Question a.0 - Implementation discussion
Discuss the implementation of SVD in this situation.

Solution: This is the first problem I've truly been concerned with computational efficiency. It has been in the back of my mind, but since we've been mostly working with toy examples it hasn't been a necessity. This changes once your matrix has dimensions larger than 3000 by 5000. Calling the pre-written and (assumedly) optimized svegd takes four minutes or so on my machine. This is even with optimization flags turned on. As such it was imperative to test with the smaller data set, who's svegd completed almost immediately. The smaller data set was still quite large, 811 x 1280, but this underscores the nonlinear increase in computation time as we increase dimensions of our matrix.

Now to move through the "sequence of tasks" one by one. The initial read in step was accomplished by re-purposing Dongwook's reading subroutine. To do this I had to add the number of rows and columns to the first line of the .dat file. After some confusion with standards of image representation (written width $\times$ height) as opposed to our usual row $\times$ column matrix representation, I was ready to start finding singular value decompositions.

To find these singular values, we called dgesvd - a prewritten routine from lapack. This was a kindness, as I'm sure my own dgesvd routine wouldn't have been efficient. This was a battle itself however as dgesvd takes roughly 8 arguments, with at least 3 of them being deeply confusing at a first glance. The most headache inducing of these entries is the work matrix, in which we allocate a space for dgesvd to do it's computations. As discussed by Ian in emails, knowing the size of this work matrix before starting your problem is a tall task. However, dgesvd allows us to call with lwork = -1 to query what size this argument should be. With this call completed, we have created the singular value decomposition in double precision

$$A = U\Sigma V^T$$

Reconstruction was assuredly the hardest part of this process for me. My first routine looked like it should work, but would produce negative values in the process (not great for an ASCII data file). Unable to get the process to work with slicing, I turned my attention to reconstruction element by element. To do this, I required a triple do loop to iterate through singular values, then essentially take an outer product of the columns of U and the rows of V^T. This takes advantage of the diagonal nature of Σ, our matrix of singular values. Since it is a diagonal matrix it commutes, and we can just take the outer product of the appropriate columns and rows of U, V^T scaled by the appropriate singular value. It is worth noting we only needed to decompose once, then the reconstruction builds up from this single decomposition. The reconstruction routine takes bounds as an argument so that we can call it on higher values without doing the previous ones again, for instance creating the approximation with $k = 40$ only required reconstructing from $k = 21, 40$, but was called on the matrix that had been reconstructed with $k = 1, 20$.

Saving/plotting each of the eight reconstructions was a bit of a pain, though very rewarding when it worked. Very rarely is one rewarded with dog as the conclusion to a math problem. Writing to a file, then pulling the file out of WSL and into matlab was a mistake. Since Windows and Linux have different end of line symbols, my data was squished into a single vector roughly 1 $\times$ 1,000,000. Reshaping alone was not enough to save me, this produced a jumbled mess. Instead I had to write out the transpose, then reshape into the appropriate dimensions. This took *a while*. Reshaping the data into appropriate dimensions and loading in the .dat files took around 10 minutes for each approximation. Beyond this, the transpose to reshape resulted in a mirrored image. Though possible to fix by swapping all columns around the center column, I decided to leave it reflected to avoid another costly permutation of my matrix.

Finally, I computed errors in Fortran, but plotted them using python. This figure is included on the next page due to space constraints and showcases the error decreasing as we increase the values of k, which is intuitive as we more faithfully are recreating our matrix.

[Please find graph of error vs k on the next page...]

