Computational Linear Algebra
Project 3

Tudor Trita Trita
CID 01199397

January 13, 2020

This is my own work unless stated otherwise.

1 Eigenvalue Algorithms

1.1 Rayleigh Quotient Iteration (RQI) for A

The implementation of the RQI algorithm is inside the file `question1.py` under the `rayleigh_quotient_iteration` function.

1.2 Cost of RQI Algorithm

The RQI algorithm is as follows:

Algorithm 1: Rayleigh Quotient Iteration Algorithm

(STEP 0.1) $v^{(0)} =$ some vector with $\left\|v^{(0)}\right\| = 1$
(STEP 0.2) $\lambda^{(0)} = (v^{(0)})^T A v^{(0)} =$ corresponding initial Rayleigh Quotient
for $k = 1, 2, \ldots$ **do**
  (STEP 1.1) Solve the system $(A - \lambda^{(k-1)} I)w = v^{(k-1)}$ for w
  (STEP 1.2) $v^{(k)} = w / \|w\|$
  (STEP 1.3) $\lambda^{(k)} = (v^{(k)})^T A v^{(k)}$
end

The FLOPS model given in lectures only considers arithmetic operations as counting towards FLOPS, ignoring other computational operations such as assignment or retrieval from memory. Let $A \in \mathbb{R}^{(n,n)}, v \in \mathbb{R}^n$.The approximate FLOPS taken for each step of the algorithm are as follows:

- STEP 0.1 - Computing the norm of a vector has approximate cost of $O(n)$.
- STEP 0.2 - This involves matrix-vector multiplication followed by a dot-product of vectors. The dot-products has an approximate cost of $O(n)$. In general, the matrix-vector multiplication has a cost approx. $O(n^2)$. In general, for a banded system the

cost will be approx $O(bn)$ where b is the number of non-zero bands in the system. For a tridiagonal matrix, we can exploit the sparse structure and the cost of this becomes approx. $O(3n)$ where 3 is the number of non-zero bands in the matrix.

- STEP 1.1 - This step involves solving a linear system. For a general, dense matrix, the approximate cost of this is $O(n^3)$. However, for a banded matrix system, the cost will be approx. $O(nmq)$, where m is the lower bandwith of the matrix, and q is the upper bandwidth. For a tridiagonal system, the cost for solving the linear system will be approx $O(n)$.
- STEP 1.2 - Similarly to STEP 0.1, this has a cost of $O(n)$.
- STEP 1.3 - Similarly to STEP 0.2, this has a cost of $O(n^2)$ for general matrices, $O(bn)$ for banded matrices with b bands and $O(3n)$ for tridiagonal systems.

Putting it all together, and taking the leading order term, we have that the approximate cost is:

- $O(n^3)$ for systems with a dense matrix A.
- $O(nmq)$ for systems with a banded matrix A.
- $O(n)$ for systems with a tridiagonal matrix A.

1.3 Behaviour of RQI w.r.t. Initial Vector

The implementation of the RQI algorithm is inside the file `question1.py` under the `question13` function.

Taking $N = 5$, then the eigenvalues and eigenvectors of A are:

```
Eigenvalues of A: [3.732 3.    2.    0.268 1.]
Eigenvectors of A:
[[ 2.887e-01  5.000e-01  5.774e-01  2.887e-01 -5.000e-01]
 [-5.000e-01 -5.000e-01 -1.220e-16  5.000e-01 -5.000e-01]
 [ 5.774e-01  6.390e-16 -5.774e-01  5.774e-01 -3.685e-16]
 [-5.000e-01  5.000e-01  2.090e-16  5.000e-01  5.000e-01]
 [ 2.887e-01 -5.000e-01  5.774e-01  2.887e-01  5.000e-01]]
```

The output of the algorithm is as follows for the following five cases:

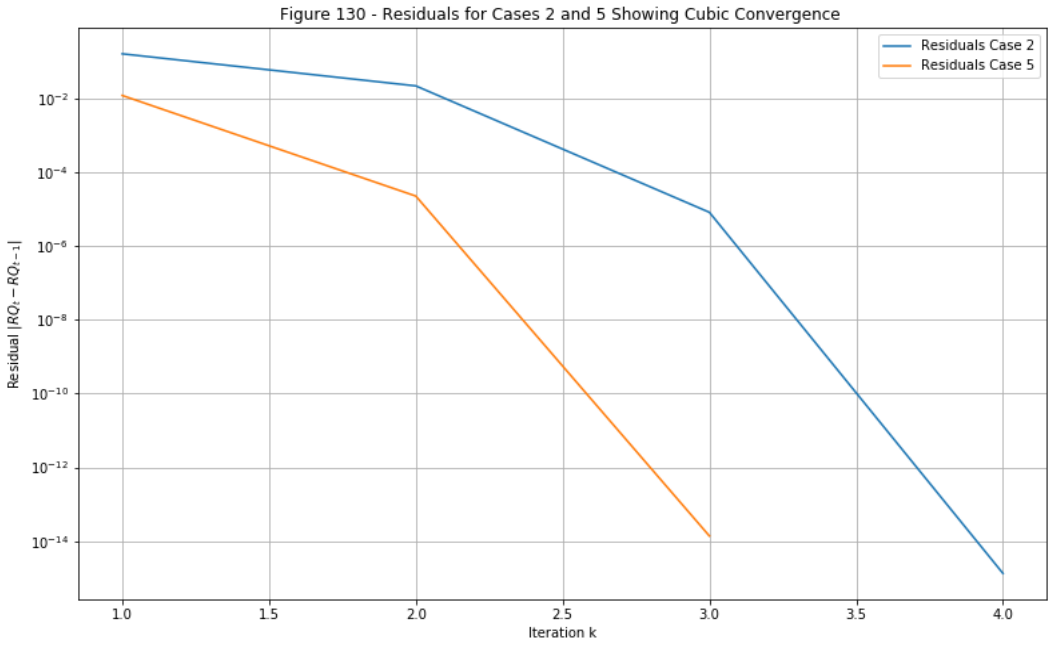
```
Case 1: x = [1 1 1 1 1]

 RQ = 0.267949
 V = [0.289 0.5   0.577 0.5   0.289]
 No. Iterations till Convergence = [0.289 0.5   0.577 0.5   0.289]
 Array of accuracy attained at each Iteration:
 [1.312e-01 8.747e-04 2.180e-10 0.000e+00]
 Raleigh Quotients:
```

```
 [0.2688238502 0.2679491926 0.2679491924 0.2679491924]

Case 2: x = [-1  4 -2  3 10]

 RQ = 2.000000
 V = [ 5.774e-01 -1.134e-24 -5.774e-01  4.983e-24  5.774e-01]
 No. Iterations till Convergence = [ 5.774e-01 -1.134e-24 -5.774e-01
                                     4.983e-24  5.774e-01]
 Array of accuracy attained at each Iteration:
 [1.627e-01 2.196e-02 8.211e-06 1.332e-15]
 Raleigh Quotients:
 [1.9780362392 1.9999917888 2.           2.          ]

Case 3: Fourth eigenvector of A
 : x = [0.289 0.5   0.577 0.5   0.289]

 RQ = 0.267949
 V = [-0.289 -0.5   -0.577 -0.5   -0.289]
 No. Iterations till Convergence = [-0.289 -0.5   -0.577 -0.5   -0.289]
 Array of accuracy attained at each Iteration:
 [5.551e-17]
 Raleigh Quotients:
 [0.2679491924]

Case 4: Fourth eigenvector of A + some noise
 : x = [0.179 0.459 0.645 0.539 0.272]

 RQ = 0.267949
 V = [-0.289 -0.5   -0.577 -0.5   -0.289]
 No. Iterations till Convergence = [-0.289 -0.5   -0.577 -0.5   -0.289]
 Array of accuracy attained at each Iteration:
 [2.675e-02 1.323e-05 2.998e-15]
 Raleigh Quotients:
 [0.2679624267 0.2679491924 0.2679491924]

Case 5: Fourth eigenvector of A + some more noise
 : x = [1.553 0.945 0.428 1.763 1.452]

 RQ = 0.267949
 V = [-0.289 -0.5   -0.577 -0.5   -0.289]
 No. Iterations till Convergence = [-0.289 -0.5   -0.577 -0.5   -0.289]
 Array of accuracy attained at each Iteration:
 [3.324e-01 1.869e-01 2.121e-02 1.763e-05 1.010e-14]
 Raleigh Quotients:
 [0.4760616283 0.289173927  0.2679668247 0.2679491924 0.2679491924]
```

```
FINISHED OUTPUT FOR PART 1.3
```

The output shows that all five cases converge to an eigenvalue/eigenvector pair. This is to be expected, as it was shown in lectures the the

The convergence of the solution is spectacular. We proceed to plot the residuals $|RQ_t - RQ_{t-1}|$ for all iterations until convergence for cases II and V:



As we can see, the residuals decay extremely rapidely and the accuracy of the solution becomes very high quickly. This is to be expected, as it was shown that the Rayleigh Quotient Algorithm behaves with cubic convergence.

