}$, for which convergence is not only guaranteed, but for which it is guaranteed to take **at most m** steps. Otherwise said, this iterative method is *at its worst*, an $\mathcal{O}(m^3)$ method, which is the same as direct methods. At its best, it can find solutions as expected in $N \ll m$ steps, therefore providing a much faster way of finding the solution $\mathbf{x}$. This is known as the **conjugate gradient method**.

A property directly related to such iterative methods is the fact that when $\mathbf{A}$ is a real, positive definite, and symmetric matrix, then the solution $\mathbf{x}$ of the linear system $\mathbf{Ax} = \mathbf{b}$ is *also* the solution of a minimizaiton problem for a quadratic form. Otherwise said, the solution to $\nabla f = 0$ is also the solution to $\mathbf{Ax} = \mathbf{b}$. It is this important property that lets us use iterative methods for minimizing quadratic forms to also solve the linear system $\mathbf{Ax} = \mathbf{b}$.

An iterative method for minimizing a quadratic form starts at a given point $\mathbf{x}^{(0)}$, and applies an algorithm of the kind $\mathbf{x}^{(k+1)} = g(\mathbf{x}^{(x)})$ that gradually decreases the distance between $\mathbf{x}^{(k)}$ and the minimum. Since the quadratic form is convex (shaped like a cup with a minimum), a simple way to proceed is to start the algorithm in a particular direction, and go the minimum of $f(\mathbf{x})$ *along* that direction, then choose another direction, and minimize $f(\mathbf{x})$ along that new direction, and continue in this process. Minimizing $f(\mathbf{x})$ along a particular direction can be done analytically for a quadratic form. Suppose the selected direction vector at step k is $\mathbf{p}^{(k)}$, starting from vector $\mathbf{x}^{(k)}$. Then the next iterate $\mathbf{x}^{(k+1)}$ will satisfy,

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{p}^{(k)} \tag{26}$$

with the key being to choose the scalar α_k so that $\mathbf{x}^{(k+1)}$ minimizes $f(\mathbf{x})$) along that line. Evaluating α_k starts with,

$$f(\mathbf{x}^{(k+1)}) = \frac{1}{2}\mathbf{x}^{(k+1)T}\mathbf{A}\mathbf{x}^{(k+1)} - \mathbf{x}^{(k+1)T}\mathbf{b} \tag{27}$$

in which, skipping intermediate steps, we set $df(\mathbf{x}^{(k+1)})/d\alpha_k = 0$ to get,

$$\mathbf{p}^{(k)T}\mathbf{A}\mathbf{x}^{(k)} + \alpha_k\mathbf{p}^{(k)T}\mathbf{A}\mathbf{p}^{(k)} - \mathbf{p}^{(k)T}\mathbf{b} = 0 \tag{28}$$

where we used the fact that $\mathbf{A}$ is symmetric to show that $\mathbf{p}^{(k)T}\mathbf{A}\mathbf{x}^{(k)} = \mathbf{x}^{(k)T}\mathbf{A}\mathbf{p}^{(k)}$. This equation has the solution:

$$\alpha_k = \frac{\mathbf{p}^{(k)T}\mathbf{r}^{(k)}}{\mathbf{p}^{(k)T}\mathbf{A}\mathbf{p}^{(k)}} \tag{29}$$

where the residual at step k is $\mathbf{r}^{(k)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(k)}$. This tells us that as long as we have picked a direction $\mathbf{A}\mathbf{p}^{(k)}$ at step k, we can find the minimum of $f(\mathbf{x})$ along this direction analytically. What remains is how to choose the directions $\mathbf{p}^{(k)}$. And this is what differs between many of the similar iterative algorithms of this type. For example, **gradient descent** uses the direction of **steepest descent**, i.e. go toward minus the gradient of f at $\mathbf{x}^{(k)}$ (the gradient always points upward and we want to go downward towards the minimum of f), which we can write as,

$$\mathbf{p}^{(k)} = -\nabla f(\mathbf{x}^{(k)}) = -(\mathbf{A}\mathbf{x}^{(k)} - \mathbf{b}) = \mathbf{r}^{(k)}. \tag{30}$$

However, the problem with gradient descent is that the algorithm can revisit a direction many times. Avoiding this phenomena is the principle behind the **conjugate gradient method**. This algorithm converges in at most m steps by making sure that directions are never re-visited.

