Computational Linear Algebra 2020/2021 **(Due: 15/01/2021)**

Coursework #2

Name: George Hutchings *CID:* 01357062

This is my own unaided work unless stated otherwise.
Classwork SHA: 6ce5e3677e70c9aa5600f1f46a12de0ba417ceb0
Coursework SHA: 17138e0eb4e0489c2b7604a5372b2c896c865a8a

Question 1 [1]: (20 %)

(a)

First consider the algorithm for Gaussian elimination (without pivoting) of a banded matrix, this is given to us in the lecture notes (Algorithm 1). Where the matrix A has upper bandwidth p and lower bandwidth q. This algorithm has been obtained from the standard LU decomposition algorithm by considering the properties of a banded A. That is, $A_{ij} = 0$ for all $i > j + q$ and $i < j - p$. Therefore all the values of A below the lower-bandwidth-diagonal are 0, and hence the corresponding values for L_k are 0 . Similarly, all the values of A above the upper-bandwidth-diagonal are 0, and hence the corresponding values for U_k are 0. Therefore, when computing L_k, U_k these values need not be updated, allowing for optimisation of the standard LU decomposition algorithm to Algorithm 1.

$U \leftarrow A$
$L \leftarrow I$
for $k = 1$ *to* $m - 1$ **do**
  for $j = k + 1$ *to* $\min(k + p,\ m)$ **do**
    $l_{j,k} \leftarrow \frac{u_{j,k}}{u_{k,k}}$
    $n \leftarrow \min(k + q,\ m)$
    $u_{j,k:n} \leftarrow u_{j,k:n} - l_{j,k} u_{k,k:n}$
  end
end

Algorithm 1: Gaussian elimination for a banded matrix (from notes)

I will apply Algorithm 1 to the matrix given in question 1, hence $p = q = 1$. This implies that $n = j = k+1$, and that we do not loop through j, since $\min(k + p,\ m) = \min(k + 1,\ m) = k + 1$ hence $j = k + 1$, i.e. it takes a single value, therefore we can simplify Algorithm 1 to Algorithm 2.

$U \leftarrow A$
$L \leftarrow I$
for $k = 1$ *to* $m - 1$ **do**
  $l_{k+1,k} \leftarrow \frac{u_{k+1,k}}{u_{k,k}}$
  $u_{k+1,k:k+1} \leftarrow u_{k+1,k:k+1} - l_{k+1,k} u_{k,k:k+1}$
end

Algorithm 2: Gaussian elimination for matrix A

By noticing the form of A we can simplify Algorithm 2 further:

- Consider $l_{k+1,k} \leftarrow \frac{u_{k+1,k}}{u_{k,k}}$

$$\frac{u_{k+1,k}}{u_{k,k}} = \frac{d}{u_{k,k}}$$

- Consider $u_{k+1,k:k+1} \leftarrow u_{k+1,k:k+1} - l_{k+1,k}u_{k,k:k+1}$

$$\begin{aligned} u_{k+1,k:k+1} - l_{k+1,k}u_{k,k:k+1} &= [u_{k+1,k},\, u_{k+1,k+1}] - l_{k+1,k}\,[u_{k,k},\, u_{k,k+1}] \\ &= [d,\, c] - l_{k+1,k}[u_{k,k},\, d] \\ &= [0,\, c - d \cdot l_{k+1,k}] \end{aligned}$$

Where c, d are as in the question. This gives what I believe to be the optimal LU factorisation algorithm (Algorithm 3), for matrices of the form given in the question.

$U \leftarrow A$
$L \leftarrow I$
for $k = 1$ *to* $m - 1$ **do**
  $l_{k+1,k} \leftarrow \frac{d}{u_{k,k}}$
  $u_{k+1,k:k+1} \leftarrow [0,\, c - d \cdot l_{k+1,k}]$
end

Algorithm 3: Optimal Gaussian elimination for matrix A

To solve the equation $Ax = b$ consider $y = Ux$, which reduces our problem to $Ax = Ly = b$, this can be solved in two steps:

1) Forward substitution to solve $Ly = b$ for y.

2) Back substitution to solve $y = Ux$ for x.

1) Considering the forward substitution algorithm for a banded matrix given to us in the lecture notes (Algorithm 4), where L has lower bandwidth p.

$y_1 \leftarrow \frac{b_1}{l_{1,1}}$
for $k = 2$ *to* m **do**
  $j \leftarrow \max(1,\, k - p)$
  $y_k \leftarrow \frac{b_k - l_{k,j:k-1}y_{j:k-1}}{l_{k,k}}$
end

Algorithm 4: Forward substitution for a banded matrix (from notes)

We can notice all diagonal elements of L are 1, and that it has a lower bandwidth $p = 1$, and hence $\max(1,\, k - p) = k - 1$, we therefore reduce Algorithm 4 to Algorithm 5.

$y_1 \leftarrow b_1$
for $k = 1$ *to* $m - 1$ **do**
  $y_{k+1} \leftarrow b_{k+1} - l_{k+1,k}y_k$
end

Algorithm 5: Forward substitution for matrix A

2) We can similarly construct an algorithm for back substitution

1 $x_m \leftarrow \frac{y_m}{u_{m,m}}$
2 **for** $k = m - 1$ *downto* 1 **do**
3 $x_k \leftarrow \frac{y_k - d \cdot x_{k+1}}{u_{k,k}}$
4 **end**

Algorithm 6: Back substitution for matrix A

(b)

We should note that if we seek only the solution to $Ax = b$, that is we do not require the explicit LU decomposition of A, we can merge the LU decomposition step and forward substitution step to form Algorithm 7.

1 $U \leftarrow A$
2 $y_1 \leftarrow b_1$
3 **for** $k = 1$ *to* $m - 1$ **do**
4 $\quad l \leftarrow \frac{d}{u_{k,k}}$
5 $\quad u_{k+1,k+1} \leftarrow c - d \cdot l$
6 $\quad y_{k+1} \leftarrow b_{k+1} - l \cdot y_k$
7 **end**

Algorithm 7: Merged LU decomposition and forward substitution to solve $Ax = b$

This can be done with particular ease since both algorithms require $m-1$ iterations, further the values of L are not required beyond each iteration hence we need only store one entry of L per iteration, lets call this value l.

We can then apply back substitution to solve the system $Ax = b$. Hence solving $Ax = b$ we carry out two steps, that is, Algorithm 7 then Algorithm 6.

(c)

To determine the operation count of the above procedure where b is a vector (Algorithm 7 then Algorithm 6), I will first consider the operation count the individual algorithms.

- Algorithm 7:
  - For loop from $k = 1$ to $m - 1$
    - Line 4 has 1 FLOP.
    - Line 5 has 2 FLOPs.
    - Line 6 has 2 FLOPs.

  This provides a total of $N^1_{\text{FLOPs}} = \sum_{k=1}^{m-1} (1 + 2 + 2) = 5(m - 1)$
- Algorithm 6:
  - Line 1 has 1 FLOP
  - For loop from $k = m - 1$ downto 1
    - Line 3 has 3 FLOPs.

  This provides a total of $N^2_{\text{FLOPs}} = 1 + \sum_{k=1}^{m-1} 3 = 1 + 3(m - 1)$

Hence the overall algorithm has:

$$N_{\text{FLOPs}} = N^1_{\text{FLOPs}} + N^2_{\text{FLOPs}} = 1 + 8(m - 1) = 8m - 7 \text{ FLOPs}$$

Which gives an asymptotic operation count of $\mathcal{O}(m)$.
Comparing this to the general solving of a matrix, that is:

1. LU decomposition
   Operation count $\sim \frac{2m^3}{3}$
2. Forward substitution
   Operation count $\sim \mathcal{O}(m^2)$
3. Back substitution
   Operation count $\sim \mathcal{O}(m^2)$

Hence the overall operation count is: $\sim \frac{2m^3}{3} + \mathcal{O}(m^2) + \mathcal{O}(m^2) \sim \mathcal{O}(m^3)$

We can therefore see that our optimised procedure is considerably more efficient, since it reduces the operation count from $\mathcal{O}(m^3)$ to $\mathcal{O}(m)$.

(d)

The Algorithm to solve $Ax = b$ (Algorithm 7 then Algorithm 6), has been applied in function `LUsolve()` found in `q1.py`. It can be noted that to solve $Ax = b$ the entire matrix of A is not needed, instead only

$$c \coloneqq \text{diagonal entry of } A,\ d \coloneqq \text{sub/super-diagonal entry of } A$$

(the dimensions of A can be found from b). And hence `LUsolve` only requires c, d, b with an optional argument whether or not to do the algorithm in place. Since the entire matrix U is not required, a vector representing the diagonal entries of U is used instead to conserve memory. It should be noted that this function allows b to be a matrix and therefore returns the appropriate matrix solution to $Ax = b$. This has been tested on random integer and float arrays of various sizes in `test_q1.py`, for which all tests are passed, suggesting that `LUsolve` has been implemented correctly.

Question 2 [1]: (20 %)

(a)

To find the constant C_1 we first find constant C.

- Finding C:

  Consider equations 4 on the Question Sheet (QS.4).

$$w^{n+1} - w^n - \frac{\Delta t}{2}\left(u_{xx}^n + u_{xx}^{n+1}\right) = 0, \quad u^{n+1} - u^n - \frac{\Delta t}{2}\left(w^n + w^{n+1}\right) = 0 \tag{QS.4}$$

  We can differentiate the second equation twice and rearrange it to:

$$u_{xx}^{n+1} = u_{xx}^n + \frac{\Delta t}{2}\left(w_{xx}^n + w_{xx}^{n+1}\right)$$

  Then substituting into the first equation gives:

$$w^{n+1} - w^n - \frac{\Delta t}{2}\left(u_{xx}^n + \left[u_{xx}^n + \frac{\Delta t}{2}\left(w_{xx}^n + w_{xx}^{n+1}\right)\right]\right) = 0$$

$$\implies w^{n+1} - w^n - u_{xx}^n \Delta t - \left(\frac{\Delta t}{2}\right)^2 \left(w_{xx}^n + w_{xx}^{n+1}\right) = 0$$

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

$$w^{n+1} - C w_{xx}^{n+1} = f \tag{QS.5}$$

  Comparing equations 1 and QS.5 we can see $C = \left(\frac{\Delta t}{2}\right)^2$ and $f = w^n + \left(\frac{\Delta t}{2}\right)^2 w_{xx}^n + u_{xx}^n \Delta t$.