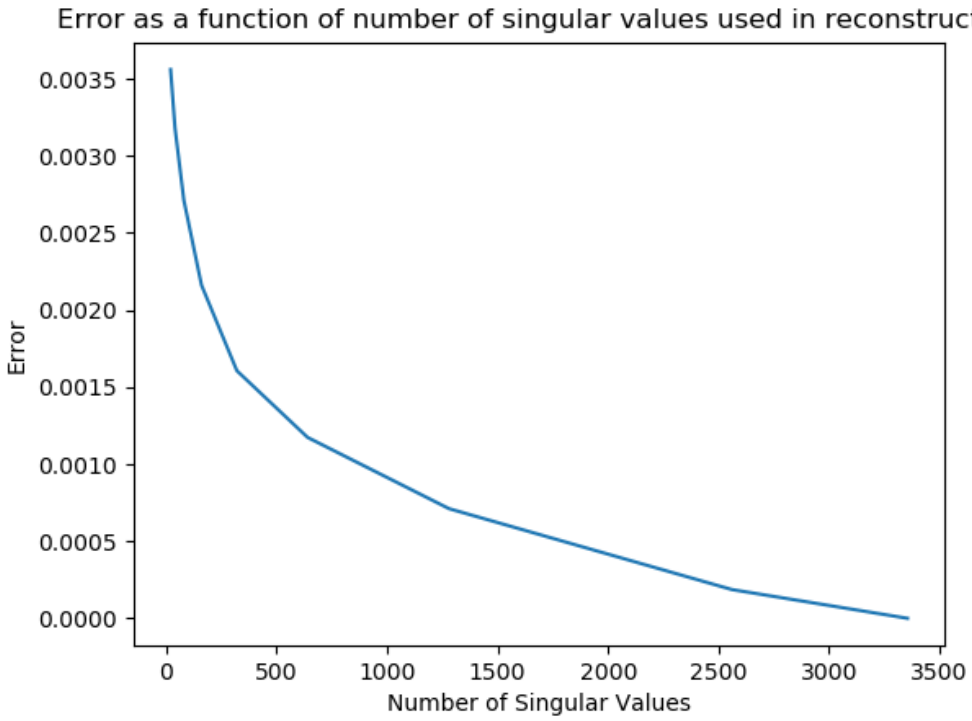

Figure 1: Error as a function of k

Question a.1
Report the first 10 largest singular values from σ_1 to σ_{10}. Also report σ_k for

$$k = 20, 40, 80, 160, 320, 640, 2560, 3355$$

Solution: Consider the following table of singular values:

k	σ_k
1	663181.73
2	85706.24
3	62123.69
4	34664.63
5	31861.71
6	21870.62
7	19628.55
8	18434.07
9	13694.46
10	12815.39
20	7529.20
40	5514.52
80	3981.03
160	2679.36
320	1518.39
640	822.37
1280	513.83
2560	179.10
3355	16.72

Question a.2
Display all eight compressed images along with the original image. Identify figures clearly with the number of singular values used for each.

Solution: First, we have the given data which creates:

Figure 2: Original data, unaltered

Next we show the compressed images, recreated with varying numbers of singular values, the specific number can be found in the figure's description underneath the image. Note the additional artifact of my computational process that reflects the images across the y-axis. This is as discussed in my implementation section.

Figure 3: Recreated with $k = 20$ singular values

Figure 4: Recreated with $k = 40$ singular values

Figure 5: Recreated with $k = 80$ singular values

Figure 6: Recreated with $k = 160$ singular values

Figure 7: Recreated with $k = 320$ singular values

Figure 8: Recreated with $k = 640$ singular values

Figure 9: Recreated with $k = 1280$ singular values

Figure 10: Recreated with $k = 2560$ singular values

Figure 11: Recreated with $k = 3355$ singular values - full recreation.

As you can see, aside from the mirroring, the final approximation using all singular values is quite accurate. Additionally, as k increase the sharpness of our image increases. For comparison purposes, the original figure 2 is placed here as well.

Figure 2: Original data unaltered (again)

Question a.3
Discuss what you see as you increase/decrease the number of singular values in calculations. Report at which k value you obtain an error lower than 10^{-3}.

As mentioned at the end of the last problem, qualitatively we can see the images getting "sharper" as we increase k. The resolution of the image seems to increase because we are using more and more singular values (which seemingly encode the essence of the image) in the recreation.