2.2.1 Conjugate Gradient Method

First starting with the notion of **conjugate directions**, not that two non-zero vectors $\mathbf{u}$ and $\mathbf{v}$ are considered **A-conjugate** if,

$$\mathbf{u}^T\mathbf{A}\mathbf{v} = 0. \tag{31}$$

In essence this identifies pairs of vectors where one is orthogonal to the image of the other after application of $\mathbf{A}$. Note that if $\mathbf{A}$ is positive definite we cal also define a new norm,

$$||\mathbf{u}||_A = \sqrt{\mathbf{u}^T\mathbf{A}\mathbf{u}}. \tag{32}$$

Furthermore, a set of vectors $\{\mathbf{p}_i\}$ forms a **conjugate set** with respect to the matrix $\mathbf{A}$ provided,

$$\mathbf{p}_i^T\mathbf{A}\mathbf{p}_j = 0 \qquad \forall \quad i \neq j. \tag{33}$$

We can conclude that *any* real symmetric matrix $\mathbf{A}$ of size $m \times m$ has a conjugate set $\{\mathbf{p}_i\}_{i=0,\ldots,m-1}$ since the eigenvectors form a conjugate set. This set forms a basis for $\mathbb{R}^m$. However, finding this set is already a $\mathcal{O}(m^3)$ task, so it is not any less expensive than solving $\mathbf{A}\mathbf{x} = \mathbf{b}$ directly. However, iteratively constructing the vectors $\mathbf{p}_0$, $\mathbf{p}_1$, etc so that $\mathbf{p}^{(0)} = \mathbf{p}_0$ is our starting vector, $\mathbf{p}^{(1)} = \mathbf{p}_1$ is the next iterate, and so forth will be a much cheaper process. At each iteration let,

$$\mathbf{x}^{(k+1)} = \sum_{i=0}^{k} \alpha_k\mathbf{p}^{(k)} = \mathbf{x}^{(k)} + \alpha_k\mathbf{p}^{(k)} \tag{34}$$

such that we are effectively constructing successive approximations to $\mathbf{x}$, which we know must converge to $\mathbf{x}$ in exactly a finite number of iterations, since $\mathbf{x}^{(m)} = \mathbf{x}$. Furthermore, if the coefficients α_k decay very rapidly with k, then $\mathbf{x}^{(k)}$ may tend to the true solution $\mathbf{x}$ much faster, using only $N \ll m$ terms in this sum, instead of all m of them. If this is the case, we only need to compute N of the conjugate directions, and the operation count will be much smaller than $\mathcal{O}(m^3)$. The core of this problem, however, is to create an algorithm to find $\mathbf{p}^{(k+1)}$ given knowledge of $\mathbf{p}^{(k)}$. To do this, we utilize what we already know from the gradient descent algorithm, but modify it slightly to make sure the directions $\mathbf{p}^{(k)}$ that are iteratively created are conjugate to one another. Skipping over the specific proof details, we are left with the encompassing theorem for the conjugate gradient algorithm, which at every iteration, we have the following properties:

1. $\mathbf{r}^{(k)T}\mathbf{p}^{(i)} = 0$ for any $i < k$, meaning the residuals are orthogonal to the previous directions.
2. $\mathbf{r}^{(k)T}\mathbf{r}^{(i)} = 0$ for any $i < k$, meaning the residuals are orthogonal to eachother.
3. $\mathbf{p}^{(k)T}\mathbf{A}\mathbf{p}^{(i)} = 0$ for any $i < k$, meaning the directions form a conjugate set.

