Computational Linear Algebra Project 3: June Version

Jonathan Cheung

June 2020

1 Eigenvalue algorithms

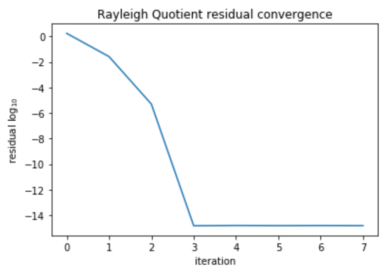
i) The function ‘CSR_construct’ will output a pentadiagonal matrix with 6s along the main diagonal, -4s on the diagonals either side of this, and 1s on the most outer diagonals. The dimension of this matrix is equal to the input ‘n’.

The function ‘Ray_QI’ takes as its input an integer ‘n’, an initial vector ‘v0’, and a number of iterations ‘iterat’. Ray_QI then constructs the tridiagonal matrix using ‘CSR_construct’ and then applies the Rayleigh Quotient Iteration for the inputted number of iterations using v0 as the initial vector. The output is a 2-D array, where each row represents a new iteration of calculated eigenvector, and a 1-D array where each value represents a new iteration of calculated eigenvalue.

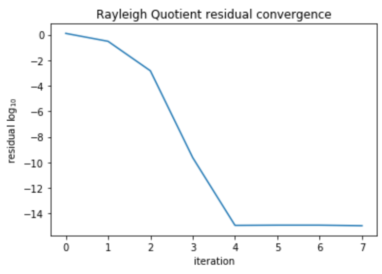
ii) The cost of this algorithm is calculated as follows: Within a single iteration, we have four main steps. The first step is forming the $(A-\lambda^{k-1}I)w = v^{k-1}$ equation. To form the matrix, it costs 2n flops (n scalar multiplications and n additions). The second step is solving this banded system with upper and lower bandwidth 2. This costs 12n flops (4n flops to do an LU decomposition, 4n flops to backward and forward substitute). The third step is to normalise this w vector and this takes 3n FLOPS (n multiplications, (n-1) additions, 1 square root, and n divisions). The last step is to calculate $\lambda^k = (v^k)^T Av^k$. A is pentadiagonal, so the first multiplication costs 9n flops (5 multiplications and 4 additions per row/column). The second multiplication costs 2n^2 -n FLOPS (n multiplications and (n-1) additions per row/column). Adding these all up we have approximately n^2 +25n flops per iteration.

iii) We can use function ‘RQ_conv’ to explore the convergence properties of our Rayleigh Quotient function. We just input in the dimension of the A matrix we want to construct and then the function will plot a graph of the residuals for some randomly chosen initial vector. The function works by calculating the true eigenvectors and then comparing it with the eigenvector estimate from

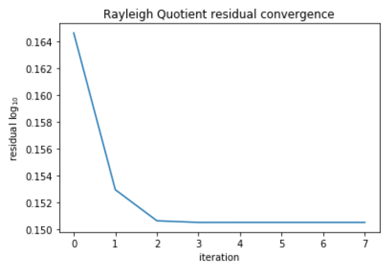
‘Ray_QI’, or in some cases the eigenvector estimate scaled with -1. Here are the first 3 plots for dimension=10 and seed=2.



(a)



(b)



(c)

Figure 1: Some residual convergence plots with dim=10 and randomly chosen starting vectors

In figure 1a and 1b we can we that the function has managed to locate the true eigenvector. And in each of these cases can see approximately cubic convergence. The y axis is a $\log_{10}$ scale and so we can see that our eigenvector estimate is gaining 3 decimal places of accuracy per iteration. The plot levels out at 1e-15 since that is the most precision we have. We also note that the function does not always converge to a true eigenvector (figure 1c). In each of these three plots, the initial vector estimate has been randomly generated and it appears that in the case of figure 1c, this initial estimate might have been too far away from any of the eigenvectors, so the algorithm was unable to converge.

iv) The eigenvalue of A that is usually computed is the eigenvalue that is closest to the first eigenvalue estimate (which has been formed from the initial vector). Our code in lines 148-174 has an experiment to confirm this. We first construct a 10x10 A matrix and use a numpy function to find the true eigenvalues. We then use our Rayleigh Quotient Iteration on this A matrix with 8 iterations (in figure 1 we saw that 8 iterations was more than enough for convergence with a 10x10 matrix). We then look at the initial eigenvalue estimate and find the true eigenvalue that it is closest to. Lastly, we compare the final eigenvalue that we converged to with this eigenvalue that was closest to the initial guess. Our code shows that we can predict which eigenvalue will be converged to with approximately 70% accuracy. We also note that we fail to converge to any eigenvalue 10% of the time. There is some code (lines 114-143) that checks whether we can predict the eigenvalue, by finding the eigenvalue that corresponds to the eigenvector closest to our initial vector, but it appears that this is only successful 30% of the time. I do not have an explanation for this and potentially the code is wrong and has not accounted for some sign errors.