This can be seen analytically by considering the following table of errors as a function of singular values. Error is taken to be the difference of our matrices in the averaged Frobenius norm, that is for each k, we compute

$$\text{Error} = \frac{||A - A_{approx}||_F}{mn}$$

where A is the original given data, A_{approx} is the recreated approximation dependent on k, and mn is the number of rows times columns of our original matrix.

k	Error
20	$3.559E-3$
40	$3.18E-3$
80	$2.71E-3$
160	$2.16E-3$
320	$1.61E-3$
640	$1.17E-3$
1280	$7.11E-4$
2560	$1.86E-4$
3355	$4.90E-16$

From this we can see that the k value where we obtain an error lower than 10^{-3} is 1280, though it starts closer than I anticipated honestly. It is worth paying special attention to the division by mn in the error computation - otherwise some of these errors would be quite large. This weighting is justified due to the sheer number of data entries we are working with. For instance, without dividing my mn our error for the $k = 20$ case would be about 63224. This indicates how different each pixel is, but doesn't capture that we were indeed somewhat close. On the other hand, I think it is interesting for the full ($k = 3355$) representation the error (unaveraged) would be $4.90531 * 10^{-16}(3355)(5295) = 8.71415 * 10^{-9}$ which is still great. It seems our full rank approximation is hardly an approximation! I'm fairly confident the naked eye couldn't tell the difference.

Question b.1.1 and b.1.3
Explain briefly what the Gauss-Jacobi and Gauss-Seidel algorithms do, how they differ, and what the convergence criterion for the algorithm is.

Discuss why some of the cases didn't converge, and compare the convergence rates of the two algorithms in light of the theoretical predictions.

Solution: The cleverness of these algorithms, like much of numerical linear algebra, lays in the avoidance of inverse calculation. For Gauss-Jacobi, henceforth referred to as Jacobi, instead of computing $Ax = b$ we compute $A = D + R$ where D is a diagonal matrix and R is the rest. This allows us to rewrite

$$Ax = b \implies (D + R)x = b \implies Dx = b - Rx \implies x = D^{-1}(b - Rx)$$

but since D is a diagonal matrix, the inverse is very easy to compute as we simply replace the elements a_{ii} with $1/a_{ii}$. We view this algorithm iteratively, meaning we start with a guess of x and replace it as we perform the above steps until $Ax - b$ is within the desired threshold of accuracy. This does mean we're storing another vector, $Ax - b$ however. Regardless, with this the problem has been reduced from order $O(m^3)$ to $O(Nm^2)$.

A key difference between Jacobi and Gauss-Seidel, henceforth referred to as Seidel, is now surfacing. In particular, note we use the whole current value of x (all entries) in computing the next value of x. In Seidel, as will be discussed shortly, we are using pieces of the next x within its own definition.

Before moving to Seidel though, we should acknowledge convergence criteria for this method. With a bit of theoretical legwork, one arrives at the theorem:

Theorem 1. *The iterative algorithm*

$$x^{(k+1)} = Tx^{(x)} + c$$

where T is a square matrix and c is a constant vector converges provided $\rho(T) < 1$.

For our particular purposes, this means we'll expect convergence for Jacobi when

$$\rho(D^{-1}R) < 1$$

and indeed, if one computes the spectral radius of the $D = 2, D = 5$ cases (diagonal entries, not matrices) we have $\rho(D^{-1}R) > 1$.

Now to examine Seidel, in which entries of x are directly overwritten as the algorithm precedes, but otherwise is rather similar. This speeds up convergence by using the updated values of the coefficients of x as they arrive to calculate the next.

The convergence of Seidel is much better than that of Jacobi. It has been shown that for any case Jacobi converges, Seidel converges faster. That is, a sufficient condition for convergence of Seidel is that spectral radius of $\rho(D^{-1}R) < 1$. Put another way, Jacobi's necessary convergence condition is sufficient for Seidel. Indeed, we see Seidel converge for more matrices than Jacobi did. This is because Seidel has been proven to converge for symmetric positive definite matrices, which as far as I can tell is the necessary condition for convergence of this method. Indeed, the cases with $D = 2, 5$ are symmetric positive definite and as such we expect (and do see) convergence for Seidel. On the next page we examine graphs of error as a function of iteration count, and as discussed above, see Seidel converging faster than Jacobi for all values of D where Jacobi even converges.

