M4N9 Coursework 2

Azmat Habibullah

December 2020

Question 1

- remove test file from cla-utils

Part a

With the matrix $A \in \mathbb{R}^{m\times m}$ defined as

$$A = \begin{pmatrix} c & d & 0 & \dots & 0 & 0 & 0 \\ d & c & d & \dots & 0 & 0 & 0 \\ 0 & d & c & \dots & 0 & 0 & 0 \\ 0 & 0 & d & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & c & d & 0 \\ 0 & 0 & 0 & \dots & d & c & d \\ 0 & 0 & 0 & \dots & 0 & d & c \end{pmatrix}$$

we compute the LU factorisation of A to solve the system $Ax = b$. There are four main steps:

1. Perform successive lower triangular transformations L_k that clear out the subdiagonal of the k-th column of A in each step, obtaining $A_k = L_k L_{k-1} ... L_1 A$, eventually obtaining an upper triangular matrix U.
2. Noting the L_k are invertible, as proved in lectures, write $A = L_1^{-1} ... L_k^{-1} U = LU$, with $L := L_1^{-1} ... L_k^{-1}$.
3. Since $Ax = LUx = b$, with U upper triangular and L lower triangular as proved in lectures, write $Ux = y$ and solve $Ly = b$ by forwards substitution.
4. Solve $Ux = y$ by backwards substitution.

We first find the form of the L_k. Write the k-th column of A_k as x_k and define $L_k = I_m - \frac{d}{x_{k,k}} e_{k+1} e_k^*$.

$$L_k x_k = x_k - \frac{d}{x_{k,k}} e_{k+1} e_k^* x_k = x_k - \frac{d}{x_{k,k}} e_{k+1} x_{k,k} = x_k - d e_{k+1}$$

which means that multiplying A by L_k for each $k = 1, ..., M-1$ clears the superdiagonal terms, meaning the result is lower triangular. With $L = (L_n \dots L_2 L_1)^{-1}$, we have $L^{-1}A = U$, so that $A = LU$.

To solve the system $Ax = b$, we proceed by writing $A = LU$, so that $Ax = b \implies LUx = b$. Defining $y = Ux$, we can solve the system $Ly = b$ by forward substitution, and then solve the system $y = Ux$ by forwards substitution to solve the original problem $Ax = b$.

To find the form of L, we first find L_k^{-1}. We claim $L_k^{-1} = I_m + \frac{d}{x_{k,k}} e_{k+1} e_k^*$. Indeed,

$$L_k L_k^{-1} = \left(I_m - \frac{d}{x_{k,k}} e_{k+1} e_k^*\right)\left(I_m + \frac{d}{x_{k,k}} e_{k+1} e_k^*\right) = I_m + \frac{d}{x_{k,k}} e_{k+1} e_k^* - \frac{d}{x_{k,k}} e_{k+1} e_k^* - \frac{d^2}{{x_{k,k}}^2} e_{k+1} e_k^* e_{k+1} e_k^* = I_m$$

as $e_k^* e_{k+1} = 0$.

We claim that $L_1^{-1}L_2^{-1}\dots L_n^{-1} = I_m + d\sum_{i=1}^{n} \frac{e_{i+1}e_i^*}{x_{i,i}}$. For $n = 1$ this is clear. Inductively,

$$\begin{aligned} L_1^{-1}L_2^{-1}\dots L_{n+1}^{-1} &= \left(I_m + d\sum_{i=1}^{n} \frac{e_{i+1}e_i^*}{x_{i,i}}\right)\left(I_m + \frac{d}{x_{n+1,n+1}} e_{n+2}e_{n+1}^*\right) \\ &= I_m + d\sum_{i=1}^{n} \frac{e_{i+1}e_i^*}{x_{i,i}} + \frac{d}{x_{n+1,n+1}} e_{n+2}e_{n+1}^* \\ &= I_m + d\sum_{i=1}^{n+1} \frac{e_{i+1}e_i^*}{x_{i,i}} \end{aligned}$$

so the claim is proven. Thus, it follows that

$$L = L_1^{-1}L_2^{-1}\dots L_m^{-1} = I_m + d\sum_{i=1}^{m} \frac{e_{i+1}e_i^*}{x_{i,i}}$$

from which it follows immediately that L only has entries on its diagonal and subdiagonal. The diagonal entries are 1 and the subdiagonals are determined by the diagonal elements of the transformed A. The multiplications by L will only affect the diagonal entries, so the superdiagonal of U is given by d.

The full algorithm for LU factorisation and solution is thus as follows, with I_1 the matrix of ones on the superdiagonal.

