Coursework 2 - Computational Linear Algebra

Marwan Riach, CID: 01349928

January 2021

To produce all figures in the report, simply run the respective $q_i.py$ files and to see whether the tests pass, run the *test.py* files.

1 Question One

1.1 a)

The specific algorithm for LU factorisation followed by forward and back substitution will be carried out in place in order to improve the memory efficiency of the algorithm. We proceed as we did for exercise 4.2 except that there is only one outer loop, given that the matrix is tridiagonal: the L matrix will have one subdiagonal non-zero (with the diagonals being equal to 1) and the U matrix will have one superdiagonal non-zero. Forward substitution is then followed by back substitution to solve for x in $Ax = b$.

$T \leftarrow A$
for $k = 1$ to $m - 1$ **do**
  $T_{k+1,k} \leftarrow \frac{T_{k+1,k}}{T_{k,k}}$
  $T_{k+1,k+1} \leftarrow T_{k+1,k+1} - T_{k+1,k} * T_{k,k+1}$
end for
$y_1 \leftarrow b_1$
for $k = 2$ to m **do**
  $y_k \leftarrow b_k - T_{k,k-1} * y_{k-1}$
end for
$x_m \leftarrow \frac{y_m}{T_{m,m}}$
for $k = m - 1$ to 1 **do**
  $x_k \leftarrow \frac{y_k - T_{k,k+1} * x_{k+1}}{T_{k,k}}$
end for

1.2 b)

The forward substitution can be merged with the first loop since they both involve iterating up to m (if the index of y is shifted by 1) and $T_{k+1,k}$ is known after the first line of the first for loop. Writing the algorithm as follows shows how LU factorisation can be merged with forward substitution:

$T \leftarrow A$
$y_1 \leftarrow b_1$
for $k = 1$ to $m - 1$ **do**
  $T_{k+1,k} \leftarrow \frac{T_{k+1,k}}{T_{k,k}}$
  $y_{k+1} \leftarrow b_{k+1} - T_{k+1,k} * y_k$
  $T_{k+1,k+1} \leftarrow T_{k+1,k+1} - T_{k+1,k} * T_{k,k+1}$
end for
$x_m \leftarrow \frac{y_m}{T_{m,m}}$
for $k = m - 1$ to 1 **do**
  $x_k \leftarrow \frac{y_k - T_{k,k+1} * x_{k+1}}{T_{k,k}}$
end for

1.3 c)

The merged algorithm's operations are dominated by the contents of the two loops which each go from 1 to $m - 1$ (albeit in different orders). Given that subtractions, multiplications and divisions are separate, we will treat the operations as individual. In the first for loop, there are a total of 5 operations for each k and in the second there are 3. Hence, the asymptotic operation count is:

$$N_{FLOPS} = \sum_{k=1}^{m-1} (3 + 5) = 8 \sum_{k=1}^{m-1} 1 = 8(m - 1) \sim 8m$$

If the structure of the matrix isn't exploited, then the asymptotic operation count is $\sim \frac{2m^3}{3}$ (from lectures), and so the algorithm we derived above is significantly faster ($O(m)$ vs $O(m^3)$). The forwards and backwards substitution for the general algorithm is $O(m^2)$ and so when forwards and backwards substitution is included in the total count for general matrices, the asymptotic operation count is unchanged (it is still $O(m^3)$).

1.4 d)

The algorithm implementing this code can be found in the *q1.py* file. It takes in as arguments c, d and b (the RHS vector) since this is all that is needed to determine the system of equations for the specific tridiagonal matrix with which we are working.

The tests check to see that the correct solution is given using this algorithm as compared to the official method which makes use of *np.linalg.inv* to solve $Ax = b$. This is carried out over matrices of 3 different sizes to validate the method.

Since all elements in the matrix A outside of the tridiagonal are 0, it is spatially inefficient to store all of these values. To improve spatial efficiency, a 3 x m matrix is used with the rows representing the superdiagonal, diagonal, and subdiagonal. In this instance, there are only 2 elements that are zero as opposed to $O(m^2)$. Here, the spatial complexity is $O(m)$ instead of $O(m^2)$.

2 Question Two

2.1 a)

We will use the fact that if a function is periodic, then so is its derivative; this will deal with the boundary cases for w when applying the central difference scheme which behaves in the same way as the boundary cases for u. The boundary cases are such that $u_i = u_{i+1}$ for all space steps i which are used at the peripheral cases.

Differentiating the second equation in (4) (from the assignment) twice with respect to space and plugging it into the first equation gives the following:

$$w^{n+1} - \frac{(\Delta t)^2}{4(\Delta x)^2} w_{xx}^{n+1} = w^n + \Delta t u_{xx}^n + \frac{(\Delta t)^2}{4} w_{xx}^n$$

The constant $C_1 = \frac{(\Delta t)^2}{4(\Delta x)^2}$ and f_i takes the following form:

$$f_i = w_i^n + \Delta t \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{(\Delta x)^2} + (\Delta t)^2 \frac{w_{i+1}^n - 2w_i^n + w_{i-1}^n}{4(\Delta x)^2}$$

2.2 b)

Writing the set of M equations in the form $Ax = b$, $b_i = f_i$ and $x_i = w_i^{n+1}$. The matrix A is constructed with the boundary conditions borne in mind.

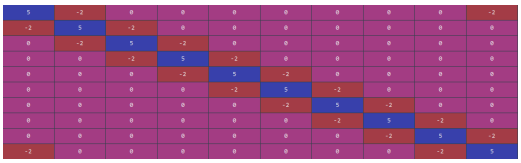
$$A = \begin{pmatrix} (1+2C_1) & -C_1 & 0 & \dots & -C_1 \\ -C_1 & (1+2C_1) & -C_1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -C_1 & 0 & \dots & -C_1 & (1+2C_1) \end{pmatrix}$$

Except for the top-right and bottom-left entries, this matrix is tridiagonal. The 2 entries that do not lie within the tridiagonal account for the periodicity of the problem. This is a sparse matrix.

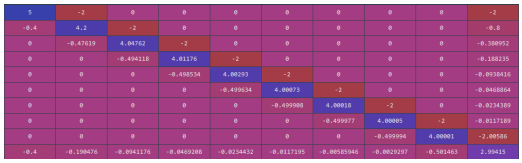
2.3 c)

The banded matrix structure relies on all entries below a certain subdiagonal or above a certain superdiagonal being zero. Since the top right and bottom left entries are non-zero, no banded matrix structure can be exploited since no banded matrix structure exists. Therefore, there is no advantage to using a banded matrix algorithm.

LU in place from cla utils was modified under the name of "LU inplace mod" so that the upper right and bottom left entries of the matrix were tracked at each iteration in the LU algorithm and returned in addition to the in place matrix.



(a) Original Matrix



(b) LU decomoposition of Matrix

Figure 1: LU decomposition when $M = 10$ and $C_1 = 2$

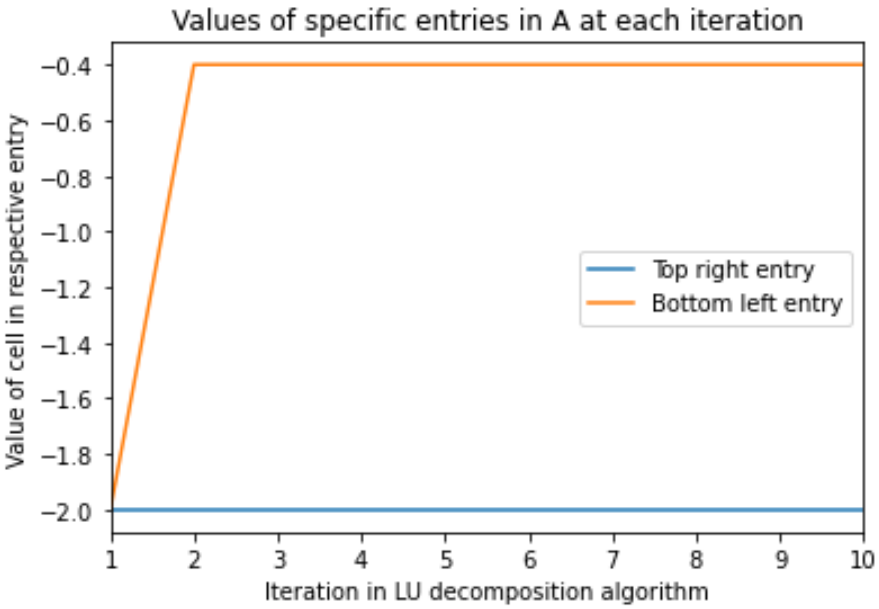


Figure 2: Value of top right and bottom left entry at each iteration for A as seen in figure 1

As can be seen from figures 1 and 2, at no point in the stages of LU decomposition on this matrix, is the matrix in a banded format as this would require both the top right and bottom left entry to be equal to zero in one iteration which figure 2 shows is not the case. It can therefore be concluded that at no stages of the LU decomposition is the matrix in banded form, rendering a banded matrix algorithm futile.

2.4 d)

A can be written as the sum of its tridiagonal component, T, and 2 outer products of vectors with a single nonzero component in the canonical basis (given below). The two outer products both contain one element that is non-zero (which is $-C_1$) with one matrix being all zeros except for the top right corner, and the other being all zeros except for the bottom left corner.

$$T = \begin{pmatrix} (1+2C_1) & -C_1 & 0 & \dots & 0 \\ -C_1 & (1+2C_1) & -C_1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & -C_1 & (1+2C_1) \end{pmatrix}, \; u_1 = \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}, \; v_1 = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ -C_1 \end{pmatrix}, \; u_2 = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 1 \end{pmatrix}, \; v_2 = \begin{pmatrix} -C_1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

If e_i represents the canonical basis in the ith coordinate, then $u_1 = e_1$, $u_2 = e_M$, $v_1 = -C_1 e_M$, and $v_2 = -C_1 e_1$.