Question b.1.2
Run the code for a 10×10 matrix A with $D = 2$, $D = 5$, $D = 10$, $D = 100$, and $D = 1000$. For each value of D where the algorithm converges produce a figure showing on a log-linear plot $||b - Ax||_2$ as a function of the iteration number for both Gauss-Jacobi and Gauss-Seidel.

Solution: The cases $D = 2, D = 5$ did not converge for Jacobi, while we did still witness convergence for Seidel. As such, the graphs for these cases are omitted. These plots were generated requiring an accuracy of $1E-10$, instead of the $1E-5$ that the driver currently uses, because I wanted to exacerbate the differences between the two methods. This gives the following:

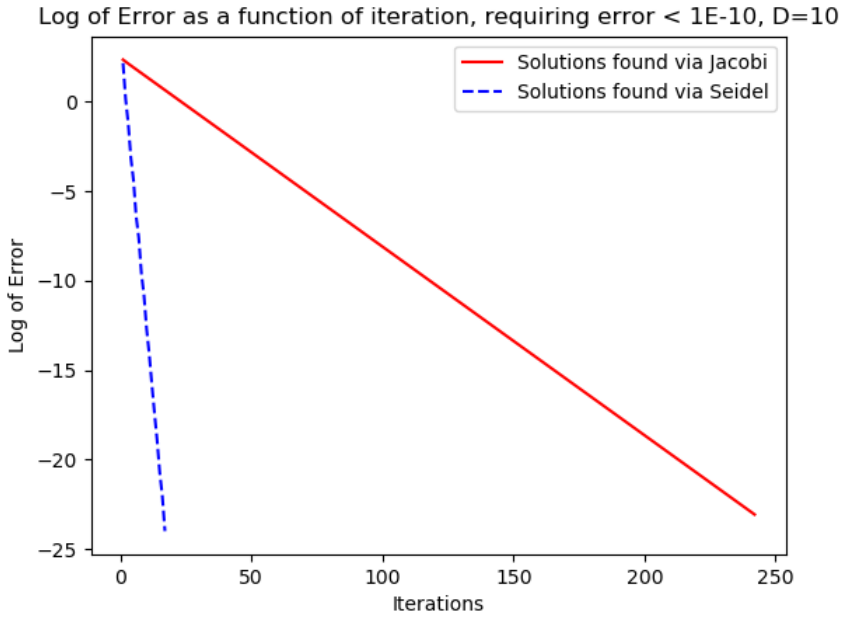

Figure 12: Error vs Iteration, D = 10

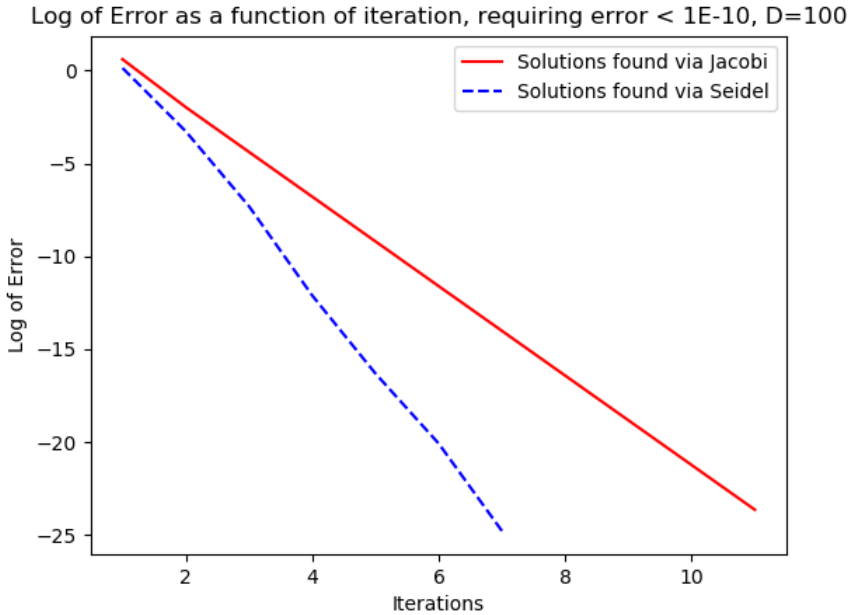

Figure 13: Error vs Iteration, D = 100

[Final figure found on next page...]