$U \leftarrow cI + dI_1$
$L \leftarrow I$
for $i = 2$ to m **do**
  $l_{i,i-1} \leftarrow d/u_{i-1,i-1}$
  $u_{i,i} \leftarrow u_{i,i} - dl_{i,i-1}$
end for
$y_1 \leftarrow b_1$
for $i = 2$ to m **do**
  $y_i \leftarrow b_i - l_{i,i-1}y_{i-1}$
end for
$x_m \leftarrow y_m/U_{k,k}$
for $i = m - 1$ to 1 BACKWARDS **do**
  $x_i \leftarrow (y_i - dx_{i+1)}/u_{i,i}$
end for

Part b

Once we compute $l_{2,1}$, we can solve for y_2, since we know the first column of L gives us the coefficients we require. We can continue with forwards substitution, solving for y in one forwards sweep, and then solve for x via backwards substitution. We present the full algorithm below:

$U \leftarrow cI + dI_1$
$L \leftarrow I$
$y_1 = b_1$
for $i = 2$ to m **do**
  $l_{i,i-1} \leftarrow d/u_{i-1,i-1}$
  $u_{i,i} \leftarrow u_{i,i} - dl_{i,i-1}$
  $y_i \leftarrow b_i - l_{i,i-1}y_{i-1}$
end for
$x_m \leftarrow y_m/u_{k,k}$
for $i = m - 1$ to 1 BACKWARDS **do**

$x_i \leftarrow (y_i - dx_{i+1)}/u_{i,i}$
end for

Part c

In the first loop, the first line yields one FLOP (division); the second yields 2 FLOPs (multiplication and subtraction) and the last yields 2 FLOPs (multiplication and subtraction). Following this we have another FLOP for division outside of a loop. In the second loop, we have 3 FLOPs (multiplication, subtraction and division). Thus in total we have

$$N_{FLOPS} = \sum_{i=2}^{m} 5 + \sum_{i=1}^{m-1} 3 + 1 = 5(m-2) + 1 + 3(m-1) = 8m - 12 = O(m)$$

This matches results in the course that for Gaussian elimination, for matrices upper bandwidth p and lower bandwidth q we have an operation count $O(mpq)$, as well as the analagous results for forwards and backwards substitution, where $p = 1$ and $q = 1$. This contrasts with the general result for LU factorisation, which is $O(m^3)$ for Gaussian elimination and $O(m^2)$ for backwards and forwards substitution, by results from the course.

Part d

We notice we only need to compute the diagonal of U and the subdiagonal of L so our implementation can be done with vectors holding these entries of L and U respectiveyly. The implementation is contained in the file `q1.py`, with tests in the file `q1_test.py`. The implementation solves a system of equations $Ax_i = b_i$ in one iteration forwards and one iteration backwards, as this is more efficient than invoking an algorithm which solves just one system $Ax = b$ multiple times - there are wasteful calculations reconstructing the same L and U.

`test_solve_tridiag_LU` verifies that our tridiagonal matrix generator is working correctly by checking the shape and the relevant entries.

`test_solve_tridiag_LU` verifies the solution we get from the implementation solves the system for multiple equations.

Question 2

Part a

With

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

we differentiate the right hand equation twice with respect to x to see

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

which, upon substituting into the left hand equation, we see

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

which simplifies to

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

$$w^{n+1} - w^n - \Delta t \cdot u_{xx}^n - \frac{(\Delta t)^2}{4}(w_{xx}^n + w_{xx}^{n+1}) = 0$$

so finally

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

so that $C = \frac{(\Delta t)^2}{4}$ and $f = w^n + \Delta t \cdot u_{xx}^n + \frac{(\Delta t)^2}{4} \cdot w_{xx}^n$.
Now under the central difference formula we have

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

so we substitute back to see

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

so that $C_1 = \frac{(\Delta t)^2}{4(\Delta x)^2}$ and $f_i = w_i^n + \frac{\Delta t}{(\Delta x)^2} \cdot (u_{i+1}^n - 2u_i^n + u_{i-1}^n) + \frac{(\Delta t)^2}{4(\Delta x)^2} \cdot (w_{i+1}^n - 2w_i^n + w_{i-1}^n)$.

Part b

This system may be equivalently written as

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

Part c

We note that since $C_1 > 0$, the top right and bottom left entries of A are always non zero. The LU factorisation does not change these values until the very last step, where they become 0. Thus, at no stage in the process does the matrix A become banded, and we expect the ratios of the successive top right and bottom left entries to be 1, until the final stage where it drops to 1. Thus there is no advantage to using a banded matrix algorithm since A is never banded.

Figure 1 confirm this behaviour for random A. The function `check_extrema_constant` produces these plots and asserts our claim.