1.4 Properties of Eigenvalue Computed

From the output of 1.3, we can see that the RQI algorithm always converges to an eigenvalue/eigenvector pair. In case, 3 of the output, we input one of the eigenvectors of A into the algorithm, and from the theory, the algorithm would have already converged. Due to rounding error, one RQ coefficient, which was the eigenvalue corresponding to the eigenvector inputted.

In case 4, we added a small amount of noise to the input of case 3, and we ran the RQI on it. The eigenvalue computed was indeed the same as in case 3. We can begin to hypothesise that the eigenvalue computed by RQI may be the one corresponding to the eigenvector 'closest' to the input vector. To check this hypothesis, in case 5 we added even more noise to the input of case 3, and indeed it converged to the same eigenvector.

It turns out, that due to the numerical instability of the RQI, the computed eigenvalue will correspond to the eigenvalue that is closest to the input vector, provided that the input vector is 'close' enough, i.e.

$$\min_j \|x_0 - v_j\| < \tau$$

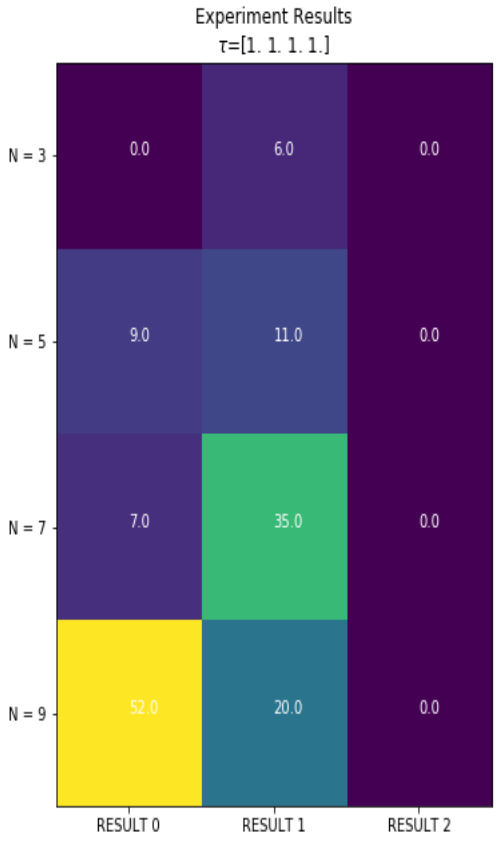
to the 'closest' eigenvector for some threshold τ, the algorithm will converge to the λ_j associated with v_j.

Design of the Experiment:

The null hypothesis is that any input vector to RQI will converge to the closest eigenvalue/eigenvector pair of A.

1. We begin with v_0, λ_0 as one of the eigenvectors/eigenvalue pairs of A as the input to the RQI. Set $u_k = v_0$.
2. Move the v_k further from the eigenvector by increasing the component orthogonal to the vector, call it u_{k+1} and input this into the RQI again.
3. If the output of the RQI is equal to λ_0, go back to 2, else we compute $\|u_{k+1} - v_j\|$ for all eigenvectors of A, and we let $\tau = \|u_{k+1} - v_0\|$, i.e. the distance with the initial eigenvector.
4. If $\tau = \min_j \|u_{k+1} - v_j\|$, then we can reject the null hypothesis. This implies that for inputs sufficiently 'far' from the closest eigenvector, the RQI will not converge to the closest eval/evec pair, and we have a threshold τ for what this cutoff distance is for that initial vector.

The results from the experiment can be summarised here. They signify: RES 0 - v_0 is still closest to u_{k+1}. RES 1 - u_{k+1} is closest to vector in the orthogonal direction that it was being pushed into. RES 2 - u_{k+1} is now closest to neither of these two vectors.



The experiment was carried out for different N looping through all pair of eigenvectors, noting what the result is and at what τ STEP 4 in the algorithm was reached.

There are two important results to note here. First, this experiment suggests that the 'loss of convergence' to v_0 occurs at $\tau = 1$, i.e. the input is outside the unit sphere to the eigenvector. Second, as N gets larger, RES 0 becomes larger.

1.5 Comments on QR Factorisation of A

The output for the QR Factorisation for N = 5 is the following.

```
The Matrix Q:
 [[-0.894 -0.359 -0.195 -0.123  0.135]
 [ 0.447 -0.717 -0.39  -0.246  0.27 ]
 [-0.     0.598 -0.586 -0.369  0.405]
 [-0.    -0.     0.683 -0.492  0.539]
 [-0.    -0.    -0.     0.739  0.674]]

The Matrix R:
 [[-2.236  1.789 -0.447  0.     0.   ]
 [ 0.    -1.673  1.912 -0.598  0.   ]
 [ 0.     0.    -1.464  1.952 -0.683]
 [ 0.     0.     0.    -1.354  1.969]
 [ 0.     0.     0.     0.     0.809]]
```

The output for the QR Factorisation for N = 7 is the following.

```
The Matrix Q:
 [[-0.894 -0.359 -0.195 -0.123 -0.085 -0.062  0.085]
 [ 0.447 -0.717 -0.39  -0.246 -0.17  -0.124  0.169]
 [-0.     0.598 -0.586 -0.369 -0.254 -0.186  0.254]
 [-0.    -0.     0.683 -0.492 -0.339 -0.248  0.338]
 [-0.    -0.    -0.     0.739 -0.424 -0.31   0.423]
 [-0.    -0.    -0.    -0.     0.777 -0.372  0.507]
 [-0.    -0.    -0.    -0.    -0.     0.806  0.592]]

The Matrix R:
 [[-2.236  1.789 -0.447  0.     0.     0.     0.   ]
 [ 0.    -1.673  1.912 -0.598  0.     0.     0.   ]
 [ 0.     0.    -1.464  1.952 -0.683  0.     0.   ]
 [ 0.     0.     0.    -1.354  1.969 -0.739  0.   ]
 [ 0.     0.     0.     0.    -1.286  1.979 -0.777]
 [ 0.     0.     0.     0.     0.    -1.24   1.985]
 [ 0.     0.     0.     0.     0.     0.     0.676]]
```

We can see a clear pattern here. The matrix Q is in upper-Hessenberg form, i.e. upper-triangular plus one band on the 2nd lower diagonal. The matrix R is an upper-triangular matrix, more specifically it contains two upper bands on the 2nd and 3rd upper diagonals.

We will now attempt to prove that the matrices Q & R will always have this structure for any

N given the the matrix in question. We begin by reminding ourselves of the QR Factorisation algorithm using Householder reflections.

Algorithm 2: Householder QR Factorisation

for $k = 1, 2, \ldots, n$ **do**
  (1) $x := A_{k:m,k}$
  (2) $v_k := \text{sign}(x_1) \|x\|_2 e_1 + x$
  (3) $v_k := v_k / \|v_k\|_2$
  (4) $A_{k:m,k:n} = A_{k:m,k:n} - 2v_k(v_k^* A_{k:m,k:n})$
end

In the case of this question, with A being Toeplitz tridiagonal, it is important to note that steps 1, 2, 3 will always be very similar. This is because for $k < n - 1$, extracting the vector $A_{k:m,k}$ leads to a vector of the form $x = (2, 1, 0, \ldots, 0)^T$, where the number of zeros in the vector will be equal to $n - k - 1$.