Theorem: The conjugate gradient algorithm, starting from $\mathbf{x}^{(0)} = 0$, $\mathbf{r}^{(0)} = \mathbf{p}^{(0)} = \mathbf{b}$, and applying the necessary iterations described below, converges to the true solution in at most m iterations. The residuals are orthogonal to one another with $\mathbf{r}^{(k)T}\mathbf{r}^{(i)} = 0$ for any $i < k$, and the directions are conjugate to one another with $\mathbf{p}^{(k)T}\mathbf{A}\mathbf{p}^{(i)} = 0$ for any $i < k$. Additionally, if we consider the error $\mathbf{e}^{(k)} = \mathbf{x} - \mathbf{x}^{(k)}$, then

$$||\mathbf{e}^{(k)}||_A = \inf_{\mathbf{u} \in \mathcal{K}^{(k)}} ||\mathbf{x} - \mathbf{u}||_A \tag{35}$$

where the A-norm is defined by Equation 32, and

$$\mathcal{K}^{(k)} = \langle \mathbf{r}^{(0)}, \mathbf{r}^{(1)}, \ldots, \mathbf{r}^{(k-1)} \rangle = \langle \mathbf{b}, \mathbf{A}\mathbf{b}, \ldots, \mathbf{A}^k\mathbf{b} \rangle. \tag{36}$$

In other words, $\mathbf{x}^{(k)}$ is the best possible approximation of the true solution $\mathbf{x}$ that lives in the subspace $\mathcal{K}^{(k)}$, at least when that distance is measured with the A-norm. The implications of this theorem are quire profound. For example, it can be shown that,

$$||\mathbf{e}^{(k+1)}||_A \leq ||\mathbf{e}^{(k)}||_A \tag{37}$$

since we are expanding the space over which $||\mathbf{x} - \mathbf{u}||_A$ is minimized at each iteration. In addition, regarding the rate of convergence of the algorithm, it can be shown that,

$$||\mathbf{e}^{(k)}||_A \leq \frac{2}{\left(\frac{\sqrt{\kappa}+1}{\sqrt{\kappa}-1}\right)^k + \left(\frac{\sqrt{\kappa}+1}{\sqrt{\kappa}-1}\right)^{-k}} ||\mathbf{e}^{(0)}||_A \tag{38}$$

where $\kappa = \text{cond}(\mathbf{A}) = \frac{\sigma_1}{\sigma_m}$. Since κ is usually a large to very large value, Equation 28 is often written more simply as,

$$||\mathbf{e}^{(k)}||_A \leq 2\left(\frac{\sqrt{\kappa}+1}{\sqrt{\kappa}-1}\right)^k ||\mathbf{e}^{(0)}||_A \tag{39}$$

in which we see the rate of convergence depends on how small $\frac{\sqrt{\kappa}+1}{\sqrt{\kappa}-1}$ is. if κ is too large, then this value is close to 1, and the convergence rate will be slow. On the other hand, if κ is close to 1, then the convergence rate can be very fast.

In summary, the **basic conjugate gradient algorithm** is as follows:

- First set $\mathbf{x} = \mathbf{x}_0$, $\mathbf{r} = \mathbf{b} - \mathbf{A}\mathbf{x}$, and $\mathbf{p} = \mathbf{r}$. Note the algorithm can start at a value $\mathbf{x}_0$ that is different from 0, which is useful if we know a good approximation to the solution.
- while $||\mathbf{r}||$ > desired accuracy, we:

  calculate and store $\mathbf{y} = \mathbf{A}\mathbf{p}$, $\alpha = \frac{\mathbf{p}^T\mathbf{r}}{\mathbf{p}^T\mathbf{y}}$, set $\mathbf{x} = \mathbf{x} + \alpha\mathbf{p}$, and $\mathbf{r} = \mathbf{r} - \alpha\mathbf{y}$. Note that forming the variable $\mathbf{y} = \mathbf{A}\mathbf{p}$ is the most expensive part, so it is initially formed and stored once.
- Next we re-calculate $||\mathbf{r}||$, calculate $\beta = -\frac{\mathbf{r}^T\mathbf{y}}{\mathbf{p}^T\mathbf{y}}$, and lastly calculate $\mathbf{p} = \mathbf{r} + \beta\mathbf{p}$.