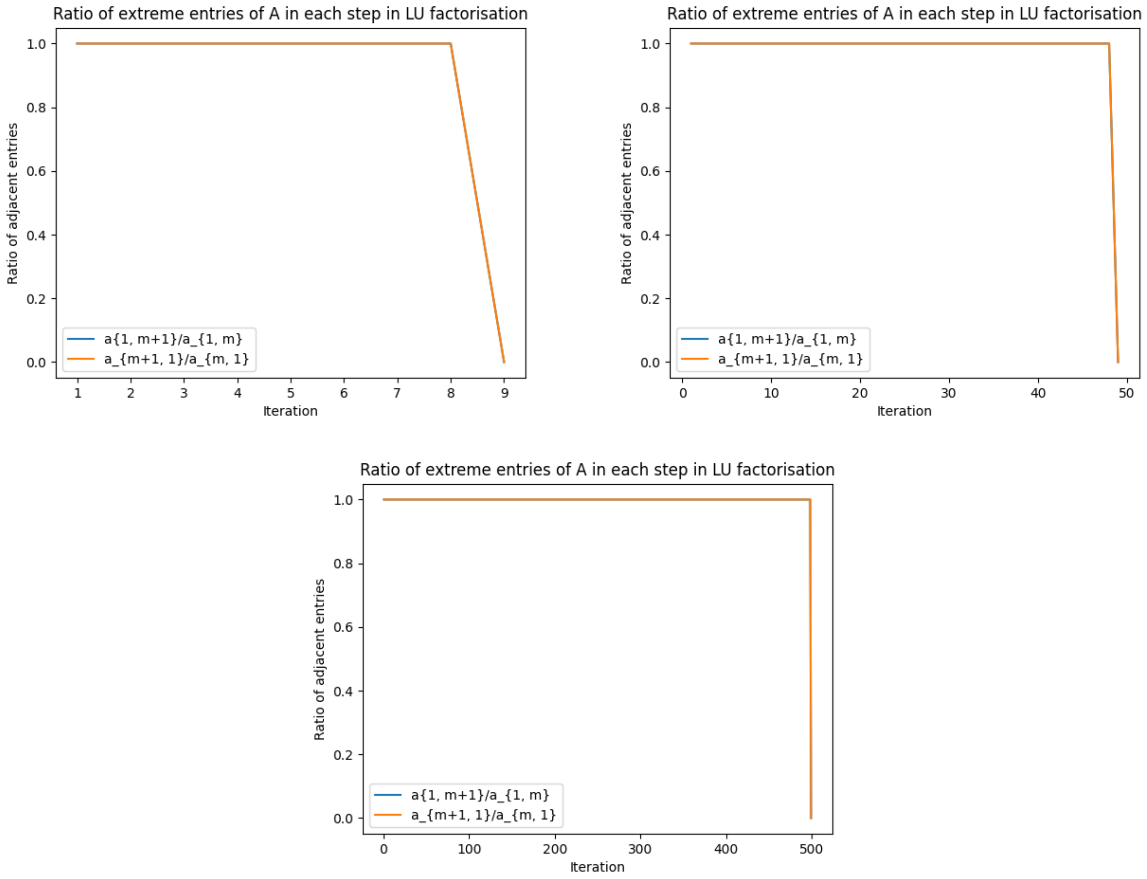

Figure 1: Experiments showing ratio of changes to top right and bottom left entries for $m = 10$ (top left), $m = 50$ (top right) and $m = 500$ (bottom middle)

Part d

With $u_1 = (1, 0, \ldots, 0, 0)^T, u_2 = (0, 0, \ldots, 0, 1)^T, v_1 = (0, 0, \ldots, 0, d)^T, v_2 = (d, 0 \ldots, 0, 0)^T \in \mathbb{R}^M$ we write $\alpha = \frac{1}{1+v_1^T T^{-1} u_1}$. Then by repeated applications of the Sherman-Morrison formula [1] we find

$$A^{-1} = T^{-1} - \alpha T^{-1} u_1 v_1^T T^{-1} - \frac{(T^{-1} - \alpha T^{-1} u_1 v_1^T T^{-1}) u_2 v_2^T (T^{-1} - \alpha T^{-1} u_1 v_1^T T^{-1})}{1 + v_2^T (T^{-1} - \alpha T^{-1} u_1 v_1^T T^{-1}) u_2}$$

whence to solve $Ax = b$, ie to retrive $x = A^{-1} b$, we form

$$x = \left(T^{-1} - \alpha T^{-1}u_1 v_1^T T^{-1} - \frac{(T^{-1} - \alpha T^{-1}u_1 v_1^T T^{-1})u_2 v_2^T (T^{-1} - \alpha T^{-1}u_1 v_1^T T^{-1})}{1 + v_2^T (T^{-1} - \alpha T^{-1}u_1 v_1^T T^{-1})u_2}\right) b$$
$$= T^{-1}b - \alpha T^{-1}u_1 v_1^T T^{-1}b - \frac{(T^{-1} - \alpha T^{-1}u_1 v_1^T T^{-1})u_2 v_2^T (T^{-1}b - \alpha T^{-1}u_1 v_1^T T^{-1}b)}{1 + v_2^T (T^{-1} - \alpha T^{-1}u_1 v_1^T T^{-1})u_2}$$

With the given forms of u_1, u_2, v_1, v_2 we can simplify this - this is done in the next part.

Part e

We can solve this system by solving the three equations $Tx = b, Tx = u_1, Tx = u_2$ and then computing the necessary matrix-vector products, which simplify.