This means, after steps 2 and 3, $v_k = \left(\frac{2+\|x\|_2}{\|x\|_2}, \frac{1}{\|x\|_2}, 0, \ldots, 0\right)^T$ and the householder reflection for iteration k will consist of a 2x2 block in the upper-left corner and 1s on the remaining diagonal entries, $H = I - 2v_k v_k^*$.

Therefore, the action of hitting the matrix $A_{k:m,k:n}$ with H on the left will have the effect of modifying the first row of $A_{k:m,k:n}$ by setting the last $(n - k) - 3$ elements equal to zero, i.e. the first elements will the only non-zero elements. After iterating, we find that the resulting matrix A will have been reduced to an upper-triangular matrix with non-zero entries on the main diagonal, 2nd and 3rd upper diagonals only.

At the last 2 iterations, the Householder algorithm will only affect the entries at the bottom 2x2 corner, and we can conclude that the resulting matrix A (now R) is upper-triangular with non-zero entries on the main diagonal, 2nd and 3rd upper diagonals only.

To obtain the structure of Q, we first note that the each consecutive householder reflector contains the same 2x2 moving matrix-block window:

$$H_{k:k+2,k+k:2} = \begin{pmatrix} 1 - 2a^2 & 2ab \\ 2ab & 1 - 2b^2 \end{pmatrix}$$

and so the product of all of these individual householder reflectors (to make the matrix Q) forms a matrix Q with upper Hessenberg form.

Computing QR by householder reflections is prefered for dense matrices. In fact, computing QR by householder reflections for sparse matrices is notoriously bad, as the reflections require indexing across whole rows/columns of the matrix and usually creates entries in places where zeros used to live in the sparse matrix. In other words, Householder reflections makes the matrix denser at each iteration. This is very expensive to do for a sparse matrix, and so a better scheme such as Givens rotations should be used for Householder.

1.6 Pure QR Algorithm

The code used in this question lies in the file `question1.py` in the functions `question16, pureQR`.

We stick with N = 5 and we print the matrix after pureQR algorithm has run; it halts after all the diagonal elements have stopped converging.

```
The matrix A after pure QR:

[[ 3.73e+00 -1.06e-04 -3.51e-16 -2.87e-16 -8.17e-18]
 [-1.06e-04  3.00e+00 -7.32e-08  4.02e-16  1.14e-16]
 [ 0.00e+00 -7.32e-08  2.00e+00 -3.11e-13 -2.82e-16]
 [ 0.00e+00  0.00e+00 -3.11e-13  1.00e+00  1.92e-16]
 [ 0.00e+00  0.00e+00  0.00e+00  3.23e-25  2.68e-01]]

Iterations taken = 43
```

Observations:

- The diagonal entries converge towards the eigenvalues of A after each iteration. In fact, the output above contains the eigenvalues of A for N=5 (see part 1.3 output).
- The entries on away from the diagonal converge towards zero. The entries away from the diagonal are on average smaller as we progress through the columns from left to right.

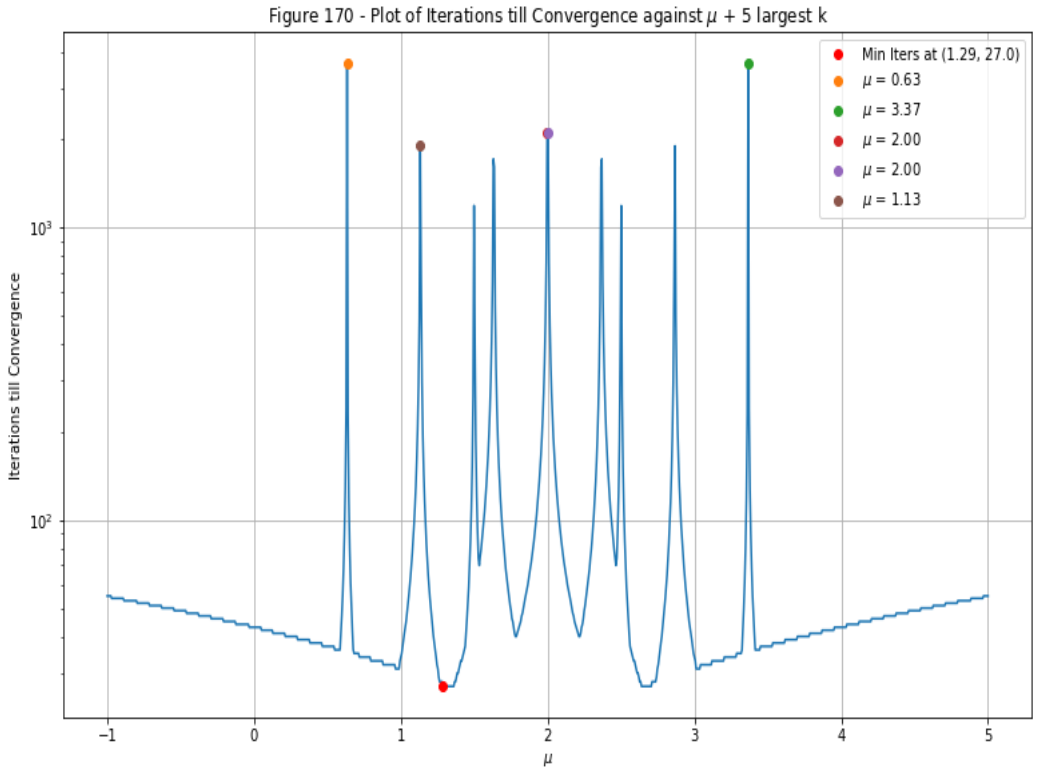
These observations are to be expected. From lectures, we know that the when the QR algorithm is performed to A, then the diagonal values are actually Rayleigh Quotients which converge to the eigenvalues of A with each iteration. As to why the entries away from the diagonal are equal to zero, this we also saw in lectures, and it is due to the fact that these entries are generalised Rayleigh Quotients of orthogonal eigenvectors of A. As they converge, these become orthogonal, therefore must converge to zero.

1.7 QR Algorithm with Constant Shift μ

We implement this and we check that the output works for the same matrix A as 1.6 and we let $\mu = 0.5$.

```
The matrix A after shifted QR (mu=0.5):

[[ 3.73e+00 -9.47e-05 -9.52e-18 -1.45e-16 -4.81e-16]
 [-9.47e-05  3.00e+00 -1.69e-08  9.71e-17  4.13e-16]
 [ 0.00e+00 -1.69e-08  2.00e+00 -1.90e-17 -2.30e-16]
 [ 0.00e+00  0.00e+00  6.07e-18  1.00e+00 -5.86e-13]
 [ 0.00e+00  0.00e+00  0.00e+00 -5.86e-13  2.68e-01]]

Iterations taken = 37
```

It is reassuring that the diagonals of the matrix have converged to the eigenvalues, and with a constant shift of 0.5, the iterations have decreased by 6. We will now explore the relationship between rate of convergence (iterations till convergence) and the constant shift parameter.



From the plot, we see that the highest rate of convergence is achieved at $\mu \approx 1.3$, with a number of iterations until convergence of 27. This is a decrease of 16 iterations compared to the pure QR. We also see that for some values of μ, the algorithm converges very slowly. We have pointed out the values of μ for which the algorithm took the longest to converge.