Letting $T = LU$ (it is in an identical form to the tridiagonal matrix in exercise 1), we require a M by M matrix B such that $A = L(I_M + B)U$. The term LBU must consist solely of $-C_1$ in the upper right and lower left corner and zero everywhere else. Writing B as follows achieves this:

$$B = \begin{pmatrix} 0 & 0 & 0 & \dots & y_1 \\ 0 & 0 & 0 & \dots & y_2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & y_{M-1} \\ x_1 & x_2 & \dots & x_{M-1} & x_M \end{pmatrix}$$

with x and y entries being defined as follows to ensure that LBU consists only of two entries:

$$\begin{aligned} x_1 &= \frac{-C_1}{U_{1,1}} \\ x_i &= \frac{-kx_{i-1}}{U_{i,i}},\ i = 2, 3, \dots, M-1 \\ x_M &= \frac{-kx_{M-1}}{U_{M,M}} - y_{M-1}L_{M,M-1} \\ y_1 &= \frac{-C_1}{U_{M,M}} \\ y_i &= -y_{i-1}L_{i,i-1},\ i = 2, 3, \dots, M-1 \end{aligned}$$

where k is any of the values in the superdiagonal of U since they are all constant due to the structure of tridiagonal matrix we are LU factorising.

The inverse of A can therefore be written as $A^{-1} = U^{-1}(I_M + B)^{-1}L^{-1}$. In order to compute the inverse of $(I_M + B)$, we will use the Woodbury matrix identity:

$$(F + DCE)^{-1} = F^{-1} - F^{-1}D\left(C^{-1} + EF^{-1}D\right)^{-1}EF^{-1}$$

(link: archive.org/details/accuracystabilit00high878/page/n288/mode/2up)

Here we will let $F = I_M$ and $C = I_2$. Defining D and E as follows gives us $(I_M + B)^{-1}$:

$$D = \begin{pmatrix} y_1 & 0 \\ \vdots & \vdots \\ y_{M-1} & 0 \\ 0 & 1 \end{pmatrix},\ E = \begin{pmatrix} 0 & 0 & 0 & \dots & 1 \\ x_1 & x_2 & x_3 & \dots & x_M \end{pmatrix},\ B = DE$$

This means $(I_M + B)^{-1}$ is written as follows:

$$(I_M + DE)^{-1} = I_M - D\left(I_2 + ED\right)^{-1}E$$

which requires us to take the inverse of a 2 by 2 matrix and is therefore computationally very cheap ($O(1)$). And so, A^{-1} can be written as:

$$A^{-1} = U^{-1}(I_M - D\left(I_2 + ED\right)^{-1}E)L^{-1}$$

The central part of this expression, $I_M - D\left(I_2 + ED\right)^{-1}E$, never deals with multiplying two M by M matrices together (which is $O(M^3)$). The number of FLOPS in producing $(I_M + DE)^{-1}$ is $O(M^2)$ which is much more efficient.

2.5 e)

To solve the problem $Ax = b$ the following algorithm is used. It is the same as in 1) b) except that there is an intermediate step which uses the vector z as a transformation of y by $I_M - D\left(I_2 + ED\right)^{-1} E$ so that backward substitution can then be used to find x. A bold letter represents the entire vector:

$T \leftarrow A$ (tri-diag component only)
$y_1 \leftarrow b_1$
for $k = 1$ to $M - 1$ **do**
  $T_{k+1,k} \leftarrow \frac{T_{k+1,k}}{T_{k,k}}$
  $y_{k+1} \leftarrow b_{k+1} - T_{k+1,k} * y_k$
  $T_{k+1,k+1} \leftarrow T_{k+1,k+1} - T_{k+1,k} * T_{k,k+1}$
end for
$\mathbf{z} \leftarrow \left(I_M - D\left(I_2 + ED\right)^{-1} E\right)\mathbf{y}$ (multiplication using matrices as described above to carry out transformation)
$x_M \leftarrow \frac{z_M}{T_{M,M}}$
for $k = M - 1$ to 1 **do**
  $x_k \leftarrow \frac{z_k - T_{k,k+1} * x_{k+1}}{T_{k,k}}$
end for

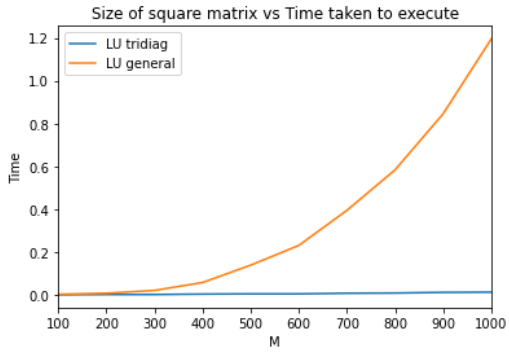
The operation count is dominated by the matrix multiplication that is used to obtain $\mathbf{z}$. This algorithm is therefore $O(M^2)$. This is far better than computing A^{-1} using standard LU decomposition as this is $O(M^3)$. This algorithm is also very efficient as at no point is the inverse of an M by M matrix calculated in order to obtain x.

2.6 f)

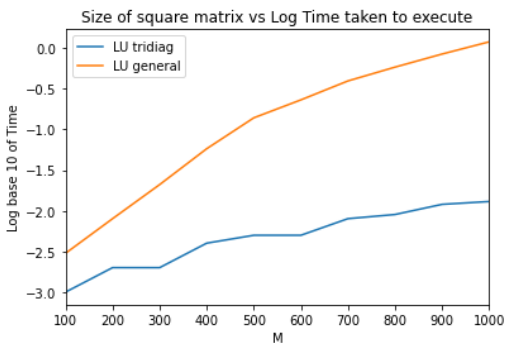
The function "LU tridiag mod" represents a modified algorithm from exercise 1 which accounts for the intermediate z step. The tests check to see that the algorithm outputs the correct solution, x, for different matrices by comapring it to the output produced by A^{-1} in python's packages.

To compare this to the generic algorithm in exercises 6 from class, we will make use of "LU inplace" and then use "solve L" and "solve U" to return x. The time taken to execute the algorithm will then be recorded and plotted alongside the time taken to perform the same task, but with our algorithm derived in the previous parts of this question for matrices of varying dimension.

As can be seen from figure 3, the time taken to carry out the algorithm without exploiting the structure of the matrix, takes a significantly longer amount of time. The log plot enables us to see the scale of difference exploiting the structure makes. For example, when $M = 1000$, the algorithm is sped up by a factor of approximately 100 when the structure is exploited.



(a) Plot of times for different M



(b) Log plot of times for different M

Figure 3: Comparison of time taken by general and tridiagonal LU algorithms

2.7 g)

The algorithm "numsol" solves the equation in 2) b) using the algorithm designed in 2) e) to invert A. The final time is T and figure 4 shows the analytical solution at time $T = 1$ alongside the numerical solution. There

is strong similarity, strengthening the legitimacy of the method. The analytical solution on which this is based is as follows:

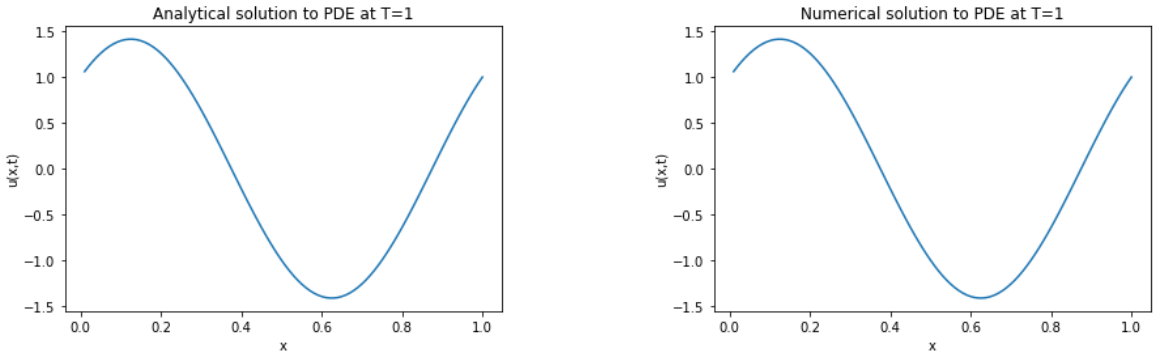
$$u(x,t) = \cos(2\pi(x+t)) + \sin(2\pi(x-t))$$

This satisfies the boundary conditions and the PDE. The initial conditions are as follows:

$$u(x,0) = \cos(2\pi x) + \sin(2\pi x)$$
$$u_t(x,0) = -2\pi(\sin(2\pi x) + \cos(2\pi x))$$

The function also takes in a vector of values which contains the timesteps at which the solution should be plotted. The default is that nothing is plotted.

The tests check to see that the analytical solution matches the numerical solution up to some tolerance for different times, T. The numpy linalg norm function of the difference between the two solutions is used to check this. Figure 5 shows how the error decreases as you shrink the space step, dx. The error is measured using the numpy linalg norm function applied to the difference between the numerical and analytical solution.



(a) Analytical solution to PDE at $T = 1$

(b) Numerical solution to PDE at $T = 1$, $dt = dx = 0.01$

Figure 4: Numerical and Analytical solutions to PDE at $T = 1$

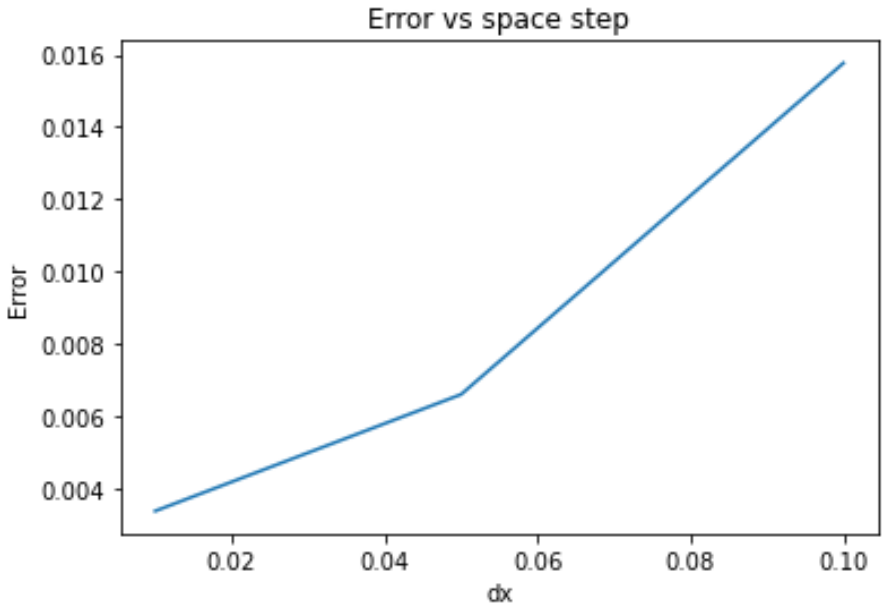


Figure 5: Error of numerical method vs dx at $T = 1$

3 Question Three

3.1 a)

Considering the QR algorithm, we will assume that $A^{k-1} = QR$ is symmetric (and real) and tridiagonal and then show that $A^k = RQ$ is symmetric and tridiagonal.