A **smarter version of this algorithm exists**, which saves some time in the computation of α and β. This algorithm is as follows:

- First set $\mathbf{x} = \mathbf{x}_0$, $\mathbf{r} = \mathbf{b} - \mathbf{A}\mathbf{x}$, and $\mathbf{p} = \mathbf{r}$, and call $E = ||\mathbf{r}||^2$
- while $\sqrt{E}$ > desired accuracy, we:

  calculate and store $\mathbf{y} = \mathbf{A}\mathbf{p}$, $\alpha = \frac{E}{\mathbf{p}^T\mathbf{y}}$, set $\mathbf{x} = \mathbf{x} + \alpha\mathbf{p}$, and $\mathbf{r} = \mathbf{r} - \alpha\mathbf{y}$.
- Next we calculate $E_{\text{new}} = ||\mathbf{r}||^2$, calculate $\beta = \frac{E_{\text{new}}}{E}$, and lastly calculate $\mathbf{p} = \mathbf{r} + \beta\mathbf{p}$, and set $E = E_{\text{new}}$.

We can indeed prove that the smart conjugate gradient algorithm is equivalent to the basic conjugate gradient algorithm. It will suffice to show that, at iteration k, $\mathbf{r}^{(k)T}\mathbf{p}^{(k)} = \mathbf{r}^{(k)T}\mathbf{r}^{(k)}$, and that $\beta_k = \frac{\mathbf{r}^{(k+1)T}\mathbf{r}^{(k+1)}}{\mathbf{r}^{(k)T}\mathbf{r}^{(k)}}$ which can be done by induction as follows:

At the first step, we pick,

$$\mathbf{r} = \mathbf{b} - \mathbf{A}\mathbf{x} \quad \text{and} \quad \mathbf{p} = \mathbf{r}$$

and therefore

$$\mathbf{r}^{(0)T}\mathbf{p}^{(0)} = \mathbf{r}^{(0)T}\mathbf{r}^{(0)}.$$

We also have defined β such that:

$$\beta_k = -\frac{\mathbf{r}^{(k+1)T}\mathbf{A}\mathbf{p}^{(k)}}{\mathbf{p}^{(k)T}\mathbf{A}\mathbf{p}^{(k)}}$$

so at this first iteration,

$$\beta_0 = -\frac{\mathbf{r}^{(1)T}\mathbf{A}\mathbf{p}^{(0)}}{\mathbf{p}^{(0)T}\mathbf{A}\mathbf{p}^{(0)}} = -\frac{\mathbf{r}^{(1)T}\mathbf{A}\mathbf{r}^{(0)}}{\mathbf{p}^{(0)T}\mathbf{A}\mathbf{r}^{(0)}} = -\frac{\mathbf{r}^{(1)T}\mathbf{A}(\mathbf{r}^{(1)} + \alpha\mathbf{A}\mathbf{p}^{(0)})}{\mathbf{r}^{(0)T}\mathbf{A}\mathbf{r}^{(0)}}$$

where we used the property relating $\mathbf{r}^{(k+1)} = \mathbf{r}^{(k)} - \alpha_k\mathbf{A}\mathbf{p}^{(k)}$. And after using the property 3 above describing the orthogonality between vectors at different iterations, we end up with,

$$\beta_0 = \frac{\mathbf{r}^{(1)T}\mathbf{r}^{(1)}}{\mathbf{r}^{(0)T}\mathbf{r}^{(0)}}$$

which is of the desired form, $\beta_k = \frac{\mathbf{r}^{(k+1)T}\mathbf{r}^{(k+1)}}{\mathbf{r}^{(k)T}\mathbf{r}^{(k)}}$.
Now, to finish the induction proof, we can generalize this for any iteration as follows:
At any further iteration k, we will have that,

$$\mathbf{r}^{(k)T}\mathbf{p}^{(k)} = \mathbf{r}^{(k)T}(\mathbf{r}^{(k)} + \beta\mathbf{p}^{(k-1)}) \tag{40}$$

$$= \mathbf{r}^{(k)T}\mathbf{r}^{(k)} + \beta\mathbf{r}^{(k)T}\mathbf{p}^{(k-1)} \tag{41}$$

$$\mathbf{r}^{(k)T}\mathbf{p}^{(k)} = \mathbf{r}^{(k)T}\mathbf{r}^{(k)} \qquad [\mathbf{r}^{(k)T}\mathbf{p}^{(i)} = 0 \quad \forall \quad i < k] \tag{42}$$