1.8 QR Algorithm to Random Tridiagonal Symmetric Matrix

The following output was observed:

```
The matrix S:
[[ 5.997  0.244  1.183  3.572  4.061]
 [ 0.244  2.511 -3.987 -0.112  0.24 ]
 [ 1.183 -3.987  8.385 -1.214  1.27 ]
 [ 3.572 -0.112 -1.214  5.579  1.902]
 [ 4.061  0.24   1.27   1.902  3.965]]

The matrix S after pure QR:
[[ 1.205e+001 -2.302e-004 -1.946e-015  1.015e-015 -1.516e-015]
 [-2.302e-004  1.049e+001 -1.655e-016  5.792e-016  7.492e-016]
 [-3.626e-032 -2.295e-028  3.247e+000 -4.353e-016  5.228e-016]
 [ 1.056e-070  8.596e-068  5.655e-040  6.504e-001  7.385e-017]
 [ 1.711e-249  1.506e-246  2.841e-218  1.261e-179  4.041e-004]]

Iterations taken = 56
The matrix S3:
[[ 5.997  0.244  0.     0.     0.   ]
 [ 0.244  2.511 -3.987  0.     0.   ]
 [ 0.    -3.987  8.385 -1.214  0.   ]
 [ 0.     0.    -1.214  5.579  1.902]
 [ 0.     0.     0.     1.902  3.965]]

The matrix S3 after pure QR:
[[ 1.066e+01  3.914e-17 -6.796e-17 -1.270e-16  9.162e-16]
 [ 9.379e-22  6.660e+00 -1.602e-04 -2.551e-16  4.941e-16]
 [ 0.000e+00 -1.602e-04  6.001e+00  1.670e-17 -1.000e-18]
 [ 0.000e+00  0.000e+00 -1.381e-38  2.701e+00 -1.000e-15]
 [ 0.000e+00  0.000e+00  0.000e+00  2.742e-87  4.138e-01]]

Iterations taken = 108
Average Iterations for pure QR Matrix S after 100 times: 36
Average Iterations for pure QR Matrix S3 after 100 times: 70
```

Observations

- The diagonal values of S & S3 are different after pure QR has been applied. This is not a surprise, as zeroing all the values away from the main three diagonals changes the eigenvalues of the matrix.
- The iterations taken for the pure QR algorithm to converge is approximately double for the matrix S3 compared to the matrix S.

2 GMRES & Preconditioning

2.1 Structure, Values of A and b

We can write the linear system described in the question as a matrix-vector system $Ax = b$, where $A \in \mathbb{R}^{(N^2,\ N^2)}, b \in \mathbb{R}^{N^2}$ and x is as described in the question.

The vector b has values

$$b = \frac{1}{(N+1)^2}(f_{1,1}, f_{2,1}, \ldots, f_{N,1}, f_{1,2}, f_{2,2}, \ldots, f_{N,2}, \ldots, f_{1,N}, \ldots, f_{N,N})^T.$$

We can write the structure of the matrix A as a block-matrix, where blank spaces denote zeroes in the matrix,

$$A = \begin{pmatrix} B & -I_N & & \\ -I_N & \ddots & \ddots & \\ & \ddots & \ddots & -I_N \\ & & -I_N & B \end{pmatrix}$$

where I_N is the NxN identity matrix and B is another tridiagonal matrix

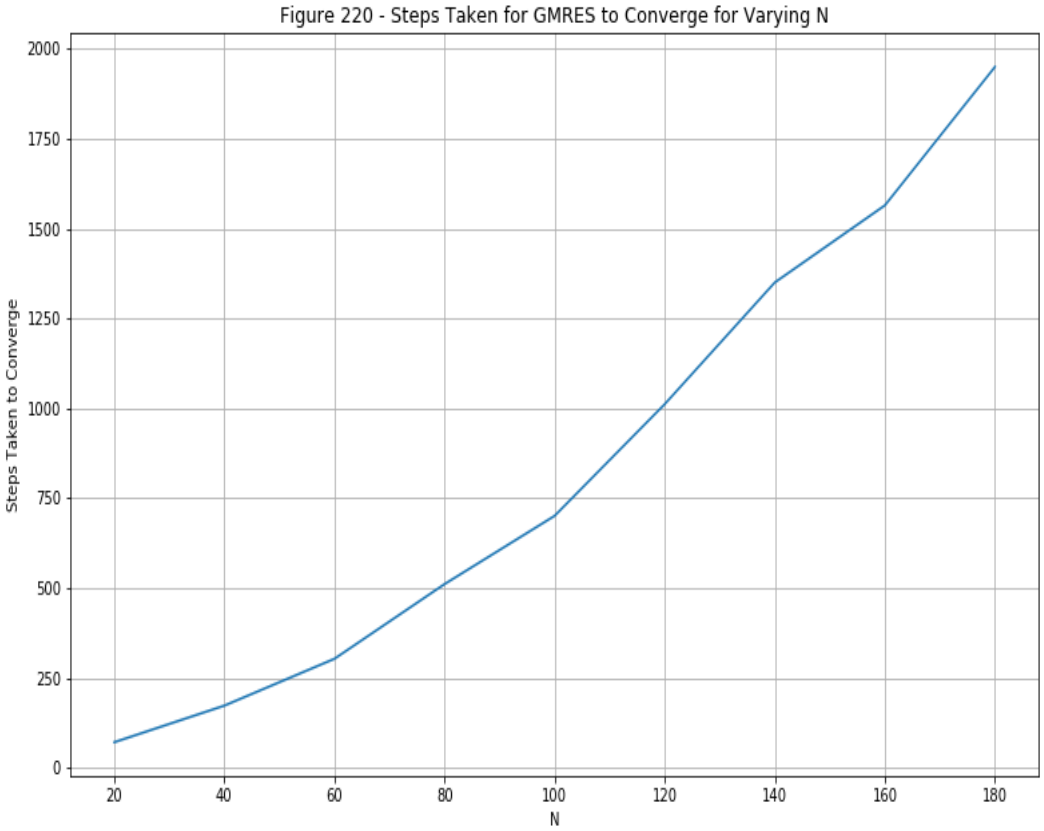
$$B = \begin{pmatrix} 4 & -1 & & \\ -1 & \ddots & \ddots & \\ & \ddots & \ddots & -1 \\ & & -1 & 4 \end{pmatrix}.$$

2.2 GMRES without Preconditioning

The code used to construct the matrix A can be found inside of the file `question2.py` in the function `construct_matrix_A`.

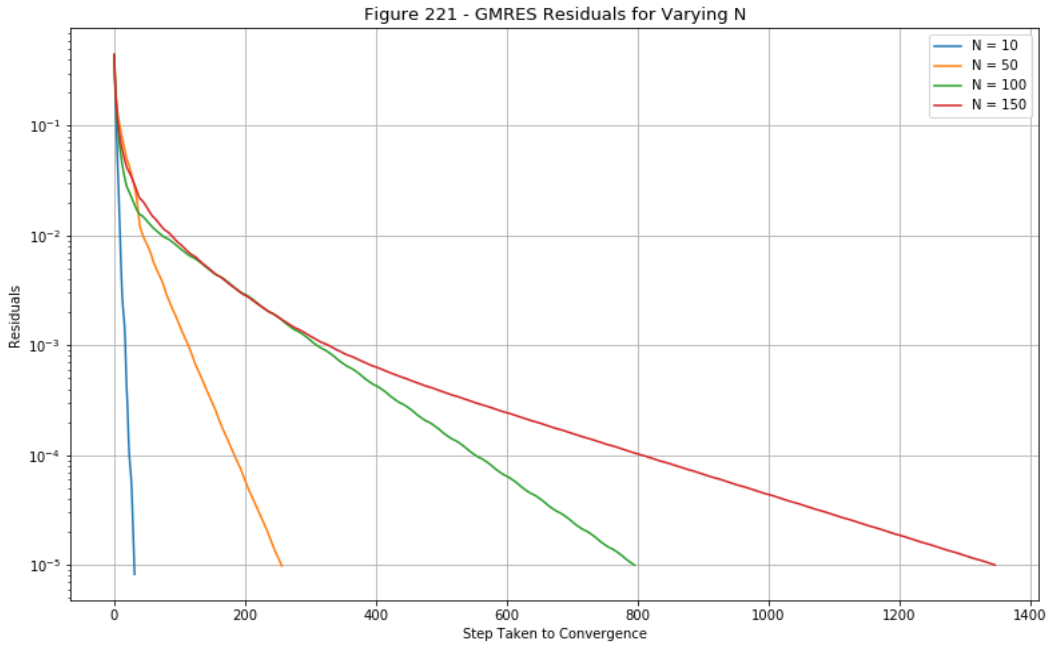
For randomly generated b, we use the function `scipy.sparse.gmres` to solve the equation using GMRES.

We first iterate over the values $N = [20, 40, 60, 80, 100, 120, 140, 160, 180]$ and plot the number of steps it takes the algorithm to convergence with a default tolerance of 10^{-5}.