To show the preservation of symmetry, we will simply make use of the properties of A^{k-1} and Q (which we know is orthogonal) as follows:

$$\begin{aligned} A^{k-1} = QR,\ A^{k-1} = (A^{k-1})^T \\ \Rightarrow QR = R^T Q^T, \\ \Rightarrow Q^T QR = Q^T R^T Q^T = R, \\ \Rightarrow RQ = Q^T R^T Q^T Q = Q^T R^T, \\ \Rightarrow RQ = (RQ)^T = A^k = (A^k)^T \end{aligned}$$

showing that A^k is symmetric. Inductively, this shows that A^{k+n} is symmetric for all $n \geq 0$ provided A^{k-1} is symmetric.

Given that the matrices are symmetric, to prove the preservation of the tridiagonal structure we need only consider the lower triangular component of A^k since if it has lower bandwidth 1, then by its symmetry, it will also have upper bandwidth 1 and thus be tridiagonal.

The proof is independent of our choice of method for QR factorisation, and so the Givens rotations method (from second year numerical analysis) will be used to show the preservation of the tridiagonal structure. As a reminder, the form of a Givens rotation is given below and our Q^T is made up of the product of $n-1$ Givens rotations with each one eliminating a subdiagonal entry to produce R, our upper triangular matrix:

$$G(i,j,\theta) = \begin{bmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & c & \cdots & -s & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & s & \cdots & c & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{bmatrix},\ i > j$$

By construction, these matrices are orthogonal and c and s represent the cosine and sine respectively evaluated at a chosen θ such that the entry (i,j) is eliminated when left multiplied. Only the rows i and j are affected when left multiplication is carried out. When G is right multiplied only the columns i and j are affected.

$$Q^T A^{k-1} = G_{n-1} G_{n-2} \cdots G_1 A^{k-1} = R$$

In order to eliminate the subdiagonal entries at each step and produce R, we require that $i = j+1$ for $j = 1$ to $n-1$. Now that we have the structure of our Givens rotations matrices and know which entries are filled in, we will consider the lower triangular entries of A^k which are obtained by multiplying our upper triangular matrix, R, on the right by the transposes of our Givens rotations in the following way:

$$R \to RG_1^T \to RG_1^T G_2^T \to RG_1^T G_2^T \cdots G_{n-1}^T = RQ = A^k$$

where $G_j = G(j+1, j, \theta_j)$ with θ_j chosen to eliminate the lower diagonal element in question (see M2AA3 notes for specific form, which is not important for the proof).

Multiplying on the right by G_j^T only changes columns j and $j+1$ of the matrix with which it is being multiplied. We show the principle of the procedure when R is right multiplied by G_1^T below with an asterisk representing an entry that isn't necessarily zero:

$$RG_1^T = \begin{bmatrix} * & \cdots & * & \cdots & * & \cdots & * \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & * & \cdots & * & \cdots & * \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & 0 & \cdots & * & \cdots & * \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & * \end{bmatrix} \begin{bmatrix} c & s & 0 & \cdots & 0 & \cdots & 0 \\ -s & c & \vdots & & \vdots & & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 0 \end{bmatrix}$$

All bar the first two columns are unaffected by the transposed Givens rotation. The upper triangular nature of R is also such that only the first two rows of R have non zero terms that interact with the c and s components of G_1^T. Therefore, the resulting matrix formed from the product is only different in the top left 2 X 2 square when compared to R; this means that the only term that may change from 0 to a non-zero term is $(2, 1)$. Similarly, for subsequent transposed Givens rotations (G_j^T) that are right multiplied and are defined as above, the only term below the diagonal that may change from 0 to a non-zero value is $(j + 1, j)$ and thus all values below the first subdiagonal remain zero under all of these transposed Givens rotations.

From this, we can conclude that after R has been multiplied $n-1$ times by the transposed Givens rotations (given above), we obtain A^k which has no non-zero entries below the first subdiagonal, meaning it has lower bandwidth 1. Given that A^k is also symmetric, we can therefore deduce that A^k is tridiagonal and hence we have shown that under the QR algorithm, tridiagonality and symmetry is preserved at each step. □

3.2 b)

Considering the in place algorithm in exercise 2.8 when applied to a tridiagonal matrix, we can see that the x that is extracted from $A_{k:m,k}$ only has 2 values that are non zero, namely $A_{k:k+1,k}$. This therefore means that the unitary matrix that removes a specific entry in the subdiagonal of A is equivalent to the identity, except for a 2 by 2 matrix which represents the Householder rotation which is referred to as F which is different for each Q_i and appears in rows i and $i+1$. Householder rotation takes the form $F = I_2 - v_k^T v_k$ with v_k defined as per the Householder algorithm given in lectures.

Given the structure of a tridiagonal, symmetric matrix, we require $n-1$ rotations with the ith matrix taking the following form:

$$Q_i = \begin{bmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & F_{1,1} & F_{1,2} & 0 & \cdots & 0 \\ 0 & \cdots & F_{2,1} & F_{2,2} & 0 & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{bmatrix}$$

When multiplying this matrix on the left to A, only rows i and $i+1$ are affected. Furthermore, only 6 terms are changed in the subsequent matrix due to multiplication up to $n-2$ and 4 terms are changed at $n-1$ (which can be seen from the algorithm in part c). This characteristic is exploited in the algorithm in part c to improve its efficiency. After $n-1$ Householder transformations of this kind, the upper triangular matrix, R has upper bandwidth 2.

If the structure of the tridiagonal matrix isn't exploited, then multiplication and addition of zeros is very frequent. From lectures, we know that the operation count for the Householder transformation (when $m = n$) is $\sim 2n^2 - \frac{2n^3}{3} = \frac{4n^3}{3}$.

If the tridiagonal structure is exploited, then for each k, we only consider vectors and slices of matrices of length 2 (or 3, but this won't change the asymptotic complexity since it is constant relative to n) and we only require that we go up to $n-1$ rather than n. This therefore makes the total operation count (using the formula derived in lectures) approximately:

$$4\sum_{k=1}^{n-1}(k+2-k)(k+2-k) = 16\sum_{k=1}^{n-1}1 = 16(n-1) \sim 16n$$

This complexity is only $O(n)$ rather than $O(n^3)$ making it significantly more efficient.

3.3 c)

See the python file for the algorithm. The algorithm does not multiply or add any terms where a zero features so as to optimise efficiency. The tests carried out investigate if the output produced is correct by comparing it to the less efficient Householder algorithm produced in exercises 3 of the class assignments which we know is correct by the cla utils tests. The tests also check to see if the R output is upper triangular.

3.4 d)

When implementing the unshifted QR algorithm and applying it to the tridiagonal matrix, the algorithm from part c will be used in order to obtain the QR factorisation.

For tridiagonal matrices, R has upper bandwidth 2 and so this is taken into account in the algorithm when right muliplying by the $m-1$ Q^T matrices in order to avoid superfluous multiplication and addition of zeros. A 2 by 2 matrix is formed at each step (which represents Q) rather than an m by m matrix and only elements that are changed by each iteration in the matrix multiplication are considered (this is either a 2 by 2 or 3 by 2 submatrix), making the operation count $O(1)$. The algorithm uses a while loop to iterate until the $(m, m-1)$ element satisfies the tolerance of 10^{-12}.

The tests for this code check that the final matrix is symmetric and that the rank and trace is preserved, as under similarity transformations these should all be preserved.

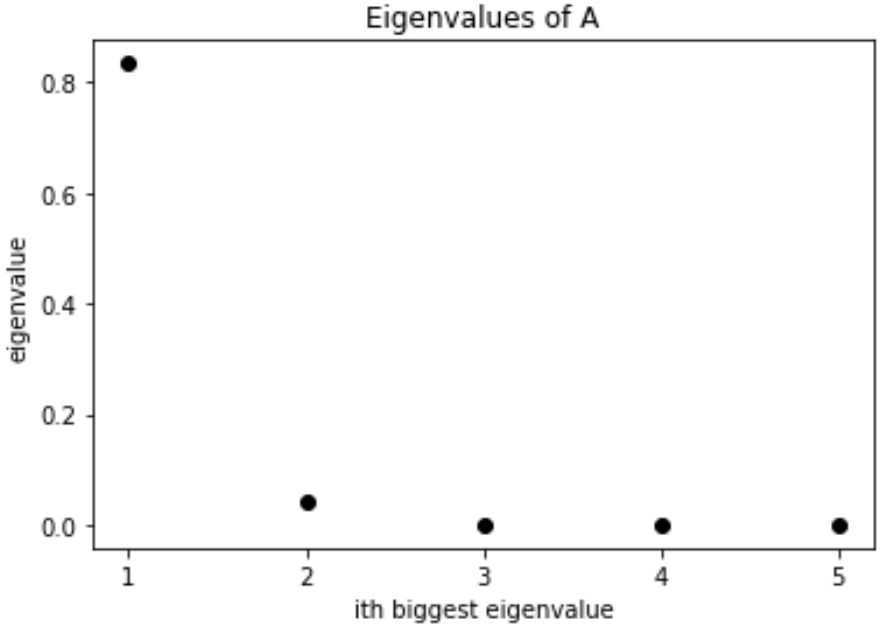


Figure 6: Approximation of eigenvalues of A_{ij} in descending order (5x5)

Applying the algorithm to A_{ij} as defined in the assignment, produces an almost upper triangular matrix which has its symmetry preserved. Prior to implementing the *qr alg tri* algorithm, the matrix was converted to Hessenberg form in order to expedite the algorithm and put it in the required form for the algorithms in this exercise. This was done using the exercises from class assignments. Given that A is symmetric, the Hessenberg matrix was tridiagonal.