and furthermore,

$$\beta_k = -\frac{\mathbf{r}^{(k+1)T}\mathbf{A}\mathbf{p}^{(k)}}{\mathbf{p}^{(k)T}\mathbf{A}\mathbf{p}^{(k)}} \tag{43}$$

$$= -\frac{\mathbf{r}^{(k+1)T}\mathbf{A}(\mathbf{r}^{(k+1)} + \beta\mathbf{p}^{(k)})}{(\mathbf{r}^{(k)} + \beta\mathbf{p}^{(k-1)})^T\mathbf{A}(\mathbf{r}^{(k)} + \beta\mathbf{p}^{(k-1)})} \tag{44}$$

$$= -\frac{\mathbf{r}^{(k+1)T}\mathbf{A}\mathbf{r}^{(k+1)}}{\mathbf{r}^{(k)T}\mathbf{A}\mathbf{r}^{(k)} + \mathbf{r}^{(k)T}\mathbf{A}\beta\mathbf{p}^{(k-1)} + \beta\mathbf{p}^{(k-1)T}\mathbf{A}\mathbf{r}^{(k)} + \beta\mathbf{p}^{(k-1)T}\mathbf{A}\beta\mathbf{p}^{(k-1)}} \tag{45}$$

where the top simplification was made due to orthogonality condition number 1, and below we can now see that orthogonality conditions can be applied to the last three terms in which they will all equal zero. Thus we will be left with,

$$\beta_k = \frac{\mathbf{r}^{(k+1)T}\mathbf{r}^{(k+1)}}{\mathbf{r}^{(k)T}\mathbf{r}^{(k)}}$$

as desired. □

2.3 Numerical Results for Iterative Methods

Let us recall the details of the problem to be solved. We are solving the equation $\mathbf{Ax} = \mathbf{b}$ for a given $m \times m$ matrix $\mathbf{A}$, and and $m-$long vector $\mathbf{b}$ using the Gauss-Jacobi (GJ), Gauss-Seidel (GS), and Conjugate Gradient (CG) methods described above. These algorithms return the solution $\mathbf{x}$ once the error $||\mathbf{b} - \mathbf{Ax}||_2$ is smaller than the required accuracy, 1×10^{-5}. We first consider the 10×10 matrix $\mathbf{A}$ to be full of ones, except on the diagonal, where $a_{ii} = D$. We also consider the vector $\mathbf{b}$ such that $b_i = i$ for $i = 1, \ldots, m$.

Table 2 compares the number total number of iterations until complete convergence between the 3 algorithms. All algorithms start at $x_0 = 0$. Note that the GJ algorithm did not converge for D = 2 and D = 5, which is represented by Na. We see that both the GJ and GS algorithms converge in fewer iterations as D increases, however the CG algorithm converges in a mere 2 iterations for all values of D. More discussion on this below.

	Total Iterations until Complete Convergence		
D	Gauss-Jacobi	Gauss-Seidel	Conjugate Gradient
2	Na	60	2
5	Na	18	2
10	138	11	2
100	7	5	2
1000	5	4	2

Table 2: Comparison of total number of iterations until complete convergence between the Gauss-Jacobi, Gauss-Seidel, and Conjugate Gradient algorithms for each value of D. All algorithms start at $x_0 = 0$.

We now focus the discussion the Gauss-Jacobi and Gauss-Seidel algorithms. For the values of D which both the GS and GJ algorithms converge ($D = 10, 100, 1000$), we see in Figures 3, 4, and 5 respectively, the error, $||\mathbf{b} - \mathbf{Ax}||_2$, as a function of the iteration number for both methods. For all 3 values of D, the GS method converges faster than the GJ method as theoretically predicted. In

addition, the GS method also converges for the values of $D = 2$ and $D = 5$ which the GJ method does not converge. Regarding the discussion and predictions in the above section, we can conclude the GJ method was not able to converge to the solution for $D = 2$ and $D = 5$ since $\rho(\mathbf{D}^{-1}\mathbf{R}) > 1$ for these values of D, i.e., the largest eigenvalue of $\mathbf{D}^{-1}\mathbf{R}$ is greater than one.