From this figure, we can see that the steps taken to converge grows quadratically with N. There is an inversely quadratic relationship between the convergence rate and N. This is to be expected, as the rate of convergence for GMRES depends quadratically on the condition number $\kappa(A)$ of the matrix, and for the matrix A in question, $\kappa(A)$ increases linearly of N.

The next plot illustrates the size of the residuals computed at each iteration for different sizes of N.



As we can see from the graph, for any N, the rate of convergence is larger at the beginning of the algorithm and it progressively gets worse until it reach a steady rate. There are methods to keep that initial fast rate of convergence at the same rate, such as using restarts or using a preconditioner to the algorithm.

2.3 Splitting of the Algorithm

If the algorithm does converge, then we can take $x^n = x^{x+\frac{1}{2}} = x^{n+1} = x$, and so equation (7) becomes

$$\begin{aligned} & (\alpha I + M)x = (\alpha I - N)x + b \\ \implies & (M + N)x = b \\ \implies & Ax = b \end{aligned}$$

and equation (8) becomes

$$\begin{aligned} & (\alpha I + N)x = (\alpha I - M)x + b \\ \implies & (N + M)x = b \\ \implies & Ax = b \end{aligned}$$

using the fact that $A = N + M$. This means that at convergence, we are always computing the solution of $Ax = b$ in both equations (7) and (8), and hence the algorithm converges to the solution of $Ax = b$ trivially.

2.4 Splitting Into N Independent Systems

We begin this section by talking about the structure of the matrices N and M. Both of the matrices inherit their structures from the matrix A. Both matrices N and M can be expressed in block-matrix format. The structure of the matrix M is simple, it is in block-diagonal format

$$\begin{pmatrix} C & & & \\ & \ddots & & \\ & & \ddots & \\ & & & C \end{pmatrix}$$

and the matrix C is

$$C = \begin{pmatrix} 2 & -1 & & \\ -1 & \ddots & \ddots & \\ & \ddots & \ddots & -1 \\ & & -1 & 2 \end{pmatrix}.$$

The matrix N has the following format: 2s on the main diagonal and -1s on the (N+1)th and -(N+1)th diagonals.

We can easily split Equation (7) into N independent tridiagonal systems of size N x N. Once the RHS of the equation has been computed, we can express the equation in block-matrix/vector format:

$$\left(\gamma I + \begin{pmatrix} C & & & \\ & \ddots & & \\ & & \ddots & \\ & & & C \end{pmatrix}\right) \begin{pmatrix} x_{i1}^{n+1/2} \\ \vdots \\ \vdots \\ x_{iN}^{n+1/2} \end{pmatrix} = \begin{pmatrix} RHS_1 \\ \vdots \\ \vdots \\ RHS_N \end{pmatrix}$$

and then we note that each of the $x_{ij}^{n+1/2} = (x_{1j}^{n+1/2}, ..., x_{Nj}^{n+1/2})^T$. As the matrix C is tridiagonal, the above can be split into N independent tridiagonal matrix systems of size N x N, each corresponding to a set of equations $(\gamma I + C)x_{ij}^{n+1/2} = RHS_j$.

To split Equation (8) into N independent tridiagonal systems of size N x N we first note that the action of the matrix N on the vector x is the same as the action of the matrix M on a different vector $\hat{x}$, which corresponds to a different ordering of the original problem.

$$\hat{x} = (p_{1,1}, p_{1,2}, \ldots, p_{1,N}, p_{2,1}, p_{2,2}, \ldots, p_{2,N}, \ldots, p_{N,1}, \ldots, p_{N,N})^T$$

After computing the RHS of equation 8, we can permute the x^ns and $RHSs$ in the way described above to achieve a different equation $(\gamma I + M)\hat{x}^{n+1} = \hat{RHS}$.

This equation involves M, which is a block-diagonal matrix, and similarly to the method in Equation (7) we can split this up into N independent tridiagonal matrix systems of size N x N, each corresponding to a set of equations $(\gamma I + C)\hat{x}_{ij}^{n+1} = R\hat{H}S_j$.

Once we have computed the solution $\hat{x}^{n+1}$, we can recover the solution to Equation (8) by permuting the indexes back, i.e.

$$x^{n+1} = (\hat{x}_{1,1}^{n+1}, \hat{x}_{2,1}^{n+1}, \ldots, \hat{x}_{N,1}^{n+1}, \hat{x}_{1,2}^{n+1}, \hat{x}_{2,2}^{n+1}, \ldots, \hat{x}_{N,2}^{n+1}, \ldots, \hat{x}_{1,N}^{n+1}, \ldots, \hat{x}_{N,N}^{n+1})^T$$

This completes the iteration using two sets of N independent tridiagonal matrix systems of size N x N.

Estimation of FLOPS of one iteration

The number of FLOPS used in each equation is the same, and we will only concentrate on one. For this part, we will not take into account the FLOPS taken to permute the $\hat{x}$s, as we can implement this by keeping two sets of solutions $x, \hat{x}$ and updating each at the end of an iteration. Also, we can precompute matrix the additions of γI to various matrices. Once these efficiencies are reached, at each iteration we will have, for each equation:

1. To compute RHS, we need to perform a matrix-vector multiplication involving a sparse tridiagonal matrix of size (N^2, N^2), which by taking advantage of the sparsity of the system can be done in approx. $3N^2$ FLOPS and an addition by the vector b also of size N^2, which is done in N^2 FLOPS. Therefore, this step is approximately $4N^2$ FLOPS.
2. We then need to compute the solution to each of the N systems, each of size N x N. Taking advantage of the sparsity of the matrix, the number of FLOPS that it takes to solve each system will be approximately 5N. This step will be approximately $5N^2$.

Putting it all together, we get an approximate number of FLOPS of $9N^2$ to obtain the solution to each of the equations (7) & (8), so the whole iteration takes approximately $18N^2$ FLOPS, which makes it $O(N^2)$.

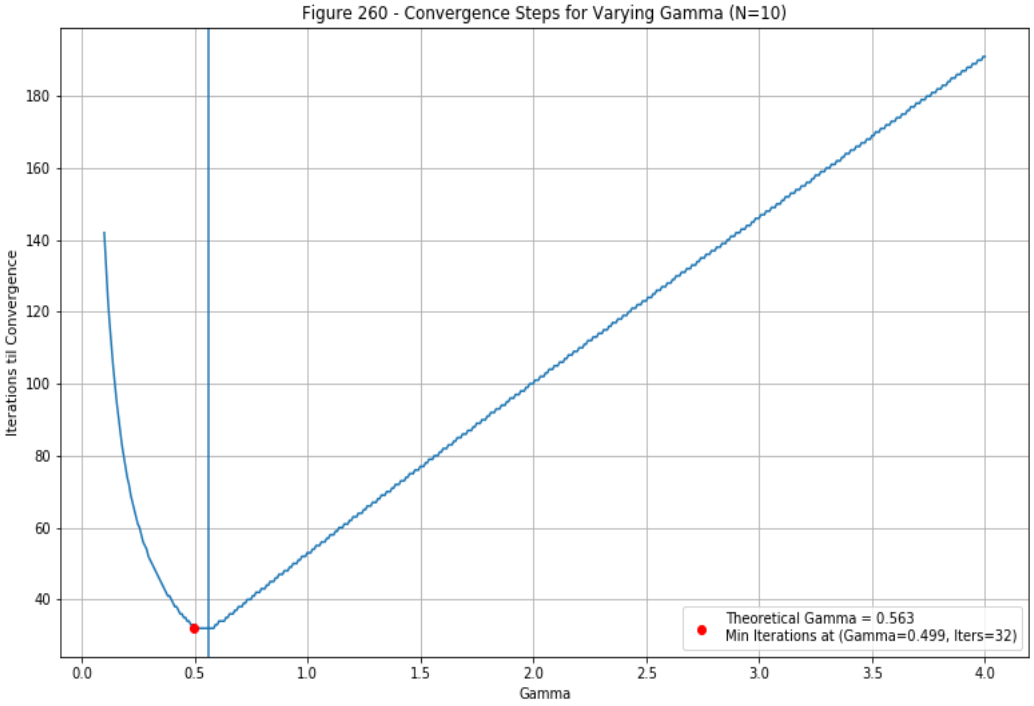
2.5 Implementation of Iterative Method

The implementation of the RQI algorithm is inside the file `question2.py` under the `alternative_iterative_method` function.