What can be seen from figure 6 is that all the eigenvalues are positive. In conjunction with this, the matrix is also symmetric and so we can deduce from this that the matrix, A, is positive definite.

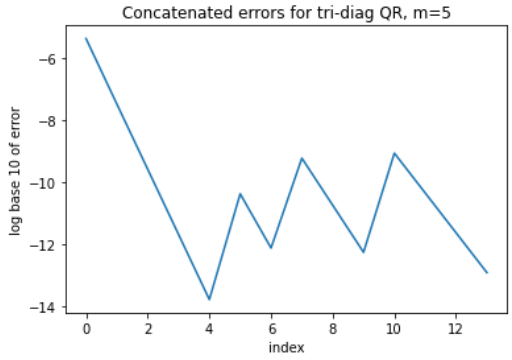
3.5 e)

The function which performs what is required in the assignment is called "Eigval iter", since the function effectively returns the eigenvalues of the matrix (that is inputted iteratively until the matrix is reduced to being 1 by 1) and the tolerances at each iteration (whose length helps us gauge the rate of convergence). The final eigenvalue is simply extracted by taking the (1,1) component of T when it is of size 2, since a 1 by one matrix is itself the eigenvalue corresponding to it. The *qr alg tri mod* returns the concatenated sequences of $|T_{m,m-1}|$; the shorter the sequences, the faster the convergence. The test for the function "Eigval iter" tests to see if the

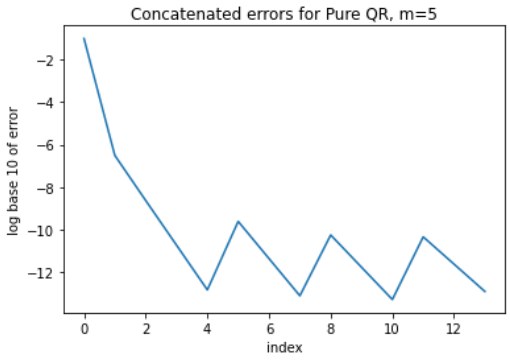
eigenvalues extracted from it are equivalent (up to a tolerance) to the eigenvalues produced by numpy's inbuilt function. The matrices used in the tests are defined in the same way as A_{ij}, except larger matrices are used, with the same principle being used.

In addition to A_{ij} for $m = 5$, we will apply the algorithm to two other positive definite matrices which we will construct by increasing the dimensions of A to 10 and 15. We also modify the pure QR algorithm from exercises 9 in order to return the sequence of errors up to the tolerance as well. The function "iter pure QR" carries out the same algorithm as described above, except it uses the pure QR algorithm instead of the QR algorithm tailored specifically to tridiagonal matrices. The tolerance of the pure QR algorithm is exactly the same as in the tridiagonal case for the purposes of direct comparison; indeed, the method used in both instances is known as deflation. The convergence of both algorithms can be seen in figures 5, 6 and 7 (for $m = 5, 10, 15$ respectively) which shows a log plot of the errors for all the concatenated errors across the iterations that are carried out in both algorithms.

What is apparent from the figures is that for initial iterations, the tridiagonal QR algorithm is superior to the pure QR algorithm, since the reduction to Hessenberg form accelerates the process and so the tolerance is reached more quickly. Beyond the first couple of iterations, there is little difference, and this is to be expected since the tolerance of 10^{-12} is low.



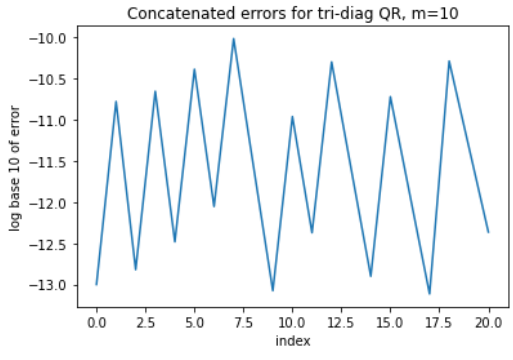
(a) Tridiagonal QR concatenated errors



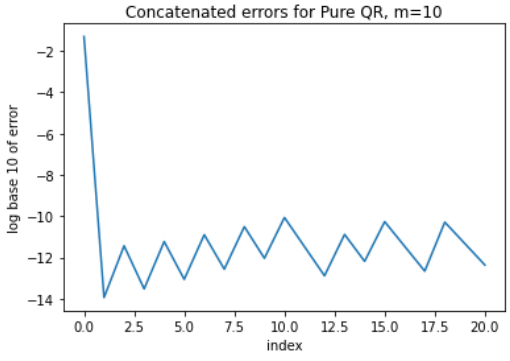
(b) Pure QR concatenated errors

Figure 7: Convergence of QR algorithms demonstrated using errors, $m = 5$

In figures 7, 8 and 9, a downwards spike represents the beginning of a new submatrix being operated on. From all the plots, it appears that the majority of the work to reduce all $T_{m,m-1}$ terms to below the tolerance is done in the first iteration of the first m by m matrix. The tridiagonal QR algorithm appears to have had most of the heavy lifting done by the Hessenberg transformation, which explains its much lower initial value as compared with the pure QR algorithm which doesn't convert the matrix to Hessenberg form.

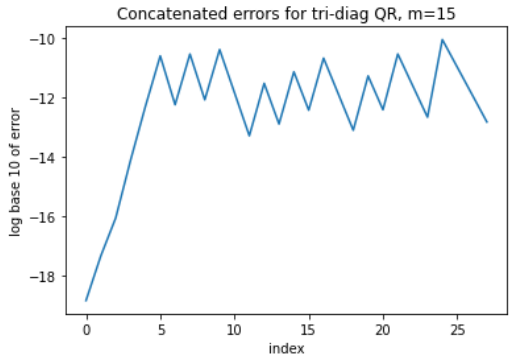


(a) Tridiagonal QR concatenated errors

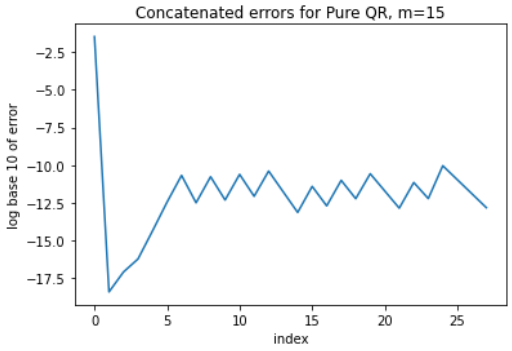


(b) Pure QR concatenated errors

Figure 8: Convergence of QR algorithms demonstrated using errors, $m = 10$



(a) Tridiagonal QR concatenated errors

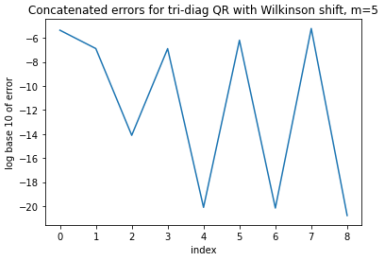


(b) Pure QR concatenated errors

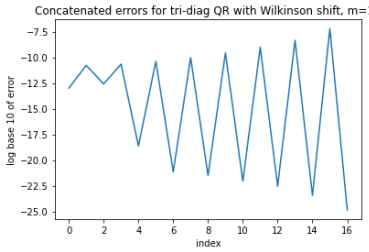
Figure 9: Convergence of QR algorithms demonstrated using errors, $m = 15$

3.6 f)

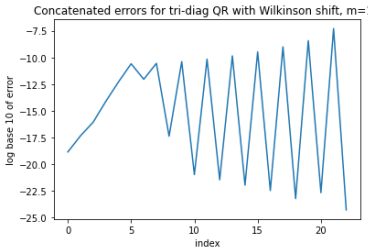
The function "qr alg tri wilk" performs just as the function in part d, except that it applies the Wilkinson shift; the Wilkinson shift does not affect the tridiagonal structure at all in the process and so the tridiagonal aspect of the algorithm will remain unaffected. A test is carried out to make sure the eigenvalues returned by the algorithm are correct (as was done in part e).



(a) $m = 5$



(b) $m = 10$



(c) $m = 15$

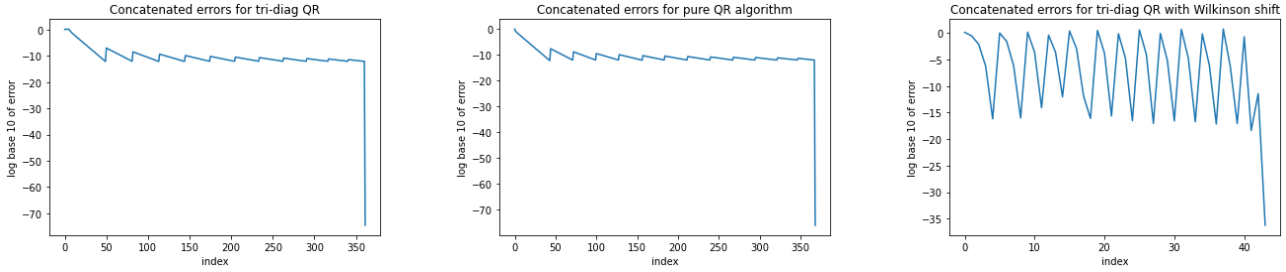
Figure 10: Convergence of tri-diag QR with Wilkinson shift algorithms demonstrated using errors

What can be seen from figure 10 is that the values reached prior to the stopping condition being put into practice are much lower with the Wilkinson shift put in place than without. This tells us that the Wilkinson shift accelerates the convergence to an upper triangular matrix; Trefethen and Bau's book on computational linear algebra (the basis of this course) confirms this since the Wilkinson shift generally has a cubic convergence rate and at worst, a quadratic one. This ensures that deflation happens at a faster rate rather than being down to chance. Furthermore, the number of indices with the shift is about 80 per cent the number of indices with the two methods that don't use a shift.