$a \leftarrow T^{-1}b$
$\mu \leftarrow T^{-1}u_1$
$\nu \leftarrow T^{-1}u_2$
$\alpha \leftarrow 1/(1 + v_1^T \mu)$
$D \leftarrow 1 + v_2^T \nu - \alpha v_2^T \mu v_1^T \nu$
$x \leftarrow a - \alpha \mu v_1^T a - \frac{1}{D}\left(\nu v_2^T a - \alpha \nu v_2^T \mu v_1^T a - \alpha \mu v_1^T \nu v_2^T a + \alpha^2 \mu v_1^T \nu v_2^T \mu v_1^T a\right)$

where D is the denominator in the right hand side of the expression for x above. a, μ, ν are solved for without forming T^{-1} using the algorithm implemented in question 1. Using the fact that u_1, u_2, v_1, v_2 are unit vectors with form as described above, we can simplify this:

$a \leftarrow T^{-1}b$
$\mu \leftarrow T^{-1}u_1$
$\nu \leftarrow T^{-1}u_2$
$\alpha \leftarrow 1/(1 + d\mu_M)$
$D \leftarrow 1 + d\nu_1 - \alpha d^2 \mu_1 \nu_M$
$x \leftarrow a - \alpha d a_M \mu - \frac{1}{D}(d a_1 \nu - \alpha d^2 \mu_1 a_M \nu - \alpha d^2 a_1 \nu_M \mu + \alpha^2 d^3 \nu_M \mu_1 a_M \mu)$

Computing the solutions to $Tx = B$ is $O(m)$. The remaining calculations are all $O(1)$ as these are just scalar multiplications, so the overall algorithm is $O(m)$.

Part f

Figure 2 shows the noticeable speed improvement of this algorithm when compared to the generic `LU_inplace` function, as expected. From the plot we can see that the new implementation is roughly $O(m)$ and the LU method is roughly $O(m^3)$.

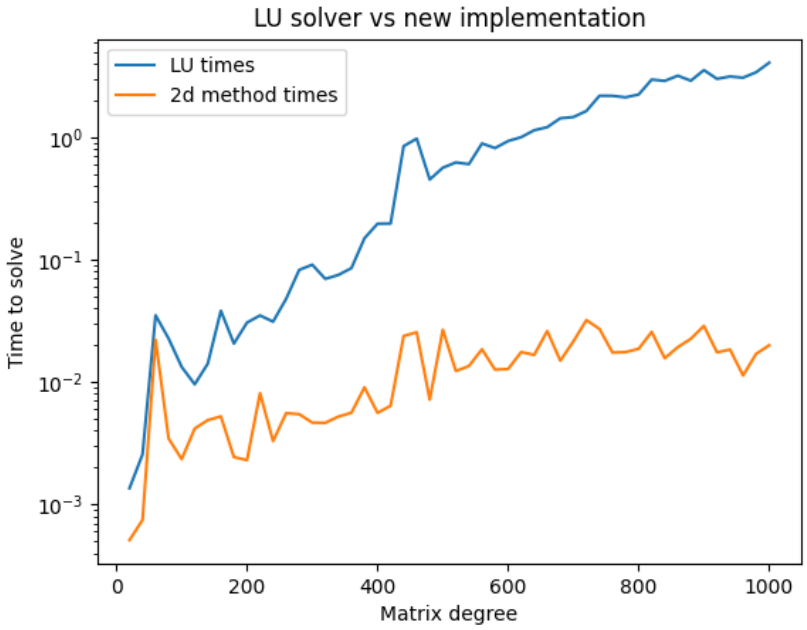

Figure 2: Comparison of time taken to solve $Ax = b$ with both implementations

The code for this question is in `q2.py`. Tests are in `q2_test.py`. In order to compare this algorithm to LU factorisation, the function `solve_LU` was written in this file too. Tests for this are also in the test file.

Part g

The code for this question is implemented in `q_2.py`. We test our wave equation solver with the initial conditions $u_0(x) = \sin()$ and $w_0(x) = 0$. The periodicity of u results in a Neumann boundary condition, as can be seen from the central difference formula:

$$u_x(0,t) = u_x(1,t) = \frac{u(x,0) - u(x,1)}{\Delta x} = 0$$

The function has the option to plot or save to disc, as explained in the documentation. The plot we see at specific timesteps is shown in Figure 3. This is clearly a travelling wave solution, with waves moving in opposite directions towards $x = 0$. This also matches the analytical solution for the wave equation with Neumann boundary conditions. Indeed, by results from M2AA2, the general solution with Neumann conditions for the wave equation on $0 \leq x \leq 1$ is

$$u(x,t) = A_0 + B_0 t + \sum_{n=1}^{\infty} (A_n \cos(n\pi t) + B_n \sin(n\pi t)) \cos(n\pi x)$$

A computation yields general solution

$$u(x,t) = 2 \sum_{n=2}^{\infty} \frac{\cos n\pi + 1}{\pi(1-n^2)} \cos(n\pi t) \cos(n\pi x) + \frac{2}{\pi}$$