2.6 Optimal Convergence Rate

We are given the result that the optimial convergence rate for this algorithm is obtained when $\gamma = \sqrt{\mu_{min}\ \mu_{max}}$, where μ_{max} is the maximum over magnitudes of eigenvalues for both M and N.

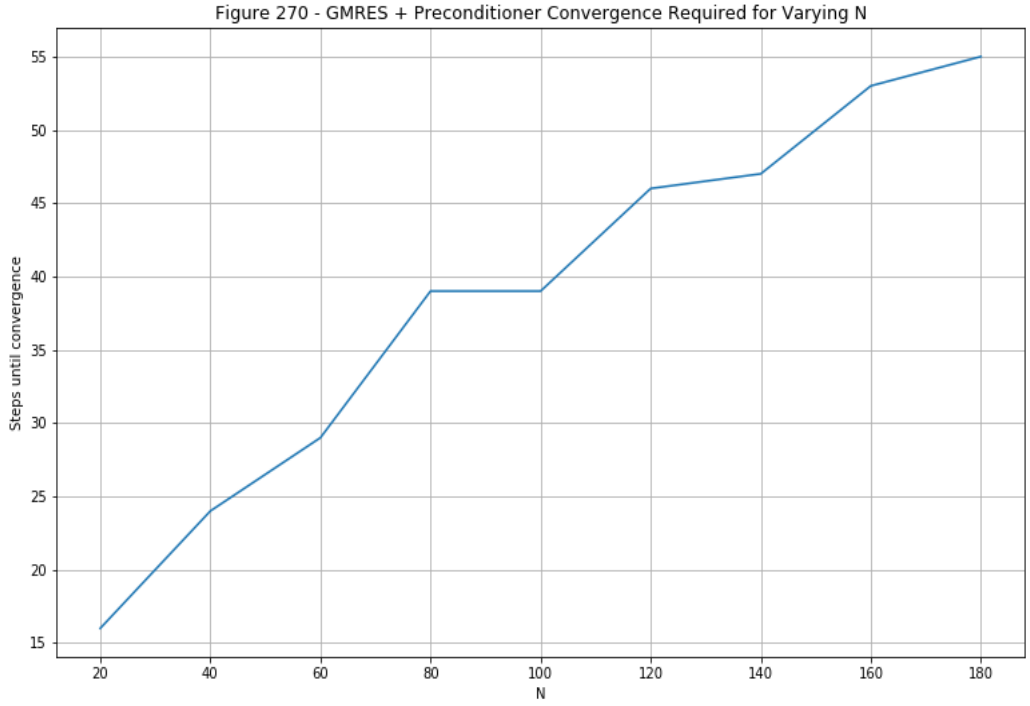
To verify this using our code, we plot a graph of iterations taken for the algorithm to converge against gamma and see whether we find a clear minimum.



For N=10, we have computed the theoretical optimal γ, which is at 0.563. The graph has a clear minimum at around this value, with values of γ of around 0.5-0.6 achieving the lowest iterations of approx 30. We plot the value of the theoretical γ computed using the maximum eigenvalues of M and N.

2.7 GMRES with Preconditioner

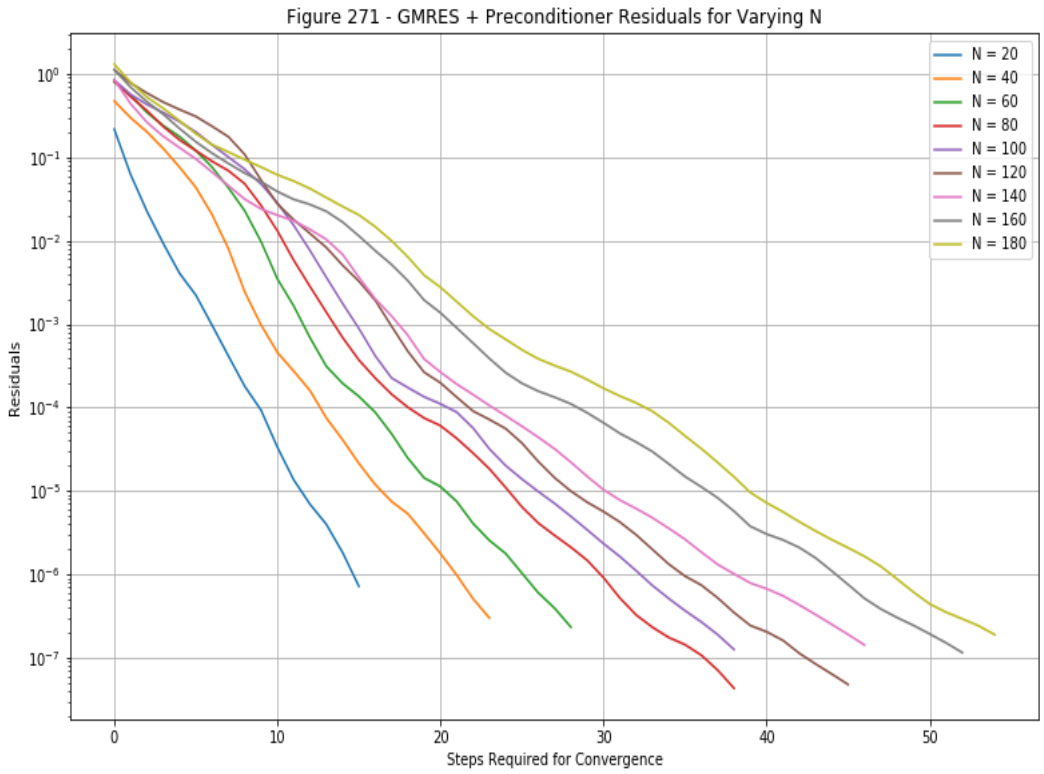
The implementation of the function `iterate` is located under as a variable inside the function `question27`. The implementation uses Scipy's spsolve routines to speed up the computations as the pre-conditioner. We present the same plots shown in Question 2.2, but this time for GMRES with pre-conditioner.



We can see immediately that the preconditioner has greatly improved the GMRES rate of convergence. The relationship between rate of convergence and N is an inversely linearly proportional one, as the rate of convergence is still determined by the condition number of the input matrix. Comparing the result found in 2.2 with this, we can summarise some key results here:

N/Method	**GMRES**	**GMRES + PreC**	**Ratio**
20	80	16	5
100	700	39	18
180	1900	55	35

This shows that by adding a preconditioner, we have been able to bring the rate of convergence from depending on N quadratically to linearly.



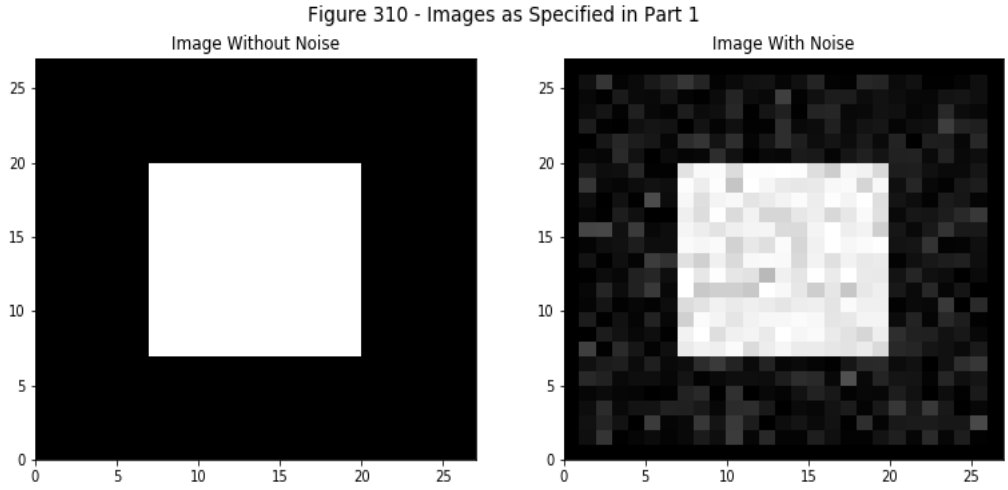
We can see that the residuals are converge at a similar exponential rate throughout the whole iteration. This is contrast with the GMRES without preconditioner residuals, where the convergence was much faster towards the beginning of the algorithm but greatly slowed down as iterations progressed.