3.7 g)

Figure 11 shows the concatenated errors for the three algorithms (pure QR, tridiag QR, tridiag QR Wilk) when applied to $A = D+O$. A point of note is that when the eigenvalues outputted are compared with the eigenvalues produced by numpy's inbuilt function (np.linalg.eig) the eigenvalues produced by tridiag QR Wilk appear in exactly the same order as with python's inbuilt function, suggesting that python's module uses the Wilkinson shift method in order to compute the eigenvalues. What can be seen is that when the shift is used, convergence is much quicker (by factor of about 10, if you consider the relative size of the indices in each case).

The speed of convergence here is more apparent than with the 5x5 matrix in the previous question (~ 10 times faster instead of ~ 1.25 times faster), since the eigenvalues are more spread out. In addition to this, the slowest method (judging by the number of indices in the concatenated array) is the pure QR algorithm which uses 369 indices; this is marginally faster than the QR algorithm with no shift which uses 362 indices.



(a) Tridiagonal QR (b) Pure QR (c) Tridiagonal QR with shift

Figure 11: Convergence of QR algorithms demonstrated using errors

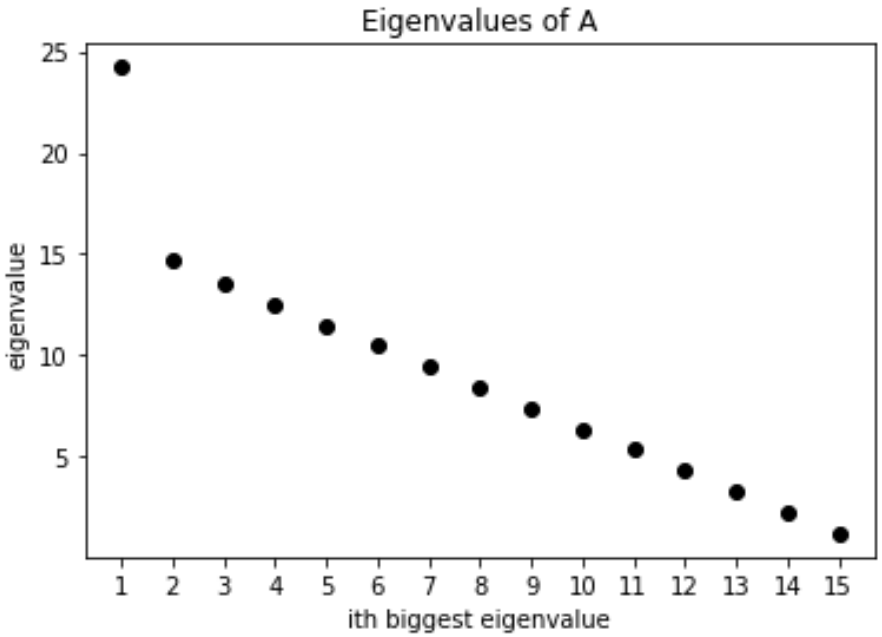


Figure 12: Approximation of eigenvalues of A in descending order (15x15)

Figure 12 shows that all eigenvalues are positive and so given that A is also symmetric, we can deduce that A is positive definite; this is a similarity between this matrix and the matrix used in part d.

4 Question Four

4.1 a)

The "apply pc" variable in this exercise will take one of three values. If the value is "None" then the "GMRES mod" function will proceed without preconditioning. If the value is 0, then the preconditioner will be the Jacobi preconditioner, which is a diagonal matrix comprised of the diagonal entries of A and is trivial to solve. If the value is 1, then the preconditioner will be the upper triangular component of A and this will be solved by backward substitution. The tests see if the residuals are monotone decreasing (as would be expected with GMRES) and that if the tolerance is hit, the size of the residual agrees with the tolerance set.

4.2 b)

The following is an immediate consequence of the properties of the operator norm:

$$\|Ax\| \leq \|A\|_2 \|x\|$$

If we fix x as a normalised eigenvector of $M^{-1}A$ and use the above property, then the following can be deduced:

$$\begin{aligned}\|(I - M^{-1}A)x\| \leq \|I - M^{-1}A\|_2\|x\| = c\|x\| = c,\\ \Rightarrow \|x - M^{-1}Ax\| \leq c,\\ \Rightarrow \|x - \lambda_i x\| \leq c,\\ \Rightarrow \|1 - \lambda_i\|\|x\| \leq c,\\ \Rightarrow |1 - \lambda_i| \leq c\end{aligned}$$

where we made use of the property of eigenvectors to get from the second line to the third line and used the fact that x is normalised and thus $\|x\| = 1$. Getting from the third line to the fourth is derived by achknowledging that $(1 - \lambda_i)$ is a real number.

x is an arbitrary eigenvector and so this inequality is true for all λ_i. From this we deduce that for all eigenvalues, λ, of $M^{-1}A$, $|1 - \lambda| \leq c$. $\square$

4.3 c)

Using the derivation from the previous part, we will consider the formulation of GMRES using matrix polynomials in terms of $M^{-1}A$. I.E.

$$\frac{\|r_n\|}{\|b\|} \leq \|p_n(M^{-1}A)\|$$

for any polynomial of degree n with the operator-2 norm being used on the RHS. As in lectures, if we assume that $M^{-1}A$ is diagonalisable, then $A = V\Lambda V^{-1}$ and hence all polynomials (with $p(0) = 1$) satisfy the following inequality:

$$\|p_n(M^{-1}A)\| = \|Vp_n(\Lambda^s)V^{-1}\| \leq \underbrace{\|V\|\|V^{-1}\|}_{=\kappa(V)}\|p_n\|_{\Lambda(M^{-1}A)}$$

with $\kappa(V)$ representing the condition number of the matrix of eigenvectors of $M^{-1}A$.

$p(z) = (1 - z)^n$ when applied to this formulation is a good upper bound. This is because it satisfies the condition that $p(0) = 1$ (trivially) and that the modulus of $p(x)$ is small when evaluated at the eigenvalues of $M^{-1}A$. The proof of this is given below and makes use of the assumed property in part b of this question. Plugging an arbitrary eigenvalue of $M^{-1}A$ yields:

$$\|p_n(\lambda_i)\| = \|1 - \lambda_i\|^n \leq \|1 - \lambda_{max}\|^n \leq c^n < 1$$

where λ_{max} is the λ that maximises $\|1 - \lambda_i\|$ across all eigenvalues of $M^{-1}A$. We are told that c is also less than one. This means that if it raised to the power of n and n is increased, c^n will approach zero as n grows arbitrarily large. This therefore means that when this algorithm is evaluated at every eigenvalue of $M^{-1}A$, $p(z) = (1-z)^n$ is relatively small and decreases as the number of iterations is increased, making it a good upper bound.

Consequently, this makes the following a derived upper bound for the convergence rate of the preconditioned GMRES (provided critera in part b are satisfied):

$$\frac{\|r_n\|}{\|b\|} \leq \kappa(V)x^n$$

with $x = \|1 - \lambda_{max}\| < 1$ and $\kappa(V)$ being the condition number as aforementioned.

4.4 d)

To investigate this upper bound, we consider a 4 by 4 Laplacian matrix (for the set up of the non-trivial, directed graph, refer to the code in the q4 python file) summed with I_4 which we refer to as A. The preconditioned matrix, M, is equal to $1.1U$ where U is the upper triangular component of A. Making use of the "operator 2 norm" function from cla utils, it can be seen that the value of $\|I_4 - M^{-1}A\|_2$ is 0.97, which is less than 1 and is therefore suitable for investigation.

Furthermore, making use of python's "linalg eig" and "matrix rank" functions, we can see that the dimension of the eigenspace of $M^{-1}A$ is equal to 4, and thus the matrix can be diagonalised, meaning that we can proceed as above in investigating the upper bound. The condition number is also computed using the "cond" function from cla utils. An additional variation of the "apply pc" parameter in "GMRES mod" has also been added; this specific paramter takes M to be $1.1U$ and uses backward substitution to solve the resulting systems of equations (the input parameter here is 2). With the specific graph we are working with, $\|1-\lambda_{max}\| \sim 0.8 < 0.97$, confirming the results previously deduced in this question. An additional test is added into the q4 python file which tests to see if the outputs due to the preconditioned and un-preconditioned are similar up to a tolerance; the test passes. The logarithm of the residuals is plotted as it is easier to see how it decreases as iterations progress when a logarithmic scale is used.

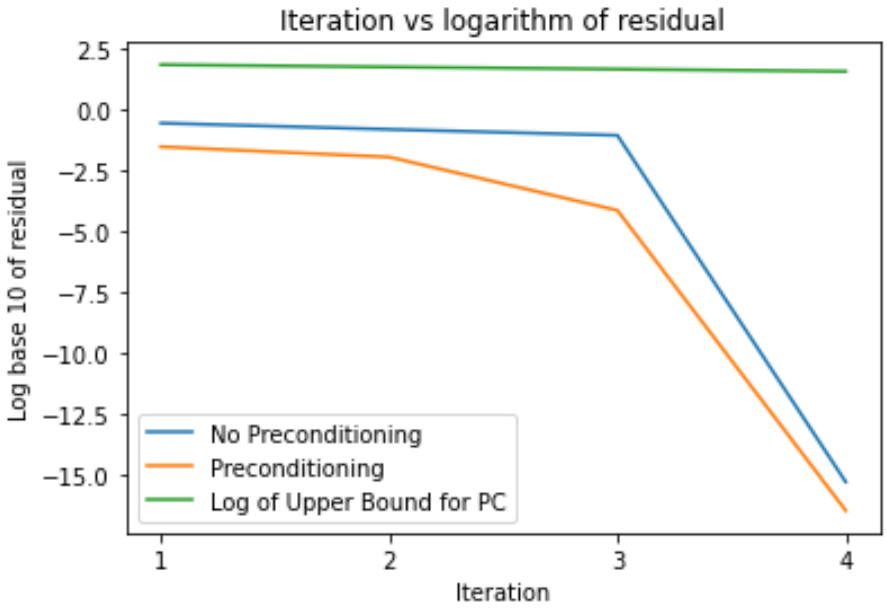


Figure 13: log_{10} of GMRES residuals with/without preconditioning alongside upper bound for preconditioning

Figure 13 shows how the upper bound decreases linearly with each iteration which is to be expected since the logarithm of $\kappa(V)x^n = nlog_{10}(x) + log_{10}(\kappa(V))$ which is itself linear and has negative gradient (since $x < 1$). What this tells us is that the upper bound decreases with each iteration and becomes a stronger upper bound the more iterations that are used; a bigger matrix would allow us to see a stronger bound (e.g. $iter = 1000$). Figure 13 also shows us how the preconditioning significantly speeds up the convergence to the true solution for the initial iterations.