v) ‘pentdiag_qr’ will compute the QR factorisation of matrix A by transforming it into dense matrix and using Householder reflections. Q is banded with maximum upper bandwidth and a lower bandwidth 2. R is a banded matrix with upper bandwidth 4 and lower bandwidth 0. Of course, the Q is orthogonal and the R is upper triangular. The bandwidths of these matrix are a consequence of the A matrix having upper and lower bandwidths 2 and also from the way that householder reflections work. The householder reflection matrices are applied to A and add zeros below the diagonal. Each subsequent householder reflection must make sure that it does not reintroduce entries into where we just introduced a zero. Since R is not just upper triangular, but is actually banded, this means our householder reflections are adding zeros to the columns and rows at the same time. By using a sparse matrix, we can ignore these additionally added zeros when adding zeros to the next column. This speeds up the process.

vi) By implementing the pure QR algorithm for A and keeping track of the diagonal values of A at each iteration with the function ‘pure_QR’, we can notice that these diagonals of A correspond to the ordered eigenvalue estimates. This happens because the pure QR algorithm is related to the simultaneous iteration, which is a simultaneous version of the power iteration. We know that the power iteration converges to eigenvalues.

vii) Our matrix A is hermitian, so we can use the algorithm for householder reduction to Hessenberg form to convert A into a similar tridiagonal matrix. Our function ‘householder_A’ performs this for us. In our code, we construct a 10x10 A matrix and we convert this into the equivalent tridiagonal matrix ‘tri_A’. We can apply our pure QR algorithm on each of these matrices and see that the third element of each (‘pure1’ and ‘pure2’) are identical. This third element is an array where each row represents an estimate of the eigenvalues, which have been extracted from the main diagonal of the iterations of A^k. These estimated eigenvalues are the same since the tridiagonal matrix is similar to the original A matrix and similar matrices have the same eigenvalues.

2 GMRES and preconditioning

i) For equation (4), A is an N^2 x N^2 matrix. This matrix is is constructed of 5 diagonals. The main diagonal has entry $2(1+\beta)$. The diagonals either side of this main diagonal have entries -1. The matrix also has two diagonals with entries $-\beta$ that either shifted N spaces from from the main diagonal. b is a column vector of length N^2 and has entries $\Delta x^2 f_{\mathrm{i,j}}$ with $i, j = 1, ..., N$.

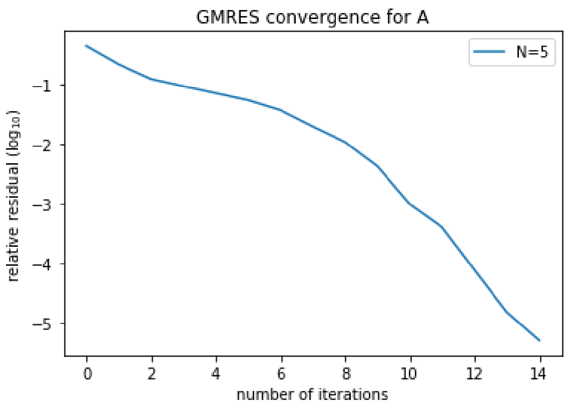
ii) The function 'A2_construct' will construct the A as described above. It takes inputs 'N' and 'B'. N decides the location of the outer diagonals and the dimension of this N^2 by N^2 matrix. B is the beta for the main and outer diagonals.

To solve the system of equations Ax=b for some randomly generated b, we can use 'GMRES_A'. This function takes N, B, and t as its input. N and B define the A matrix as described above. When the residual reaches the tolerance t, the GMRES will stop iterating. It is set at 1e-05 by default and the following plots were all set with t=1e-05. The solution of the system is output as x. We also have an output info that returns 0 to confirm that the tolerance was reached, otherwise info will correspond to the number of iterations that were attempted before exiting.

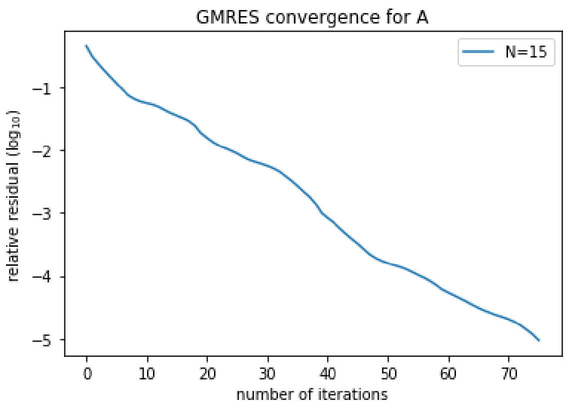
We can use the function 'resid_plot' to plot the convergence of the relative residuals for our matrix A with some specified N and B value. Running this function with N=5 and N=15 (B=1 in both cases) returns figures 2a and 2b respectively. From this we immediately see that the GMRES took more steps to converge when N=15. This is not too surprising since the matrix system being solved in figure 2b has dimension 225, which is much larger than the dimension 25 matrix of figure 2a. More specifically, it takes 14 iterations for GMRES to reach our tolerance level (1e-5). To discover where this number comes from, we will use the 'eig_plot' function. Setting the input to 5, we return a scatter plot (figure 12) of the eigenvalues of the A matrix when N=5. We have ploted them on the X axis so we can imagine a polynomial that will try and fit these points and the point (0,1) as well as possible with as low degree as possible. In this plot we can count 14 main clusters.