3 Image Denoising

3.1 Setting Initial Image

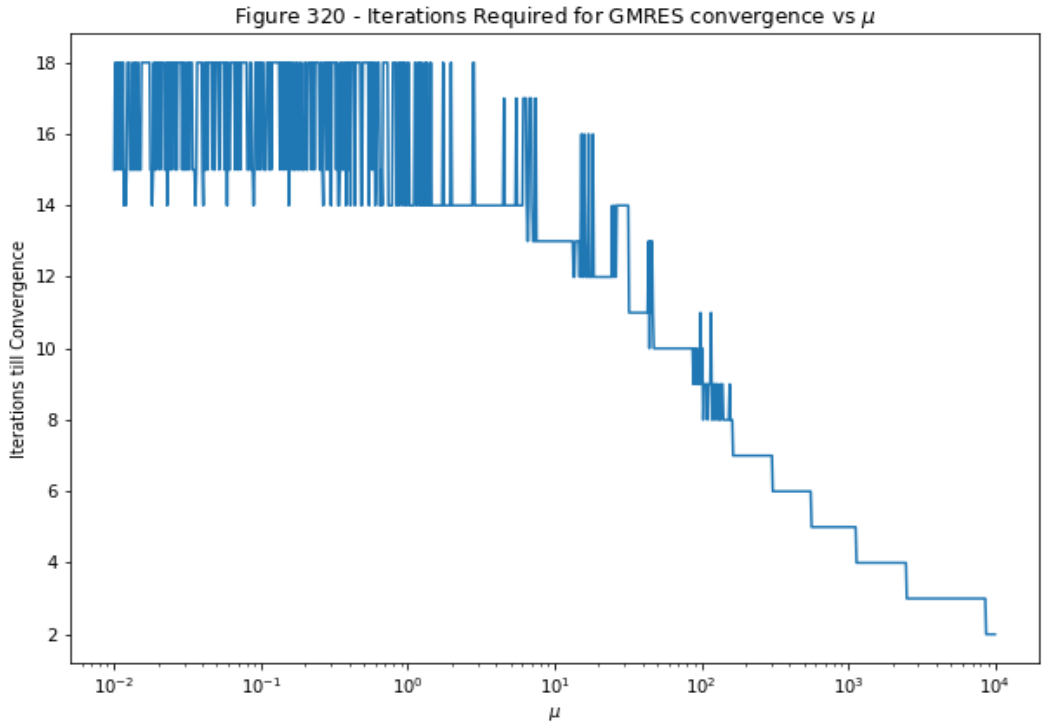
The code used to generate the initial image with some noise is the following:

```
IMAGE = np.zeros((N+2, N+2))
B1 = int(np.floor(N*0.75 - 1e-12))  # Add some noise to not be singular
B2 = B1 + 2
B1 = N - B1
IMAGE[B1:B2, B1:B2] = 1

# Adding noise:
IMAGE[1:-1, 1:-1] += np.random.normal(loc=0.0, scale=0.1, size=(N, N))
IMAGE[IMAGE > 1] -= 2*(IMAGE[IMAGE > 1] - 1)
IMAGE[IMAGE < 0] *= -1
```



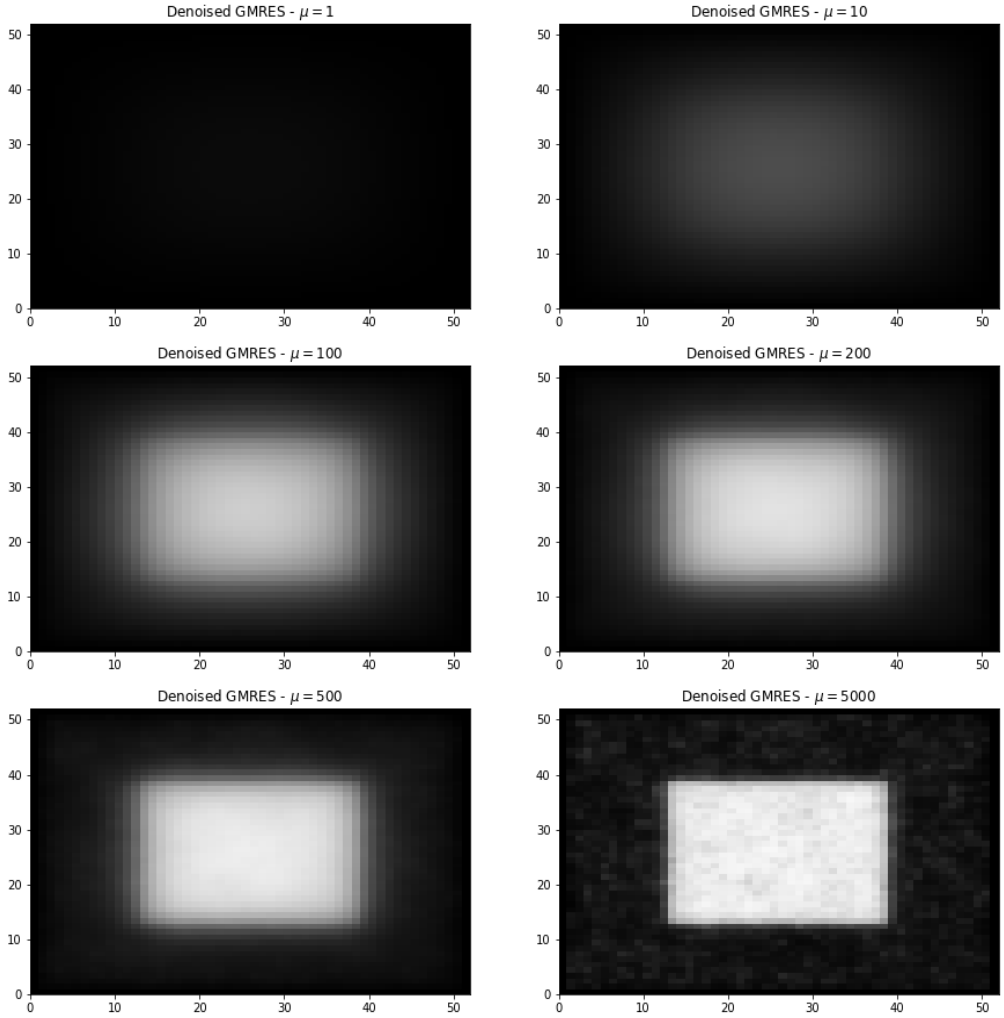
3.2 Adaptation of GMRES

We recognise the form of Equation (13) to be very similar to discretisation performed in Equation (3). We can easily adapt the GMRES solver by adding $\mu\Delta x^2$ to the diagonal of A and b, as well as $(\mu\Delta x^2)/2$ to the diagonals of the matrices M & N in the preconditioner of GMRES. The routine for denoising images using GMRES can be found inside the `denoise_GMRES` function.



Let k_μ denote the iterations until convergence for corresponding μ. Figure 320 shows the relationship between μ and k_μ. For $\mu < 10$, we can see that the algorithm iterates for between 14 to 18 iterations. For μ the algorithm converges faster, until a point where it converges so fast that $k_{10^4} = 2$. Figure 321 below contains the output of the denoising method for different values of μ.