This is in line with what we expect, as the eigenvalues are clustered much more closely together after having been preconditioned, and this can be easily seen from Figure 14, with the preconditioning bringing the eigenvalues closer together. The complex component is disregarded since in both cases, the complex values are very close to being zero and so the distance of the eigenvalues on the complex plane is accurately represented by just considering the real axis.

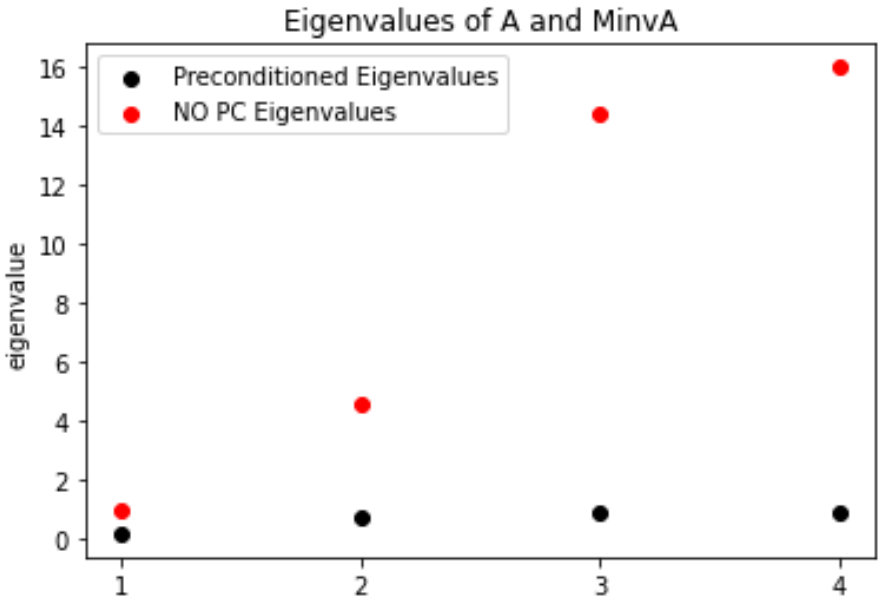


Figure 14: Real parts of eigenvalues of preconditioned Matrix, $M^{-1}A$ and A

5 Question Five

5.1 a)

Implementing the following scheme (with initial conditions for u and w known) across $j = 0, 1, \ldots, N-1$ will present us with solutions for u_i^n and w_i^n for all $i = 1, \ldots, M$ and $n = 1, \ldots, N$:

$$w_i^{j+1} - w_i^j - \frac{\Delta t}{2(\Delta x)^2}(u_{i+1}^j + u_{i+1}^{j+1} - 2(u_i^j + u_i^{j+1}) + u_{i-1}^j + u_{i-1}^{j+1}) = 0 \tag{1}$$

$$u_i^{j+1} - u_i^j - \frac{\Delta t}{2}(w_i^j + w_i^{j+1}) = 0 \tag{2}$$

We will attempt to write (1) and (2) in the form $AU = (r\ 0 \ldots\ 0)^T$ with :

$$A = \begin{pmatrix} I & 0 & 0 & \ldots & 0 \\ -I & I & 0 & \ldots & 0 \\ 0 & -I & I & \ldots & 0 \\ \vdots & & & & \vdots \\ 0 & \ldots & 0 & -I & I \end{pmatrix} + \frac{1}{2}\begin{pmatrix} B & 0 & 0 & \ldots & 0 \\ B & B & 0 & \ldots & 0 \\ 0 & B & B & \ldots & 0 \\ \vdots & & & & \vdots \\ 0 & \ldots & 0 & B & B \end{pmatrix}$$

and the forms of U, p_i and q_i are all given in the assignment. I is $2M$ by $2M$ (I_{2M}) and we can regard A as being an N by N matrix with each entry being a a $2M$ by $2M$ matrix.

In this format, the 2nd to nth rows of A when multiplied by U represent equations (1) and (2) directly with the RHS being equal to zero. The first row (when $j = 0$) of A rewrites (1) and (2) by isolating the u^1 and w^1 terms on the LHS and the u^0 and w^0 terms on the RHS which are represented by r, a constant since the initial condition is known.

To work out the form of B (which we split into 4 M by M submatrices as given in the assignment), we will consider the system of equations at $j = 1$ to $N-1$. The $-I$ and I components in the first part of A represent the $w_i^{j+1} - w_i^j$ and $u_i^{j+1} - u_i^j$ aspect of our system of equations. In equations (1) and (2), we can also see that the u and w components do not interact with themselves in the part multiplied by Δt. This therefore means that $B_{11} = B_{22} = 0$.

Setting B_{21} and B_{12} as follows allows us to write the system of equations in the desired form:

$$B_{21} = -\frac{\Delta t}{(\Delta x)^2}\begin{pmatrix} -2 & 1 & 0 & \dots & 1 \\ 1 & -2 & 1 & \dots & 0 \\ 0 & 1 & -2 & \dots & 0 \\ \vdots & & & & \vdots \\ 1 & \dots & 0 & -1 & -2 \end{pmatrix}, \; B_{12} = -\Delta t I_M$$

B_{21} resembles very much what was done in question 2; it is tridiagonal except for the top right and bottom left entry due to the periodicity at the boundary.

Now that we have the form of B, the value of r takes a similar form except that it represents u^0 and w^0 when written on the RHS of (1) and (2) (in the case $j = 0$):

$$r = (I_{2M} - \frac{1}{2}B)(p_0 \; q_0)^T = \begin{pmatrix} u_1^0 + \frac{\Delta t}{2} w_1^0 \\ \vdots \\ u_M^0 + \frac{\Delta t}{2} w_M^0 \\ w_1^0 + \frac{\Delta t}{2(\Delta x)^2}(u_M^0 - 2u_1^0 + u_2^0) \\ \vdots \\ w_1^M + \frac{\Delta t}{2(\Delta x)^2}(u_{M-1}^0 - 2u_M^0 + u_1^0) \end{pmatrix}$$

We have successfully written the system of equations as a matrix vector system which solves all of these equations at once.

5.2 b)

Considering the following iterative method:

$$(C_1^{(\alpha)} \otimes I + C_2^{(\alpha)} \otimes B)U^{k+1} = \begin{pmatrix} r + \alpha(-I + \frac{1}{2}B)\begin{pmatrix} p_N^k \\ q_N^k \end{pmatrix} \\ 0 \\ \vdots \\ 0 \end{pmatrix} \tag{3}$$

we will assume that this iterative method converges for sufficiently small α. If this does converge, then this means that $\|U_k - U_{k+1}\| \to 0$ as $k \to \infty$. For this to be the case, we require that all of the components of U_k and U_{k+1} become equal in the limit. This therefore means that in the limit, $U_k = U_{k+1}$ and $\begin{pmatrix} p_N^k \\ q_N^k \end{pmatrix} = \begin{pmatrix} p_N^{k+1} \\ q_N^{k+1} \end{pmatrix}$.

Plugging this into (3) yields:

$$(C_1^{(\alpha)} \otimes I + C_2^{(\alpha)} \otimes B)U^{k} = \begin{pmatrix} r + \alpha(-I + \frac{1}{2}B)\begin{pmatrix} p_N^k \\ q_N^k \end{pmatrix} \\ 0 \\ \vdots \\ 0 \end{pmatrix}, \; k \to \infty \tag{4}$$

To show that (4) is equivalent to the system of equations in part a, we will rewrite the LHS of (4) using the property of bilinearity of the tensor product. To do this we will write $C_1^{(\alpha)}$ and $C_2^{(\alpha)}$ as follows:

$$C_1^{(\alpha)} = C_1 + \begin{pmatrix} 0 & 0 & 0 & \dots & -\alpha \\ 0 & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & \dots & 0 \\ \vdots & & & & \vdots \\ 0 & \dots & 0 & 0 & 0 \end{pmatrix}$$

$$C_2^{(\alpha)} = C_2 + \begin{pmatrix} 0 & 0 & 0 & \dots & \frac{\alpha}{2} \\ 0 & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & \dots & 0 \\ \vdots & & & & \vdots \\ 0 & \dots & 0 & 0 & 0 \end{pmatrix}$$

Using the tensor product and its bilinearity property, we obtain:

$$C_1^{(\alpha)} \otimes I = C_1 \otimes I + \begin{pmatrix} 0 & 0 & 0 & \dots & -\alpha I \\ 0 & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & \dots & 0 \\ \vdots & & & & \vdots \\ 0 & \dots & 0 & 0 & 0 \end{pmatrix}$$

$$C_2^{(\alpha)} \otimes B = C_2 \otimes B + \begin{pmatrix} 0 & 0 & 0 & \dots & \frac{\alpha}{2} B \\ 0 & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & \dots & 0 \\ \vdots & & & & \vdots \\ 0 & \dots & 0 & 0 & 0 \end{pmatrix}$$

Plugging this into (4), results in:

$$(C_1 \otimes I + C_2 \otimes B)U^k + \begin{pmatrix} 0 & 0 & 0 & \dots & -\alpha I \\ 0 & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & \dots & 0 \\ \vdots & & & & \vdots \\ 0 & \dots & 0 & 0 & 0 \end{pmatrix} U^k + \begin{pmatrix} 0 & 0 & 0 & \dots & \frac{\alpha}{2} B \\ 0 & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & \dots & 0 \\ \vdots & & & & \vdots \\ 0 & \dots & 0 & 0 & 0 \end{pmatrix} U^k = \begin{pmatrix} r + \alpha(-I + \frac{1}{2}B) \begin{pmatrix} p_N^k \\ q_N^k \end{pmatrix} \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

The two matrices on the LHS when multiplied out, only interact with $\begin{pmatrix} p_N^k \\ q_N^k \end{pmatrix}$ in U^k.