Thus, with fundamental theorem of algebra, we know we it will take a polynomial of degree 14 to pass through the centre of these 14 or 15 clusters of points. This implies, after 14 iterations of GMRES, we should reach our desired tolerance. This seems to agree with the last plot. To explain why we needed so many more iterations for GMRES to converge when N=15, we look at the eigenvalue scatter plot for N=15 (figure 2d) and see that the eigenvalues are more spread out; there are no clear clusters. As a result of this, we will need a much higher degree polynomial to fit these points and will need more iterations to reach our desired convergence.

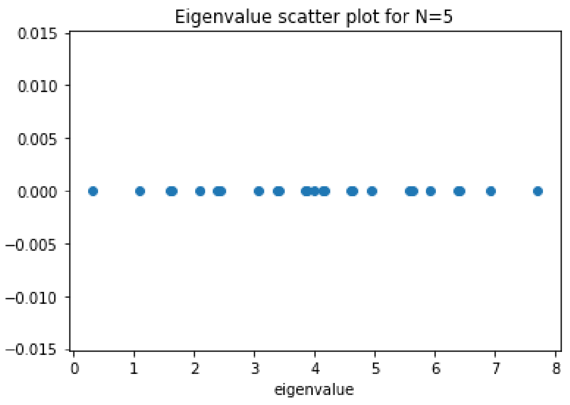
Since the convergence is a lot slower for larger N, perhaps it would be beneficial to implement a preconditioner to this GMRES function when solving for large N.



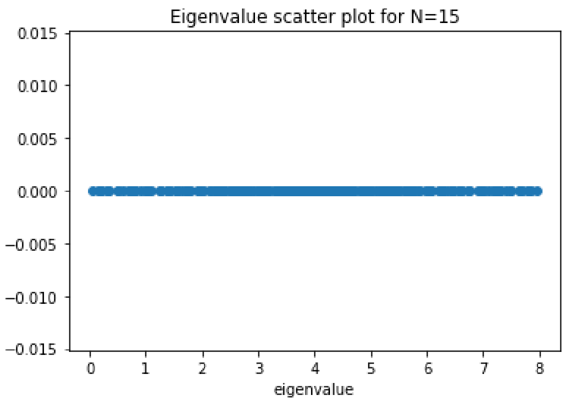
(a)



(b)



(c)



(d)

Figure 2: GMRES convergence and eigenvalues scatter plots for N=5,15 and B=1

Figure 3 shows the convergence for other values of N. As we expect, the convergence is slower for larger N. This is because the A matrices, for larger N, have their eigenvalues more evenly spread. We also note that the convergence is monotonic in all cases. This is because the GMRES algorithm adds an extra dimension each iteration and so we are guaranteed to have a better fit each iteration.

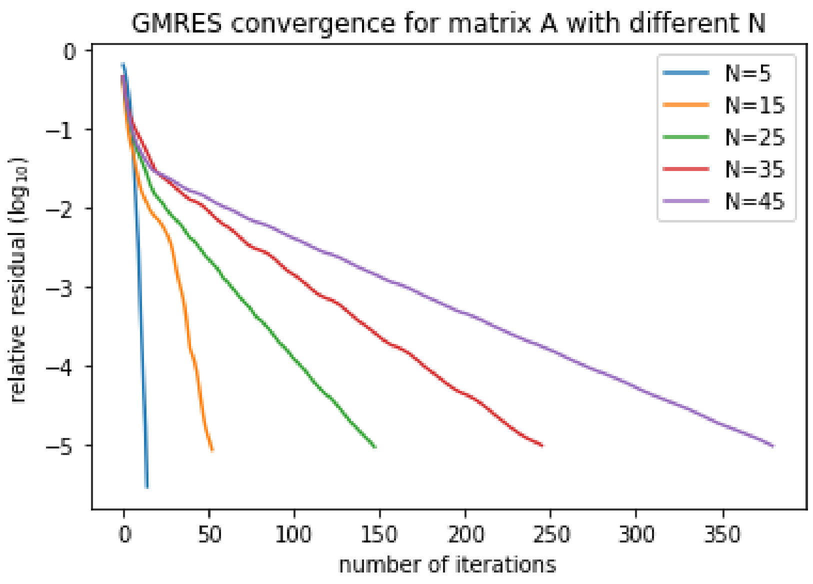


Figure 3:

iii) Suppose the algorithm converges, so x^n converges to some limit L. And we can see that as n tends to infinity, the difference between $x^{n+1/2}$ and x^n decreases. So equations (7) and (8) can be rewritten as