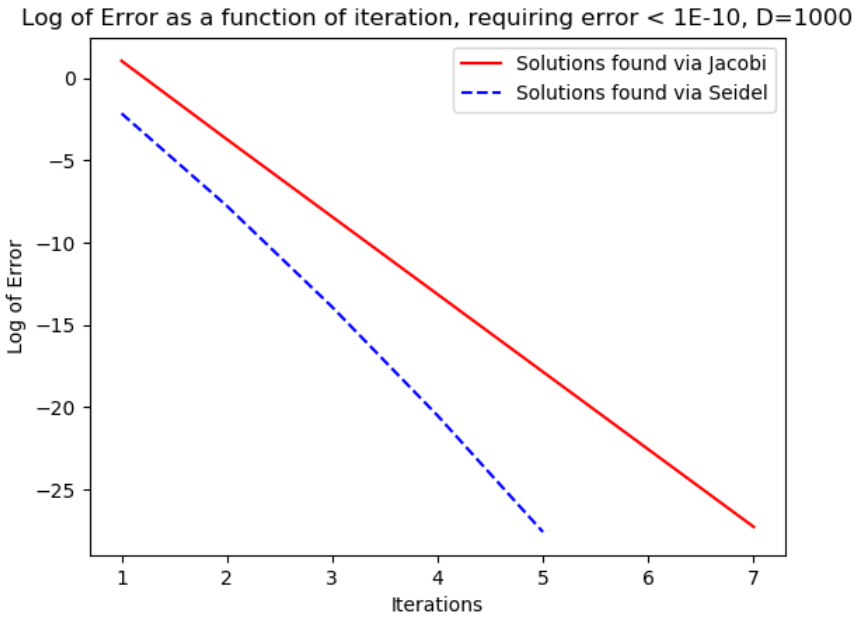

Figure 14: Error vs Iteration, D = 1000

Question b.1.4
Modify the program so that A is the matrix A full one ones, except on the diagonal where $a_{ii} = i$. Run the program for a 10×10 matrix A. Do either of the algorithms converge?

Solution: No, both the algorithms do not converge. Jacobi doesn't converge in this instance, while Seidel does. This is because the matrix in question has spectral radius $\rho(D^{-1}R) > 1$, thus not satisfying the necessary condition for convergence using the Jacobi method. On the other hand, Seidel converges because the matrix is symmetric positive definite, satisfying the convergence criterion for this algorithm. Seidel converged in 28 iterations to an accuracy of $6.27E-8$, as it was required the error be less than $10E-5$.

[Note: problem b.1.3 was answered alongside b.1.1]

Question b.2.1
Explain briefly what the conjugate gradient algorithm does and why it is guaranteed to converge in a finite number of iterations.

Solution: A key requirement of this method is that our matrix be symmetric positive definite. This method takes the method of gradient descent as inspiration, but constructs conjugate sets on which to do this process. Assuming we can find a good conjugate set, which we in fact find is always possible for a symmetric positive definite matrix, we effectively construct successive approximations to x. By construction, $x^{(m)} = x$, meaning that after a finite number of iterations it converges to the desired solution. At the heart of this method is the construction of conjugate directions however, and moreover we want to do so iteratively. Thus the main goal to keep in mind is to find the direction closest to the steepest descent that is still conjugate to the previous directions. The requirement that this algorithm converges in a finite number of steps is captured in the theorem:

Theorem 2. *The conjugate gradient algorithm, starting from $x^{(0)} = 0$, $r^{(0)} = p^{(0)} = b$, and applying the iterations described above, converges to the true solution in at most m iterations. In other words, $x^{(k)}$ is the best possible approximation of the true solution x that lives within the Krylov subspace $\kappa^{(k)}$.*

Question b.2.2
Prove that the smart conjugate gradient algorithm is equivalent to the basic conjugate gradient algorithm.