- Finding C_1

  I seek to approximate w_{xx}^{n+1} as a function of w^{n+1}. I will do this by repeatedly applying the central difference formula [5].

$$\varphi'(a) \approx \frac{\varphi(a+h) - \varphi(a-h)}{2h} \qquad \text{(Central difference formula)}$$

  Applying the central difference formula to itself we can obtain a version for second derivatives:

$$\varphi''(a) \approx \frac{\varphi(a+2h) - 2\varphi(a) + \varphi(a-2h)}{(2h)^2} \tag{2}$$

  For fixed t, consider the below application of (2).

$$\text{Set } \varphi = w^{n+1},\ a = i\,(\Delta x)\,,\ h = \frac{\Delta x}{2}$$

$$\implies w_{xx,i}^{n+1} \approx \frac{w_{i+1}^{n+1} - 2w_i^n + w_{i-1}^{n+1}}{(\Delta x)^2} \tag{3}$$

  Evaluating (1) at $x = i\Delta x$ we can substitute the approximation for $w_{xx,i}^{n+1}$ (equation 3) to simplify this to:

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

$$w_i^{n+1} - C_1\left(w_{i+1}^{n+1} - 2w_i^{n+1} + w_{i-1}^{n+1}\right) = f_i \tag{QS. 6}$$

Where: $f_i \coloneqq w_i^n + \left(\frac{\Delta t}{2}\right)^2 w_{xx,i}^n + u_{xx,i}^n \Delta t$. Hence we can clearly see $C_1 = \left(\frac{\Delta t}{2\Delta x}\right)^2$. We can similarly apply (2) to f_i:

$$\implies f_i = w_i^n + \left(\frac{\Delta t}{2\Delta x}\right)^2 (w_{i+1} - 2w_i + w_{i-1}) + \frac{u_{i+1} - 2u_i + u_{i-1}}{(\Delta x)^2}\Delta t \tag{5}$$

(b) It should be noted due to the periodicity and boundary conditions of u in x (and therefore u_t) we can conclude:

I $$u(0) = u(M\Delta x, t),\ w(0,t) = w(M\Delta x, t)$$

II $$u((M+1)\Delta x, t) = u(\Delta x, t),\ w((M+1)\Delta x, t) = w(\Delta x, t)$$

Hence we can write (4) in matrix form:

$$Ax = b \tag{6}$$

Where:

$$A \coloneqq \begin{bmatrix} c & d & 0 & 0 & \cdots & 0 & 0 & d \\ d & c & d & 0 & \cdots & 0 & 0 & 0 \\ 0 & d & c & d & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & c & d & 0 \\ 0 & 0 & 0 & 0 & \cdots & d & c & d \\ d & 0 & 0 & 0 & \cdots & 0 & d & c \end{bmatrix} \tag{7}$$

$$x \coloneqq \begin{bmatrix} w_1^{n+1} \\ w_2^{n+1} \\ \vdots \\ w_M^{n+1} \end{bmatrix}, \quad b \coloneqq \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_M \end{bmatrix}$$

With $c \coloneqq 1 + 2C_1$, $d \coloneqq -C_1$.

(c)

We can apply the LU decomposition for banded matrices algorithm (Algorithm 1) to A (Equation 7), we notice that its bandwidth is $p = q = m - 1$. And hence Algorithm 1 collapses to the standard LU decomposition algorithm (since there doesn't exist any bands for which A is always 0 diagonally above or below them). And so we would expect there to be no advantage to use the banded algorithm.

Analysing the operation count: we know from lectures that the operation count for the LU decomposition of banded matrices is $\mathcal{O}(mpq)$, applied to A this is $\mathcal{O}(m(m-1)^2) \sim \mathcal{O}(m^3)$. Compare this to the standard LU decomposition algorithm which we know from lectures notes to also have an operation count $\mathcal{O}(m^3)$, we can see there is no difference in the order of the operation counts. This is shown by implementing the function `timing_LU_comparison`, which finds the average computational time (for n repeats) of the LU decomposition of a random matrix of the form $A \in \mathbb{R}^{m \times m}$. This is found by running `q2.py`. When I run this:

- For 200 repeats and $m = 100$:

  ```
  Avg (200x) percentage difference of computational time of LU_banded against LU_inplace on
  100x100 matrices is 1.31%
  ```

- For 200 repeats and $m = 200$:

  ```
  Avg (200x) percentage difference of computational time of LU_banded against LU_inplace on
  200x200 matrices is 0.63%
  ```

- For 200 repeats and $m = 300$:

  ```
  Avg (200x) percentage difference of computational time of LU_banded against LU_inplace on
  300x300 matrices is -0.09%
  ```

As we can see there is negligible difference in the test times, further providing evidence that there is no advantage to using a banded algorithm.

(d) We note that the below satisfy the equation stated in the question, $A = T + u_1 v_1^T + u_2 v_2^T$.

$$T \coloneqq \begin{bmatrix} c & d & 0 & 0 & \cdots & 0 & 0 & 0 \\ d & c & d & 0 & \cdots & 0 & 0 & 0 \\ 0 & d & c & d & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & c & d & 0 \\ 0 & 0 & 0 & 0 & \cdots & d & c & d \\ 0 & 0 & 0 & 0 & \cdots & 0 & d & c \end{bmatrix} \tag{8}$$

$$u_1 \coloneqq v_2 \coloneqq \begin{bmatrix} d \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \quad u_2 \coloneqq v_1 \coloneqq \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix}$$

Consider:

$$\hat{U} \coloneqq \begin{bmatrix} u_1 & u_2 \end{bmatrix} \in \mathbb{R}^{m \times 2}, \; \hat{V} \coloneqq \begin{bmatrix} v_1^T \\ v_2^T \end{bmatrix} \in \mathbb{R}^{2 \times m}$$

Hence $\hat{U}\hat{V} = u_1 v_1^T + u_2 v_2^T$. Considering the LU decomposition of T, that is $T = LU$, let us define:

$$\overline{U} \coloneqq L^{-1}\hat{U}, \; \overline{V} \coloneqq \hat{V}U^{-1}$$

$$\implies A = L\left(I + \overline{UV}\right)U$$
$$\implies A^{-1} = U^{-1}\left(I + \overline{UV}\right)^{-1}L^{-1}$$

Hence we require $\left(I + \overline{UV}\right)^{-1}$, this is found in Theorem 3.3

Lemma 3.1. *[2] $(I + A)$ has:*

- *Left inverse equal to: $I - (I + A)^{-1}A$*
- *Right inverse equal to: $I - A(I + A)^{-1}$*

Proof.

- Left inverse:

$$I = (I + A)^{-1}(I + A) = (I + A)^{-1} + (I + A)^{-1}A$$
$$\implies (I + A)^{-1} = I - (I + A)^{-1}A$$

- Right inverse: Similar.

□

Lemma 3.2. *[2] $B(I + CB)^{-1} = (I + BC)^{-1}B$*

Proof.

$$B + BCB = B(I + CB) = (I + BC)B$$

Multiplying on the right by $(I + CB)^{-1}$ and the left by $(I + BC)^{-1}$.

$$\implies B(I + CB)^{-1} = (I + BC)^{-1}B \tag{9}$$

□

Theorem 3.3. *[2] $(I + \overline{UV})^{-1} = I - \overline{U}(I + \overline{VU})^{-1}\overline{V}$*

Proof. We aim to find the inverse of $(I + \overline{UV})^{-1}$, I shall do this by considering the left and right inverse of $I + \overline{UV}$ using Lemma 3.1 and then showing both are equal to $I - \overline{U}(I + \overline{VU})^{-1}\overline{V}$ using Lemma 3.2.

- By Lemma 3.1 the left inverse of $I + \overline{UV}$ equals:

$$I - (I + \overline{UV})^{-1}\overline{UV} \overset{\text{Lemma 3.2}}{=} I - \overline{U}(I + \overline{UV})^{-1}\overline{V}$$

- By Lemma 3.2 the right inverse of $I + \overline{UV}$ equals:

$$I - \overline{UV}(I + \overline{UV})^{-1} \overset{\text{Lemma 3.2}}{=} I - \overline{U}(I + \overline{UV})^{-1}\overline{V}$$

□

And hence, continuing our derivation of A^{-1}:

$$\begin{aligned} A^{-1} &= U^{-1}\left(I - \overline{U}(I + \overline{VU})^{-1}\overline{V}\right)L^{-1} \\ &= T^{-1} - T^{-1}\hat{U}(I + \hat{V}T^{-1}\hat{U})^{-1}\hat{V}T^{-1} \end{aligned}$$

It should be noted that $(I + \hat{V}T^{-1}\hat{U})^{-1}$ can easily be calculated since, $I + \hat{V}T^{-1}\hat{U} \in \mathbb{R}^{2\times 2}$ and hence its inverse is tractable.

(e)

Solving $Ax = b$ is equivalent to the multiplication of b by A^{-1} on the left, that is $A^{-1}b$

$$\begin{aligned} A^{-1}b &= \left(T^{-1} - T^{-1}\hat{U}(I + \hat{V}T^{-1}\hat{U})^{-1}\hat{V}T^{-1}\right)b \\ &= T^{-1}b - T^{-1}\hat{U}(I + \hat{V}T^{-1}\hat{U})^{-1}\hat{V}T^{-1}b \end{aligned}$$

Forming T^{-1} is computationally expensive, however we only need $T^{-1}b$ and $T^{-1}\hat{U}$ hence these can be found by the merged LU decomposition of T with forward substitution then back substitution, as we found in question 1. As mentioned prior $(I + \hat{V}T^{-1}\hat{U})^{-1}$ is tractable and inexpensive to compute as it is the inverse of a 2×2 matrix. This entire process is written in (Algorithm 3) and explained below.

1 Find $T^{-1}\hat{U}$ and $T^{-1}b$
2 Find $(I + \hat{V}T^{-1}\hat{U})^{-1}$
3 $x = T^{-1}b - T^{-1}\hat{U}(I + \hat{V}T^{-1}\hat{U})^{-1}\hat{V}T^{-1}b$

Algorithm 8: Solving $Ax = b$

Now considering each of line individually

- Line 1 is solved by first combining $\hat{U}$ and b to form one matrix, that is:

$$\begin{bmatrix} | & | \\ \hat{U} & b \\ | & | \end{bmatrix} \tag{10}$$

This matrix is then solved: $Tx = \begin{bmatrix} | & | \\ \hat{U} & b \\ | & | \end{bmatrix}$ Giving x of the form $\begin{bmatrix} | & | \\ T^{-1}\hat{U} & T^{-1}b \\ | & | \end{bmatrix}$. These can then be separated to give $T^{-1}\hat{U}$ and $T^{-1}b$ without explicitly forming T^{-1} (which is computationally expensive). This is done by passing c, d representing the diagonal, and sub/super-diagonals (and corners) of T respectively, as well as the concatenated $\hat{U}$, b into the function `q1.LUsolveA` which efficiently solves this problem taking advantage of the form of T. It further is preferable to call this function on the concatenated array $\hat{U}b$ as opposed to $\hat{U}$ and b individually, since then we can take advantage of solving them simultaneously.

We showed in Question 1 the operation count of solving $Ax = b$ in this way is $\mathcal{O}(m)$, this order is retained, given that $\hat{U}b$ does not have a significant number of columns (in this case it has only 3 which is insignificant in comparison to (large) m).

- Line 2 is solved by multiplying to get $I + \hat{V}(T^{-1}\hat{U})$, then inverting it, where $T^{-1}\hat{U}$ has been found in previously in line 1. It should be noted that due to the sparsity of $\hat{V}$ (it only has 2 non-zero elements). $\hat{V}(T^{-1}\hat{U})$ can be simplified without the need to multiply $\hat{V}$ and $(T^{-1}\hat{U})$.

$$\hat{V}(T^{-1}\hat{U}) = \begin{bmatrix} 0 & 0 & \cdots & 0 & 1 \\ d & 0 & \cdots & 0 & 0 \end{bmatrix} \begin{bmatrix} (T^{-1}\hat{U})_{1,1} & (T^{-1}\hat{U})_{1,2} \\ \vdots & \vdots \\ (T^{-1}\hat{U})_{m,1} & (T^{-1}\hat{U})_{m,2} \end{bmatrix} = \begin{bmatrix} (T^{-1}\hat{U})_{m,1} & (T^{-1}\hat{U})_{m,2} \\ d \cdot (T^{-1}\hat{U})_{1,1} & d \cdot (T^{-1}\hat{U})_{1,2} \end{bmatrix}$$

  This forms a 2×2 matrix and hence its inverse is found easily by applying the formula for the inverse of a 2×2 matrix, this is done so in `inv2by2`.

  The operation count of this line is of constant order, since to form $\hat{V}(T^{-1}\hat{U})$ we only require 2 multiplications, then two more additions to add ones to the diagonal (that is $+I$), and finally 9 operations to form the 2×2 inverse.

- Line 3 is solved by simply multiplying the respective components together where $T^{-1}\hat{U}$, $T^{-1}b$, $\left(I + \hat{V}T^{-1}\hat{U}\right)^{-1}$ and $\hat{V}(T^{-1}\hat{U})$ have been found previously.