$$(\gamma I + M)L = (\gamma I - N)L + b \tag{1}$$

$$(\gamma I + N)L = (\gamma I - M)L + b \tag{2}$$

Matrix multiplication is distributive so the γL on both sides can cancel out and both equations become

$$ML = -NL + b \tag{3}$$

$$NL = -ML + b \tag{4}$$

which can both be rewritten as

$$\begin{aligned}(M + N)L &= b \\ AL &= b\end{aligned}$$

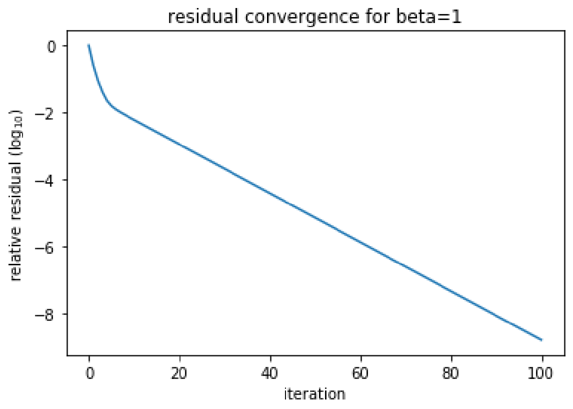
Hence, if the algorithm converges, then it converges to the solution of Ax=b

iv) Each of the equations (7) and (8) are of the form Tx=c, where T is a N^2 by N^2 matrix. And T is formed from adding gamma to the main diagonal of either M or N. M is tridiagonal with entries in the main diagonal. So, adding gamma to the main diagonal of M will keep the tridiagonal structure. And since this whole square matrix has dimension N^2. we can take n nxn blocks and solve these tridiagonal systems independently. Equation (8) is not tridiagonal, but only has 3 main diagonals. We can modify the way we solve this banded system to solve it in the same time as a normal tridiagonal system. If we consider the banded system solver as a modified gaussian elimination algorithm, with the range of loop restricted to the bandwidth. We can further modify this loop of the banded system solver and only operate on exactly the element that is of bandwith n away from the main diagonal since we know there are only zeros between the main and outer diagonals.

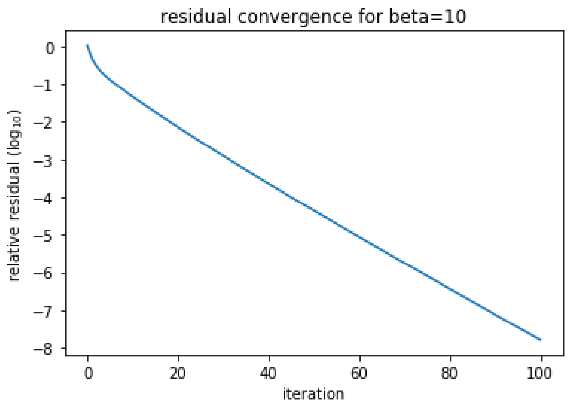
For a single iteration of this algorithm we have two equations that need to be solved. Each equation is formed and solved in three main parts. Note that our matrix has n^2 rows and columns. The first step is to form $(\gamma I + M)$. This requires $2n^2$ FLOPS (n^2 to multiply γ and n^2 additions). To form the right side of the equation we need $7n^2$ FLOPS (n^2 subtractions of γI and then $5n^2$ flops to multiply a tridiagonal matrix with a vector, and then n^2 additions to add b to this new vector. The last step is to solve n nxn tridiagonal systems and this requires $5n^2$ FLOPS (since each of the n systems requires n flops for a LU decomposition and 2n flops for each of the forward and backward substitution algorithms). So for two equations we require two times the total of these flops. It takes $28n^2$ FLOPS per iteration.

v) The function 'split_solve' will implement this algorithm to solve Ax=b. A is formed with inputs n and B. The column vector b must also be put into the function. This is a bit different to 'GMRES_A', which generated the random b vector within the function. But, this has been done so that we can reuse this function as a preconditioner later. Iterations is the number of iterations of equations (7) and (8) that will be completed. Lastly, gamma chooses the γ in the M and N matrices. The output of this function is an array X, where the ith row represents the ith iterated solution.

