Computational Linear Algebra Coursework 2

George Hilton CID: 01494579

1

a

During this question we assume a solution always exists and that $A \in \mathbb{R}^{mxm}$ and $x, b \in \mathbb{R}^m$.

We're tasked with using a variation of LU factorisation to solve the system $Ax = b$ where A is a tridiagonal symmetric matrix. As A is tridiagonal and $A = LU$ with L lower triangular and U upper triangular, we can see that the forms of L and U are as follows.

Noting firstly that,

$$A = \begin{bmatrix} c & d & 0 & 0 & \dots & 0 & 0 & 0 \\ d & c & d & 0 & \dots & 0 & 0 & 0 \\ 0 & d & c & d & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & d & c & d \\ 0 & 0 & 0 & 0 & \dots & 0 & d & c \end{bmatrix} \in \mathbb{R}^{mxm}$$

we must have,

$$L = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 & 0 \\ l_2 & 1 & 0 & \dots & 0 & 0 \\ 0 & l_3 & 1 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & l_m & 1 \end{bmatrix}, \quad U = \begin{bmatrix} u_1 & d & 0 & \dots & 0 & 0 \\ 0 & u_2 & d & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & u_{m-1} & d \\ 0 & 0 & 0 & \dots & 0 & u_m \end{bmatrix}.$$

to ensure that all the conditions are met. Let's construct the method then derive the algorithm for it.

Method:

1. Write $Ax = b$ as $LUx = b$ using LU Decomposition.
2. Set $Ux = y$ and observe that $LUx = b$ becomes $Ly = b$.
3. Solve $Ly = b$ for y using forward substitution.
4. Solve $Ux = y$ for x using back substitution.

Now, let's derive the new algorithms. For $A = LU$ we see that:

1. $c = u_1 \implies u_1 = c$
2. $d = l_k u_{k-1} \implies l_k = \frac{d}{u_{k-1}}$ for $k = 2, ..., n$
3. $c = l_k d + u_k \implies u_k = c - l_k d$ for $k = 2, ..., n$

In addition to this, we can see that there can be significant reductions to the complexity of forward and back substitution due to the number of zeros in L and U. So, we can remove all computations which involve zeros in these calculations.

At this point we can write the pseudo-code for the LU variation and method to solve the matrix vector system.

Algorithm 1 LU Factorisation for Tridiagonal Matrices

$u_1 = c$
for $k = 2, 3, \ldots, m$ **do**
    $l_k = \frac{d}{u_{k-1}}$
    $u_k = c - l_k d$
end for

Algorithm 2 Forward substitution to solve Ly=b

$y_1 = b_1$
for $k = 2, 3, \ldots, m$ **do**
    $y_k = b_k - l_k y_{k-1}$
end for

Algorithm 3 Back substitution to solve Ux=y

$x_m = \frac{y_m}{u_m}$

for $k = m-1, m-2, \ldots, 1$ **do**

    $x_k = \frac{y_k - dx_{k+1}}{u_k}$

end for

b

Considering Algorithm 1 and Algorithm 2 above, we notice that the 'for' loops can be combined; they both iterate over $k = 2, 3, ..., m$. So, ensuring that we define y_k and u_k after l_k we can combine the algorithms into one, which solves the tridiagonal matrix-vector system.

Algorithm 4 Combination of Algorithm 1,2 and 3

$u_1 = c$

$y_1 = b_1$

for $k = 2, 3, \ldots, m$ **do**

    $l_k = \frac{d}{u_{k-1}}$

    $u_k = c - l_k d$

    $y_k = b_k - l_k y_{k-1}$

end for

$x_m = \frac{y_m}{u_m}$

for $k = m-1, m-2, \ldots, 1$ **do**

    $x_k = \frac{y_k - dx_{k+1}}{u_k}$

end for

c

For the new LU factorisation, there are three main operations:

1. $\frac{d}{u_{k-1}}$
2. $l_k * d$
3. $c - l_k d$.

These are completed over $(m-1)$ loops so we have $3(m-1)$ FLOPS. For the new forward substitution, there are two operations

1. $l_k * y_{k-1}$

2. $b_k - l_k y_{k-1}$.

These are also completed over $(m-1)$ loops so we have $2(m-1)$ FLOPS. For the new back substitution there are three main operations

1. $d * x_{k+1}$

2. $y_k - dx_{k+1}$

3. $\frac{y_k - dx_{k+1}}{u_k}$.

Once again, this is performed for $(m-1)$ loops and as such, there are $3(m-1)$ FLOPS.

Asymptotically, we have a total number of FLOPS of $\approx 8m$ or we can say the algorithm has a time complexity of $\mathcal{O}(m)$. From lectures, we know that classical LU Factorisation has $\frac{2}{3}m^3$ FLOPS, both classical forward and back substitution have m^2 FLOPS. Hence, we see that the time complexity of the regular LU method is $\mathcal{O}(m^3)$. Clearly, for very large m there is a significant reduction in computation time for the LU variation derived above.

d

These algorithms have been implemented in q1.py and tested in test_q1.py; I used an in-place system to reduce memory usage and improve efficiency. Running the tests we see that tridiagLUsolve works correctly to solve $Ax = b$ systems when A is symmetric tridiagonal and tridiagLU correctly performs LU decomposition on tridiagonal matrices.

2

a

In this question all vectors we use have dimension m and all matrices have dimension mxm.

Starting at equation (4) within the questions we have

$$w^{n+1} - w^n - \frac{\Delta t}{2}(u_{xx}^n + u_{xx}^{n+1}) = 0 \quad (\dagger^2), \quad u^{n+1} - u^n - \frac{\Delta t}{2}(w^n + w^{n+1}) = 0$$

where $w^{n+1}(x)$ is the approximation to $w(x,(n+1)\Delta t)$ and similar for $u^{n+1}(x)$. Taking partial derivatives, we see

$$\begin{aligned}
&\frac{\partial^2}{\partial x^2}(u^{n+1}-u^n-\frac{\Delta t}{2}(w^n+w^{n+1}))=0\\
\implies\ &u_{xx}^{n+1}-u_{xx}^n-\frac{\Delta t}{2}(w_{xx}^n+w_{xx}^{n+1})=0 \quad (\dagger\dagger^2).
\end{aligned}$$

At this point we can add together equations $(\dagger^2)$ and $(\dagger\dagger^2)$ and manipulate as follows,

$$\begin{aligned}
&\left[\frac{2}{\Delta t}(w^{n+1}-w^n)-(u_{xx}^n+u_{xx}^{n+1})\right]+\left[u_{xx}^{n+1}-u_{xx}^n-\frac{\Delta t}{2}(w_{xx}^n+w_{xx}^{n+1})\right]=0\\
&\implies \frac{2}{\Delta t}(w^{n+1}-w^n)-2u_{xx}^n-\frac{\Delta t}{2}(w_{xx}^n+w_{xx}^{n+1})=0\\
&\implies w^{n+1}-w^n-\Delta t u_{xx}^n-\left(\frac{\Delta t}{2}\right)^2(w_{xx}^n+w_{xx}^{n+1})=0\\
&\implies w^{n+1}-\frac{(\Delta t)^2}{4}w_{xx}^{n+1}=w^n+\Delta t u_{xx}^n+\frac{(\Delta t)^2}{4}w_{xx}^n. \quad (\dagger\dagger\dagger^2)
\end{aligned}$$

At this point we have the form of equation (5) with $C=\frac{(\Delta t)^2}{4}$ and $f=w^n+\Delta t u_{xx}^n+\frac{(\Delta t)^2}{4}w_{xx}^n$.

Now we discretise this equation using the central difference formula [1]. We have $w_i^{n+1}\approx w(i\Delta x,(n+1)\Delta t)$ and the central difference formula of $f''(x)=\frac{f(x+h)-2f(x)+f(x-h)}{h^2}$.

Setting $h=\Delta x$, we have

$$w_{xx}^{n+1}=\frac{w_{i+1}^{n+1}-2w_i^{n+1}+w_{i-1}^{n+1}}{(\Delta x)^2},\quad u_{xx}^n=\frac{u_{i+1}^n-2u_i^n+u_{i-1}^n}{(\Delta x)^2}.$$

Subbing these expressions into equation (5) and noting that $\Delta x=M^{-1}$ we get,

$$\begin{aligned}
\text{LHS of } (\dagger\dagger\dagger^2)&=w^{n+1}-\frac{(\Delta t)^2}{4}w_{xx}^{n+1}\\
&\implies =w_i^{n+1}-\frac{(\Delta t)^2}{4}\left[\frac{w_{i+1}^{n+1}-2w_i^{n+1}+w_{i-1}^{n+1}}{(\Delta x)^2}\right]\\
&\implies =w_i^{n+1}-\frac{(M\Delta t)^2}{4}\left(w_{i+1}^{n+1}-2w_i^{n+1}+w_{i-1}^{n+1}\right)
\end{aligned}$$

and,

$$\begin{aligned}
\text{RHS of } (\dagger\dagger\dagger^2)&=w^n+\Delta t u_{xx}^n+\frac{(\Delta t)^2}{4}w_{xx}^n\\
&\implies =w_i^n+\Delta t\left[\frac{u_{i+1}^n-2u_i^n+u_{i-1}^n}{(\Delta x)^2}\right]+\frac{(\Delta t)^2}{4}\left[\frac{w_{i+1}^n-2w_i^n+w_{i-1}^n}{(\Delta x)^2}\right]\\
&\implies =w_i^n+M^2\Delta t\left(u_{i+1}^n-2u_i^n+u_{i-1}^n\right)+\frac{(M\Delta t)^2}{4}\left(w_{i+1}^n-2w_i^n+w_{i-1}^n\right).
\end{aligned}$$