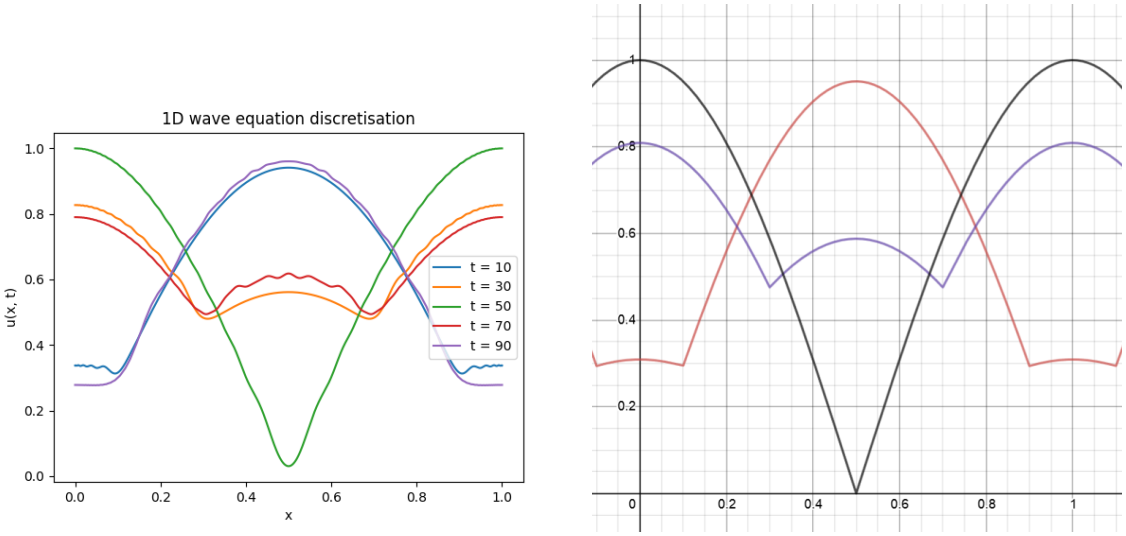

Figure 3: Output for given initial conditions with $M = 1000$, $\Delta t = 0.01$ (left) and same timesteps for analytical solution (right) - note that due to symmetry 2 of the timesteps have overlapping lines. The black line corresponds to $t = 50$, the purple line $t = 70, t = 30$ and the red line $t = 10, t = 90$.

Run `q_2.py` to see this plot. `q2_test.py` also tests this function.

Question 3

Part a

Suppose we have $A^{(n-1)}$ tridiagonal and symmetrical. Then $A^{(n-1)}_{ij} = 0$ for $i > j+1$. The QR factorisation $A^{(n-1)} = Q^{(n)}R^{(n)}$ here has $Q^{(n)}_{ij} = 0$ for $i > j+1$, since the algorithm orthogonalises with respect to previous columns, but these have 0 entries for $i > j+1$.

Now $Q^{(n)}$ is orthogonal, so $R^{(n)} = (Q^T)^{(n)}A^{(n-1)}$. Due to the zero entries below the subdiagonal of $Q^{(n)}$ we have $R^{(n)}_{ij} = 0$ for $j > i+2$. Now note that $A^{(n)} = R^{(n)}Q^{(n)} = (Q^T)^{(n)}A^{(n-1)}Q^{(n)}$ satisfies $(A^{(n)})^T = (Q^T)^{(n)}(A^{(n-1)})^T Q^{(n)} = A^{(n)}$, since $A^{(n-1)}$ is symmetric, so $A^{(n)}$ is symmetric. But we know that $A^{(n)}_{ij} = 0$ for $i > j+1$, since $Q^{(n)}$ satisfies this property. Thus, entries below the subdiagonal are 0, and by symmetry entries above the superdiagonal are 0. Thus we have that $A^{(n)}$ is also a symmetric tridiagonal matrix.

Part b

We first notice that if we were to use a full Householoder reflection v, only two components will be non-zero. This is because $v = \operatorname{sgn} x||x||e_1 + x$, with x the entries of A_k below the $k-1$th element. But since A is tridiagonal, this is only ever two elements, so we can just consider 2×2 matrices applied to the rows of A. We further see that the only entries that will be changed are the 2×3 submatrices of A.

By results from the course, the operation count of Householder generally is $2mn^2 - \frac{2n^3}{3}$. To calculate v_k we have 3 FLOPs to calculate the norm of x, followed by 2 multiplications and an addition, all for 2 components, so this is 14 FLOPs. The norm is 3 FLOPs, dividing is a 4th. m=k+2, n=k+3. By results from the course the flop count is

$$4\sum_{k=1}^{n} 2(3)(2) = 48n = O(n)$$

which is considerably more efficient than the general algorithm.

Part c

The code for this question is in `q3.py` and tests are in `q3_test.py`. `test_qr_factor_tri` verifies that this algorithm is working correctly by checking R is upper triangular and that $R^TR = A^TA$ as expected since Q is orthogonal. It also uses another function to form Q from the vectors v and checks that the constructed Q is corrected - Q is orthonormal, upper triangular and the QR factorisation is accurate. This function to construct Q is not used later; it is solely for testing the implementation of `qr_factor_tri`.

Part d

The code for this question is in `q3.py` and tests are in `q3_test.py`. `test_qr_alg_tri` verifies that this algorithm is working correctly by checking that the output is Hermitian and tridiagonal and also preserves trace.

From Figure 4 we see that $c = |T_{m,m-1}|$ decreases linearly on a log based plot, ie c decreases exponentially, and is below the tolerance after just 4 iterations.

To see this plot, as well as all outputs for the remainder of this question, run `q3.py`.