Rearranging, multiplying out the two LHS matrices and isolating $(C_1 \otimes I + C_2 \otimes B)U^k$ gives us:

$$(C_1 \otimes I + C_2 \otimes B)U^k = \begin{pmatrix} r + \alpha(-I + \frac{1}{2}B) \begin{pmatrix} p_N^k \\ q_N^k \end{pmatrix} \\ 0 \\ \vdots \\ 0 \end{pmatrix} + \begin{pmatrix} \alpha I \begin{pmatrix} p_N^k \\ q_N^k \end{pmatrix} \\ 0 \\ \vdots \\ 0 \end{pmatrix} + \begin{pmatrix} -\alpha \frac{1}{2} B \begin{pmatrix} p_N^k \\ q_N^k \end{pmatrix} \\ 0 \\ \vdots \\ 0 \end{pmatrix} = \begin{pmatrix} r \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

which equivalent to the system of equations as described in part a. Thus, this iterative scheme converges to the solution of (13) as given in the assignment. □

5.3 c)

Showing that the system of N vectors is an eigenvector is simply a matter of showing that when the system of vectors is left multiplied by $C_1^{(\alpha)}$ and $C_2^{(\alpha)}$ a scalar multiple of the vector is outputted. We will proceed to show this for x_j which is defined as follows:

$$x_j = \begin{pmatrix} 1 \\ \alpha^{-1/N} e^{\frac{2\pi ij}{N}} \\ \alpha^{-2/N} e^{\frac{4\pi ij}{N}} \\ \vdots \\ \alpha^{-(N-1)/N} e^{\frac{2(N-1)\pi ij}{N}} \end{pmatrix}$$

Starting with $C_1^{(\alpha)}$:

$$\begin{aligned} C_1^{(\alpha)} x_j &= \begin{pmatrix} 1 - \alpha * \alpha^{-(N-1)/N} e^{\frac{2(N-1)\pi ij}{N}} \\ \alpha^{-1/N} e^{\frac{2\pi ij}{N}} - 1 \\ \alpha^{-2/N} e^{\frac{4\pi ij}{N}} - \alpha^{-1/N} e^{\frac{2\pi ij}{N}} \\ \vdots \\ \alpha^{-(N-1)/N} e^{\frac{2(N-1)\pi ij}{N}} - \alpha^{-(N-2)/N} e^{\frac{2(N-2)\pi ij}{N}} \end{pmatrix} \\ &= (1 - \alpha^{1/N} e^{-\frac{2\pi ij}{N}}) \begin{pmatrix} 1 \\ \alpha^{-1/N} e^{\frac{2\pi ij}{N}} \\ \alpha^{-2/N} e^{\frac{4\pi ij}{N}} \\ \vdots \\ \alpha^{-(N-1)/N} e^{\frac{2(N-1)\pi ij}{N}} \end{pmatrix} \\ &= \lambda_{1,j} x_j \end{aligned}$$

with the jth eigenvalue being equal to $(1-\alpha^{1/N} e^{-\frac{2\pi ij}{N}})$. We made use of the fact in this proof that $e^{2\pi ij} = 1$ for all $j \in \{0, 1, \ldots, N-1\}$.

Similarly, with $C_2^{(\alpha)}$:

$$\begin{aligned} C_2^{(\alpha)} x_j &= \frac{1}{2} \begin{pmatrix} 1 + \alpha * \alpha^{-(N-1)/N} e^{\frac{2(N-1)\pi ij}{N}} \\ \alpha^{-1/N} e^{\frac{2\pi ij}{N}} + 1 \\ \alpha^{-2/N} e^{\frac{4\pi ij}{N}} + \alpha^{-1/N} e^{\frac{2\pi ij}{N}} \\ \vdots \\ \alpha^{-(N-1)/N} e^{\frac{2(N-1)\pi ij}{N}} + \alpha^{-(N-2)/N} e^{\frac{2(N-2)\pi ij}{N}} \end{pmatrix} \\ &= \frac{1}{2}(1 + \alpha^{1/N} e^{-\frac{2\pi ij}{N}}) \begin{pmatrix} 1 \\ \alpha^{-1/N} e^{\frac{2\pi ij}{N}} \\ \alpha^{-2/N} e^{\frac{4\pi ij}{N}} \\ \vdots \\ \alpha^{-(N-1)/N} e^{\frac{2(N-1)\pi ij}{N}} \end{pmatrix} \\ &= \lambda_{2,j} x_j \end{aligned}$$

with the jth eigenvalue being equal to $\frac{1}{2}(1+\alpha^{1/N} e^{-\frac{2\pi ij}{N}})$. We made use of the fact in this proof that $e^{2\pi ij} = 1$ for all $j \in \{0, 1, \ldots, N-1\}$.

For both $C_1^{(\alpha)}$ and $C_2^{(\alpha)}$, we have N distinct eigenvalues. This implies that all the eigenvectors are linearly independent and hence form a basis. Expressing each of $C_1^{(\alpha)}$ and $C_2^{(\alpha)}$ in the basis of V (the matrix of all eigenvectors, x_j) is achieved by diagonlisation and corresponding the jth column of V with the jth diagonal entry of D_1 and D_2 forming a diagonal matrix of eigenvalues for $C_1^{(\alpha)}$ and $C_2^{(\alpha)}$ respectively.

Given that we have shown $C_1^{(\alpha)}$ and $C_2^{(\alpha)}$ to be similar to D_1 and D_2 respectively with the change of basis matrix, V, being the N by N matrix of eigenvectors, we can rewrite $C_1^{(\alpha)} \otimes I + C_2^{(\alpha)} \otimes B$ as follows (via replacement):

$$C_1^{(\alpha)} \otimes I + C_2^{(\alpha)} \otimes B = (VD_1V^{-1}) \otimes I + (VD_2V^{-1}) \otimes B$$

To show the equivalence of $(VD_1V^{-1}) \otimes I + (VD_2V^{-1}) \otimes B$ with $(V \otimes I)(D_1 \otimes I + D_2 \otimes B)(V^{-1} \otimes I)$, we will make use of the mixed-product property of the tensor product along with its associative property which is given below:

$$\begin{aligned} A \otimes (B + C) &= A \otimes B + A \otimes C \\ (B + C) \otimes A &= B \otimes A + C \otimes A \\ (A \otimes B)(C \otimes D) &= (AC) \otimes (BD) \end{aligned}$$

(link: https://en.wikipedia.org/wiki/Kroneckerproduct)

Starting with $(V \otimes I)(D_1 \otimes I + D_2 \otimes B)(V^{-1} \otimes I)$, we obtain $(VD_1V^{-1}) \otimes I + (VD_2V^{-1}) \otimes B$ using the above properties:

$$\begin{aligned} &(V \otimes I)(D_1 \otimes I + D_2 \otimes B)(V^{-1} \otimes I) \\ &= (V \otimes I)(D_1 \otimes I)(V^{-1} \otimes I) + (V \otimes I)(D_2 \otimes B)(V^{-1} \otimes I) \\ &= (VD_1 \otimes I)(V^{-1} \otimes I) + (VD_2 \otimes B)(V^{-1} \otimes I) \\ &= (VD_1V^{-1} \otimes I) + (VD_2V^{-1} \otimes B) \\ &= (VD_1V^{-1}) \otimes I + (VD_2V^{-1}) \otimes B \end{aligned}$$

showing equivalence of all the expressions. The first to the second line use the associative property and the other lines of the above proof use the mixed-product property.

5.4 d)

To see how the Fourier transform can be used to obtain $(V \otimes I)U$, we will write $(V \otimes I)U$ in terms of the above, deduced eigenvectors which use the exponential function. Starting off, we consider the definition of matrix multiplication as per the beginning of the course, which sees the output as a linear combination of the columns of the matrix with the coefficients given by the vector with which it is being multiplied.

We will denote x_j as the $(j+1)$th column of V and $v_{i,j}$ as the $(i+1, j+1)$ entry of V whose value we know from part c of the assignment.

$$\begin{aligned} (V \otimes I)U &= ((x_0, x_1, \ldots, x_{N-1}) \otimes I)U \\ &= \sum_{j=0}^{N-1} (x_j \otimes I) \begin{pmatrix} p_{j+1} \\ q_{j+1} \end{pmatrix} \\ &= \sum_{j=0}^{N-1} \begin{pmatrix} v_{0,j} \begin{pmatrix} p_{j+1} \\ q_{j+1} \end{pmatrix} \\ \vdots \\ v_{N-1,j} \begin{pmatrix} p_{j+1} \\ q_{j+1} \end{pmatrix} \end{pmatrix} \\ &= \begin{pmatrix} \alpha^{-0/N} \sum_{j=0}^{N-1} e^{\frac{2\pi i j * 0}{N}} \begin{pmatrix} p_{j+1} \\ q_{j+1} \end{pmatrix} \\ \vdots \\ \alpha^{-(N-1)/N} \sum_{j=0}^{N-1} e^{\frac{2\pi i j * (N-1)}{N}} \begin{pmatrix} p_{j+1} \\ q_{j+1} \end{pmatrix} \end{pmatrix} \end{aligned}$$

$$= \begin{pmatrix} \alpha^{-0/N} & 0 & 0 & \dots & 0 \\ 0 & \alpha^{-1/N} & 0 & \dots & 0 \\ 0 & 0 & \alpha^{-2/N} & \dots & 0 \\ \vdots & & & \ddots & \vdots \\ 0 & \dots & 0 & 0 & \alpha^{-(N-1)/N} \end{pmatrix} \begin{pmatrix} \sum_{j=0}^{N-1} e^{\frac{2\pi ij*0}{N}} \begin{pmatrix} p_{j+1} \\ q_{j+1} \end{pmatrix} \\ \vdots \\ \sum_{j=0}^{N-1} e^{\frac{2\pi ij*(N-1)}{N}} \begin{pmatrix} p_{j+1} \\ q_{j+1} \end{pmatrix} \end{pmatrix}$$

$$= \begin{pmatrix} N\alpha^{-0/N} & 0 & 0 & \dots & 0 \\ 0 & N\alpha^{-1/N} & 0 & \dots & 0 \\ 0 & 0 & N\alpha^{-2/N} & \dots & 0 \\ \vdots & & & \ddots & \vdots \\ 0 & \dots & 0 & 0 & N\alpha^{-(N-1)/N} \end{pmatrix} \begin{pmatrix} \frac{1}{N}\sum_{j=0}^{N-1} e^{\frac{2\pi ij*0}{N}} \begin{pmatrix} p_{j+1} \\ q_{j+1} \end{pmatrix} \\ \vdots \\ \frac{1}{N}\sum_{j=0}^{N-1} e^{\frac{2\pi ij*(N-1)}{N}} \begin{pmatrix} p_{j+1} \\ q_{j+1} \end{pmatrix} \end{pmatrix}$$