  It should be noted that $\hat{V}(T^{-1}b)$ can be simplified by again considering the sparsity of $\hat{V}$, giving:

$$\hat{V}(T^{-1}b) = \begin{bmatrix} (T^{-1}b)_m \\ d \cdot (T^{-1}b)_1 \end{bmatrix}$$

  We should also notice that when computing $(T^{-1}\hat{U})(I + \hat{V}T^{-1}\hat{U})^{-1}\hat{V}T^{-1}b$ it is advisable to first compute $(I + \hat{V}T^{-1}\hat{U})^{-1}\hat{V}T^{-1}b$ (of which we have computed all the constituent parts) and then multiply this result on the left by $(T^{-1}\hat{U})$, since this only requires one multiplication of dimension of m items.

  The operation count of this line is $\mathcal{O}(m)$: the computation of $\hat{V}(T^{-1}b)$ is of constant order due to the simplifications made because of the sparsity of $\hat{V}$, the computation of $(I + \hat{V}T^{-1}\hat{U})^{-1}\hat{V}T^{-1}b$ is also of constant order, since this requires the multiplication of previously calculated: $(I + \hat{V}T^{-1}\hat{U})^{-1}$ and $\hat{V}(T^{-1}b)$, which are of size 2×2, 2×1, respectively.

  Hence our operation count is influenced by: the matrix-vector multiplication of: $[T^{-1}\hat{U}][(I + \hat{V}T^{-1}\hat{U})^{-1}\hat{V}T^{-1}b]$, which has operation count $\mathcal{O}(m)$, and then the addition of this result to another m dimensional vector (which again is $\mathcal{O}(m)$)

This gives us the final solution to $Ax = b$, and an overall operation count of $\mathcal{O}(m)$.

(f) I have implemented an efficient method to solve $Ax = b$, for A of the form given in the question, this is done in `LUsolveq2A` in `q2.py`, similarly to as in question one the entire matrix A is not required, only c and d. This is tested with various integer and float arrays in `test_q2.py`. To compare this solver to the naive solving method I created a function that implements the `LU_inplace`, `solve_L`, `solve_U` from Exercises 6 to naively solve this problem (without taking advantage of the structure of A). I tested that this function indeed finds the valid solution in `test_q2.py`. I further created a function, analogous to one used in part c, to compare the average time taken to solve various random arrays by both methods, `timing_solve_comparison`

When I run this:

- For 200 repeats and $m = 100$:

  ```
  Avg (200x) percentage difference of computational time of LUsolveq2A against solve_inplace
  on 100x100 matrices is -72.78%
  ```

- For 200 repeats and $m = 200$:

  ```
  Avg (200x) percentage difference of computational time of LUsolveq2A against solve_inplace
  on 200x200 matrices is -84.11%
  ```

- For 200 repeats and $m = 300$:

  ```
  Avg (200x) percentage difference of computational time of LUsolveq2A against solve_inplace
  on 300x300 matrices is -90.01%
  ```

It is clearly evident that using `LUsolveq2A` as opposed to naively finding the LU decomposition then performing forward and back substitution offers a large computational time decrease.

(g)[3]

Below I formulate a method to solve the problem posed in Question 2.

1. Discretise the initial conditions: $u_i^0 = u_0(i\Delta x),\ u_i^0 = u_1(i\Delta x)$
2. Repeat below steps for required number of timesteps
   (a) Calculate f_i as per equation 5
   (b) Solve equation QS. 6 for w_i^{n+1}.
   (c) Solve for u^{n+1} as per the second equation of QS.4

Where $i = 1, 2, \ldots M$, and the implications of the boundary conditions as mentioned in part b: I, II.
However when computing this it is preferable to consider all i's simultaneously, this transforms our steps to performing the operations on the vector forms of the variables: $\boldsymbol{x}, \boldsymbol{w}^n, \boldsymbol{u}^n, \boldsymbol{f}$. And hence our method is slightly amended to:

1. Discretise the initial conditions: $u_i^0 = u_0(\boldsymbol{x}),\ u_i^0 = u_1(\boldsymbol{x})$
2. Repeat below steps for required number of timesteps, incrementing n each time
   (a) Calculate $\boldsymbol{b} = \boldsymbol{f}$ as per Equation 5
   (b) Solve equation 6 for $\boldsymbol{w}^{n+1}$.
   (c) Solve for $\boldsymbol{u}^{n+1}$ as per the second equation of QS.4

As required I have written a function to implement this as well as the options to plot and save, this is done so in function `solve_wave_equ` In Figures 1, 2, 3, I have applied the above algorithm to some examples.

- Figure 1, For `M=200, tsteps=1000, dt=0.001, interval=10`, considering $u(x,t) = sin(2\pi x)sin(2\pi t)$, we can see this is a valid solution to the wave equation and satisfies our boundary conditions, and we can derive the initial conditions that represent this solution: $u_0 = 0$, $u_1 = 2\pi sin(2\pi x)$. We can see that as time increases the wave appears to be oscillating, as we would expect for these initial conditions. Visually, the solutions at various time-steps appear to satisfy the boundary conditions, further providing evidence that `solve_wave_equ` has been implemented successfully.
- Figure 2, For `M=200, tsteps=1000, dt=0.001, interval=10`, considering $u(x,t) = sin(2\pi x)cos(2\pi t)$, we can see this is a valid solution to the wave equation and satisfies our boundary conditions, and we can derive the initial conditions that represent this solution: $u_0 = sin(2\pi x)$, $u_1 = 0$. We can see that as time increases the wave appears to be oscillating, as we would expect for these initial conditions. Visually, the solutions at various time-steps appear to satisfy the boundary conditions, further providing evidence that `solve_wave_equ` has been implemented successfully.
- Figure 3, For `M=200, tsteps=500, dt=0.001, interval=5`, considering $u(x,t) = sin(2\pi(x+t))$, we can see this is a valid solution to the wave equation and satisfies our boundary conditions, and we can derive the initial conditions that represent this solution: $u_0 = sin(2\pi x)$, $u_1 = 2\pi cos(2\pi x)$. We can see that as time increases the wave appears to be travelling left, as we would expect for these initial conditions. Visually, the solutions at various time-steps appear to satisfy the boundary conditions, further providing evidence that `solve_wave_equ` has been implemented successfully.

To further test this function, in `tests_q2.py` I have compared the solution found by `solve_wave_equ`, with initial conditions for which the solutions can easily calculated analytically to the analytical solution (evaluated at the corresponding x and t), I have done this for multiple different Δt, M and timesteps. The solution given by `solve_wave_equ` passes these tests leading me to believe that it provides a valid approximation to the solution to the wave equation.

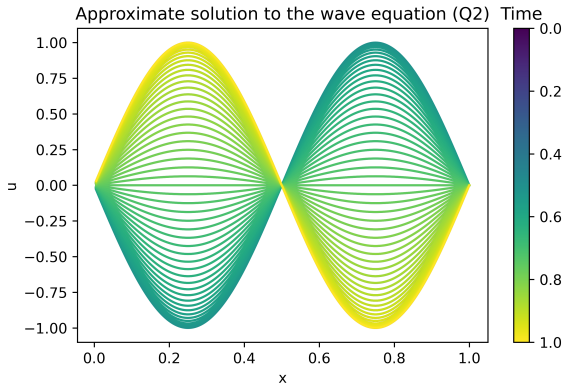

Figure 1: $u_0 = 0$, $u_1 = 2\pi sin(2\pi x)$

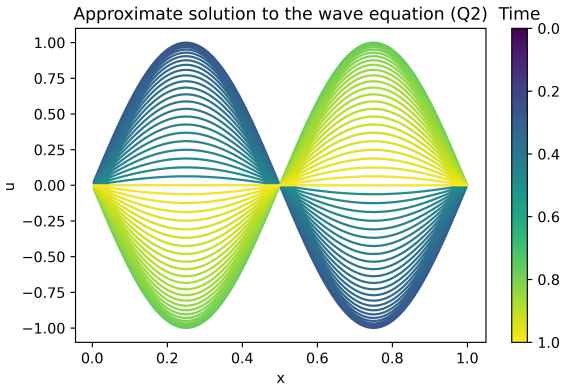

Figure 2: $u_0 = sin(2\pi x)$, $u_1 = 0$

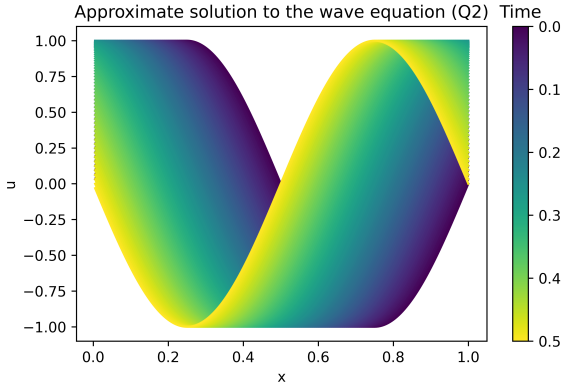

Figure 3: $u_0 = sin(2\pi x)$, $u_1 = 2\pi cos(2\pi x)$

Question 3 [1]: (20 %)

(a) I will prove that at each step of the QR algorithm A_k is tridiagonal (given that A_0 is tridiagonal), this will be proved by induction:

1. A_0 is tridiagonal by assumption.
2. Assume A_k is tridiagonal.
3. I will show A_{k+1} is tridiagonal.

   Consider the QR factorisation of A_k. R is upper triangular by the nature of the algorithm.

   Claim: Q is upper Hessenberg.

   Proof: By considering the Gram Schmidt orthogonalisation algorithm to produce Q we see Qe_j is a linear combination of all columns of A, $1 \to j$, since A is tridiagonal, this implies the claim, that Q is upper Hessenberg.

   I will show RQ is both upper Hessenberg and symmetric which implies RQ is tridiagonal.

   - Showing RQ is upper Hessenberg:
     Recall:

     $$\text{R is upper triangular} \implies r_{i,k} = 0 \; \forall i > k$$
     $$Q \text{ is upper Hessenberg} \implies q_{k,j} = 0 \; \forall k > j+1$$

     Consider RQ, that is $(RQ)_{i,j} = \sum_k r_{i,k} q_{k,j}$ Then:

     $$(RQ)_{i,j} \neq 0 \implies i \leq k \leq j+1 \implies i \leq j+1$$

which implies RQ is upper Hessenberg.

- Showing RQ is symmetric:
  Consider: $RQ = Q^T AQ$
  $(RQ)^T = \left(Q^T AQ\right)^T = Q^T A^T Q = Q^T AQ = RQ$ $\square$

(b) As hinted in the question, it is inefficient to use the full householder QR algorithm due to the sparsity (that is tridiagonal structure) of A. We know that at the j^{th} iteration householder reflectors are applied to A, eliminating all entries below the diagonal in the j^{th} column of A. Since A is tridiagonal, there is only one such entry that is not already known to be zero, when applying the Householder reflector to the relevant sub-matrix of A, that is $a_{j+1,j}$, and hence it would be useless to apply the entire Householder reflector. Instead we apply the reflector, (found from $A_{k:k+1,k}$) to $A_{k:k+1,k:k+2}$. This simplifies the general householder algorithm to Algorithm 9.

$R \leftarrow A$
for $k = 1$ *to* $m - 1$ **do**
  $x \leftarrow R_{k:k+1,k}$
  $v_k \leftarrow sign(x_1)\|x\|_2 e_1 + x$
  $v_k \leftarrow \frac{v_k}{\|v_k\|}$
  $j \leftarrow min(k + 2, m)$
  $R_{k:k+1,k:j} \leftarrow R_{k:k+1,k:j} - 2v_k(v_k^T \cdot R_{k:k+1,k:j})$
end

Algorithm 9: Householder QR decomposition algorithm for symmetrical tridiagonal matrices A

Briefly considering the operation count of Algorithm 9, we can see that each iteration performs a constant number of operations (lets call this constant c), that is there are no operations that are dependent on the size of A (this makes sense since we are considering only 2 dimensional Householder reflectors as opposed to a size comparable to m). Hence the operation count is of the form $\sum_{k=1}^{m-1} c = (m-1)c \sim \mathcal{O}(m)$. Compared to the operation count for the general Householder algorithm, which we know from notes to be: $\mathcal{O}(m^3)$, we see that our algorithm is significantly more efficient.