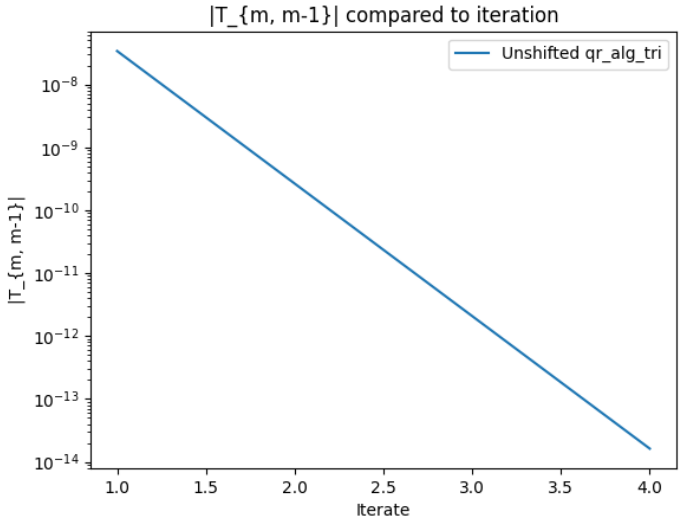

Figure 4: c from `qr_alg_tri` for given A

Part e

For A as defined in the question, Figure 5 verifies that `pure_qr` requires more iterations. The norm of $|T_{m,m-1}$ is significantly lower for `pure_qr` as we need the entire diagonal to have norm below the tolerance, so that we have convergence to a diagonal matrix. This results in the norm of $|T_{m,m-1}$ being significantly lower, and is thus wasteful. However, since the entire subdiagonal has a low norm, each entry will be low so concatenation will result in a jump to a norm which is still below the tolerance.

We also see that with a random symmetric matrix we get different results. In this case, `pure_qr` converges quicker than `qr_alg_tri`. This shows that our algorithm does not always perform better than the implementation in the course: there are improvements to be made, which we will do in the next subsection.

The code for this question is in `q3.py` and tests are in `q3_test.py`. `test_concatenate` verifies that this algorithm is working correctly by checking that the iterates for an $m \times m$ matrix have at least m values less than 10^{-12}, which is expected by the convergence criteria, for both `pure_qr` and `qr_alg_tri`.

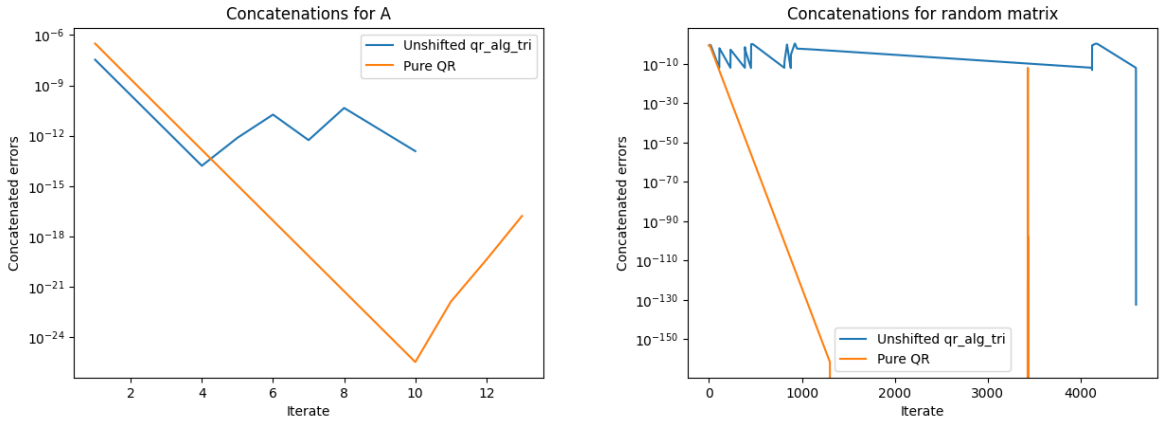

Figure 5: Pure QR and unshifted concatenation for A in the question (left) and random 12×12 matrix (right)

Part f

Using the Wilkinson shift, Figure 6 shows a significant improvement in number of timesteps required compared with both `pure_qr` and unshifted `qr_alg_tri`. The plots do not show the same pattern of spiking up, as with the unshifted `qr_alg_tri`. This is because the order of iterates is disrupted by the detaching.

This function was implemented by passing an additional parameter to `qr_alg_tri`. The test for this function checks the same criteria as for the unshifted algorithm.