As desired, our equation ($\dagger\,\dagger\,\dagger^2$) now takes the form of (6) with,

$$C_1 = \frac{(M\Delta t)^2}{4}$$
$$f_i = w_i^n + M^2\Delta t\left(u_{i+1}^n - 2u_i^n + u_{i-1}^n\right) + \frac{(M\Delta t)^2}{4}\left(w_{i+1}^n - 2w_i^n + w_{i-1}^n\right).$$

b

We can write the equation $w_i^{n+1} - C_1(w_{i+1}^{n+1} - 2w_i^{n+1} + w_{i-1}^{n+1}) = f_i$ in matrix vector form. Firstly, note that for $i = 2, 3, ..., M-1$ we have

$$f_i = -C_1 w_{i+1}^{n+1} + (1 + 2C_1)w_i^{n+1} - C_1 w_{i-1}^{n+1}.$$

For the cases when $i = 1$ and $i = M$ we have to recall the periodic conditions outlined in the question. As such, we have $w_0^{n+1} = w_M^{n+1}$ and $w_{M+1}^{n+1} = w_1^{n+1}$. This gives

$$\begin{aligned} f_1 &= -C_1 w_2^{n+1} + (1 + 2C_1)w_1^{n+1} - C_1 w_0^{n+1} \\ \implies f_1 &= -C_1 w_2^{n+1} + (1 + 2C_1)w_1^{n+1} - C_1 w_M^{n+1} \\ f_M &= -C_1 w_{M+1}^{n+1} + (1 + 2C_1)w_M^{n+1} - C_1 w_{M-1}^{n+1} \\ \implies f_M &= -C_1 w_1^{n+1} + (1 + 2C_1)w_M^{n+1} - C_1 w_{M-1}^{n+1} \end{aligned}$$

It is now simple to write this system of equations in matrix-vector form as

$$\begin{bmatrix} 1+2C_1 & -C_1 & 0 & \dots & 0 & -C_1 \\ -C_1 & 1+2C_1 & -C_1 & \dots & 0 & 0 \\ 0 & -C_1 & 1+2C_1 & \dots & 0 & 0 \\ 0 & 0 & -C_1 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 1+2C_1 & -C_1 \\ -C_1 & 0 & 0 & \dots & -C_1 & 1+2C_1 \end{bmatrix} \begin{bmatrix} w_1^{n+1} \\ w_2^{n+1} \\ w_3^{n+1} \\ w_4^{n+1} \\ \vdots \\ w_{M-1}^{n+1} \\ w_M^{n+1} \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \\ \vdots \\ f_{M-1} \\ f_M \end{bmatrix}.$$

c

If we write the above system as $Aw = f$ for simplicity, we see there is no advantage in using a banded matrix LU-Decomposition Algorithm. Although the majority of the elements of A lie on the sub, upper and main diagonal, there are elements in the top right and bottom left so the matrix has full bandwidth. As such, a banded algorithm would have to iterate as many times as the normal algorithm to reach these elements and is no more efficient.

d

Setting,

$$u_1 = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ -C_1 \end{bmatrix}, \quad v_1 = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \quad u_2 = \begin{bmatrix} -C_1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \quad v_2 = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix}$$

We have that,

$$u_1 v_1^T = \begin{bmatrix} 0 & \dots & 0 \\ \vdots & \ddots & \vdots \\ -C_1 & \dots & 0 \end{bmatrix}, \quad u_2 v_2^T = \begin{bmatrix} 0 & \dots & -C_1 \\ \vdots & \ddots & \vdots \\ 0 & \dots & 0 \end{bmatrix}.$$

In addition to this, setting

$$T = \begin{bmatrix} 1+2C_1 & -C_1 & 0 & \dots & 0 & 0 \\ -C_1 & 1+2C_1 & -C_1 & \dots & 0 & 0 \\ 0 & -C_1 & 1+2C_1 & \dots & 0 & 0 \\ 0 & 0 & -C_1 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 1+2C_1 & -C_1 \\ 0 & 0 & 0 & \dots & -C_1 & 1+2C_1 \end{bmatrix}$$

we see very clearly that $A = T + u_1 v_1^T + u_2 v_2^T$ where T is tridiagonal.

Now we want to write A^{-1} in terms of T^{-1}. Writing $M = T + u_1 v_1^T$ and noting the Sherman-Morrison Formula [2], $(A + uv^T)^{-1} = A^{-1} - \frac{A^{-1}uv^T A^{-1}}{1+v^T A^{-1}u}$, we have

$$M^{-1} = T^{-1} - \frac{T^{-1}u_1 v_1^T T^{-1}}{1 + v_1^T T^{-1} u_1}.$$

Now looking at A^{-1} we see,

$$A^{-1} = (M + u_2 v_2^T)^{-1}$$
$$\implies A^{-1} = M^{-1} - \frac{M^{-1}u_2 v_2^T M^{-1}}{1 + v_2^T M^{-1} u_2}$$

Subbing the formula for M^{-1} into this equation gives

$$A^{-1} = T^{-1} - \frac{T^{-1}u_1v_1^TT^{-1}}{1+v_1^TT^{-1}u_1} - \frac{\left(T^{-1} - \frac{T^{-1}u_1v_1^TT^{-1}}{1+v_1^TT^{-1}u_1}\right)u_2v_2^T\left(T^{-1} - \frac{T^{-1}u_1v_1^TT^{-1}}{1+v_1^TT^{-1}u_1}\right)}{1+v_2^T\left(T^{-1} - \frac{T^{-1}u_1v_1^TT^{-1}}{1+v_1^TT^{-1}u_1}\right)u_2}$$

e

Now we are faced with solving the matrix-vector system $Ax = b$ where A takes the from as above. To improve computational efficiency we also do not want to explicitly construct T^{-1} at any point.

To do this we solve $x = A^{-1}b$; we will consider the form of A^{-1} involving M^{-1}. So,

$$\begin{aligned} & x = A^{-1}b \\ \implies & x = M^{-1}b - \frac{M^{-1}u_2v_2^TM^{-1}b}{1+v_2^TM^{-1}u_2} \end{aligned}$$

notice the structure of M^{-1} applied to a vector,

$$M^{-1}y = T^{-1}y - \frac{T^{-1}u_1v_1^TT^{-1}y}{1+v_1^TT^{-1}u_1}.$$

We see that to solve $A^{-1}b$ we need to construct $M^{-1}b$ and $M^{-1}u_2$. To do this we must first compute $T^{-1}b$, $T^{-1}u_1$ and $T^{-1}u_2$. Each of which can be obtained by solving $Ty_1 = b$, $Ty_2 = u_1$ and $Ty_3 = u_2$ respectively. It is very efficient to solve these using the method devised in Q1. Then, combining these components as above involves: the inner product of m dimensional vectors, scalar multiplication and addition of the vectors as well as scalar addition. Hence, solving the system $Ax = b$ in this case be reduced to these much more simple, computationally efficient problems.

With regards to the operation count, we have to first construct $T^{-1}b$, $T^{-1}u_1$ and $T^{-1}u_2$ - this uses 24(m-1) FLOPS ($T \in \mathbb{R}^{mxm}$). Then we have several inner products and scalar-vector multiplications which are $(2m-1)$ and m FLOPS respectively. In addition to this there is addition of vectors and scalars, scalar addition is negligible in this environment and the vector addition uses m FLOPS . So, the asymptotic time complexity of the system is $\mathcal{O}(m)$.

f

I have implemented this algorithm in q2funcs.py; we also test whether the code produces a correct solution in test_q2.py - it passses the test so we know the algorithm is working as we want. Another file is included namely, q2timings.py, which compares the time taken for solve_LUP and q2solve to solve the same system. Running the file, we see timings printed for both functions in the $m = 25$ and $m = 100$ case (Figure: 1). The q2solve.py function is more efficient and has a shorter computation time as we would expect based on the reduced time complexity; it appears to be approximately 10 times faster than solve_LUP.

g

To reduce the number of global variables I created a function to perform the duties of the script mentioned in the question. This is implemented in q2.py: the function takes $N, M, \Delta T, r$ as its arguments, as well as a truth value which governs whether the function saves the solutions (at different timesteps) or plots them. The variables are defined as we would expect, r gives the number of solutions we plot/save. A brief check to confirm that the function is working as we want can be completed using the plot variation of the function. Setting, $u_0(x) = 0$ and $u_1(x) = cos(2\pi x)$ should produce a standing wave of the same for as $cos(2\pi x)$ - and running the function in the file we see just that. As such, we can be relatively confident that algorithm/code is working correctly.

3

a

Claim:

QR Algorithm applied to a tridiagonal symmetric matrix preserves the tridiagonal structure.

Proof:

Let A be a tridiagonal symmetric matrix.

$Q^T A = R \implies Q_n ... Q_2 Q_1 A = R$ where we have used Householder reflections. We know

R is non-zero strictly on the diagonal and two super diagonals above that. When performing the QR Algorithm at each step we compute $A_k = QR$ then set $A_{k+1} = RQ$, from this we can prove that the tridiagonal and symmetric structure is preserved at each iteration.

Firstly we'll show that if A_k is symmetric then so is A_{k+1}. Assume $A_k = QR$ is symmetric.

$$A_{k+1} = RQ = Q^T(QR)Q = Q^{-1}A_kQ.$$

As A_k is similar to A_{k+1} we know that our assumption - A_k is symmetric - implies that A_{k+1} is also symmetric. Now, we will now prove by induction that A_k is tridiagonal at each step.

Base case is trivial as we set A to be tridiagonal.

Assume that A_k is tridiagonal for $k = 0, 1, ..., m$. So,

$$A_m = Q_1^T...Q_m^TR \implies A_{m+1} = RQ_1^T...Q_m^T.$$

As R is upper triangular, the only non-zero subdiagonal entry in RQ_1^T is in (2,1). For $RQ_1^TQ_2^T$ the only non-zero subdiagonal entries are in (2,1) and (3,2). So, inductively we see that the only non-zero subdiagonal entries of $RQ_1^T...Q_m^T$ are in $(j+1, j)$ for $j = 2, ..., m$. So we see the lower triangular part of A_{m+1} only has zeros on the subdiagonal. As we know A_{m+1} is symmetrical, we see that A_{m+1} is tridiagonal. So, our inductive hypothesis is true. We see that the QR Algorithm applied to symmetric tridiagonal matrices preserves the tridiagonal structure. □

b

If we consider a symmetric tridiagonal matrix of the form

$$A = \begin{bmatrix} a & b & 0 & 0 & \dots & 0 & 0 & 0 \\ b & a & b & 0 & \dots & 0 & 0 & 0 \\ 0 & b & a & b & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & b & a & b \\ 0 & 0 & 0 & 0 & \dots & 0 & b & a \end{bmatrix} \in \mathbb{R}^{mxm}$$

we can improve the efficiency of QR Decomposition by householder reflections. To do this, we ignore the non-zero entries beneath the subdiagonal and apply 2x2 householder reflectors to

each 2x3 submatrix along the diagonal of A (A[k:k+1, k:k+2] for $k = 1, ..., m-2$) and then on the final 2x2 submatrix. This will convert A inplace to the upper triangular matrix, R ($A = QR$).