On the other hand, $\rho(\mathbf{D}^{-1}\mathbf{R}) < 1$ for $D = 10, 100, 1000$, and thus the algorithm converges. Since by design, $\mathbf{D}^{-1}\mathbf{R}$ will be a matrix with all zeros on the diagonal, and values of $1/D$ everywhere else, we can easily verify by the Gershgorin theorem that the largest eigenvalue will be bounded by $(m-1)(|1/D|)$, which is the sum of the absolute values of the non-diagonal entries in the i-th row (or j-th column). Thus for a value of say $D = 10$, the largest eigenvalue will be bounded by $9(1/10) = 9/10$, which is indeed less than 1. Another way of confirm this is to use a smaller 5×5 matrix $\mathbf{A}$. Now we see that the GJ algorithm converges for $D = 5$ ($\lambda_{\max}$ is bounded by $4(1/5)$ meaning it must be less than 1). We cannot confirm using the Gershgorin theorem precisely what the largest eigenvalue will be, however it can be computed that for $D = 2$, the largest eigenvalue of $\mathbf{D}^{-1}\mathbf{R}$ is still greater than one, $\lambda_{\max} = 2$, and thus the GJ algorithm does not converge. We can indeed conclude from these results that generally, matrices whose diagonal coefficients are large compared with the non-diagonal ones are well-behaved from the point of view of GJ convergence. In regards to the Gauss-Seidel algorithm, our results indeed agree with the fact that this algorithm converges for any symmetric, positive definite matrix (which is not always the case for GJ algorithm).

Lastly, we modify the program so that $\mathbf{A}$ is the matrix $\mathbf{A}$ full of ones, except on the diagonal where, $a_{ii} = i$ for $i = 1, \ldots, m$. We run the program for a 10×10 matrix $\mathbf{A}$. For a starting point $x_0 = 0$, we find is that the Gauss-Jacobi algorithm does not converge, while the Gauss-Seidel algorithm converges in 22 iterations. Again, these results agree with the above discussion in regards to the specific convergence criteria for each algorithm.

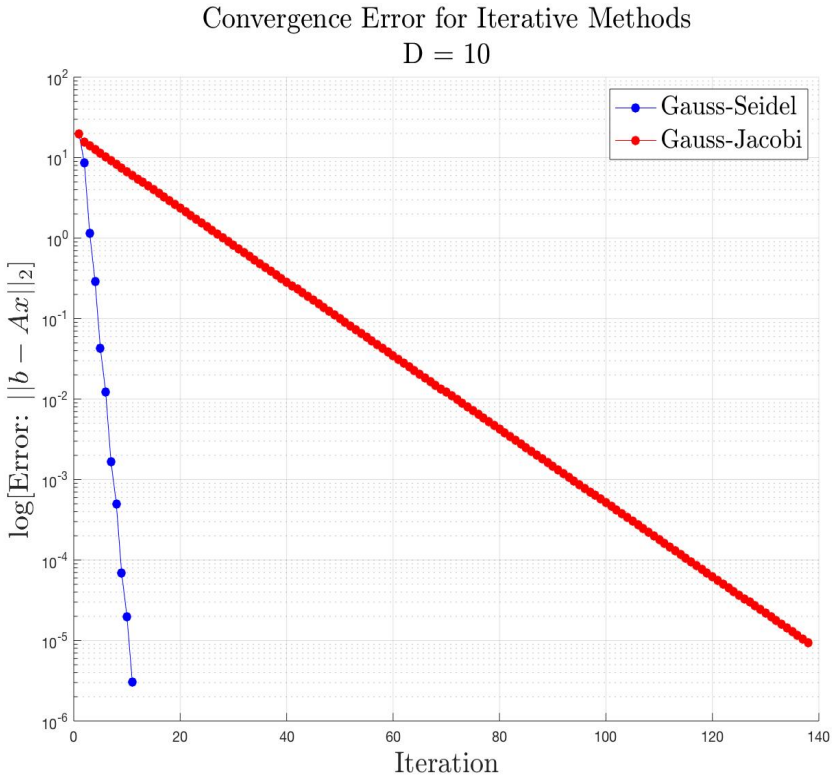

Figure 3: Log-linear plot of the error, $||\mathbf{b} - \mathbf{Ax}||_2$, as a function of the iteration number for the Gauss-Jacobi and Gauss-Seidel iteration methods for a value of $D = 10$. Both algorithms start at $x_0 = 0$.