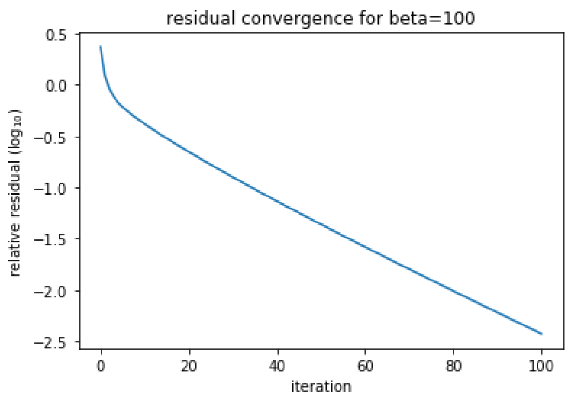
vi) Our code in lines 152-166 will generate a series of residual plots (figure 4) when Ax=b is solved for different β values. n (this is the n to generate A) has been fixed at 10 and we stop the function after 100 iterations. As β tends to infinity, we can see that the rate at which the residuals converge slow down significantly. For $\beta = 1, 10$ 100 iterations is enough to reach a reasonable tolerance. But it is clearly not enough for the larger values of β. The other thing we can notice about these graphs is that the convergence is fastest for the first few iterations, but then settles down. Perhaps this is due to the fact that we have chosen a random starting vector to iterate from and so there is more progress to made per iteration, for the first few iterations.



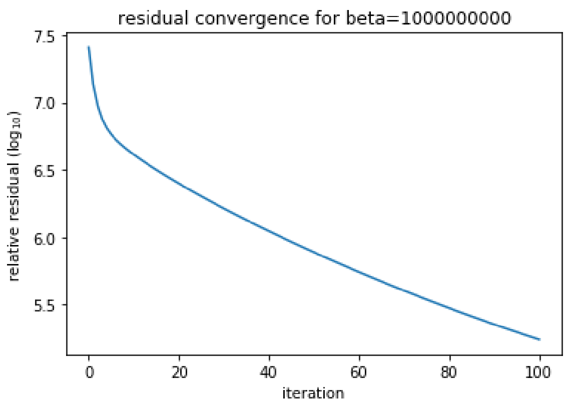
(a)



(b)



(c)

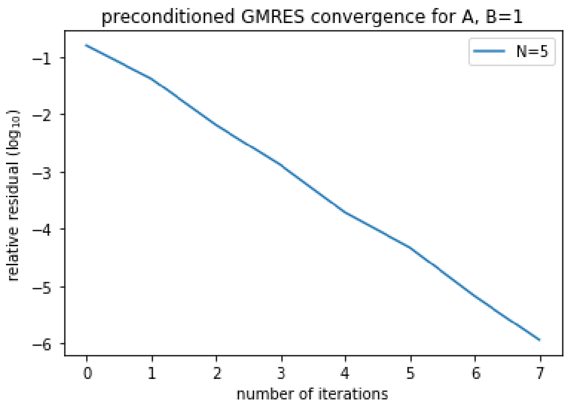


(d)

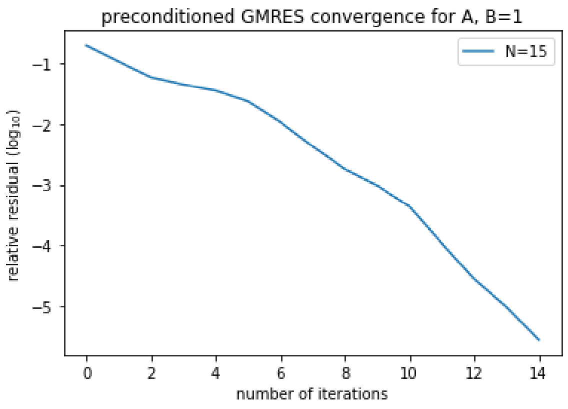
Figure 4: residual convergence using 'split_solve' for fixed N=10, 100 iterations and varying β

vii) We can write a function 'iterate' which will perform a single iteration of 'split_solve' with b=r_k and x0 set to a vector of zeros . This function can now be used as a preconditioner for the GMRES algorithm. Our precondtioned GMRES function is called 'GMRES_p'. Once again, this function solves Ax=b. b is defined by the user. A is generated with parameters n and B. gamma decides how the A matrix is split in the preconditioner function.

Using the function 'resid_plot_p' we can plot the convergence of the residuals when using this preconditioned GMRES function. From figure 4 it is imediately apparent that preconditioning speeds up the convergence. With the parameters (B=1, N=5,15), convergence takes 7 and 14 iterations to reach our desired tolerance level. This is much faster than the 14 and 74 iterations it took in figure 2a and 2b, where we used the same parameters.



(a)

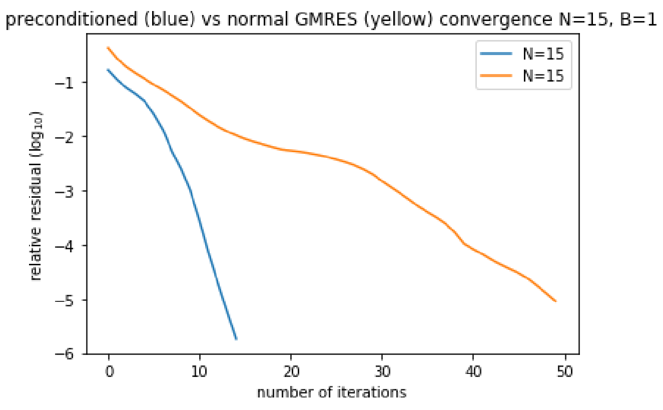


(b)