The operation count of performing this new QR Decomposition is far smaller than the traditional method. The work is dominated by two separate processes. One is constructing v_k each iteration and the other, is applying the householder reflections. Constructing v_k we multiply two constants with a 2-d vector, then add another 2-d vector. So we have $(1+2+2) = 5$ FLOPS over the first $(m-1)$ loops which gives $5(m-1)$ total FLOPS. For the householder reflector we perform 30 multiplications and additions over the first $(m-2)$ loops and 20 from the final one. Giving a total of $30(m-2)+20$ FLOPS. Asymptotically we see this gives $\approx 30m$ FLOPS for the householder component and $\approx 5m$ FLOPS for the construction of v_k. So in total we have approximately $35m$ FLOPS or a time complexity of $\mathcal{O}(m)$. Traditional householder has approximately $2m^3 - \frac{2}{3}m^3$ FLOPS or a time complexity of $\mathcal{O}(m^3)$ so we have created a more efficient method for $m \geq 6$, particularly for large m.

c

In q3funcs.py I have implemented the qr_factor_tri function. In test_q3.py the function is tested - we assert whether R is upper triangular and whether $\left\|R^T R - A^T A\right\| < 1.0\text{x}10^{-6}$. It passes all the tests, so we know the function is working correctly. As stipulated in the question, it stores the vectors v_k at each iteration and never explicitly computes $Q \in \mathbb{C}^{mxm}$.

d

In q3funcs.py I have implemented the qr_alg_tri function and tested it within test_q3.py, for this question we want both boolean arguments to be set to 'False'. We assert whether the norm of a vector containing the predicted eigenvalues is similar to the norm of a vector containing the actual eigenvalues. The tests pass so we know the function is working corrrectly and returning the eigenvalues along the diagonal of the matrix.

As specified within the question, we want to test the function applied to a specific matrix, namely $A_{ij} = 1/(1+i+j)$ where $A \in \mathbb{R}^{5\text{x}5}$. To study the results I wrote a scipt in q3.py which

displays a heatmap (Figure: 2) of the resulting matrix , T as well as a calculation of the 'error' (Figure: 3). Here, I have chosen to determine $error = \|Diagonal(T)\| - \|V\|$. Where V is a vector of the eigenvalues determined using numpy.linalg.eig. Initially, viewing Figure: 3 we see that the error in the calculation is of order 10^{-11}; this demonstrates clearly that our algorithm is working very accurately on this matrix. Viewing Figure: 2, the heatmap of T we see that two of the eigenvalues are notably larger than zero, but beyond that we see the last three eigenvalues (lower three diagonal elements) are almost/equivalent to zero.

e

The script referenced in this question is implemented by a combination of qr_alg_tri with the 'mod' argument set to 'True' and qr_alg with the 'shift' argument set to 'False'.

The code is tested within test_q3.py in the same way as the previous part, the function passes all the tests so we know it is working correctly.

To study how the function works, we plotted the $|T_{m,m-1}|$ values for a random SPD matrix (reduced to Hessenberg) and the aforementioned A_{ij}. This code is included in q3.py and the plots are Figure: 4 and Figure: 5. For all cases we see there is very rapid decrease of $|T_{m,m-1}|$ towards zero with slight deviation in pattern for each example. In Figure : 4 we see the values approach zero linearly and very quickly and then stay very close to zero. In Figure: 5 there is more variability and a small second peak but ultimately $|T_{m,m-1}|$ decays very quickly to zero once again.

We also want to compare the convergence of this algorithm to pure_QR. The convergence criteria is different for both algorithms but as both generate eigenvalues we can consider convergence in a slightly different way. Ultimately, the faster the convergence, the less time the functions will take to return the eigenvalues. As such, I chose to determine convergence in this fashion - timing both functions on several different size matrices (note we set the tolerance to 10^{-12} in pure_QR). The results are plotted in Figure: 6; we see that on a logarithmic scale there is a clear decrease in computation time from pure_QR to our algorithm; this suggests that convergence is faster.

f

The next task was to implement the previous algorithm but with a Wilkinson Shift described in the question. This is implemented in qr_alg by setting the boolean argument to 'True'; this is found within the q3funcs.py file. It passes the same type of test once again, found in test_q3.py so we know it is working correctly. We can compare how this modification changes the $|T_{m,m-1}|$ plots for A_{ij} using Figure: 7. We see that the line initially drops less rapidly for the shifted algorithm than the original however, it reaches zero sooner. So, convergence occurs more quickly for the shifted algorithm.

In addition, we can compare how the shifted algorithm operates on the random SPD matrix. This result is very interesting. Viewing Figure: 8 we now see two large peaks and mostly zero for the rest of the values. We see very rapid convergence at all parts but less consistency when compared to the normal algorithm which is most likely due to the randomness of the matrices we're testing on.

A final and interesting comparison of convergence between the shifted and unshifted version of qr_alg can be seen in Figure: 9 where I timed the functions like before. Once again, I used a logarithmic scale for clarity; as we might expect, the shifted algorithm converges far quicker than the unshifted version for all sizes of matrices tested.

g

In this question, we make some comparisons between plots for A_{ij} and a new matrix, A. We define $A = D + O$ where,

$$A = \begin{bmatrix} 16 & 1 & 1 & \dots & 1 & 1 \\ 1 & 15 & 1 & \dots & 1 & 1 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 1 & 1 & 1 & \dots & 3 & 1 \\ 1 & 1 & 1 & \dots & 1 & 2 \end{bmatrix}.$$

Viewing the $|T_{m,m-1}|$ plots for the unshifted algorithm applied to A in Figure: 10 there is a small upwards spike then an extremely rapid decrease of $|T_{m,m-1}|$ downwards with this realisation of the function converging quickly. Comparing with Figure: 11 (the shifted version) we see no upward spike but a downward line. The normal qr_alg algorithm $|T_{m,m-1}|$ values drop more rapidly initially, but take a greater time to converge to the tolerance than the shifted version.

For A there is far faster reduction in $|T_{m,m-1}|$ both with and without a Wilkinson shift compared to A_{ij}. It also appears that the Wilkinson shift preserves the way the $|T_{m,m-1}|$ values convergence. For example it appears the values converge linearly in the A_{ij} case and much faster for the A case.

4

a

The changes requested have been implemented by adding an optional boolean argument to GMRES within cla_utils . When this argument apply_pc is 'None' it uses the normal GMRES algorithm however, if we set apply_pc to be a function (which works as defined in the question) it uses the new method. I tested the function in test_q4 .py using diagonal preconditioning; we assert whether the solution it generates to a matrix-vector system is correct. The function passes the test, so we can be confident that it is working as we would expect it to.

b

We will use without proof, the fact that $M^{-1}A$ is diagonalisable during this question.

Assume that the matrix A and the preconditioning matrix M satisfy

$$\left\|I - M^{-1}A\right\|_2 = c < 1$$

and let v be a normalised eigenvector of $M^{-1}A$ and λ its corresponding eigenvalue. We want to show that $|1-\lambda| \leq c$. Firstly,

$$\begin{aligned}(I - M^{-1}A)v &= Iv - \lambda v \\ &= v - \lambda v \\ &= (1-\lambda)v.\end{aligned}$$

Now, we have

$$\begin{aligned}(1-\lambda)v &= (I - M^{-1}A)v \\ \implies \|(1-\lambda)v\| &= \left\|(I - M^{-1}A)v\right\| \\ \implies |1-\lambda|.\|v\| &\leq \left\|I - M^{-1}A\right\|.\|v\| \\ \implies |1-\lambda| &\leq \left\|I - M^{-1}A\right\| \\ \implies |1-\lambda| &\leq c\end{aligned}$$

We know $M^{-1}A$ is diagonalisable so it has full rank; in addition the number of its eigenvectors is equal to this rank. As we chose an arbitrary eigenvector and its corresponding eigenvalue we know this inequality holds for all eigenvalues of $M^{-1}A$.

c

From the lecture notes [3], we know that the residual of GMRES becomes $r_n = p(M^{-1}A)b$ for some polynomial $p(z)$, with several conditions that must be satisfied,

1. $p(z) = 1 - zp'(z)$
2. $Deg(p(z)) \leq n$
3. $p(0) = 1$.

In addition to this, we want all of our eigenvalues of $M^{-1}A$ to be clustered in small groups and the polynomial to be zero near these groups. As we have chosen M, such that $M^{-1}A \approx I$, we expect the eigenvalues to be close to 1. So, if $p(z)$ also satisfies $p(1) = 0$ these facts will also hold.

Setting our polynomial to be $p(z) = (1-z)^n$ we trivially see that all of these conditions are met and so, $p(z)$ is a good upper bound for the optimal polynomial in the polynomial formulation

of GMRES used to derive error estimates.

From this, we can derive an upper bound for the convergence rate of preconditioned GMRES. Firstly, let's recall that as $M^{-1}A$ is diagonalisable, it can be written as $M^{-1}A = V\Lambda V^{-1}$, where V a matrix of the eigenvectors of $M^{-1}A$ and Λ is a diagonal matrix of the eigenvalues of $M^{-1}A$.

Now, we know from the lecture notes that our upper bound for the rate of convergence is governed by,

$$\frac{\|r_n\|}{\|b\|} \leq \kappa(V) \sup_{z \in \Lambda(M^{-1}A)} \|p(z)\|.$$

Recall that $\kappa(V)$ is the condition number, a constant. This means it will not affect the convergence rate - just the magnitude of error at each iteration. As such we can ignore it. Using the polynomial upper bound we just derived and $p(z) = (1-z)^n$ we have,

$$\begin{aligned} \frac{\|r_n\|}{\|b\|} &\leq \sup_{z \in \Lambda(M^{-1}A)} \|(1-z)^n\| \\ \implies &\leq |1-\lambda|^n \\ \implies &\leq c^n \end{aligned}$$

d

For clarity, the c referenced in part (b) remains as c and the one referenced in part (d), I relabelled to μ.

For this question we wanted to investigate convergence for the matrix $A = I + L$. To do this I constructed a function in q4.py called ' laplacetest '. Firstly, I'll discuss the case when conv=False; in this situation the function produces plots of the residual norm at each stage of the iteration as well as our upper bound, c^n. Viewing Figures: 12, 13, 14 we see a clear trend within all of them. The preconditioned GMRES is converging consistently, in fewer iterations and with a smaller residual throughout than the unconditioned GMRES. This is to be expected as we have optimised our algorithm with the intention of making it more efficient. Similarly, we see in all cases that beyond iteration 5 our residuals norms dip below c^n demonstrating that it acts as a good upper bound.