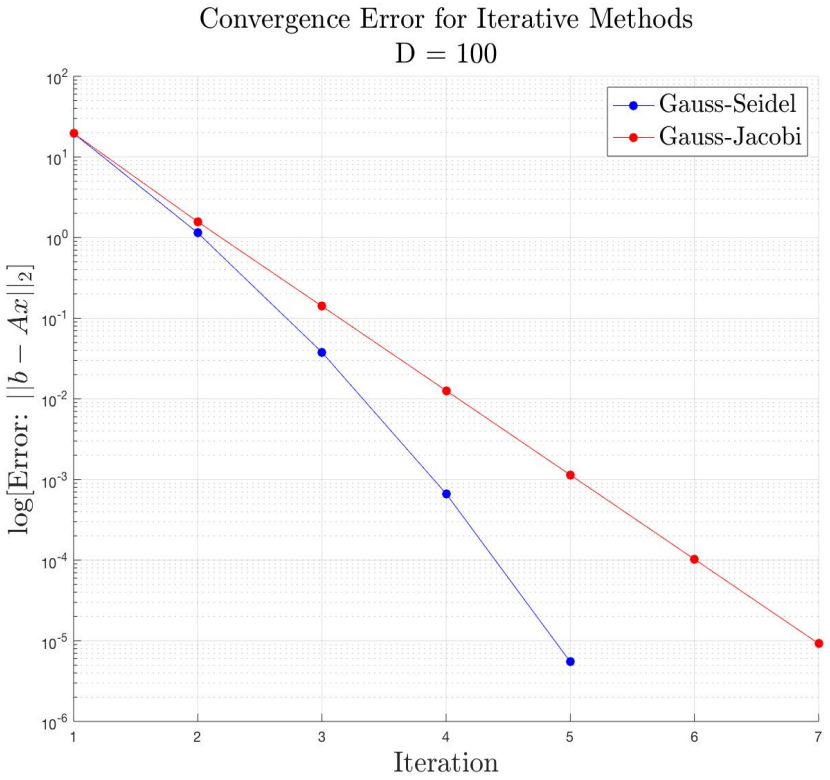

Figure 4: Log-linear plot of the error, $||\mathbf{b} - \mathbf{Ax}||_2$, as a function of the iteration number for the Gauss-Jacobi and Gauss-Seidel iteration methods for a value of $D = 100$. Both algorithms start at $x_0 = 0$.