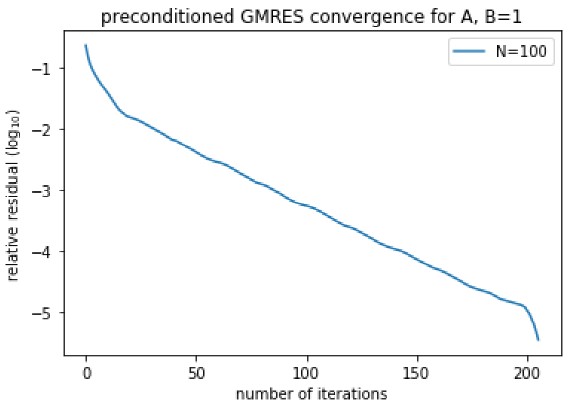
Figure 5: residual convergence using preconditioner for fixed $\beta = 1$ and varying N

In fact we can plot the lines for N=5 on the same graph (figure 6a) just to demonstrate this further. This speed up in convergence comes from the fact that we are solving Ax=b, by transforming it into an equivalent problem which is easier to solve with GMRES. Our preconditioner is also fairly easy to compute, since it is just solving a sparse linear system. So overall this seems to be a good choice of preconditioner; it is easy to compute and significantly reduces the number of iterations.

We can also see the convergence for even larger N in figure 6b. Here N=100 and B=1 (so we are operating on a 10000x10000 sized matrix). For this large matrix, it only takes 200 iterations for convergence. In figure 3, we saw that 200 iterations was not enough to converge for N=35 and the same β value when we did not use a preconditioner.



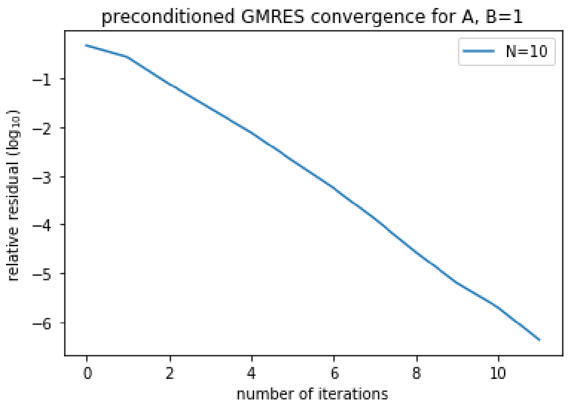
(a)



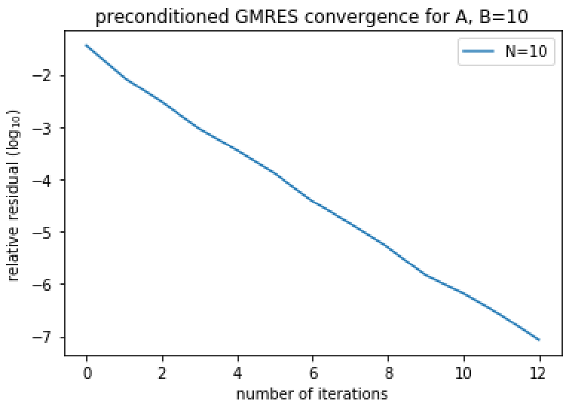
(b)

Figure 6: residual convergence using preconditioner for fixed $\beta = 1$ and varying N

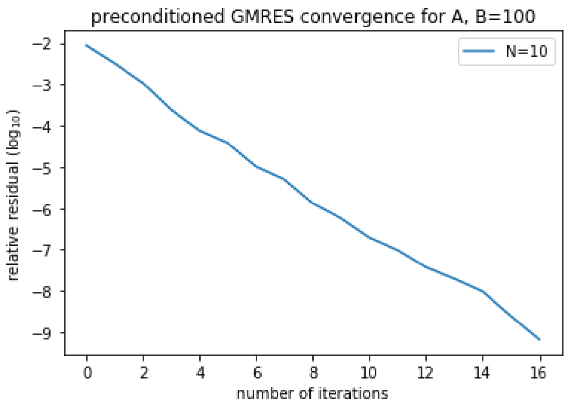
Lastly, we can see how increasing β does not slow down the rate of convergence like it did for the 'split_solve' function. The graphs below (figure 7) were generated by solving the equation with N=10, just like in figure 4. We have also used the same β values. In each case, we manage to reach our desired convergence in well under 100 iterations, whereas 'split_solve' did not manage to converge in 100 iterations for all β values.