The second part of the function, when we set conv=True, returns a different set of results as can be seen in Figure: 15. This function iterates through 10 matrices of different sizes and each time calculates the order of convergence based on the following formula [4]

$$q \approx \frac{log\left(\frac{|x_{n+1}-x_n|}{|x_n-x_{n-1}|}\right)}{log\left(\frac{|x_n-x_{n-1}|}{|x_{n-1}-x_{n-2}|}\right)}.$$

I set x_{n+1} to be the final residual norm value as convention in each. Viewing the plot we see consistently higher convergence order for the preconditioned version as would expect. In particular, this is emphasised for the smaller matrices; we see that the orders become closer together and even cross over for the final value. I expect this is a result of poorly conditioned large matrices which have unusual residuals.

5

a

To give useful context for the rest of the question I will include the form of the equation system. We are solving

$$AU = \left(\begin{bmatrix} I & 0 & 0 & \dots & 0 & 0 \\ -I & I & 0 & \dots & 0 & 0 \\ 0 & -I & I & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & -I & I \end{bmatrix} + \frac{1}{2} \begin{bmatrix} B & 0 & 0 & \dots & 0 & 0 \\ B & B & 0 & \dots & 0 & 0 \\ 0 & B & B & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & B & B \end{bmatrix} \right) U = \begin{bmatrix} r \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

We begin by adding equations (10) and (11) together, to give

$$w_i^{n+1}\left[1-\left(\frac{\Delta t}{2}\right)\right] + w_i^n\left[-1-\left(\frac{\Delta t}{2}\right)\right]$$
$$+ u_i^{n+1} - \frac{\Delta t}{2(\Delta x)^2}(u_{i-1}^{n+1} - 2u_i^{n+1} + u_{i+1}^{n+1}) + u_i^n - \frac{\Delta t}{2(\Delta x)^2}(u_{i-1}^n - 2u_i^n + u_{i+1}^n) = 0.$$

Now, we consider the action of the block row of A - applied to U - which is only non-zero at the n^{th} and $(n+1)^{\text{th}}$ $(n>1)$ time slice,

$$\left[(-I_{2M}\ \ I_{2M}) + \frac{1}{2}(B\ \ B)\right]\begin{bmatrix} p_n \\ q_n \\ p_{n+1} \\ q_{n+1}\end{bmatrix} = 0$$

$$\implies \left[-I_{2M} + \frac{B}{2}\right]\begin{bmatrix} p_n \\ q_n\end{bmatrix} + \left[I_{2M} + \frac{B}{2}\right]\begin{bmatrix} p_{n+1} \\ q_{n+1}\end{bmatrix} = 0.$$

Then breaking each matrix into its block components we have

$$\left(\begin{bmatrix} -I_M & 0 \\ 0 & -I_M\end{bmatrix} + \frac{1}{2}\begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22}\end{bmatrix}\right)\begin{bmatrix} p_n \\ q_n\end{bmatrix} + \left(\begin{bmatrix} I_M & 0 \\ 0 & I_M\end{bmatrix} + \frac{1}{2}\begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22}\end{bmatrix}\right)\begin{bmatrix} p_{n+1} \\ q_{n+1}\end{bmatrix} = 0$$

Now, lets look at the terms involving q_n and q_{n+1} and compare to the relevant terms in (10)+(11).

$$\left[I_M + \frac{1}{2}(B_{22}+B_{12})\right]q_{n+1} + \left[-I_M + \frac{1}{2}(B_{22}+B_{12})\right]q_n = \left(1-\frac{\Delta t}{2}\right)\begin{bmatrix} w_1^{n+1} \\ \vdots \\ w_M^{n+1}\end{bmatrix} + \left(-1-\frac{\Delta t}{2}\right)\begin{bmatrix} w_1^{n} \\ \vdots \\ w_M^{n}\end{bmatrix}.$$

From this, we can see clearly that $B_{22}+B_{12} = -\Delta t I_M$.

Similarly, we can compare the terms involving p_{n+1}, p_n to the relevant terms in $(10)+(11)$

$$u_i^{n+1} - \frac{\Delta t}{2(\Delta x)^2}(u_{i-1}^{n+1} - 2u_i^{n+1} + u_{i+1}^{n+1}) + u_i^n - \frac{\Delta t}{2(\Delta x)^2}(u_{i-1}^n - 2u_i^n + u_{i+1}^n) = 0$$
$$\implies \left[I_M + \frac{1}{2}(B_{11}+B_{21})\right]p_{n+1} + \left[-I_M + \frac{1}{2}(B_{11}+B_{21})\right]p_n = 0.$$

From this we see that

$$B_{11}+B_{21} = \frac{\Delta t}{(\Delta x)^2}\begin{bmatrix} 2 & -1 & 0 & \dots & 0 & -1 \\ -1 & 2 & -1 & \dots & 0 & 0 \\ 0 & -1 & 2 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 2 & -1 \\ -1 & 0 & 0 & \dots & -1 & 2\end{bmatrix} \in \mathbb{R}^{M\text{x}M}.$$

so we set

$$B_{21} = 0,\ \ B_{12} = 0,\ \ , B_{22} = -\Delta t I_M,\ \ B_{11} = \frac{\Delta t}{(\Delta x)^2} \begin{bmatrix} 2 & -1 & 0 & \dots & 0 & -1 \\ -1 & 2 & -1 & \dots & 0 & 0 \\ 0 & -1 & 2 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 2 & -1 \\ -1 & 0 & 0 & \dots & -1 & 2 \end{bmatrix} \in \mathbb{R}^{M \text{x} M}$$

With $B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$.

We also, want to construct r explicitly. To do so, we consider the action of the first block row of A applied to U. The non-zero elements of this are

$$\left(I_{2M} + \frac{1}{2}B\right) \begin{bmatrix} p_1 \\ q_1 \end{bmatrix} = r$$

$$\implies \left(\begin{bmatrix} I_M & 0 \\ 0 & I_M \end{bmatrix} + \frac{1}{2} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} \right) \begin{bmatrix} p_1 \\ q_1 \end{bmatrix} = r$$

$$\implies \begin{bmatrix} p_1 + \frac{1}{2}(B_{11}p_1 + B_{12}q_1) \\ q_1 + \frac{1}{2}(B_{21}p_1 + B_{22}q_1) \end{bmatrix} = r$$

b

We want to show that if U^k converges, it converges to our solution U. Let's assume $U^k \to U^*$. Looking at the first block row of (17) we see that we have

$$\left(I_{2M} + \frac{B}{2}\right)(p_1^{k+1}\ \ q_1^{k+1})^T + \alpha\left(-I_{2M} + \frac{B}{2}\right)(p_N^{k+1}\ \ q_N^{k+1})^T = r + \alpha\left(-I_{2M} + \frac{B}{2}\right)(p_N^{k}\ \ q_N^{k})^T$$

But, as we have convergence we know that at some point both sides will converge to U^*, so this equation becomes

$$\left(I_{2M} + \frac{B}{2}\right)(p_1^{*}\ \ q_1^{*})^T + \alpha\left(-I_{2M} + \frac{B}{2}\right)(p_N^{*}\ \ q_N^{*})^T = r + \alpha\left(-I_{2M} + \frac{B}{2}\right)(p_N^{*}\ \ q_N^{*})^T$$

$$\implies \left(I_{2M} + \frac{B}{2}\right)(p_1^{*}\ \ q_1^{*})^T = r.$$

This is exactly the same as the the first block row of A applied to U as we see in the first question. So, if this also holds for all other rows, we know that $U^* = U$. Looking at the application of the

block row of $\left((C_1^{(\alpha)} \otimes I) + (C_2^{(\alpha)} \otimes B)\right)$ on U^* which is non-zero only at the n^{th} and $(n+1)^{\text{th}}$ $(n > 1)$ time slice, we have

$$\left(-I + \frac{B}{2}\right)(p_n^* \;\; q_n^*)^T + \left(I + \frac{B}{2}\right)(p_{n+1}^* \;\; q_{n+1}^*)^T = 0$$

Once again, this mirrors the expression we derived in the previous part. As such, we know that if U^k converges, it converges to our desired solution U.

c

For a matrix A and vector v, we say v is an eigenvector of A if there exists a constant λ such that $Av = \lambda v$. We claim that

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

for $j = 0, 1, 2, ..., N-1$ are the linearly independent eigenvectors of $C_1^{(\alpha)}$ and $C_2^{(\alpha)}$. To prove this, we must simply show that the definition holds, i.e. find constants $\lambda_1^{(j)}$ and $\lambda_2^{(j)}$ such that $C_i^{(\alpha)} v_j = \lambda_i^{(j)}$. Let's consider the case for $C_1^{(\alpha)}$ first.

$$C_1^{(\alpha)} v_j = \begin{bmatrix} 1 - \alpha^{\frac{1}{N}} e^{\frac{2(N-1)\pi ij}{N}} \\ -1 + \alpha^{\frac{-1}{N}} e^{\frac{2\pi ij}{N}} \\ -\alpha^{\frac{-1}{N}} e^{\frac{2\pi ij}{N}} + \alpha^{\frac{-2}{N}} e^{\frac{4\pi ij}{N}} \\ \vdots \\ -\alpha^{\frac{-(N-2)}{N}} e^{\frac{2(N-2)\pi ij}{N}} + \alpha^{\frac{-(N-1)}{N}} e^{\frac{2(N-1)\pi ij}{N}} \end{bmatrix}$$

From the first element, we suspect that $\lambda_1^{(j)} = 1 - \alpha^{\frac{1}{N}} e^{\frac{2(N-1)\pi ij}{N}}$. Let's double check for the other elements of v_j which take the form $\alpha^{\frac{-m}{N}} e^{\frac{2m\pi ij}{N}}$ for $m = 1, 2, ..., N-1$.

$$\begin{aligned} \lambda_1^{(j)} v_j^{(m)} &= \alpha^{\frac{-m}{N}} e^{\frac{2m\pi ij}{N}} \left(1 - \alpha^{\frac{1}{N}} e^{\frac{2(N-1)\pi ij}{N}}\right) \\ &= \alpha^{\frac{-m}{N}} e^{\frac{2m\pi ij}{N}} - \alpha^{\frac{1-m}{N}} e^{\frac{2\pi ij}{N}(m+N-1)} \\ &= -\alpha^{\frac{-(m-1)}{N}} e^{\frac{2\pi ij(m-1)}{N}} e^{2\pi ij} + \alpha^{\frac{-m}{N}} e^{\frac{2m\pi ij}{N}} \\ &= -\alpha^{\frac{-(m-1)}{N}} e^{\frac{2\pi ij(m-1)}{N}} + \alpha^{\frac{-m}{N}} e^{\frac{2m\pi ij}{N}} \end{aligned}$$