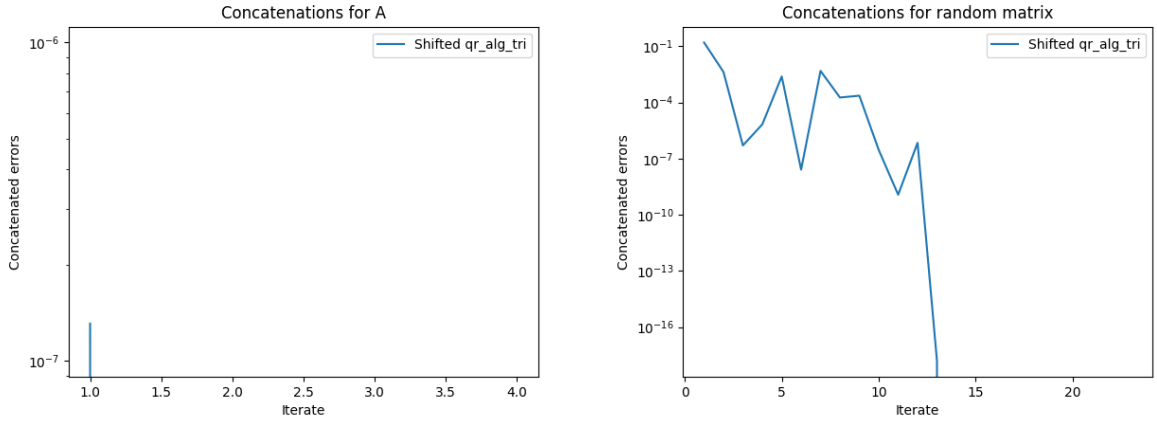

Figure 6: Wilkinson concatenation for A in the question (left) and random 12×12 matrix (right)

Part g

In Figure 7 we see that the shifted QR algorithm converges much quicker than the unshifted one.

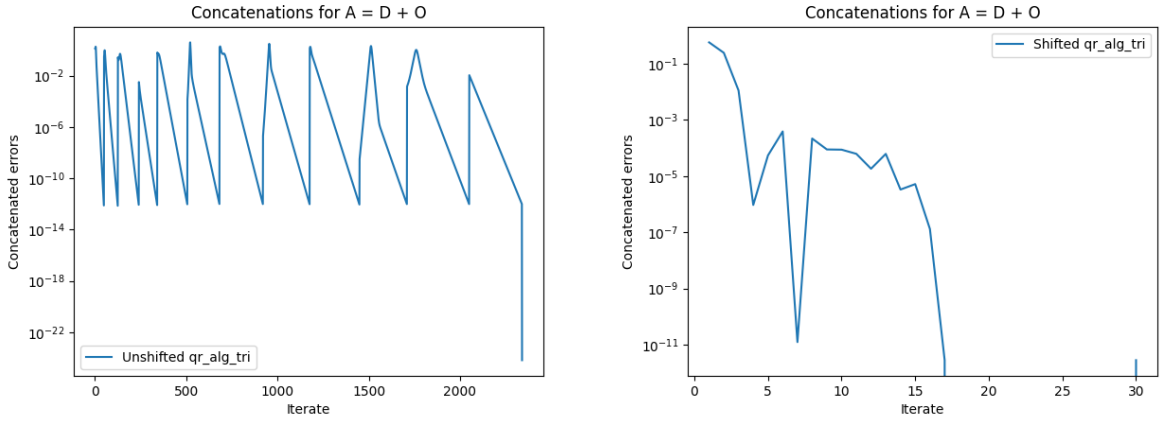

Figure 7: Shifted and unshifted QR algorithm for 15×15 matrix $A = D + O$

Question 4

Part a

The code for this question is in the `q4.py` file, as well as `exercises_10.py`. The tests are in the `q4_test.py` file. The tests check that the preconditioned GMRES algorithm returns solutions to the system $Ax = b$ for a given tolerance. To do this we have also built an upper triangular preconditioner, which has its own test to check that it is correctly solving upper triangular systems $Mx = b$.

Part b

Suppose $M^{-1}Ax = \lambda x$. Then $x - M^{-1}Ax = x - \lambda x \implies (I - M^{-1}A)x = (1 - \lambda)x$. Taking the operator 2-norm of both sides of this equality, we have $||(I - M^{-1}A)x|| = ||(1 - \lambda)x||$.

Since $||Ax|| \leq ||A|| \cdot ||x||$ for the operator 2-norm by results from the course, $||(I - M^{-1}A)x|| \leq ||(I - M^{-1}A)|| \cdot ||x|| = c||x||$. Also, by definition of the norm we have that $||(1 - \lambda)x|| = |1 - \lambda| \cdot ||x||$.

Thus, $|1 - \lambda| \cdot ||x|| \leq c||x||$, so $(|1 - \lambda| - c) \cdot ||x|| < 0$. We know the norm is non-negative definite, so for this inequality to hold we require $|1 - \lambda| - c \leq 0$, from which it follows that $|1 - \lambda| \leq c$.

Part c

By results from the course, GMRES will converge quickly if V is well-conditioned and $p(z)$ is small for all $z \in \Lambda(A)$, with the condition that $p(0) = 1$. We see that $p(z) = (1 - z)^n$ clearly satisfies $p(0) = 1$. Further, from part b we know that $|1 - \lambda| \leq c$ for $c < 1$, so for $x \in \Lambda(M^{-1}A)$ we have

$$|p(x)| = |(1 - x)^n| = |1 - x|^n \leq |1 - x^*|^n \leq c^n < 1$$

where x^* is the eigenvalue maximising the norm above. Indeed as $c < 1$ we have $c^n \to 0$ as $n \to \infty$ monotonically. It follows that $p(z) = (1 - z)^n$ is a good upper bound for the optimal polynomial as the residual decreases as the number of iterations is increased.