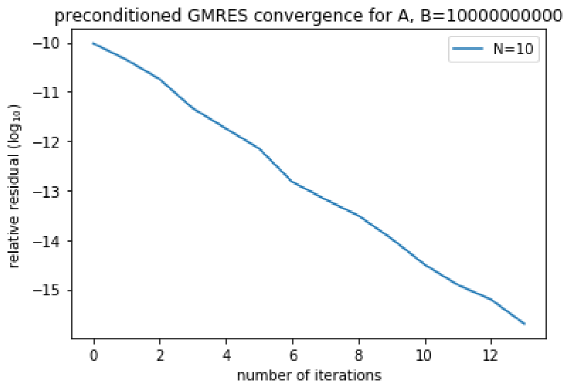
(a)



(b)



(c)



(d)

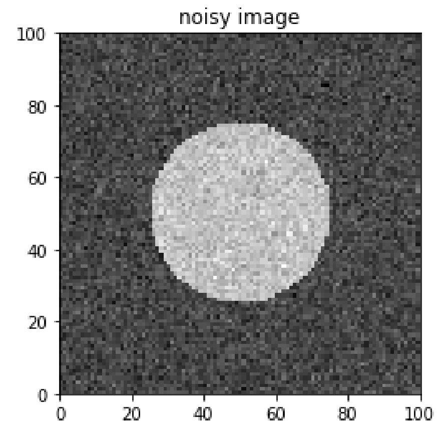
Figure 7: residual convergence using 'GMRES_p' for fixed N=10, 100 iterations and varying β

3 Image denoising

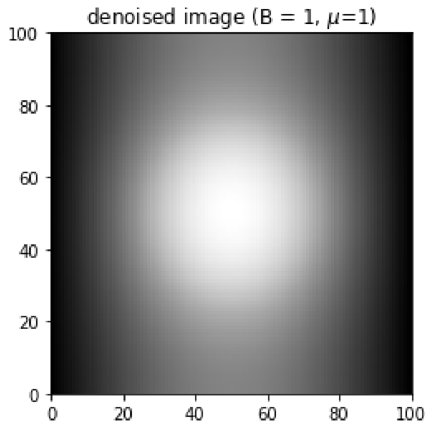
i) We can generate an NxN resolution gray scale image by using the function ‘generate’. This image is constructed by passing an NxN grid of coordinates into function (11) on the assignment. Normally distributed noise is then added with $\sigma^2 = 0.15^2$.

ii) Given the noisy image, we can denoise it by minimising function (12), which can be done by solving equation (13). We can modify our preconditioned GMRES solver from part 2 to solve this equation (13). The modification is adding $\mu\delta x^2$ to the main diagonal of the A matrix. This entry is then included in the M, when A is split into A=M+N for preconditioning. Also, for our GMRES solver to work, we have to convert the 2d array, that corresponds to the noisy image, into a 1d array. Numpy.reshape has been used for this and we have set order=’F’ so that the matrix is converted into a vector by pasting the the columns together, not the rows. These steps are completed by the function ‘GMRES_3’. This function will plot both the noisy and denoised images.

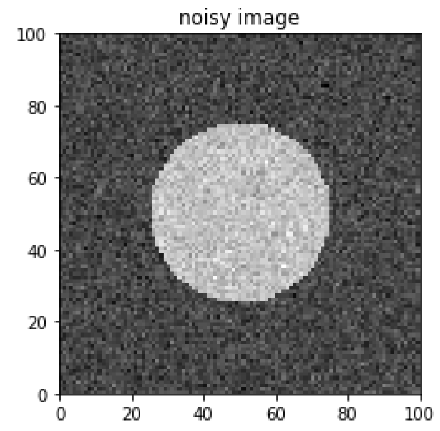
We can investigate how changing the β and μ parameters affect the behaviour of our function. The left images are the noisy ones generated by our ‘generate’ function. The right images correspond to the denoised images that we have constructed by using our modified preconditioned GMRES solver. In figure 8a and 8b we can see what happens when β and μ are set to 1. Although we have denoised the image, the border of our circle has been smoothed out and we have lost detail. We can increase μ significantly and see what happens in figure 8c and 8d. With these parameters, the image has not changed that much, perhaps it has been denoised slightly since there is slightly more contrast in 8d.



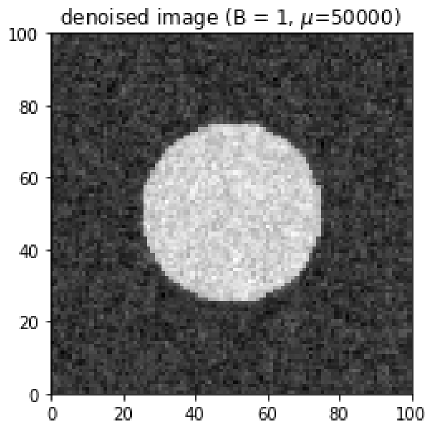
(a)



(b)



(c)



(d)

Figure 8: Left images are the noisy images, right images are corresponding denoised images

If we increase β, we can see in 9b that our circle has been deformed into a square. But again, a lot of the noise has been removed and the image looks smooth. The falloff from white to black is more gradual in the horizontal axis than in the vertical axis.