This is clearly the same form as above. So we know the vector v_j is an eigenvector for $C_1^{(\alpha)}$ with eigenvalue $\lambda_1^{(j)} = 1 - \alpha^{\frac{1}{N}} e^{\frac{2(N-1)\pi ij}{N}}$. Let's repeat the process for $C_2^{(\alpha)}$.

$$C_2^{(\alpha)} v_j = \begin{bmatrix} \frac{1}{2}\left(1 + \alpha^{\frac{1}{N}} e^{\frac{2(N-1)\pi ij}{N}}\right) \\ \frac{1}{2} + \frac{1}{2}\left(\alpha^{\frac{-1}{N}} e^{2\pi ij}\right) \\ \frac{1}{2}\left(\alpha^{\frac{-1}{N}} e^{2\pi ij}\right) + \frac{1}{2}\left(\alpha^{\frac{-2}{N}} e^{4\pi ij}\right) \\ \vdots \\ \frac{1}{2}\left(\alpha^{\frac{-(N-2)}{N}} e^{2(N-2)\pi ij}\right) + \frac{1}{2}\left(\alpha^{\frac{-(N-1)}{N}} e^{2(N-1)\pi ij}\right) \end{bmatrix}$$

Similar to before, we suspect $\lambda_2^{(j)} = \frac{1}{2}\left(1 + \alpha^{\frac{1}{N}} e^{\frac{2(N-1)\pi ij}{N}}\right)$. Let's confirm this for the other rows of v_j which take the form $\alpha^{\frac{-m}{N}} e^{\frac{2m\pi ij}{N}}$ for $m = 1, 2, ..., N-1$.

$$\begin{aligned} \lambda_2^{(j)} v_j^{(m)} &= \frac{1}{2}\alpha^{\frac{-m}{N}} e^{\frac{2m\pi ij}{N}} \left(1 + \alpha^{\frac{1}{N}} e^{\frac{2(N-1)\pi ij}{N}}\right) \\ &= \frac{1}{2}\left(\alpha^{\frac{-m}{N}} e^{\frac{2m\pi ij}{N}} + \alpha^{\frac{-(m-1)}{N}} e^{\frac{2\pi ij(m+N-1)}{N}}\right) \\ &= \frac{1}{2}\left(\alpha^{\frac{-m}{N}} e^{\frac{2m\pi ij}{N}} + \alpha^{\frac{-(m-1)}{N}} e^{\frac{2\pi ij(m-1)}{N}} e^{2\pi ij}\right) \\ &= \frac{1}{2}\left(\alpha^{\frac{-m}{N}} e^{\frac{2m\pi ij}{N}} + \alpha^{\frac{-(m-1)}{N}} e^{\frac{2\pi ij(m-1)}{N}}\right) \end{aligned}$$

Which is once again the same form as above. So, we know v_j is an eigenvector of $C_2^{(\alpha)}$ with eigenvalue $\lambda_2^{(j)} = \frac{1}{2}\left(1 + \alpha^{\frac{1}{N}} e^{\frac{2(N-1)\pi ij}{N}}\right)$.

As both $C_1^{(\alpha)}$ and $C_2^{(\alpha)}$ have N rows and columns, as well as N linearly independent eigenvectors we know from standard linear algebra that it is diagonalisable. In addition to this, it is a standard result that if a matrix, say M, is diagonalisable it can be written as $M = VDV^{-1}$. Where the columns of V are the eigenvectors of M and D is a diagonal matrix with the corresponding eigenvalues along the diagonal. Hence, if we set

$$V = \begin{bmatrix} \uparrow & & \uparrow \\ v_0 & \dots & v_{N-1} \\ \downarrow & & \downarrow \end{bmatrix}, \quad D_i = \begin{bmatrix} \lambda_i^{(0)} & & \\ & \ddots & \\ & & \lambda_i^{(N-1)} \end{bmatrix}$$

we can write $C_1^{(\alpha)} = VD_1V^{-1}$ and $C_2^{(\alpha)} = VD_2V^{-1}$.

Now, using a useful property of the Kronecker Product [5]: $(A \otimes B)(C \otimes D) = (AC) \otimes (BD)$ we

can generate an important result.

$$\begin{aligned}(C_1^{(\alpha)} \otimes I) + (C_2^{(\alpha)} \otimes B) &= (VD_1V^{-1} \otimes I) + (VD_2V^{-1} \otimes B) \\ &= ((VD_1)V^{-1} \otimes II) + ((VD_2)V^{-1} \otimes BI) \\ &= (VD_1 \otimes I)(V^{-1} \otimes I) + (VD_2 \otimes B)(V^{-1} \otimes I) \\ &= \{(VD_1 \otimes II) + (VD_2 \otimes IB)\}(V^{-1} \otimes I) \\ &= \{(V \otimes I)(D_1 \otimes I) + (V \otimes I)(D_2 \otimes B)\}(V^{-1} \otimes I) \\ &= (V \otimes I)\{(D_1 \otimes I) + (D_2 \otimes B)\}(V^{-1} \otimes I)\end{aligned}$$

d

Firstly, note that $V_{kj} = \alpha^{\frac{-(k-1)}{N}} e^{\frac{2(k-1)\pi i(j-1)}{N}}$ is the kj^{th} element of V where $k, j = 1, ..., N$.

After viewing the form of $(V \otimes I)U$ and the IDFT we can relate the two. Firstly, we note that $\mathcal{F}^{-1}\left[\{T_k\}_{k=0}^{N-1}\right]_{(n)} = \frac{1}{N}\sum_{k=0}^{N-1} T_k e^{\frac{2\pi ikn}{N}}$ for $n = 0, 1, ..., N-1$. Now, consider,

$$\begin{aligned}(V \otimes I)U &= \begin{bmatrix} V_{11}I_{2m} & \dots & V_{1N}I_{2m} \\ \vdots & \ddots & \vdots \\ V_{N1}I_{2m} & \dots & V_{NN}I_{2m} \end{bmatrix} \begin{bmatrix} T_1 \\ \vdots \\ T_N \end{bmatrix}, \quad T_i = \begin{bmatrix} p_i \\ q_i \end{bmatrix} \\ &= \begin{bmatrix} V_{11}T_1 + ... + V_{1N}T_N \\ \vdots \\ V_{N1}T_1 + ... + V_{NN}T_N \end{bmatrix}.\end{aligned}$$

From the previous question we know the form of V, so can expand this further.

$$(V \otimes I)U = \begin{bmatrix} 1T_1 + 1T_2 + ... + 1T_N \\ \alpha^{\frac{-1}{N}} e^0 T_1 + \alpha^{\frac{-1}{N}} e^{\frac{2\pi i}{N}} T_2 + ... + \alpha^{\frac{-1}{N}} e^{\frac{2(N-1)\pi i}{N}} T_N \\ \alpha^{\frac{-2}{N}} e^0 T_1 + \alpha^{\frac{-2}{N}} e^{\frac{4\pi i}{N}} T_2 + ... + \alpha^{\frac{-2}{N}} e^{\frac{4(N-1)\pi i}{N}} T_N \\ \vdots \\ \alpha^{\frac{-(N-1)}{N}} e^0 T_1 + \alpha^{\frac{-(N-1)}{N}} e^{\frac{2(N-1)\pi i}{N}} T_2 + ... + \alpha^{\frac{-(N-1)}{N}} e^{\frac{2(N-1)^2\pi i}{N}} T_N \end{bmatrix}$$

$$= \begin{bmatrix} 1 & & & & \\ & \alpha^{\frac{-1}{N}} & & & \\ & & \alpha^{\frac{-2}{N}} & & \\ & & & \ddots & \\ & & & & \alpha^{\frac{-(N-1)}{N}} \end{bmatrix} \begin{bmatrix} \sum_{k=0}^{N-1} T_k e^0 \\ \sum_{k=0}^{N-1} T_K e^{\frac{2k\pi i}{N}} \\ \vdots \\ \sum_{k=0}^{N-1} T_k e^{\frac{2(N-1)k\pi i}{N}} \end{bmatrix}$$

$$= \begin{bmatrix} N & & & & \\ & N\alpha^{\frac{-1}{N}} & & & \\ & & N\alpha^{\frac{-2}{N}} & & \\ & & & \ddots & \\ & & & & N\alpha^{\frac{-(N-1)}{N}} \end{bmatrix} \begin{bmatrix} \mathcal{F}^{-1}\left[\{T_k\}_{k=0}^{N-1}\right]_{(0)} \\ \mathcal{F}^{-1}\left[\{T_k\}_{k=0}^{N-1}\right]_{(1)} \\ \vdots \\ \mathcal{F}^{-1}\left[\{T_k\}_{k=0}^{N-1}\right]_{(N-1)} \end{bmatrix}$$

Thus, we see that applying $(V \otimes I)$ to U is the same as applying a diagonal matrix (let's call this D^{-1}) to the IDFT matrix as shown above (let's call this F^{-1}). So, $(V \otimes I)U = D^{-1}F^{-1}$.