(c)

`qr_factor_tri` is implemented in `q3.py`, it is then tested in `test_q3.py`, it is tested on several random matrices A to ensure it produces a valid QR factorisation, and also the assumptions made and proved in part a are correct. The function passes all the tests, suggesting that `qr_factor_tri` has been implemented correctly.

(d)

`qr_alg_tri` is implemented in `q3.py`, it is again tested in `test_q3.py`, it is tested on several random matrices, A, to ensure it preserves the trace of A, and that element $|T_{m,m-1}| < 10^{-12}$, we later test that its diagonal approaches the eigenvalues of A. The function passes all the tests, suggesting that `qr_alg_tri` has been implemented correctly. I applied `qr_alg_tri` to the matrix as described in the question, and when comparing the diagonal of the matrix after it has been passed through the algorithm to the eigenvalues of the given matrix, as computed by numpy, their absolute difference is:

$$\begin{bmatrix}6.54788232\text{e}{-18} & 2.54042122\text{e}{-17} & 1.15359111\text{e}{-16} & 4.85722573\text{e}{-17} & 1.33226763\text{e}{-15}\end{bmatrix}$$

which we can see is very small, and hence our algorithm finds the eigenvalues well, (this is to be expected because our algorithm produces a similar matrix which is almost diagonal).

Taking a closer look at the eigenvalues:

$$\begin{bmatrix}1.79889252\text{e}{-07} & 2.29961954\text{e}{-05} & 1.29982425\text{e}{-03} & 4.30978838\text{e}{-02} & 8.33789794\text{e}{-01}\end{bmatrix}$$

We see that all the eigenvalues are tightly packed and very close to zero, as briefly discussed in the lecture notes this means that it is relatively easy to find such eigenvalues. And although our algorithm performed well in finding the eigenvalues, this is what we should expect with such eigenvalues

(e)

This code is implemented in function `q3e`, and tested (by comparing its diagonal to the actual eigenvalues, computed by numpy, in `tests_q3.py`), leading us to believe that it has been implemented correctly. `plot3e` allows for the easy plotting of `q3e`, it takes in as its arguments the matrix A for the algorithm to be performed on, as well as other arguments concerning whether shift is required, the `pure_qr` algorithm should be performed instead and concerning graph aesthetics. Figures 4 and 5 (where A is the matrix as described in Question 3e), show us how the errors, that is $|T_{m-1,m}|$ change with each iteration of `qr_alg_tri` and `pure_QR` respectively. It should be noted that I amended `pure_QR` so that after each iteration it returns the error, that is how close the A_k is to being upper triangular. Where its error is defined as the norm of the lower triangular (not including the diagonal) part of the A_k at each iteration (and hence the tolerance being compared are not quite like for like). We can see `qr_alg_tri` takes fewer iterations to reach its tolerance, than `pure_QR`, and follows a regular saw-tooth pattern. The jumps seen in Figure 4 are due to the fact that once the tolerance is reached for $|T_{m,m-1}|$ the last rows and columns of that matrix are removed, and so effectively $|T_{m-1,m-2}|$ is now considered.

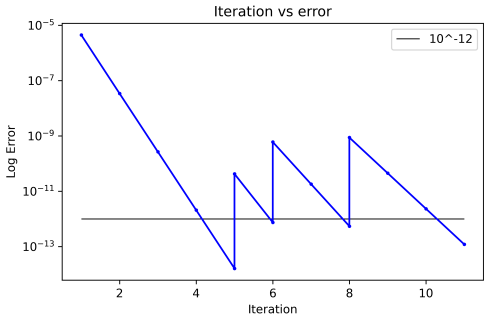

Figure 4: `qr_alg_tri` (unshifted) repeatedly applied to A from 3d

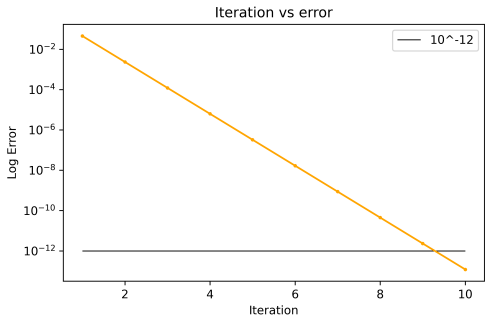

Figure 5: `pure_QR` applied A from 3d

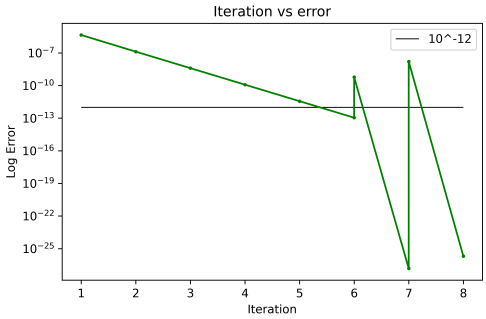

Figure 6: `qr_alg_tri` (shifted) repeatedly applied to A from 3d

I further compare the `pure_QR` algorithm to the unshifted `qr_alg_tri` in Figures 7, 8 (where A is a random 30×30 matrix) we can see that the number of iterations required to bring the error below the tolerance is approximately the same. This is further confirmed when comparing Figures 10, 11, that is the respective algorithms applied to the matrix given in part g. Hence overall we can conclude that there is nominal difference in iterations between: `pure_QR` and `qr_alg_tri` for unshifted matrices. However the `pure_QR` algorithm performs each iteration on the entire matrix A, compared to `qr_alg_tri` which performs some iterations on sub-matrices.

(f) I have created a function, `wilk_shift` that takes in a matrix A and outputs the Wilkinson shift for that matrix. I have then implemented this in my function `q3e` such that if `shift=True` the shift is used. This is tested in `tests_q3.py` and passes all the tests suggesting that the shift has been implemented successfully. Now to compare the shifted algorithm to its unshifted counterpart and the pure QR algorithm.

- The 5×5 matrix from part d. That is comparing Figure 6 to Figures 4, 5. We see that the shifted algorithm reaches its tolerance in fewer iterations than its unshifted counterpart, that is 3 fewer (and so also fewer than the pure QR algorithm). The shifted algorithm takes longer to bring the first $|T_{m-1,m}|$ below the tolerance than the unshifted algorithm. Reducing the tolerance of all following $|T_{m-1,m}|$ is done in only one iteration each (compared to 1,2 or 3 iterations for the unshifted algorithm). Further, the shifted algorithm requires fewer cuts of the main matrix into sub matrices, since iterations seem to reduce

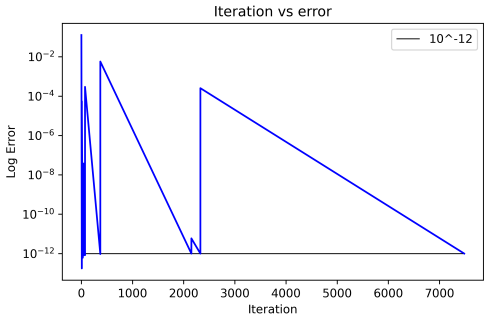

Figure 7: `qr_alg_tri` (unshifted) repeatedly applied to a random 30×30 matrix

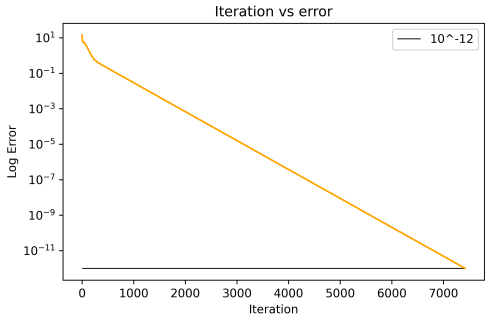

Figure 8: `pure_QR` applied to a random 30×30 matrix

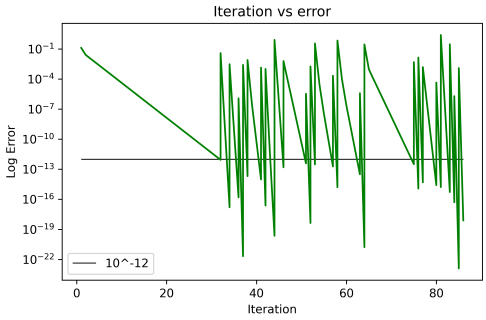

Figure 9: `qr_alg_tri` (shifted) repeatedly applied to a random 30×30 matrix

the absolute values of other sub diagonal entries more.

- The random 30×30 matrix. That is comparing Figure 9 to Figures 7, 8. Here the shifted algorithm reaches its tolerance in significantly fewer (about 7000 fewer) iterations than in the unshifted algorithm (and therefore also the pure QR algorithm). We can see that the unshifted algorithm follows a more regular saw tooth pattern than the shifted algorithm, which seems to reduce the tolerance slowly at first and then more rapidly as the algorithm progresses.
- $A = D + O$ matrix mentioned in part f. Comparison with this matrix has been left to part f.

It should be noted that all the above matrices reduce the error at a linear rate according to the graphs, that is the errors decrease in straight lines, although because we are using a *log* y axis the values are in fact decreasing in an exponential fashion.

(g)

Figures 10, 11, 12 show the implementation of the unshifted algorithm, the unshifted algorithm and the pure QR algorithm, applied to the matrix $A = D + O$. It can again be seen that the shifted algorithm requires far fewer iterations than its unshifted and pure QR counterparts. It is interesting to see that it takes the shifted algorithm longer than the unshifted algorithm to bring the first error down below tolerance, however once this has been done, the shifted algorithm requires very few iterations to reduce the following sub-matrices below the tolerance. The unshifted algorithm follows a regular saw tooth pattern with there being no significant difference in the number of iterations taken to reduce different errors to below the tolerance. I found the eigenvalues of the matrix to be approximately:

$$\begin{bmatrix} 0.215 & 1.257 & 2.288 & 3.314 & 4.339 & 5.363 & 6.387 & 7.412 & 8.439 & 9.468 \\ 10.501 & 11.540 & 2.590 & 13.664 & 23.223 & & & & & \end{bmatrix}$$

That is they are not very tightly packed, especially compared to the eigenvalues of the 5×5 matrix mentioned earlier. Hence, I believe without a shift to help locate the eigenvalues, it requires many iterations to successfully find them. Whereas the shift helps seek out the eigenvalues reducing the iterations required.

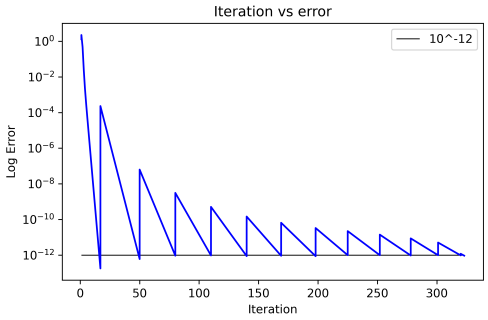

Figure 10: `qr_alg_tri` (unshifted) repeatedly applied to $A = D + O$

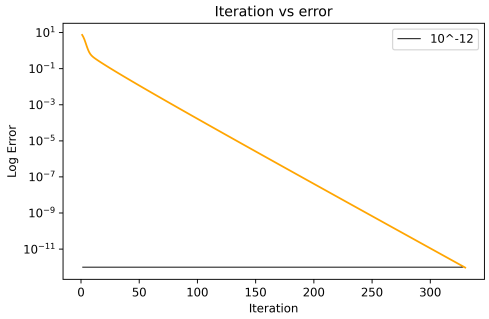

Figure 11: `pure_QR` applied to $A = D + O$

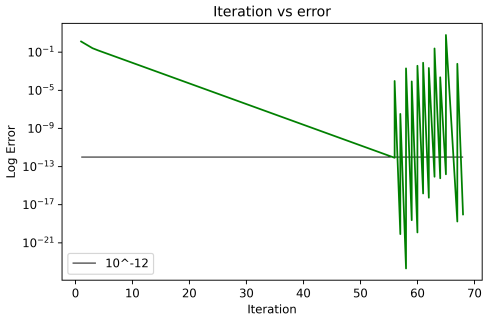

Figure 12: `qr_alg_tri` (shifted) repeatedly applied to $A = D + O$

Question 4 [1]: (20 %)

(a)

My `GMRES` function from the exercises has been modified as required, so that it can perform preconditioning. The algorithm checks the tolerance of the preconditioned system (when `apply_pc` is given), that is testing against $\|M^{-1}Ax - M^{-1}b\|$ and returns these norms. However to analyse the algorithm for preconditioning I amended `GMRES` such that if an `apply_pc` is provided, and `return_residual_norms` is true, the norms returned are those of the preconditioned system. This is tested in `test_q4.py`, where the preconditioned problem is seen to give a valid solution.