Proof. As stated in the lecture notes, showing this requires induction. We aim to show for arbitrary iteration $k \in \mathbb{N}$ it follows that

$$r^{(k)}p^{(k)} = r^{(k)T}r^{(k)}$$

and

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

To that end,

Base case: $k = 0$ Note that

$$r^{(0)}p^{(0)} = rp = (r)(r + r)$$

by definitions of r and p since we start the algorithm by letting $p = r$. So we're only left with $r \cdot r$. Thus we're taking a dot product of the residual with itself, which may be written as $r^T r$. Then

$$r^{(0)}p^{(0)} = r^T r$$

as desired.

Next, we rearrange the equation given in the notes to solve for β_0.

$$p^{(1)} = r^{(1)} + \beta_0 p^{(0)}$$

$$\implies r^{(1)T}p^{(1)} = r^{(1)T}r^{(1)} + r^{(1)T}\beta_0 p^{(0)}$$

Unfortunately, out of brain power and short on time, this is where my rigorous proof ends. I believe the way to show the base case is to apply A to both sides and invoke the orhthogonality of p somehow. To conclude the proof, suppose the pattern holds up to iteration k, then show that this supposition implies the pattern holds for $k + 1$. It's been a long week.

□

Question b.2.3
Run the code using the conjugate gradient algorithm for a 10×10 square matrix A with $D = 2, 5, 10, 100, 1000$. What happens this time? Make a table comparing the number of iterations until complete convergence between the 3 algorithms. Explain the trends that you see.

Solution: Consider the following table showing number of iterations until convergence (Error $< 10^{-5}$) for varying diagonal entries. Recall Jacobi doesn't converge for $D = 2, 5$ and as such their entries are filled with N/A.

D	Jacobi	Seidel	CG
2	N/A	77	2
5	N/A	22	2
10	176	13	2
100	9	6	2
1000	5	4	2

As we can see, the conjugate gradient method (CG) does wonderfully regardless of the diagonal entry. For all cases, we saw convergence to a reasonably accurate solution in just two iterations. Moreover, after these two iterations, all errors were smaller than $4.69E-15$, meaning we could have set our threshold to be 10^{-10} and we would have seen convergence in the same number of iterations. This is *not* the case for either Jacobi or Seidel. While the other methods see an increase in rate of convergence as D is increased, CG is so good already it has anything to improve upon.

Question b.2.4
Explain why the diagonal pre-conditioner is not very useful for these matrices.

Solution: The diagonal preconditioner, where we take the diagonal elements of A as the diagonal of our preconditioning matrix, is not helpful in this situation as the diagonals of our matrix are all the same value. In multiplying by a matrix with $1/c$ along the main diagonal and nothing else, we simply scale the whole system by $1/c$. Rates of convergence are dependent on the ratio of the first and last singular value, and in multiplying the whole system by the same diagonal matrix we have no impact on this ratio.

Consider the toy example where A is a 5×5 matrix with 2 along the main diagonal, and 1 in the other entries. Then the eigenvalues are $6, 1, 1, 1, 1$ and as such the ratio of our singular values (our matrix is symmetric positive definite so these *are* our singular values) is 6. If we multiply by the preconditioner M with zeroes in all entries aside from the main diagonal, which contains $1/2$ (the reciprocal of our diagonal in A), then the eigenvalues come out to be $3, 1/2, 1/2, 1/2, 1/2$. Both have the same ratio of first and last singular value, yielding 6 in either case. As such our convergence is not sped up. We conclude diagonal preconditioners are for matrices who don't have the same value repeated along the whole diagonal.

Question b.2.5
Finally modify the preogram so that A is the matrix full of ones, except on the diagonal where $a_{ii} = i$. Run the program on a 10×10 and a 100×100 matrix. In each case, does the CG algorithm converge? If yes in how many iterations?

Solution: Yes, the CG algorithm does indeed converge for all cases. This is not a surprise to me, the algorithm is the closest thing to magic I've found in the real world. For the 10×10 case, CG acheived convergence to an accuracy of $10E-5$ in 10 iterations with an error of $2.08E-11$. The 100×100 case saw convergence in 67 iterations, finishing with an error of $9.83E-8$.

As the theory suggested, both converge in m or less steps, taking the maximum amount allotted for the 10×10 case. These are matrices who could be preconditioned to speed up convergence, most likely with the diagonal preconditioner discussed earlier, as the diagonal entries actually change.