For the next part we want to write $(V^{-1} \otimes I)U$ in a similar format involving DFT and multiplication by a diagonal matrix. Firstly, let's recall that $\mathcal{F}\left[\{T_k\}_{k=0}^{N-1}\right]_{(n)} = \sum_{k=0}^{N-1} T_k e^{\frac{-2\pi i k n}{N}}$. Now, using a useful result that $(A \otimes B)^{-1} = (A^{-1} \otimes B^{-1})$ [5] we see that $(V^{-1} \otimes I) = (V \otimes I)^{-1}$. As such, we want to apply the inverse procedure to the previous case: we must firstly multiply U by D and the take the DFT in a similar way to before (denote this F). We know that $\mathcal{F}\{\mu X\} = \mu\mathcal{F}\{X\}$ trivially, so the procedure outlined above is equivalent to taking the DFT of

U then multiplying by the diagonal matrix D, i.e. DW. This can be expressed as follows

$$(V^{-1}\otimes I)U = \begin{bmatrix} \frac{1}{N} & & & & \\ & \frac{1}{N}\alpha^{\frac{1}{N}} & & & \\ & & \frac{1}{N}\alpha^{\frac{2}{N}} & & \\ & & & \ddots & \\ & & & & \frac{1}{N}\alpha^{\frac{N-1}{N}} \end{bmatrix} \begin{bmatrix} \mathcal{F}\left[\{T_k\}_{k=0}^{N-1}\right]_{(0)} \\ \mathcal{F}\left[\{T_k\}_{k=0}^{N-1}\right]_{(1)} \\ \vdots \\ \mathcal{F}\left[\{T_k\}_{k=0}^{N-1}\right]_{(N-1)} \end{bmatrix}.$$

Note: In both diagonal matrices for $(V^{-1}\otimes I)U$ and $(V\otimes I)U$ each non-zero element represents a $2m$-dimensional diagonal matrix consisting of that element. I use the more brief notation for simplicity.

e

Let's recall some useful facts before demonstrating how to solve (17)

$$(V\otimes I)^{-1} = (V^{-1}\otimes I)$$
$$U^k \to U \text{ by the iterative method in (17).}$$

Now, let's work through the steps needed to solve (17), noting that we will demonstrate this for a vector U but this applies to all iterations $U^1, ..., U^k, ..., U$.

$$\left[(C_1^{(\alpha)}\otimes I) + (C_2^{(\alpha)}\otimes B)\right]U = R$$
$$\left[(V\otimes I)\left[(D_1\otimes I) + (D_2\otimes B)\right](V^{-1}\otimes I)\right]U = R$$

Multiply both sides by $(V\otimes I)^{-1}$.

$$\left[(V\otimes I)^{-1}(V\otimes I)\right]\left[(D_1\otimes I) + (D_2\otimes B)\right](V^{-1}\otimes I)U = (V\otimes I)^{-1}R$$
$$\left[(D_1\otimes I) + (D_2\otimes B)\right](V^{-1}\otimes I)U = (V^{-1}\otimes I)R.$$

Now set $\hat{R} = (V^{-1}\otimes I)R$ and $\hat{U} = (V^{-1}\otimes I)U$ where,

$$\hat{R} = \begin{bmatrix} \hat{r}_1 \\ \vdots \\ \hat{r}_N \end{bmatrix}, \quad \hat{U} = \begin{bmatrix} \hat{p}_1 \\ \hat{q}_1 \\ \vdots \\ \hat{p}_N \\ \hat{q}_N \end{bmatrix}$$

Recalling that D_i, $i = 1, 2$ are diagonal matrices with elements $d_{ik} = \lambda_i^{(k-1)}$, $k = 1, 2, ..., N$ along the diagonal we have,

$$[(D_1 \otimes I) + (D_2 \otimes B)]\hat{U} = \hat{R}$$

$$\implies \left(\begin{bmatrix} d_{11}I_{2m} & & & \\ & d_{12}I_{2m} & & \\ & & \ddots & \\ & & & d_{1N}I_{2m} \end{bmatrix} + \begin{bmatrix} d_{21}B & & & \\ & d_{22}B & & \\ & & \ddots & \\ & & & d_{2N}B \end{bmatrix} \right) \hat{U} = \hat{R}$$

$$\implies \begin{bmatrix} d_{11}I_{2m} + d_{21}B & & & \\ & d_{12}I_{2m} + d_{22}B & & \\ & & \ddots & \\ & & & d_{1N}I_{2m} + d_{2N}B \end{bmatrix} \hat{U} = \hat{R}$$

Looking at the action of the k^{th} block matrix row applied to the k^{th} time slice of $\hat{U}$, we see that solving the system for $\hat{U}$ is equivalent to solving $(d_{1k}I_{2m} + d_{2k}B)(\hat{p}_k \;\; \hat{q}_k)^T = \hat{r}_k$ for $k = 1, 2, ..., N$.

Hence, we can compute U as follows

$$\begin{aligned} \hat{U} &= (V^{-1} \otimes I)U \\ \implies U &= (V^{-1} \otimes I)^{-1}\hat{U} \\ \implies U &= (V \otimes I)\hat{U}. \end{aligned}$$

Thus, we have shown how to solve each stage of the iterative process defined in (17). The steps detailed above can be written concisely as:

1. Compute $\hat{R} = (V^{-1} \otimes I)R$
2. Solve $(d_{1k}I_{2m} + d_{2k}B)(\hat{p}_k \;\; \hat{q}_k)^T = \hat{r}_k$, where $\hat{p}_k, \hat{q}_k$ is the k^{th} time slice of $\hat{U}$, for $k = 1, 2, ..., N$.
3. Compute $U = (V \otimes I)\hat{U}$.

f

Regretfully, due to time constraints as a result of illness and workload I was not able to implement the algorithm.

6 Gallery

Figure 1: Timings for Q2

```
Time for q2solve=0.0002286434173583984, Time for solve_LUP=0.0011081695556640625
Time for q2solve=0.0007369518280029297, Time for solve_LUP=0.0021660327911376953
```