(b) Consider an arbitrary eigenvalue λ, and its respective eigenvector v for the matrix $\left(M^{-1}A\right)$. Hence:

$$\begin{aligned} \lambda v &= M^{-1}Av \\ \implies v - \lambda v &= v - M^{-1}Av \\ \implies \|v - \lambda v\| &= \|v - M^{-1}Av\| \\ \implies |1-\lambda|\|v\| &\leq \|I - M^{-1}A\|\|v\| \\ \implies |1-\lambda| &\leq \|I - M^{-1}A\| = c \ \square \end{aligned}$$

Since $\|I - M^{-1}A\| = c < 1$

(c)

Recall from section 6.4 of the lecture notes we showed for diagonalisable A, that is $A = V\Lambda V^{-1}$, where Λ is diagonal. The following is true:

- The residual of the the system $r_n = Ax_n - b$ can be thought of in terms of polynomials, that is

$$\|r_n\| \leq \|p_n(A)\|\|b\|$$

Consider $p_n(x) = (1-x)^n$ and the preconditioned problem: $M^{-1}Ax = M^{-1}b$ where we define $\tilde{b} \coloneqq M^{-1}b$, we shall consider the residual to this problem, assuming $M^{-1}A$ is diagonalisable and the inequality stated in the question holds.

$$\|r_n\| \leq \|P_n(M^{-1}A)\|\|\tilde{b}\|$$
$$\|P_n(M^{-1}A)\| = \|\left(I - M^{-1}A\right)^n\| \leq \|I - M^{-1}A\|^n = c^n$$
$$\implies \|r_n\| \leq c^n\|\tilde{b}\|$$

Hence we can see that the convergence of the residual to 0 is exponential.

To calculate the convergence rate we consider

$$\lim_{n\to\infty} \frac{\|r_{n+1}\|}{\|r_n\|} \leq \lim_{n\to\infty} \frac{c^{n+1}\|M^{-1}b\|}{c^n\|M^{-1}b\|} \leq c < 1$$

And hence c provides an upper bound for the rate of convergence. Further it important to note $p_n(x)$ is valid since $p(0) = 1\ \forall n$.

(d)

I first decided to find the optimal c_1, such that $|\lambda - 1| = c < 1$ where λ is any eigenvalue of $M^{-1}A$, ($M \coloneqq c_1U$, and U is defined to be the upper triangular part of A, including the diagonal). I further picked the c_1 that minimised c. I did this by plotting $|\lambda - 1|_{\max}$ while varying c_1 for various sized matrices, this is seen in Figure 13. It seems as the matrix size increases c_1 tends to 0.5 from above.

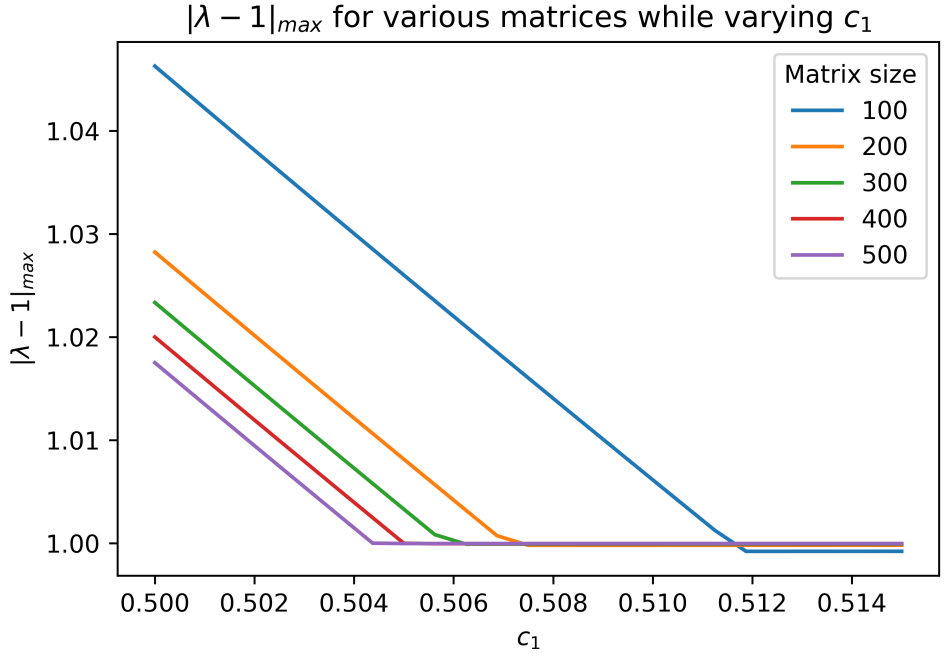

Figure 13: c_1 against $|\lambda - 1|_{\max}$

Figure 13 is plotted using `q4.py`. I now consider 500×500 matrices, running `q4.py` shows us the optimal c_1 : `For 500x500 matrices the optimal` c_1 `is 0.505`.

I now compare the solutions of $Ax = b$, for A created as suggested in Question 4 ($A = I + L$), with the preconditioned GMRES and the non-preconditioned GMRES. With 5 500×500 matrices the non-preconditioned

algorithm doesn't converge (whereas the preconditioned one does) : this is clear since my script returns:

```
Iteration:1 Non-preconditioned system doesnt converge
Iteration:2 Non-preconditioned system doesnt converge
Iteration:3 Non-preconditioned system doesnt converge
Iteration:4 Non-preconditioned system doesnt converge
Iteration:5 Non-preconditioned system doesnt converge
```

And doesn't return this message at all for the preconditioned system. The script also returns if the estimate for the solution is far from the actual solution to the system but this message is not shown either, suggesting that despite the non preconditioned system not converging it does produce an acceptable estimate for the solution to $Ax = b$.

I consider the eigenvalues of both the non-preconditioned system and the preconditioned system:

- My script returns for the non-preconditioned system: `Max abs eigenvalue = 136859.32, min abs eigenvalue = 1.00 without pc.` Hence the eigenvalues are not very tightly packed, there is at least one about 1 away from the origin and at least one other that is 136859.32 away from the origin. This suggests that the non preconditioned system is ill conditioned and hence it is not surprising that it doesn't converge.
- My script returns for the non-preconditioned system: `Max abs eigenvalue = 2.00, min abs eigenvalue = 0.00 with pc.` Hence the eigenvalues are fairly tightly packed, we see in our calculation of c that all the eigenvalues are such that $|\lambda - 1| < c$ and $c \approx 0.99997$ The more tightly packing of the eigenvalues implies that this preconditioned system should be easier to estimate the solution of, and hence it is not surprising that the preconditioned system converges when the non-preconditioned system does not.

Consider the upper bound of the preconditioned system residuals, that is $\|r_n\| \leq \|M^{-1}b\|c^n$ This is plotted in Figure 14, which is produced in `q4.py`. As we can see from the figure that $\|M^{-1}b\|c^n$ doesn't provide a very good upper bound for the system since it is considerably greater than the errors for all iterations. This is likely due to the fact that c is very close to 1 and so the upper bound provides a negligible decrease at each iteration. Perhaps for extremely large matrices or problems where significant numbers of iterations are required this upper bound would have more use. But in our case and probably in most cases it is of little use, despite it being a valid upper bound.

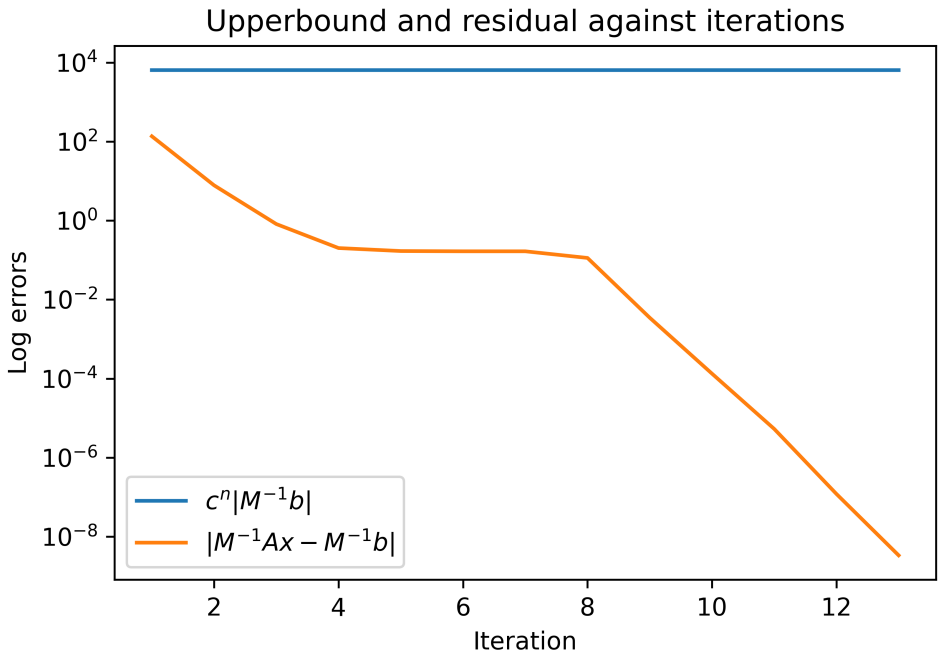

Figure 14: Plot to help determine if $\|M^{-1}b\|c^n$ provides a good upper bound for the residuals.

Question 5 [1]: (20 %)

(a)

$$w_i^{n+1} - w_i^n - \frac{\Delta t}{2\Delta x^2}\left(u_{i-1}^n + u_{i-1}^{n+1} - 2\left(u_i^n + u_i^{n+1}\right) + u_{i-1}^n + u_{i-1}^{n+1}\right) = 0 \tag{QS.10}$$

$$u_i^{n+1} - u_i^n - \frac{\Delta t}{2}\left(w_i^n + w_i^{n+1}\right) = 0 \tag{QS.11}$$

Consider $B \in \mathbb{C}^{2M\times 2M}$ and $I \in \mathbb{C}^{2M\times 2M}$, with $B_{ij} \in \mathbb{C}^{M\times M}$ We find the form of $r \in \mathbb{C}^{2M}$ and B_{ij} by considering the case of the first $2M$ rows (that is the first block of rows) and the subsequent rows separately.