Figure 321 - Visualising GMRES Denoising Method (N=50)

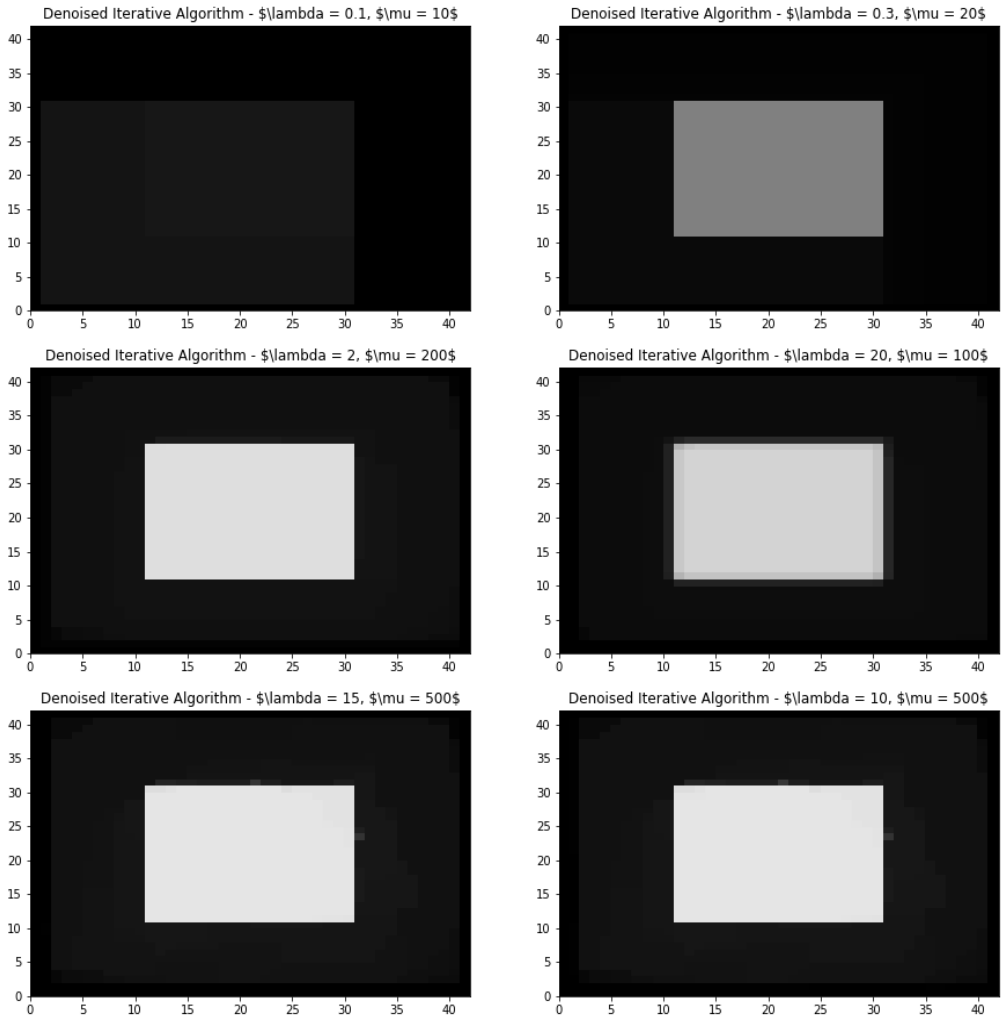


We can see that μ has a blurring + dimming effect on the image. For small μ, the image becomes very dim and smooth, losing the well-defined hard boundaries of the square. For large μ, the denoising is not very effective, and we begin to recover the original image containing noise. Overall, this algorithm seems to be prone to softening image edges a lot for all values of μ.

3.3 Alternative Iterative Algorithm for Denoising

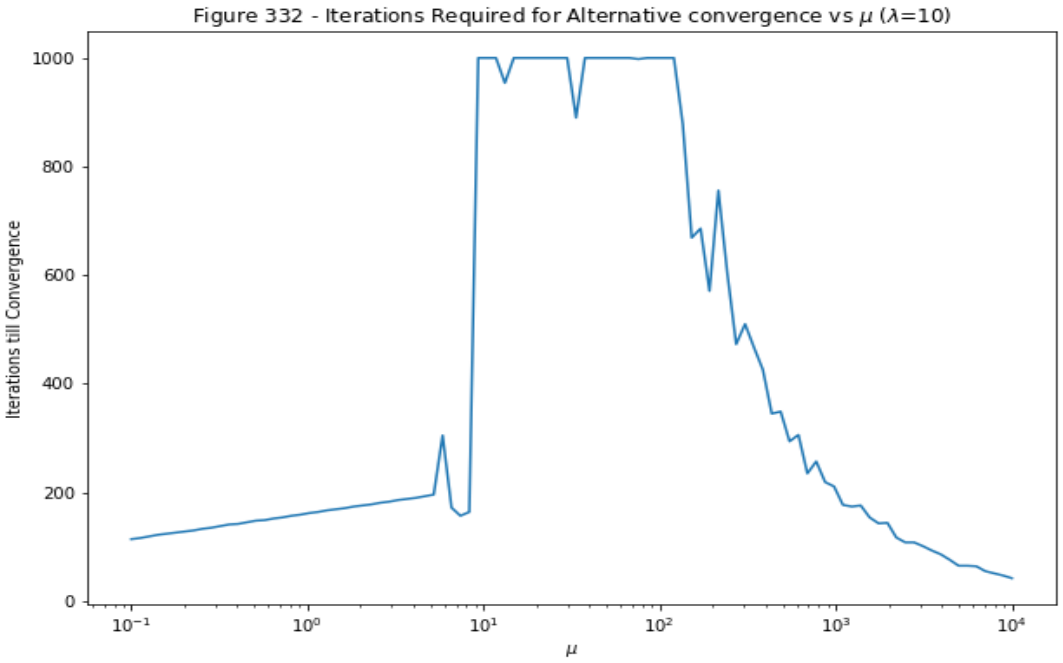
The implementation for this section can be found in the file `question3.py` with the function `denoise_iterative_method`.

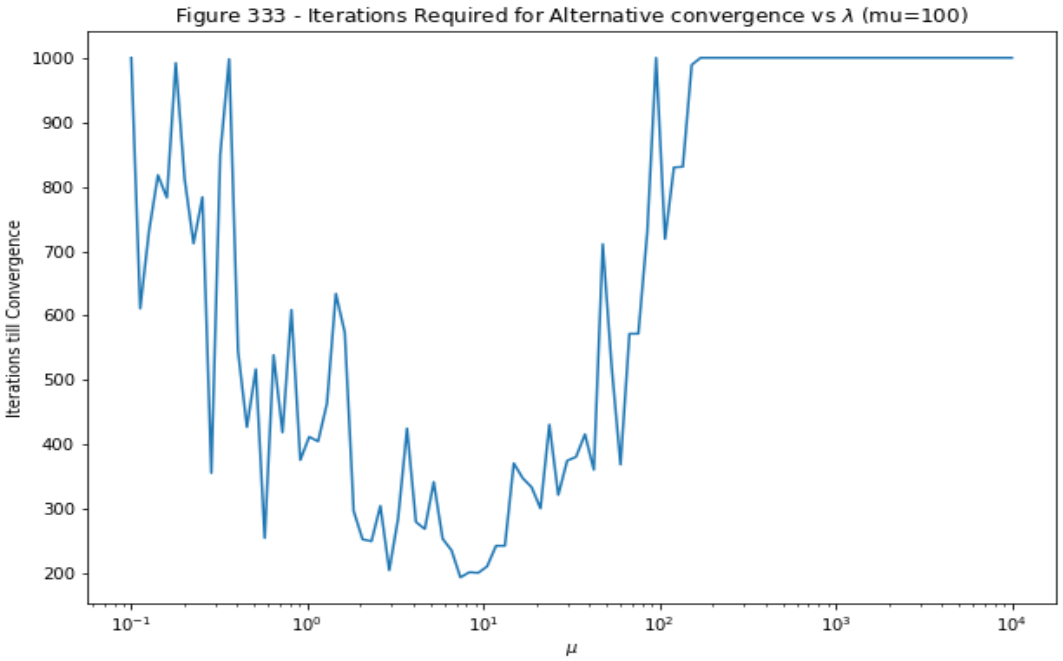
Figure 331 - Visualising Iterative Algorithm Denoising Method (N=50)



The alternative method seems to perform better than GMRES. It is able to cope with edges much better.

We now plot the convergence of the method vs λ and μ.





We can see that both parameters have a large influence in convergence.