Figure 2: Heatmap of T

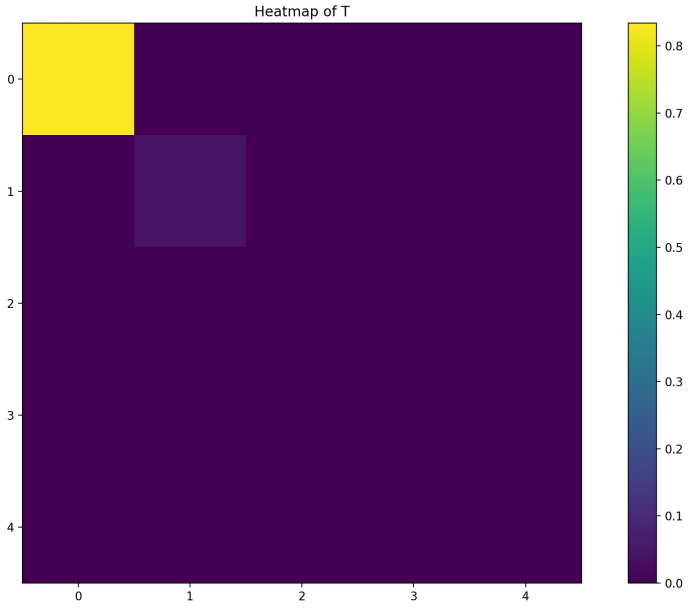


Figure 3: Error of Q3d Eigenvalue calculation

Error of eigenvalues calculation
-4.767519712345347e-11

Figure 4: 3e Plots of A_{ij}

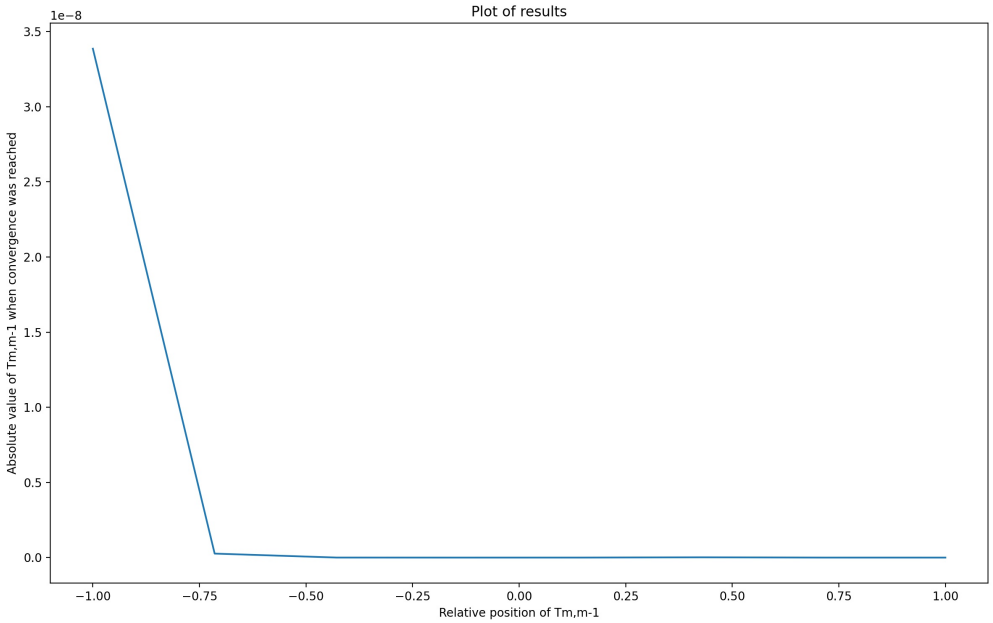


Figure 5: Q3e Plots of a Random SPD Matrix, $m = 16$

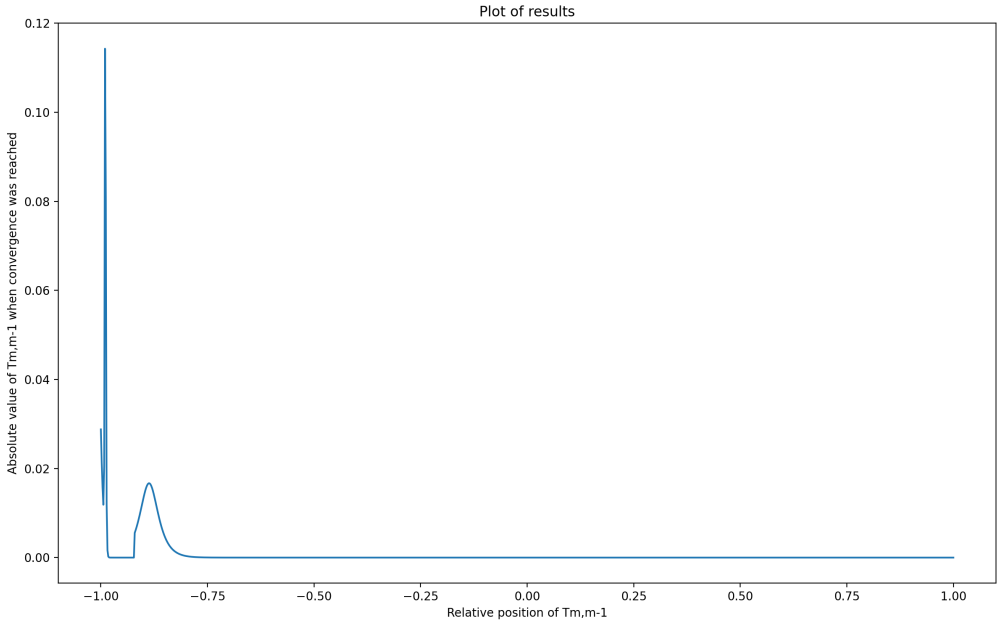


Figure 6: Timing plot for Q3e

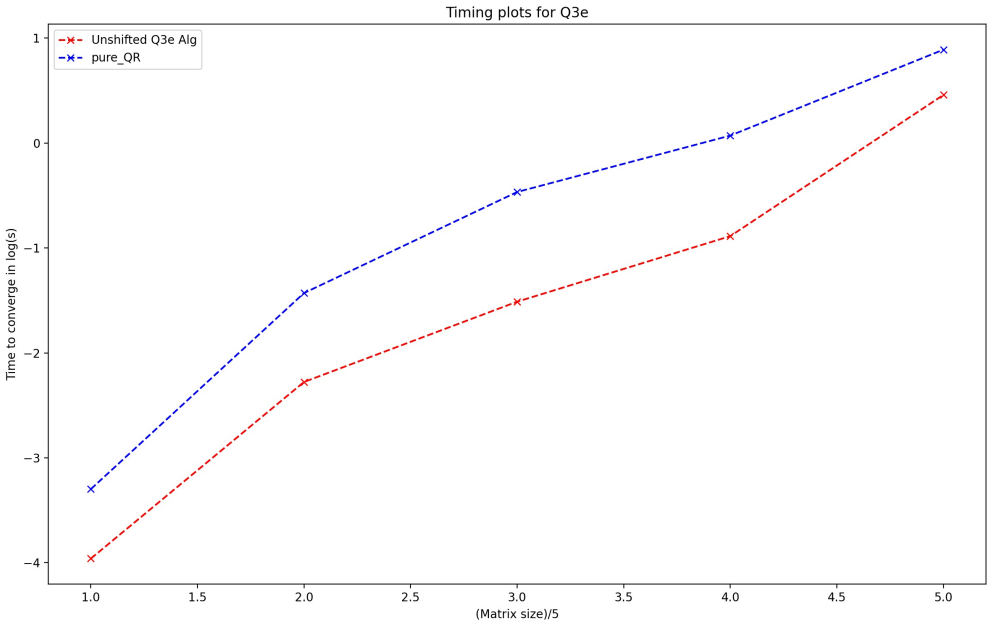


Figure 7: qr alg with Wilkinson Shift Applied to A_{ij}

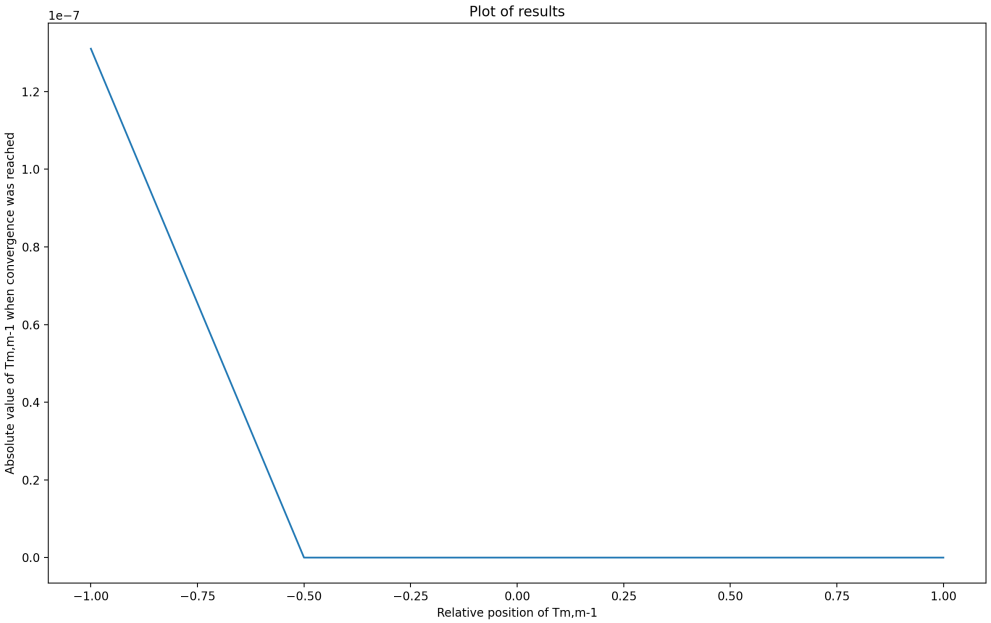


Figure 8: qr alg with Wilkinson Shift Applied to a Random SPD Matrix

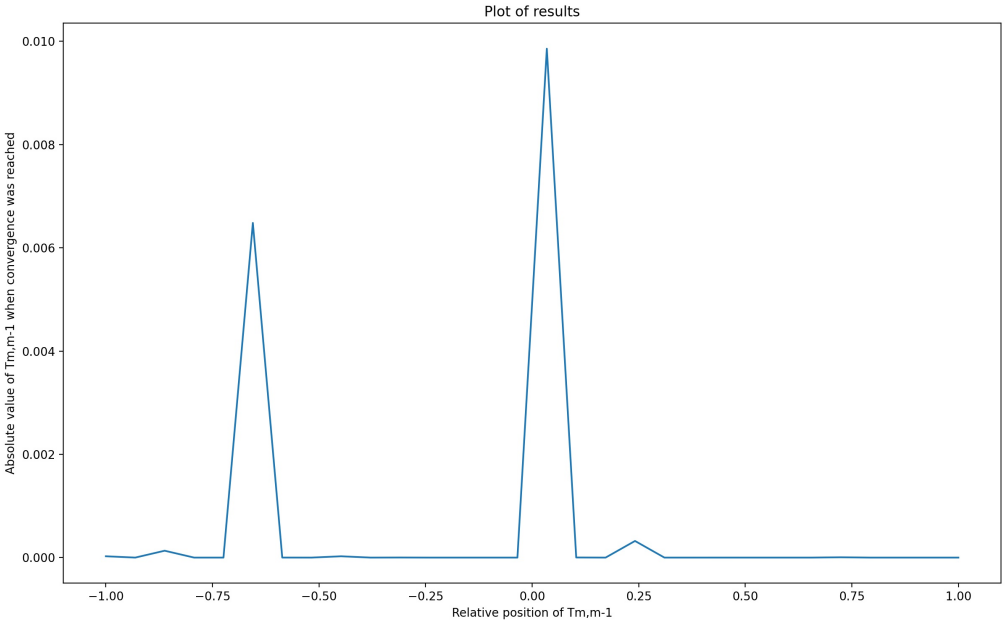


Figure 9: Timing Comparisons for Shifted and Unshifted qr alg

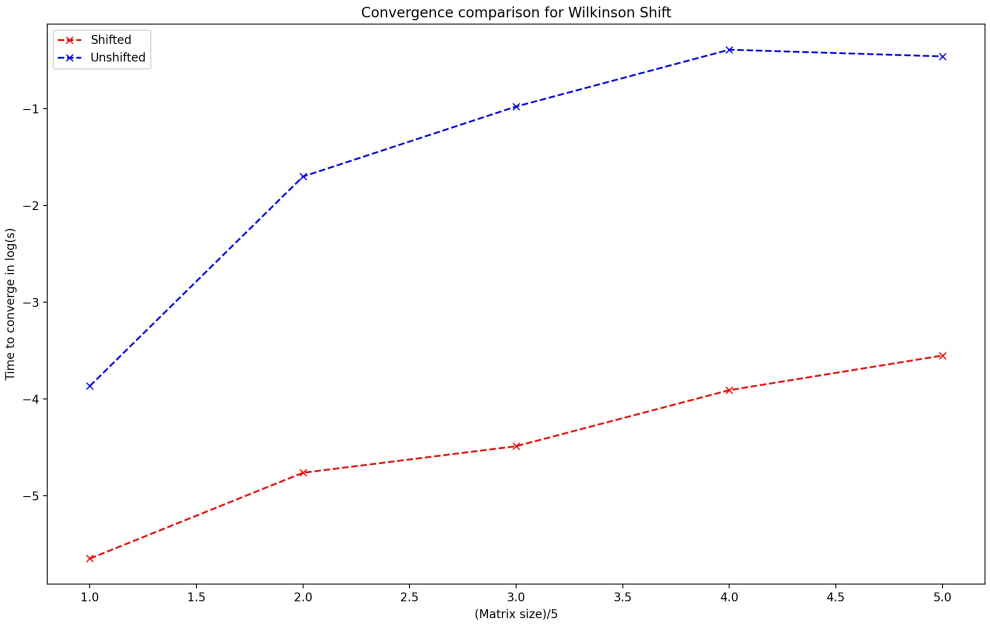


Figure 10: qr alg Applied to Matrix A from Q3g

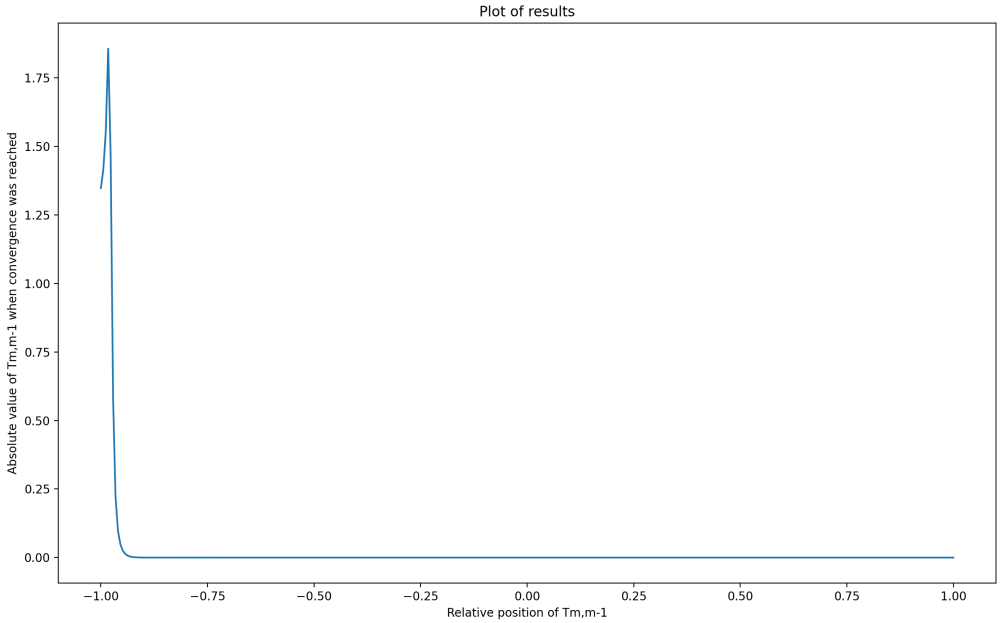


Figure 11: qr alg with Wilkinson Shift Applied to Matrix A from Q3g

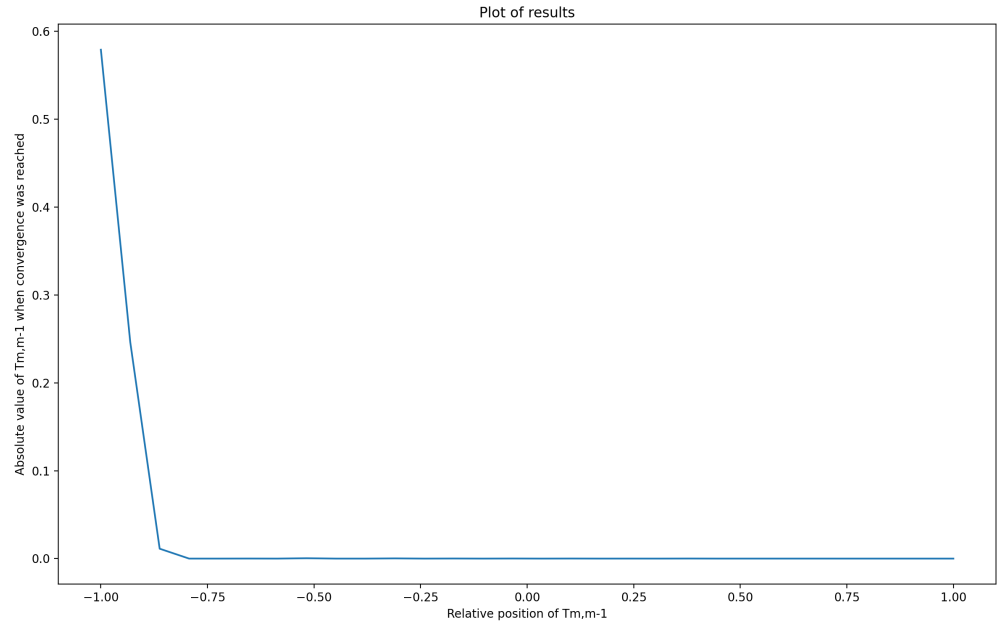


Figure 12: Plots of Residual Norms and Upper Bound for m=10

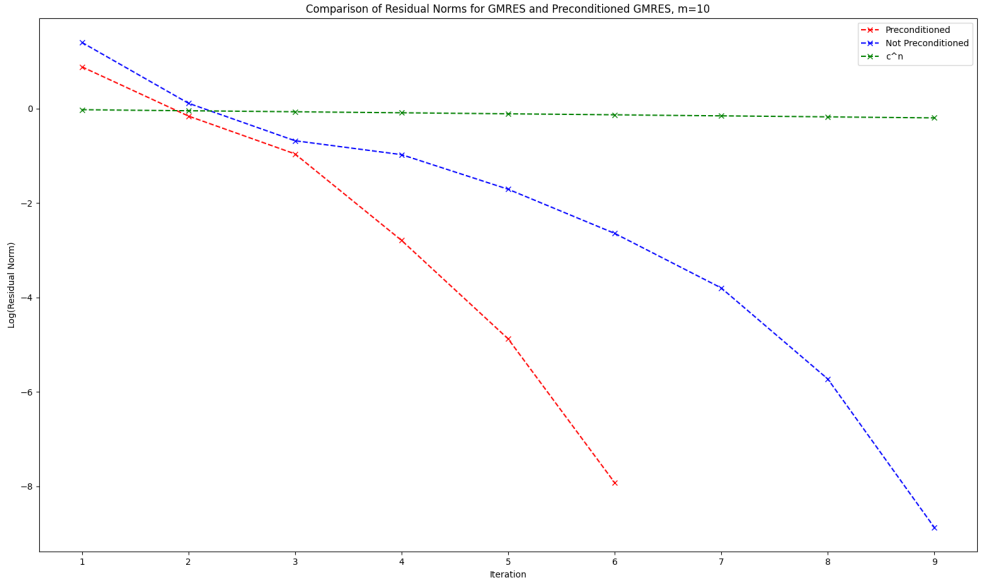