- First $2M$ rows:

  Firstly let us split r into $r \coloneqq \begin{bmatrix} r_1 \\ r_2 \end{bmatrix}$, where $r_1,\, r_2 \in \mathbb{C}^M$

$$\begin{aligned} &\left(\begin{bmatrix} I & 0 & \cdots & 0 \end{bmatrix} + \frac{1}{2}\begin{bmatrix} B & 0 & \cdots & 0 \end{bmatrix}\right) U = \begin{bmatrix} r_1 \\ r_2 \end{bmatrix} \\ \implies &\left(I + \frac{1}{2}B\right)\begin{bmatrix} p_1 \\ q_1 \end{bmatrix} = \begin{bmatrix} r_1 \\ r_2 \end{bmatrix} \\ \implies &\begin{bmatrix} p_1 \\ q_1 \end{bmatrix} + \frac{1}{2}\begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix}\begin{bmatrix} p_1 \\ q_1 \end{bmatrix} = \begin{bmatrix} r_1 \\ r_2 \end{bmatrix} \\ \implies &\begin{bmatrix} p_1 \\ q_1 \end{bmatrix} + \frac{1}{2}\begin{bmatrix} B_{1,1}p_1 + B_{1,2}q_1 \\ B_{2,1}p_1 + B_{2,2}q_1 \end{bmatrix} = \begin{bmatrix} r_1 \\ r_2 \end{bmatrix} \end{aligned} \tag{11}$$

  We can see that the first and second block rows of Equation 11, are equivalent to Equations QS.10, QS.11 respectively, where $n = 0$. This is an edge case since it requires p_0 and q_0 which are analogous to our initial conditions, this is represented in the vector r.

  – First block of rows (M rows):
  Comparing $p_1 + \frac{1}{2}\left(B_{1,1}p_1 + B_{1,2}q_1\right) = r_1$ to Equation QS.11 we see that:

$$B_{1,1} = 0,\; B_{1,2} = -\Delta t \cdot I \tag{12}$$

$$r_1 = u^0 - \frac{1}{2}B_{1,2}w^0$$

  Where u^0 and w^0 represent discretised vectors of the initial conditions: $u(x,0)$ and $u_t(x,0)$ respectively.

  – Second block of rows (M rows):
  Comparing $q_1 + \frac{1}{2}\left(B_{2,1}p_1 + B_{2,2}q_1\right) = r_2$ to Equation QS.10 we see that:

$$B_{2,1} = \begin{bmatrix} c & d & 0 & 0 & \cdots & 0 & 0 & d \\ d & c & d & 0 & \cdots & 0 & 0 & 0 \\ 0 & d & c & d & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & c & d & 0 \\ 0 & 0 & 0 & 0 & \cdots & d & c & d \\ d & 0 & 0 & 0 & \cdots & 0 & d & c \end{bmatrix} \quad \text{where } d = -\frac{\Delta t}{\Delta x^2},\; c = \frac{2\Delta t}{\Delta x^2},\; B_{2,2} = 0 \tag{13}$$

$$r_2 = w^0 - \frac{1}{2}B_{2,1}u^0$$

- The subsequent rows (after the first $2M$ rows):

$$\left(\begin{bmatrix}-I & I & 0 & \cdots & 0\end{bmatrix} + \frac{1}{2}\begin{bmatrix}B & B & 0 & \cdots & 0\end{bmatrix}\right) U = 0$$

$$\implies \left(\begin{bmatrix}-I & I\end{bmatrix} + \frac{1}{2}B\begin{bmatrix}I & I\end{bmatrix}\right)\begin{bmatrix}p_{i-1}\\ q_{i-1}\\ p_i\\ q_i\end{bmatrix} = 0$$

$$\implies -\begin{bmatrix}p_{i-1}\\ q_{i-1}\end{bmatrix} + \begin{bmatrix}p_i\\ q_i\end{bmatrix} + \frac{1}{2}B\left(\begin{bmatrix}p_{i-1}\\ q_{i-1}\end{bmatrix} + \begin{bmatrix}p_i\\ q_i\end{bmatrix}\right) = 0$$

$$\implies \begin{bmatrix}p_i - p_{i-1}\\ q_i - q_{i-1}\end{bmatrix} + \frac{1}{2}\begin{bmatrix}B_{1,1} & B_{1,2}\\ B_{2,1} & B_{2,2}\end{bmatrix}\begin{bmatrix}p_{i-1} + p_i\\ q_{i-1} + q_i\end{bmatrix} = 0$$

$$\implies \begin{bmatrix}p_i - p_{i-1}\\ q_i - q_{i-1}\end{bmatrix} + \frac{1}{2}\begin{bmatrix}B_{1,1}(p_i + p_{i-1}) + B_{1,2}(q_i + q_{i-1})\\ B_{2,1}(p_i + p_{i-1}) + B_{2,2}(q_i + q_{i-1})\end{bmatrix} = 0 \tag{14}$$

We can see that the first and second block rows of Equation 14 are equivalent to Equations QS.11, QS.10 respectively, for $n = i = 1, 2, \ldots, N$.

- First block of rows:
  Comparing $p_i - p_{i-1} + \frac{1}{2}\left(B_{1,1}(p_i + p_{i-1}) + B_{1,2}(q_i + q_{i-1})\right) = 0$ to Equation QS.11 we see that the previous derivations of $B_{1,1}$ and $B_{1,2}$ are valid and solve the equation of the first block of rows of Equation 14 analogous to Equation QS.11.
- Second block of rows:
  Comparing $q_i - q_{i-1} + \frac{1}{2}\left(B_{2,1}(p_i + p_{i-1}) + B_{2,2}(q_i + q_{i-1})\right) = 0$ to Equation QS.10 we see that the previous derivations of $B_{2,1}$ and $B_{2,2}$ are valid and solve the equation of the second block of rows of Equation 14 analogous to Equation QS.10.

Hence

$$B \coloneqq \begin{bmatrix}0 & -\Delta t \cdot I\\ B_{2,1} & 0\end{bmatrix} \text{ For } B_{2,1} \text{as in Equation 12}$$

$$r = \begin{bmatrix}r_1\\ r_2\end{bmatrix} = \begin{bmatrix}u^0 + \frac{\Delta t}{2}w^0\\ w^0 - \frac{1}{2}B_{2,1}u^0\end{bmatrix}$$

(b) Assume that U^k converges to U^∞, by the iterative method in the question. I will show that this is the solution to Equation 13 from the question sheet.

To prove this, we will again consider the first block of rows and then all successive rows separately.

- First block of rows (first $2M$ rows)

$$r + \alpha\left(-I + \frac{B}{2}\right)\begin{bmatrix}p_N^\infty\\ q_N^\infty\end{bmatrix} = \left(\begin{bmatrix}I & 0 & \cdots & 0 & -\alpha I\end{bmatrix} + \frac{1}{2}\begin{bmatrix}B & 0 & \cdots & 0 & \alpha B\end{bmatrix}\right) U^\infty$$

$$= \left(\begin{bmatrix}I & -\alpha I\end{bmatrix} + \frac{1}{2}\begin{bmatrix}B & \alpha B\end{bmatrix}\right)\begin{bmatrix}p_1^\infty\\ q_1^\infty\\ p_N^\infty\\ q_N^\infty\end{bmatrix}$$

$$= \begin{bmatrix}p_1^\infty\\ q_1^\infty\end{bmatrix} - \alpha\begin{bmatrix}p_N^\infty\\ q_N^\infty\end{bmatrix} + \frac{1}{2}B\begin{bmatrix}p_1^\infty\\ q_1^\infty\end{bmatrix} + \frac{\alpha}{2}B\begin{bmatrix}p_N^\infty\\ q_N^\infty\end{bmatrix}$$

$$= \begin{bmatrix}p_1^\infty\\ q_1^\infty\end{bmatrix} + \frac{B}{2}\begin{bmatrix}p_1^\infty\\ q_1^\infty\end{bmatrix} + \alpha\left(-I + \frac{B}{2}\right)\begin{bmatrix}p_N^\infty\\ q_N^\infty\end{bmatrix}$$

$$\implies r = \begin{bmatrix}p_1^\infty\\ q_1^\infty\end{bmatrix} + \frac{B}{2}\begin{bmatrix}p_1^\infty\\ q_1^\infty\end{bmatrix} \tag{15}$$

We can see by comparing Equation 15 and Equation 11, that

$$\begin{bmatrix}p_1^\infty\\ q_1^\infty\end{bmatrix} = \begin{bmatrix}p_1\\ q_1\end{bmatrix}$$

which is a valid solution of the first two rows of Equation 13 from the question sheet.

- Successive rows (rows $2M+1$ and onwards)

  It can be noticed that for successive rows, Equation 17 from the question sheet is identical to Equation 15 (and so 13) (from the question sheet), since $C_1^{(\alpha)}$ differs from C_1 by its first row only, and so (for arbitrary matrix X) $C_1^{(\alpha)} \otimes X$ differs from $C_1 \otimes X$ by only its first block of rows, similar can be said for $C_2^{(\alpha)}$ and C_2. Hence $U^{\infty}_{2M+1:2MN}$ is a solution to Equation 17 from the question sheet implies it also solves Equation 13 from the question sheet.

(c)

$$v_j \coloneqq \begin{bmatrix} 1 \\ \alpha^{-\frac{1}{N}} e^{\frac{2\pi ij}{N}} \\ \alpha^{-\frac{2}{N}} e^{\frac{4\pi ij}{N}} \\ \vdots \\ \alpha^{-\frac{(N-1)}{N}} e^{\frac{2(N-1)\pi ij}{N}} \end{bmatrix} \tag{QS.19}$$

- I will show v_j is an eigenvalue of $C_1^{(\alpha)}$.

$$\begin{aligned}
C_1^{(\alpha)} v_j &= \begin{bmatrix} 1 & 0 & 0 & \cdots & -\alpha \\ -1 & 1 & 0 & \cdots & 0 \\ 0 & -1 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & -1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ \alpha^{-\frac{1}{N}} e^{\frac{2\pi ij}{N}} \\ \alpha^{-\frac{2}{N}} e^{\frac{4\pi ij}{N}} \\ \ddots \\ \alpha^{-\frac{(N-1)}{N}} e^{\frac{2(N-1)\pi ij}{N}} \end{bmatrix} \\
&= \begin{bmatrix} 1 - \alpha^{1-\frac{N-1}{N}} e^{\frac{2(N-1)\pi ij}{N}} \\ -1 + \alpha^{-\frac{1}{N}} e^{\frac{2\pi ij}{N}} \\ -\alpha^{-\frac{1}{N}} e^{\frac{2\pi ij}{N}} + \alpha^{-\frac{2}{N}} e^{\frac{4\pi ij}{N}} \\ \vdots \\ -\alpha^{-\frac{(N-2)}{N}} e^{\frac{2(N-2)\pi ij}{N}} + \alpha^{-\frac{(N-1)}{N}} e^{\frac{2(N-1)\pi ij}{N}} \end{bmatrix} \\
&= \begin{bmatrix} 1 - \alpha^{\frac{1}{N}} e^{\frac{-2\pi ij}{N}} \\ -1 + \alpha^{-\frac{1}{N}} e^{\frac{2\pi ij}{N}} \\ -\alpha^{-\frac{1}{N}} e^{\frac{2\pi ij}{N}} + \alpha^{-\frac{2}{N}} e^{\frac{4\pi ij}{N}} \\ \vdots \\ -\alpha^{-\frac{(N-2)}{N}} e^{\frac{2(N-2)\pi ij}{N}} + \alpha^{-\frac{(N-1)}{N}} e^{\frac{2(N-1)\pi ij}{N}} \end{bmatrix} \\
&= \left(1 - \alpha^{\frac{1}{N}} e^{\frac{-2\pi ij}{N}}\right) \begin{bmatrix} 1 \\ \alpha^{-\frac{1}{N}} e^{\frac{2\pi ij}{N}} \\ \alpha^{-\frac{2}{N}} e^{\frac{4\pi ij}{N}} \\ \vdots \\ \alpha^{-\frac{(N-1)}{N}} e^{\frac{2(N-1)\pi ij}{N}} \end{bmatrix} \\
&= \left(1 - \alpha^{\frac{1}{N}} e^{\frac{-2\pi ij}{N}}\right) v_j \\
&= \lambda_{1,j} v_j \ \square
\end{aligned}$$