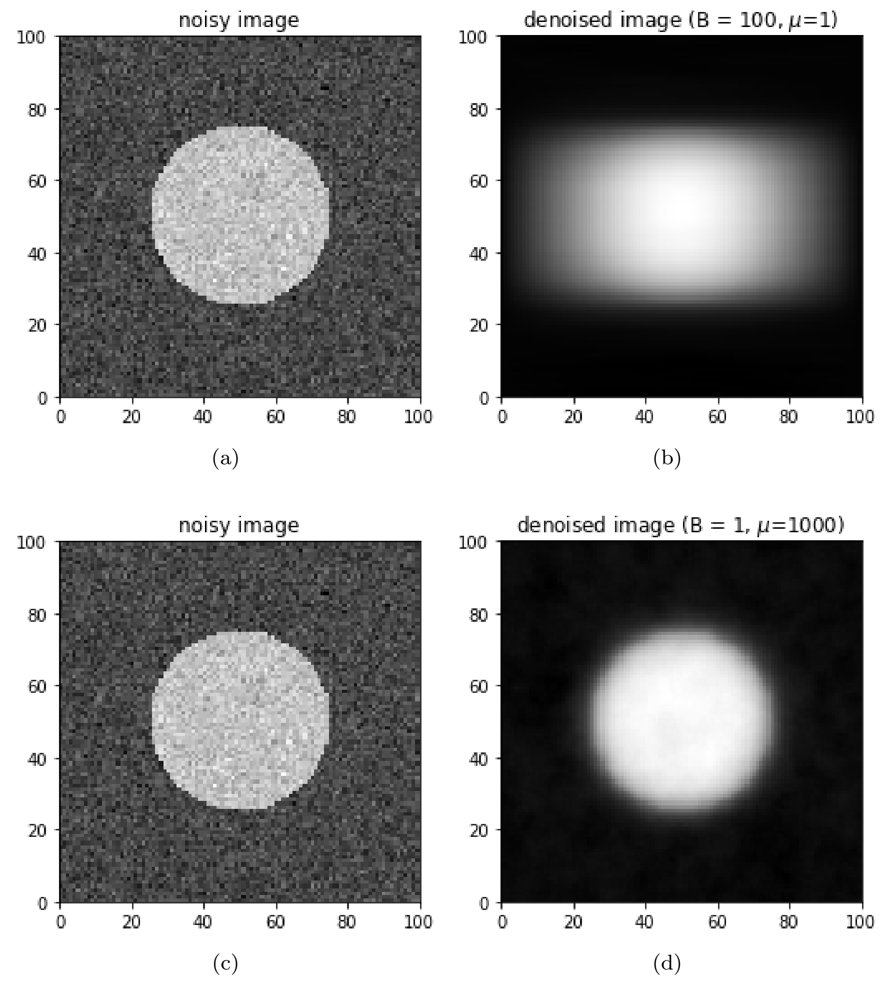


Figure 9: Left images are the noisy images, right images are corresponding denoised images

These observations can be explained by looking at equation (12). The μ parameter scales the last term which is the difference between the noisy and denoised image. By increasing μ we are placing a large penalty on this difference, which causes the denoised image to be very similar to the noisy image. In the case of figure 8d, the penalty is so large, our function has not made many changes to 8c. To avoid this we can not set the value to be too low, otherwise we end up with the scenario in figure 8b; the penalty for similarity to the noisy image is so low, the function has prioritised smoothness and we end up with a loss in detail.

To explain the role of β, we look at the equation we are minimising as see that this parameter applies a penalty onto horizontal smoothness. Thus, using a larger β value like in figure 9b will create an image that prioritises horizontal smoothness over vertical smoothness.

Now that we have seen what the extreme choices of these parameters result in, we will try and find the optimum choice of parameters for our noisy circle image. Since a circle is vertically and horizontally symmetric, we expect same amount of horizontal and vertical smoothness. We also want a good balance between noise reduction and similarity to the original image. Thus, we set $\beta = 1$ and $\mu = 1000$ to produce the image in figure 9d.

iii) The code in lines 147-317 is an attempt to implement the iterative method in 3.3 into code. It has been commented out so that the rest of the file can be ran within a reasonable amount of time. My attempt to implement the method is not vectorised, so it is very slow to run and potentially incorrect. We have not been able to produce any graphs for this part.

Nevertheless, we will still look at equation (14) and attempt to figure out how these parameters affect the solution of the equation. We can see that μ is the penalty applied to the difference/similarity between the noisy and denoised image. Just like in 3.2, we can expect that large μ will cause the denoised image to look very similar to the noisy image. Small μ will cause the denoised image to be more smooth. Perhaps λ is used to define the threshold at which we modify some image pixel. If λ is small, then d^{h} and d^{v} will take value 0 and this will result in less change for the denoising process. I would estimate that the optimum parameters would be μ approximately 1000, just like in the figure 9d, and an extreme value of lambda. Our image is piecewise constant, so it has a very big discontinuity at the border of the circle. Perhaps a very small value of λ will cause the function to only update an image pixel if there is a clear change to be made.