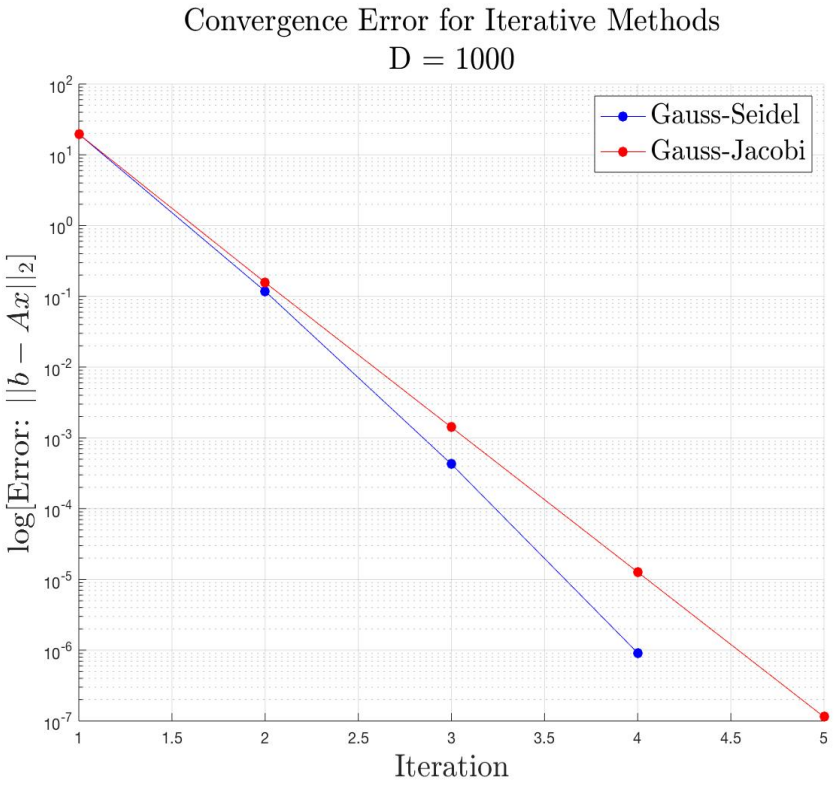

Figure 5: Log-linear plot of the error, $||\mathbf{b} - \mathbf{Ax}||_2$, as a function of the iteration number for the Gauss-Jacobi and Gauss-Seidel iteration methods for a value of $D = 1000$. Both algorithms start at $x_0 = 0$.

We now incorporate the Conjugate Gradient results into our discussion. As per the results in Table 2, we find this method converges in 2 iterations for all values of D. Again, the matrix $\mathbf{A}$ is all ones except on the diagonal where $a_{ii} = D$. This is very fast convergence in comparison to the other methods, and is expected given that matrix $\mathbf{A}$ has all the same diagonal values.

The diagonal pre-conditioner for the CG method is not very useful for these matrices because all the diagonal entries are the exact same. Because $\mathbf{A}$ is a real, symmetric, and positive definite matrix of this form (all elements are ones except diagonals are all D), it will have two resulting eigenvalues. One will have a multiplicity of $m-1$, and the other a multiplicity of 1. The eigenvalues are not widely dispersed, and furthermore $\mathbf{A}$ is not poorly conditioned. When the eigenvalues do differ widely in size and the matrix is poorly conditioned, this is when diagonal pre-conditioners are useful in speeding up the algorithms convergence. With this algorithm converging in 2 iterations for all tested values of D, we can already see there is no need to implement any pre-conditioner.

Now we modify the program as before, so that $\mathbf{A}$ is the matrix $\mathbf{A}$ full of ones, except on the diagonal where now, $a_{ii} = i$ for $i = 1, \ldots, m$. Again we use $x_0 = 0$. For a 10×10 matrix $\mathbf{A}$, we find that the CG method converges in 10 iterations. For a 100×100 matrix $\mathbf{A}$, the CG method converges in 61 iterations. Since the diagonal elements are $a_{ii} = i$ for $i = 1, \ldots, m$, the eigenvalues will be more widely dispersed, the largest and smallest eigenvalues will differ in size by more, and thus the convergence of the CG algorithm is slower. This would be a matrix for which diagonal pre-conditioning would be useful.

Provided $\mathbf{A}$ is diagonally dominant, the following preconditioner fits the requirements as it is a good approximation to $\mathbf{A}$, and its inverse is also easy to find. The slight adjustment to the previous CG algorithm is essentially the additional variable $\mathbf{z} = \mathbf{M}^{-1}\mathbf{r}$, where $\mathbf{r} = \mathbf{b} - \mathbf{A}\mathbf{x}$ upon the first iteration. This $m \times m$ matrix $\mathbf{M}$ is:

$$\mathbf{M} = \begin{bmatrix} a_{11} & & & \\ & a_{22} & & \\ & & \ddots & \\ & & & a_{mm} \end{bmatrix} \rightarrow \mathbf{M}^{-1} = \begin{bmatrix} a_{11}^{-1} & & & \\ & a_{22}^{-1} & & \\ & & \ddots & \\ & & & a_{mm}^{-1} \end{bmatrix}$$

We implement this method on the same matrix $\mathbf{A}$ above which is all ones except on the diagonal where now, $a_{ii} = i$ for $i = 1, \ldots, m$. Starting with an initial value $x_0 = 0$, for a 10×10 matrix $\mathbf{A}$, we find that in comparison to the CG method converging in 10 iterations, the diagonally preconditioned CG algorithm converges in 8 iterations. For a 100×100 matrix $\mathbf{A}$, the CG method converges in 61 iterations, where the preconditioned CG algorithm converges in just 10 iterations. This shows the importance of preconditioning the matrix $\mathbf{A}$ in regards to significantly speeding up the algorithm when the convergence rate of the standard CG algorithm is slower, likely due to a matrix with a fairly large condition number.