  Where $\lambda_{1,j} \coloneqq \left(1 - \alpha^{\frac{1}{N}} e^{\frac{-2\pi ij}{N}}\right)$ for $j = 1, 2, ..., N-1$

- I will show v_j is an eigenvector of $C_2^{(\alpha)}$.

$$
\begin{aligned}
C_2^{(\alpha)} v_j &= \begin{bmatrix} \frac{1}{2} & 0 & 0 & \cdots & \frac{\alpha}{2} \\ \frac{1}{2} & \frac{1}{2} & 0 & \cdots & 0 \\ 0 & \frac{1}{2} & \frac{1}{2} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \frac{1}{2} & \frac{1}{2} \end{bmatrix} \begin{bmatrix} 1 \\ \alpha^{-\frac{1}{N}} e^{\frac{2\pi i j}{N}} \\ \alpha^{-\frac{2}{N}} e^{\frac{4\pi i j}{N}} \\ \vdots \\ \alpha^{-\frac{(N-1)}{N}} e^{\frac{2(N-1)\pi i j}{N}} \end{bmatrix} \\
&= \frac{1}{2} \begin{bmatrix} 1 + \alpha^{1-\frac{N-1}{N}} e^{\frac{2(N-1)\pi i j}{N}} \\ 1 + \alpha^{-\frac{1}{N}} e^{\frac{2\pi i j}{N}} \\ \alpha^{-\frac{1}{N}} e^{\frac{2\pi i j}{N}} + \alpha^{-\frac{2}{N}} e^{\frac{4\pi i j}{N}} \\ \vdots \\ \alpha^{-\frac{(N-2)}{N}} e^{\frac{2(N-2)\pi i j}{N}} + \alpha^{-\frac{(N-1)}{N}} e^{\frac{2(N-1)\pi i j}{N}} \end{bmatrix} \\
&= \frac{1}{2} \begin{bmatrix} 1 + \alpha^{\frac{1}{N}} e^{\frac{-2\pi i j}{N}} \\ 1 + \alpha^{-\frac{1}{N}} e^{\frac{2\pi i j}{N}} \\ \alpha^{-\frac{1}{N}} e^{\frac{2\pi i j}{N}} + \alpha^{-\frac{2}{N}} e^{\frac{4\pi i j}{N}} \\ \vdots \\ \alpha^{-\frac{(N-2)}{N}} e^{\frac{2(N-2)\pi i j}{N}} + \alpha^{-\frac{(N-1)}{N}} e^{\frac{2(N-1)\pi i j}{N}} \end{bmatrix} \\
&= \frac{1}{2} \left(1 + \alpha^{\frac{1}{N}} e^{\frac{-2\pi i j}{N}}\right) \begin{bmatrix} 1 \\ \alpha^{-\frac{1}{N}} e^{\frac{2\pi i j}{N}} \\ \alpha^{-\frac{2}{N}} e^{\frac{4\pi i j}{N}} \\ \vdots \\ \alpha^{-\frac{(N-1)}{N}} e^{\frac{2(N-1)\pi i j}{N}} \end{bmatrix} \\
&= \frac{1}{2} \left(1 + \alpha^{\frac{1}{N}} e^{\frac{-2\pi i j}{N}}\right) v_j \\
&= \lambda_{2,j} v_j \ \square
\end{aligned}
$$

Where $\lambda_{2,j} \coloneqq \frac{1}{2} \left(1 + \alpha^{\frac{1}{N}} e^{\frac{-2\pi i j}{N}}\right)$ for $j = 1, 2, ..., N-1$

We can clearly see that for $j = 0, 1, \ldots N-1$ the eigenvalues are distinct, both for $C_1^{(\alpha)}$ and $C_2^{(\alpha)}$, and therefore they both are diagonalisable.

$$
\begin{aligned}
C_1^{(\alpha)} &= V D_1 V^{-1} \\
C_1^{(\alpha)} &= V D_2 V^{-1}
\end{aligned}
$$

Where

$$
V \coloneqq \begin{bmatrix} | & | & & | \\ v_0 & v_1 & \cdots & v_{N-1} \\ | & | & & | \end{bmatrix}
$$

$$
D_1 \coloneqq \begin{bmatrix} \lambda_{1,0} & & & \\ & \lambda_{1,1} & & \\ & & \ddots & \\ & & & \lambda_{1,N-1} \end{bmatrix}
$$

$$
D_2 \coloneqq \begin{bmatrix} \lambda_{2,0} & & & \\ & \lambda_{2,1} & & \\ & & \ddots & \\ & & & \lambda_{2,N-1} \end{bmatrix}
$$

Hence we have shown the first equality of QS.20.

$$
C_1^{(\alpha)} \otimes I + C_2^{(\alpha)} \otimes B = \left(V D_1 V^{-1}\right) \otimes I + \left(V D_2 V^{-1}\right) \otimes B = (V \otimes I) \left(D_1 \otimes I + D_2 \otimes B\right) \left(V^{-1} \otimes I\right) \quad \text{(QS.20)}
$$

To show the second equality we first prove Theorem 5.1.

Theorem 5.1.

$$(A \otimes B)(C \otimes D) = AC \otimes BD$$

Proof. I will show $[(A \otimes B)(C \otimes D)]_{qr} = [AC \otimes BD]_{qr}$.

$$\begin{aligned}
[(A \otimes B)(C \otimes D)]_{qr} &= [(A \otimes B)]_{qj} [(C \otimes D)]_{jr} \\
&= a_{qj} B c_{jr} D \\
&= a_{qj} c_{jr} BD \\
&= [AB]_{qr} BD \\
&= [AB \otimes BD]_{qr}
\end{aligned}$$

□

Using Theorem 5.1 can now clearly see that the second inequality is also satisfied, since

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

(d)[4]

I first aim to find V in the form of a Fourier transform and multiplication by a diagonal matrix. Let us note that a discrete Fourier transform can be represented as a matrix, denoted W below. Let us denote the diagonal matrix as $\hat{D}$.

We should notice that

$$\hat{D}^{-1}W^{-1} = V \left(\implies V^{-1} = W\hat{D} \right)$$

where $\omega \coloneqq e^{\frac{2\pi i}{N}}$ and:

$$\hat{D}^{-1} \coloneqq \begin{bmatrix} N & & & \\ & N\alpha^{-\frac{1}{N}} & & \\ & & \ddots & \\ & & & N\alpha^{-\frac{N-1}{N}} \end{bmatrix}$$

$$W^{-1} = \frac{1}{N} \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{N-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{N-1} & \omega^{2(N-1)} & \cdots & \omega^{(N-1)(N-1)} \end{bmatrix}$$

$$\implies W = N \cdot \left(W^{-1}\right)^*$$

$$\implies \hat{D} = \begin{bmatrix} \frac{1}{N} & & & \\ & \frac{\alpha^{\frac{1}{N}}}{N} & & \\ & & \ddots & \\ & & & \frac{\alpha^{\frac{N-1}{N}}}{N} \end{bmatrix}$$

It may seem unusual that we are first defining the inverses of matrices, however this is because it allows for a closer link to Discrete Fourier transform functions. It should be noted that the factor of $\frac{1}{N}$is used due to convention. Hence multiplication Vx is the equivalent to applying an inverse discrete Fourier transform to x and then left multiplication by the diagonal matrix $\hat{D}^{-1}$, that is $\hat{D}^{-1}x$. Similarly left multiplication by V^{-1}, that is $V^{-1}x = W\hat{D}$ is equivalent to left multiplication of x by the diagonal matrix $\hat{D}$ then applying the discrete Fourier transform to $\hat{D}x$.

Hence

$$(V \otimes I) = \left(\hat{D}^{-1}W^{-1} \otimes I\right)$$
$$\left(V^{-1} \otimes I\right) = \left(W\hat{D} \otimes I\right)$$

I shall generalise this simplification by considering the equation $(X \otimes I)\,U$, where X is an arbitrary matrix of the same dimensions as V, this will then allow me to simplify $(V \otimes I)\,U$ and $\left(V^{-1} \otimes I\right)$.

Consider the i^{th} block of rows (that is $2M$ rows) of $(X \otimes I)\,U$.

Let:

$$y_j \coloneqq \begin{bmatrix} p_j \\ q_j \end{bmatrix} \in \mathbb{C}^{2M}$$

$$\begin{aligned} \begin{bmatrix} x_{i,1}I & x_{i,2}I & \cdots & x_{i,N}I \end{bmatrix} U &= \begin{bmatrix} x_{i,1}I & x_{i,2}I & \cdots & x_{i,N}I \end{bmatrix} \begin{bmatrix} y_1 \\ \vdots \\ y_N \end{bmatrix} \\ &= x_{i,1}y_1 + \cdots + x_{i,N}y_N \\ &= x_{i,j}y_j \end{aligned}$$

Here I thought this looks similar to matrix vector multiplication if only the dimensions were more amiable. Further in the question it hints at the use of reshaping to solve this problem, hence I transposed y_j, so $y_j^T \in \mathbb{C}^{1\times 2M}$

$$\begin{aligned} &= x_{i,j}y_j^T \\ &= \begin{bmatrix} x_{i,1} & x_{i,2} & \cdots & x_{i,N} \end{bmatrix} \begin{bmatrix} y_1^T \\ y_2^T \\ \vdots \\ y_N^T \end{bmatrix} \\ &= \begin{bmatrix} x_{i,1} & x_{i,2} & \cdots & x_{i,N} \end{bmatrix} \begin{bmatrix} p_1^T & q_1^T \\ p_2^T & q_2^T \\ \vdots & \vdots \\ p_N^T & q_N^T \end{bmatrix} \end{aligned}$$

Informally we have taken each block of $2M$ rows of U, transposed them and taken them to be the first row of our new U, lets call it $\overline{U} \coloneqq \begin{bmatrix} p_1^T & q_1^T \\ p_2^T & q_2^T \\ \vdots & \vdots \\ p_N^T & q_N^T \end{bmatrix}$. We have found that this means that $(X \otimes I)\,U$ is roughly equivalent to $X\overline{U}$, roughly since further reshaping is required to make them identical.

$$\begin{gathered} \text{Reshape } \left(X\overline{U}\right) \text{ to a } 2MN \text{ vector, retaining the order of the rows.} \\ = \\ (X \otimes I)\,U \end{gathered} \tag{16}$$

- Substituting X for V we see $(V \otimes I)\,U \equiv$ Reshape $\left(V\overline{U}\right)$ to a $2MN$ vector, retaining the order of the rows.
- Similarly, substituting X for V^{-1} we see $\left(V^{-1} \otimes I\right)U \equiv$ Reshape $\left(V^{-1}\hat{U}\right)$ to a $2MN$ vector, retaining the order of the rows.

Hence overall:

1. $(V \otimes I)\,U$ can be simplified to:
   (a) Form $\overline{U}$ by reshaping U such that the i^{th} row is $\begin{bmatrix} p_i^T & q_i^T \end{bmatrix}$

(b) Compute $V\overline{U} = \hat{D}^{-1}W^{-1}\overline{U}$ by applying the inverse Fourier transform to $\overline{U}$ and then multiplying this on the left by the diagonal matrix $\hat{D}^{-1}$

(c) Reshape (row wise) the previously calculated $V\overline{U}$ such that it returns to a $2MN$ length vector.

2. Similarly, $\left(V^{-1} \otimes I\right) U$ can be simplified to:

(a) Form $\overline{U}$ by reshaping U such that the i^{th} row is $\begin{bmatrix} p_i^T & q_i^T \end{bmatrix}$

(b) Compute $V^{-1}\overline{U} = W\hat{D}\overline{U}$ by first multiplying $\overline{U}$ on the left by the diagonal matrix $\hat{D}$ then applying the Fourier transform to this product.