The final line above is the product of a diagonal matrix with a vector whose elements all resemble the discrete inverse Fourier transform which transforms a sequence of numbers Y_n into another sequence y_n. As a reminder, the discrete inverse Fourier transform takes the following form:

$$y_n = \frac{1}{N}\sum_{j=0}^{N-1} Y_j \cdot e^{\frac{2\pi ijn}{N}}$$

Defining $Y_j = \begin{pmatrix} p_{j+1} \\ q_{j+1} \end{pmatrix}$ and setting $y_n = \frac{1}{N}\sum_{j=0}^{N-1} e^{\frac{2\pi ijn}{N}} \begin{pmatrix} p_{j+1} \\ q_{j+1} \end{pmatrix}$ allows us to obtain $(V \otimes I)U$ via the product of a diagonal matrix with a vector composed of the inverse Fourier transform of $\begin{pmatrix} p_{j+1} \\ q_{j+1} \end{pmatrix}$.

Obtaining $(V^{-1} \otimes I)U$ is equivalent to inverting the above process since $(V \otimes I)U^{-1} = (V^{-1} \otimes I)U$. This simply entails multiplying U on the left by D^{-1} (given below) and then performing the Fourier transform on the result (which is also given below):

$$D^{-1} = \begin{pmatrix} \frac{1}{N}\alpha^{0/N} & 0 & 0 & \dots & 0 \\ 0 & \frac{1}{N}\alpha^{1/N} & 0 & \dots & 0 \\ 0 & 0 & \frac{1}{N}\alpha^{2/N} & \dots & 0 \\ \vdots & & & \ddots & \vdots \\ 0 & \dots & 0 & 0 & \frac{1}{N}\alpha^{(N-1)/N} \end{pmatrix}$$

$$Y_n = \sum_{j=0}^{N-1} y_j \cdot e^{-\frac{2\pi ijn}{N}}$$

5.5 e)

The inverse of the tensor product of two matrices is used in showing how steps (i), (ii) and (iii) in the assignment provide a method for solving (3). The inverse is given below and the result is obtained from the aforementioned link:

$$(A \otimes B)^{-1} = A^{-1} \otimes B^{-1}$$

Using the result obtained from part c of this question, we can rewrite (3) as follows:

$$(V \otimes I)(D_1 \otimes I + D_2 \otimes B)(V^{-1} \otimes I)U^{k+1} = R$$

Multiplying on the left by $(V \otimes I)^{-1}$ and using the inverse of the tensor product of two matrices yields:

$$(V \otimes I)^{-1}(V \otimes I)(D_1 \otimes I + D_2 \otimes B)(V^{-1} \otimes I)U^{k+1} = (V \otimes I)^{-1}R$$
$$\Rightarrow (D_1 \otimes I + D_2 \otimes B)(V^{-1} \otimes I)U^{k+1} = (V^{-1} \otimes I)R = \hat{R}$$

giving us step (i).

Defining $\hat{U}^{k+1} = (V^{-1} \otimes I)U^{k+1}$, we can write the previous result as:

$$(D_1 \otimes I + D_2 \otimes B)\hat{U}^{k+1} = \hat{R}$$

Given the structure of the diagonal matrices when tensor multiplied by another matrix, the jth time slice (for $j = 1, 2, \ldots, N$) interacts only with the jth diagonal entry of D_1 and D_2 which we call $d_{1,j}$ and $d_{2,j}$ respectively. Solving the system for each time slice j leads to to step (ii).

With $\hat{U}^k$ computed, all that is left to do is to return back to U^k which is achieved by the following:

$$\begin{aligned} \hat{U}^{k+1} &= (V^{-1} \otimes I)U^{k+1} \\ \Rightarrow (V^{-1} \otimes I)^{-1}\hat{U}^{k+1} &= U^{k+1} \\ \Rightarrow (V \otimes I)\hat{U}^{k+1} &= U^{k+1} \end{aligned}$$

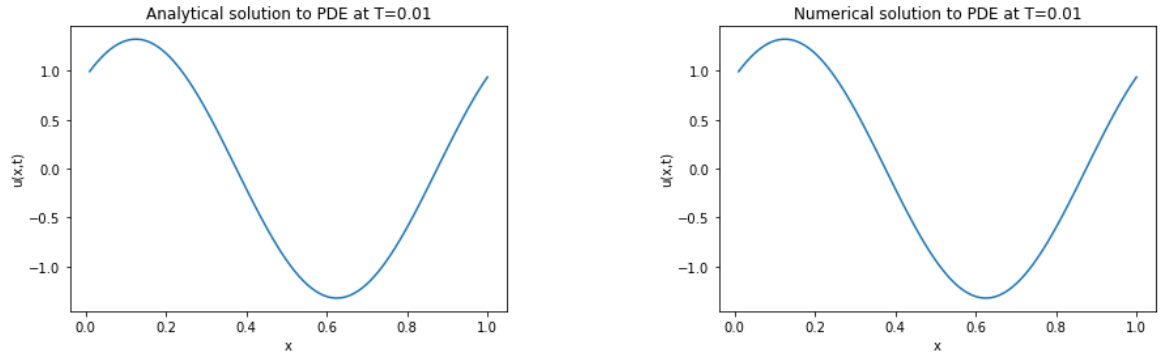
giving us the final step (iii).

5.6 f)

The iterative scheme is stable and so the vector of zeros can be used as our initial guess for U^0. The function that implements the above algorithm is called "numsol q5" and this uses the same periodic function as in question 2, namely:

$$u(x,t) = \cos(2\pi(x+t)) + \sin(2\pi(x-t))$$

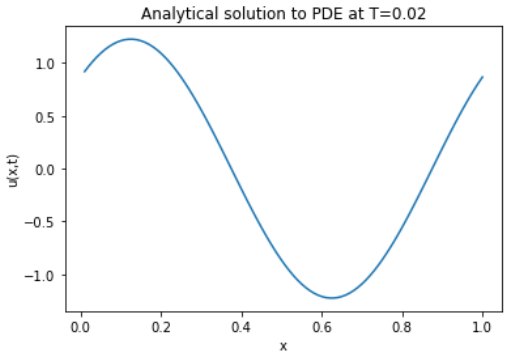
The output is an M by N matrix which gives the solutions from time dt to $N * dt$ across all space steps between 0 and 1, incremented by dx. To test the convergence of U, the error between each successive iteration is recorded using the norm function and if it is below a predetermined tolerance, then the algorithm terminates and uses the latest value of U^k. As the iterative method should converge to a true solution, one would expect the errors to monotonically decrease at each iteration; this property is tested.



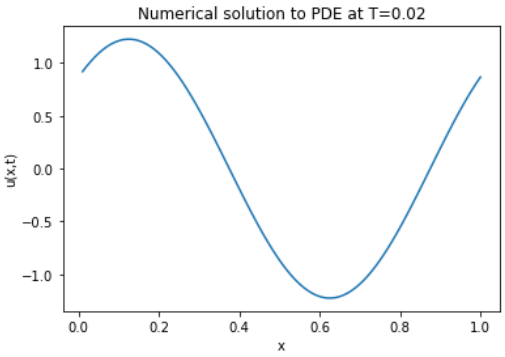
(a) Analytical Solution

(b) Numerical Solution, $\alpha = 0.1$ $dt = 0.01$

Figure 15: Analytical solution (left) and Numerical solution to PDE, $dx = 0.01$ at $T = 0.01$

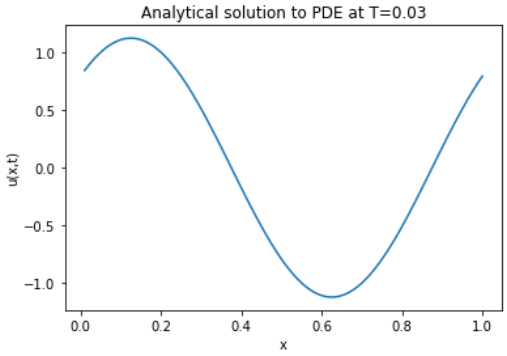


(a) Analytical Solution

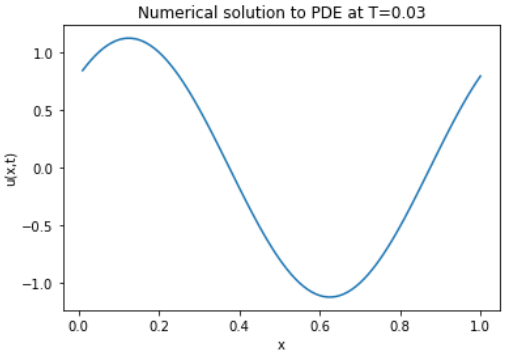


(b) Numerical Solution, $\alpha = 0.1$ $dt = 0.01$

Figure 16: Analytical solution (left) and Numerical solution to PDE, $dx = 0.01$ at $T = 0.02$



(a) Analytical Solution



(b) Numerical Solution, $\alpha = 0.1$ $dt = 0.01$

Figure 17: Analytical solution (left) and Numerical solution to PDE, $dx = 0.01$ at $T = 0.03$

As can be seen from figures 15, 16 and 17, the numerical and analytical solutions bear much resemblance, corroborating the method used in this exercise to simultaneously solve the PDE at different time steps. This method is also tested by comparing the difference between the numerical and analytical solution, ensuring that they lie within a certain tolerance.