Figure 13: Plots of Residual Norms and Upper Bound for m=35

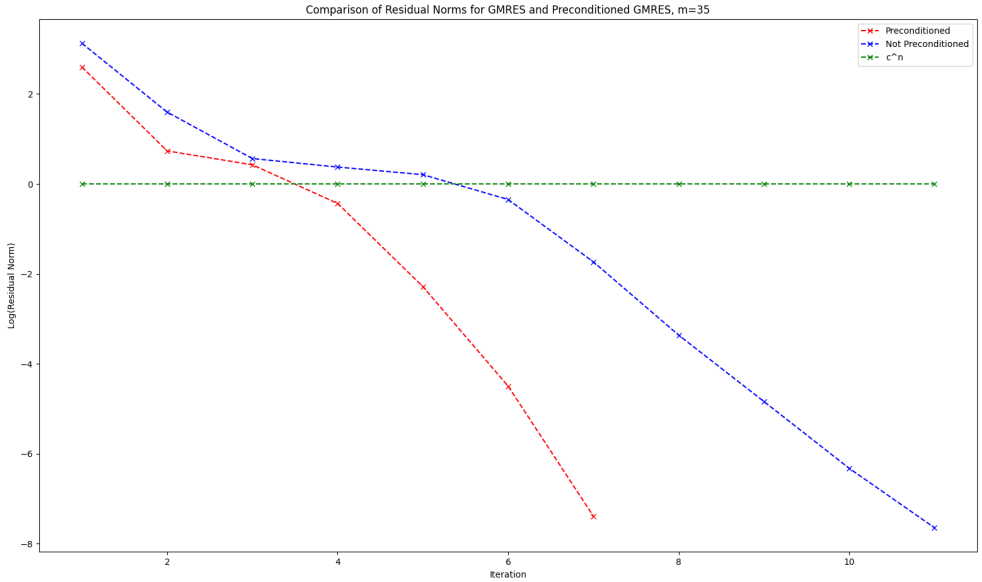


Figure 14: Plots of Residual Norms and Upper Bound for m=100

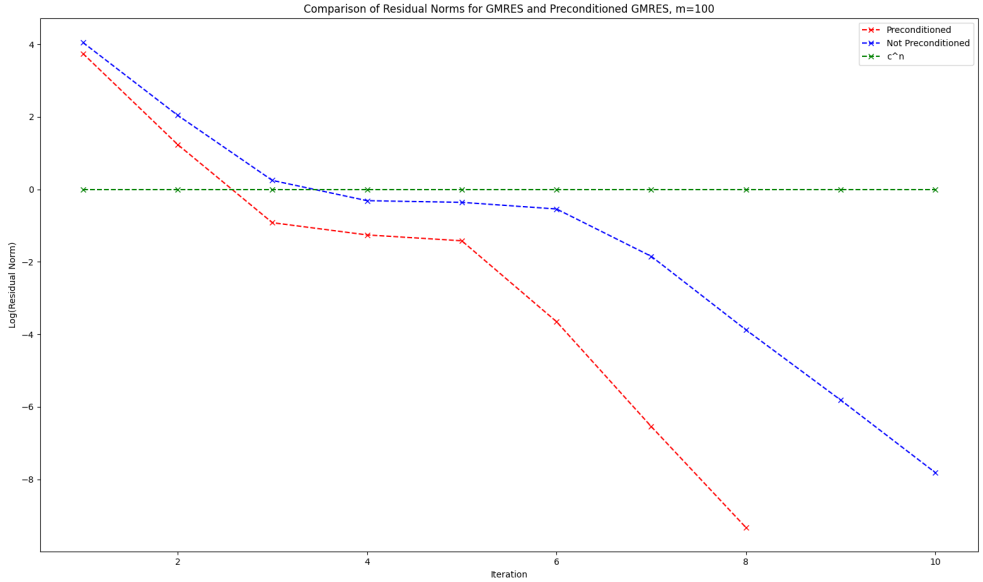
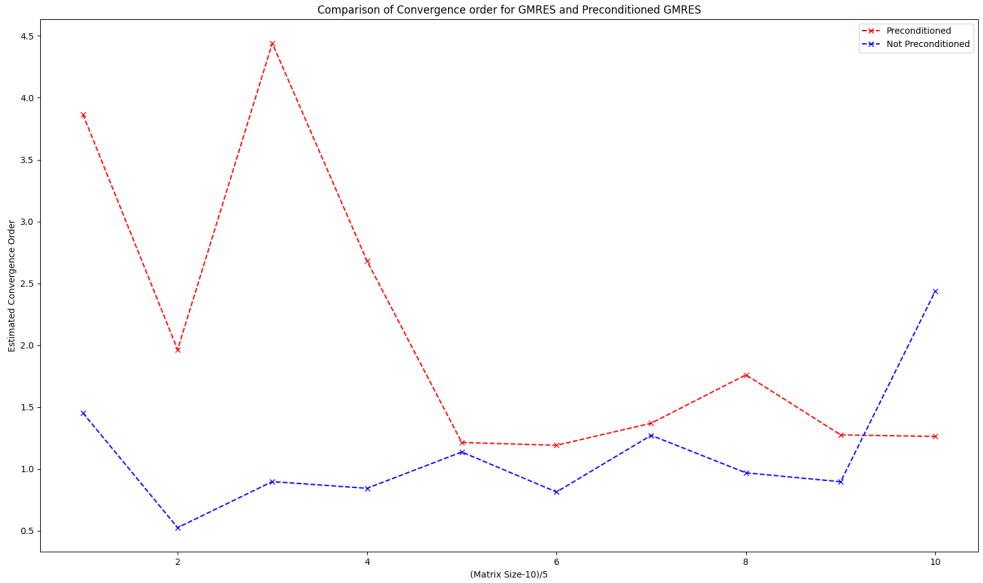


Figure 15: Plot of Estimated Convergence Orders for GMRES on Multiple Matrices



References

[1] Rao SS. FINITE DIFFERENCE METHODS. In: Encyclopedia of Vibration. Elsevier; 2001. p. 520–530. Available from: https://doi.org/10.1006/rwvb.2001.0002.

[2] Hager WW. Updating the Inverse of a Matrix. SIAM Review. 1989;31(2):221–239. Available from: https://epubs.siam.org/doi/abs/10.1137/1031049.

[3] Cotter C. Computational Linear Algebra Lecture Notes;. Available from http://comp-lin-alg.github.io/ (2020).

[4] SENNING JR. Computing and Estimating the Rate of Convergence;. Available from: http://www.math-cs.gordon.edu/courses/ma342/handouts/rate.pdf.

[5] Horn RA, Johnson CR. Topics in Matrix Analysis. Cambridge: Cambridge University Press; 1991.