(c) Reshape (row wise) the previously calculated $V^{-1}\overline{U}$ such that it returns to a $2MN$ length vector.

(e)

We aim to solve:

$$\left(C_1^{(\alpha)} \otimes I + C_2^{(\alpha)} \otimes B\right) U^{k+1} = R$$

NB. Theorem 5.1 $\implies \left(X \otimes I\right)^{-1} = \left(X^{-1} \otimes I\right)$

$$\begin{aligned}
\left(C_1^{(\alpha)} \otimes I + C_2^{(\alpha)} \otimes B\right) U^{k+1} &= R \\
\left(V \otimes I\right)\left(D_1 \otimes I + D_2 \otimes B\right)\left(V^{-1} \otimes I\right) U^{k+1} &= R \\
\implies \left(D_1 \otimes I + D_2 \otimes B\right)\left(V^{-1} \otimes I\right) U^{k+1} &= \left(V^{-1} \otimes I\right) R \qquad (17) \\
\implies \left(V^{-1} \otimes I\right) U^{k+1} &= \left(D_1 \otimes I + D_2 \otimes B\right)^{-1}\left(V^{-1} \otimes I\right) R \qquad (18) \\
\implies U^{k+1} &= \left(V \otimes I\right)\left(D_1 \otimes I + D_2 \otimes B\right)^{-1}\left(V^{-1} \otimes I\right) R \qquad (19)
\end{aligned}$$

I shall now formulate the steps i-iii as they are in the question

- Step i, we can see this refers to Equation 17, that is finding $\hat{R} = \left(V^{-1} \otimes I\right) R$ this is done using the method as in Item 1, explained in part d.
- Step ii, consider Equation 17, that is calculating $\hat{U} = \left(D_1 \otimes I + D_2 \otimes B\right)^{-1} \hat{R}$. This equivalent to solving $\left(D_1 \otimes I + D_2 \otimes B\right) x = \hat{R}$ for $x \in \mathbb{C}^{2MN}$ Since D_1 and D_2 are diagonal matrices, $D_1 \otimes I$ and $D_2 \otimes B$ are block diagonal matrices.

  Hence considering the kth block of rows (that is $2M$ rows), this problem reduces, for $d_{1,k} \coloneqq D_{1,(k,k)}$, $d_{2,k} \coloneqq D_{2,(k,k)}$ to:

$$\left(d_{1,k} I + d_{2,k} B\right) \begin{bmatrix} \hat{p}_k \\ \hat{q}_k \end{bmatrix} = \hat{r}$$

  We can further simplify this

$$\begin{aligned}
\left(d_{1,k} I + d_{2,k} \begin{bmatrix} 0 & B_{1,2} \\ B_{2,1} & 0 \end{bmatrix}\right) \begin{bmatrix} \hat{p}_k \\ \hat{q}_k \end{bmatrix} &= \hat{r} \\
d_{1,k} \begin{bmatrix} \hat{p}_k \\ \hat{q}_k \end{bmatrix} + d_{2,k} \begin{bmatrix} B_{1,2}\hat{q}_k \\ B_{2,1}\hat{p}_k \end{bmatrix} &= \hat{r}
\end{aligned}$$

  Considering the rows separately: For $\hat{r} \coloneqq \begin{bmatrix} \hat{r}_1 \\ \hat{r}_2 \end{bmatrix}$

  – First Row

$$\begin{aligned}
d_{1,k}\hat{p}_k + d_{2,k}B_{1,2}\hat{q}_k &= \hat{r}_1 \\
\implies \hat{p}_k &= \frac{\hat{r}_1 - d_{2,k}B_{1,2}\hat{q}_k}{d_{1,k}} \qquad (20)
\end{aligned}$$

– Second Row

$$d_{1,k}\hat{q}_k + d_{2,k}B_{2,1}\hat{p}_k = \hat{r}_2$$

Substituting in 20 for $\hat{p}_k$ gives:

$$\begin{aligned}
&d_{1,k}\hat{q}_k + d_{2,k}B_{2,1}\left(\frac{\hat{r}_1 - d_{2,k}B_{1,2}\hat{q}_k}{d_{1,k}}\right) = \hat{r}_2 \\
\implies &d_{1,k}\hat{q}_k - \frac{d_{2,k}^2}{d_{1,k}}B_{2,1}B_{1,2}\hat{q}_k = \hat{r}_2 - \frac{d_{2,k}}{d_{1,k}}B_{2,1}\hat{r}_1 \\
\implies &d_{1,k}\hat{q}_k - \frac{-\Delta t \cdot d_{2,k}^2}{d_{1,k}}B_{2,1}\hat{q}_k = \hat{r}_2 - \frac{d_{2,k}}{d_{1,k}}B_{2,1}\hat{r}_1 \\
\implies &\left(d_{1,k}I + \frac{\Delta t \cdot d_{2,k}^2}{d_{1,k}}B_{2,1}\right)\hat{q}_k = \hat{r}_2 - \frac{d_{2,k}}{d_{1,k}}B_{2,1}\hat{r}_1 \\
\implies &\left[\left(\frac{d_{1,k}}{d_{2,k}}\right)^2 I + \Delta t \cdot B_{2,1}\right]\hat{q}_k = \frac{d_{1,k}}{d_{2,k}^2}\hat{r}_2 - \frac{1}{d_{2,k}}B_{2,1}\hat{r}_1
\end{aligned}$$

We can notice that $\left[\left(\frac{d_{1,k}}{d_{2,k}}\right)^2 I + \Delta t \cdot B_{2,1}\right]$ is almost tridiagonal, that is of identical form to 8, this is because I is of this form, and so is $B_{2,1}$, therefore addition and scalar multiplication do not disrupt this form. It is useful to notice that this matrix has only real values with the exception of those on its diagonal. It has element

$$c \coloneqq \left(\frac{d_{1,k}}{d_{2,k}}\right)^2 + \Delta t \cdot \frac{2\Delta t}{\Delta x^2} = \left(\frac{d_{1,k}}{d_{2,k}}\right)^2 + \cdot\frac{2\Delta t^2}{\Delta x^2}$$

on its diagonal, and element:

$$d \coloneqq \Delta t \cdot -\frac{\Delta t}{\Delta x^2} = -\left(\frac{\Delta t}{\Delta x}\right)^2$$

for its sub/super diagonals.

We can therefore solve this equation similarly to as we did with the analogous problem in question 2.

$\hat{q}_k$ can then be substituted into equation 20 to give $\hat{p}_k$

- Step iii, we can see this refers to Equation 19, that is finding $U = (V \otimes I)\,\hat{U}$ Using reshaping as in Equation 16, we can see $U \equiv V^{-1}\overline{U}$, with of course further reshaping required (as in 16) to give an equality. [1]

(f)

The problem has been implemented in `q5.py`, as function `solve_wave_equ5`. I have applied it to the same initial condition functions as in Question 2 to allow for comparison. Since the algorithm only converges to a solution, termination is given either by the argument `maxit` or once the largest element wise difference between U_k and U_{k+1} is less than `tol`. The observations and graphs are near identical to those from Question 2 suggestion that the method has been implemented correctly.

In Figures 15, 16, 17, I have applied the above algorithm to some examples.

- Figure 15, For `M=100, tsteps=500, dt=0.002, interval=5`, as noted in question to we shall use: $u_0 = 0$, $u_1 = 2\pi sin(2\pi x)$. We can see that as time increases the wave appears to be oscillating, as we would expect for these initial conditions. Visually, the solutions at various time-steps appear to satisfy the boundary conditions, further providing evidence that `solve_wave_equ5` has been implemented successfully.
- Figure 16, For `M=100, tsteps=500, dt=0.002, interval=5`, as noted in question to we shall use:$u_0 = sin(2\pi x)$, $u_1 = 0$. We can see that as time increases the wave appears to be oscillating, as we would expect for these initial conditions. Visually, the solutions at various time-steps appear to satisfy the boundary conditions, further providing evidence that `solve_wave_equ5` has been implemented successfully.

- Figure 17, For `M=100, tsteps=500, dt=0.001, interval=5`, as noted in question to we shall use: $u_0 = sin(2\pi x)$, $u_1 = 2\pi cos(2\pi x)$. We can see that as time increases the wave appears to be travelling left, as we would expect for these initial conditions. Visually, the solutions at various time-steps appear to satisfy the boundary conditions, further providing evidence that `solve_wave_equ5` has been implemented successfully.

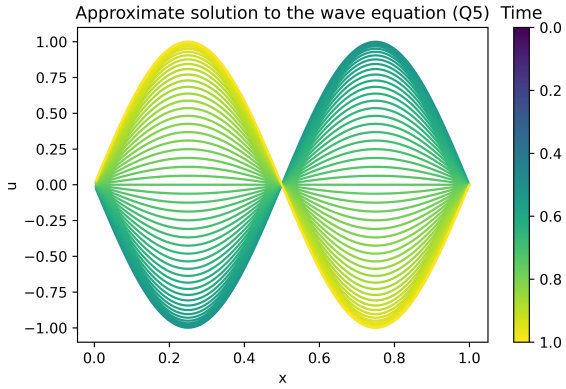

Figure 15: $u_0 = 0$, $u_1 = 2\pi sin(2\pi x)$

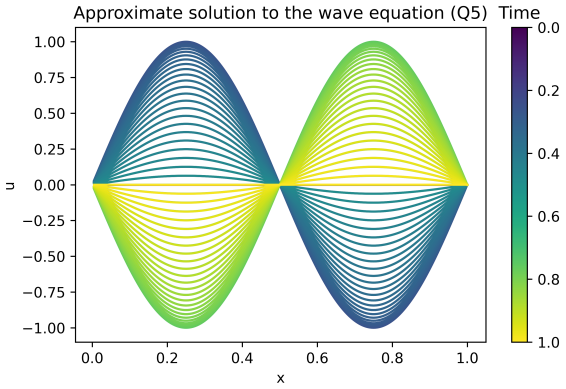

Figure 16: $u_0 = sin(2\pi x)$, $u_1 = 0$

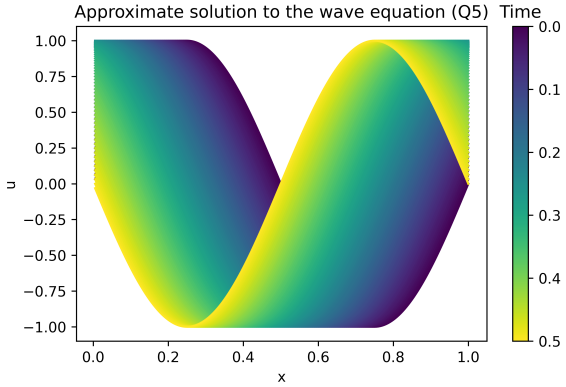

Figure 17: $u_0 = sin(2\pi x)$, $u_1 = 2\pi cos(2\pi x)$

All these figures can be seen to be almost identical to the corresponding ones in Question 2. To further test this function, in `tests_q5.py` I have compared the solution found by `solve_wave_equ5`, with initial conditions for which the solutions can easily calculated analytically to the analytical solution (evaluated at the corresponding x and t), the solution given by `solve_wave_equ5` passes these tests leading me to believe that it provides a valid approximation to the solution to the wave equation.

References

[1] C Cotter. Computational linear algebra notes, 2020.

[2] H. V. Henderson and S. R. Searle. On deriving the inverse of a sum of matrices. *SIAM Review*, 23(1):53–60, 1981.

[3] TEH (https://physics.stackexchange.com/users/234259/teh). Initial conditions for wave equation. Physics Stack Exchange. URL:https://physics.stackexchange.com/q/486595 (version: 2019-06-18).

[4] Wikipedia contributors. Dft matrix — Wikipedia, the free encyclopedia, 2020. [Online; accessed 15-January-2021].

[5] Wikipedia contributors. Finite difference — Wikipedia, the free encyclopedia, 2020. [Online; accessed 15-January-2021].