It follows that

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

with $x = ||1 - x^*|| < 1$ and $\kappa(V)$ the condition number. Thus we have exponential convergence, bounded by the number of iterations required.

Part d

We investigate 2 different graphs: a random 6x6 matrix with entries between 0 and 1, and a random 100x100 matrix with integer entries. These cases correspond to different eigenvalues of $M^{-1}A$ and thus different behaviours for GMRES.

From Figure 8 we see that in both cases, the preconditioned residual errors are lower than the non-preconditioned errors. The number of iterations required for convergence varies depending on the matrix. For some matrices, preconditioning in this form accelerates GMRES, whereas in others it doesn't. This is due to the fact that depending on the graph we investigate, $M^{-1}A$ will have a different number of clusters of eigenvalues. We see that preconditioning does not decrease performance, which matches what we expect, by results from the course. Figure 9 shows that there is a greater effect for the 100×100 matrix since the eigenvalues are compressed significantly, without having to use a large c. The preconditioning brings the eigenvalues a lot closer which results in preconditioned GMRES performing better. On the other hand, for the 6×6 matrix, the effect is not as large.

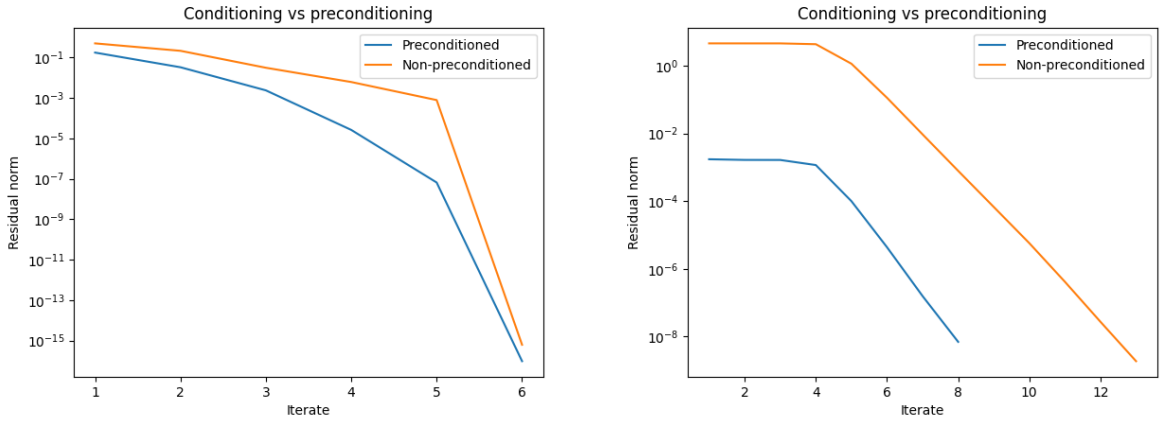

Figure 8: Examples of preconditioning vs conditioning. The left hand plot is a random 6×6 graph. The right hand side plot is a 100×100 random integer graph.

This matches the behaviour seen in Figure 10. The 100×100 matrix has many more of its eigenvalues clustered closely together, so preconditioning is more effective, and less iterates are required. On the other hand, the 6×6 matrix has its eigenvalues more spaced apart, so preconditioning does not speed up GMRES as much. However, the upper bound is still realised.

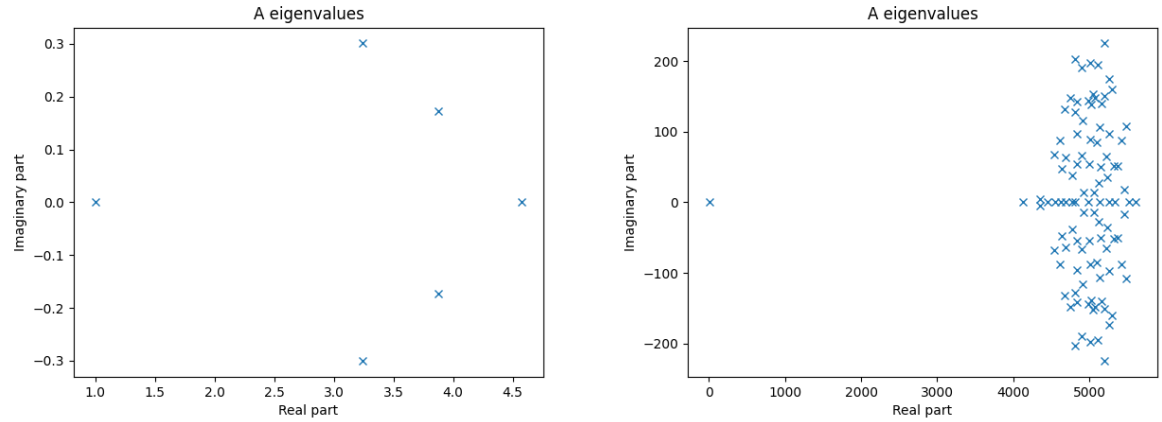

Figure 9: Eigenvalues of A corresponding to plots above

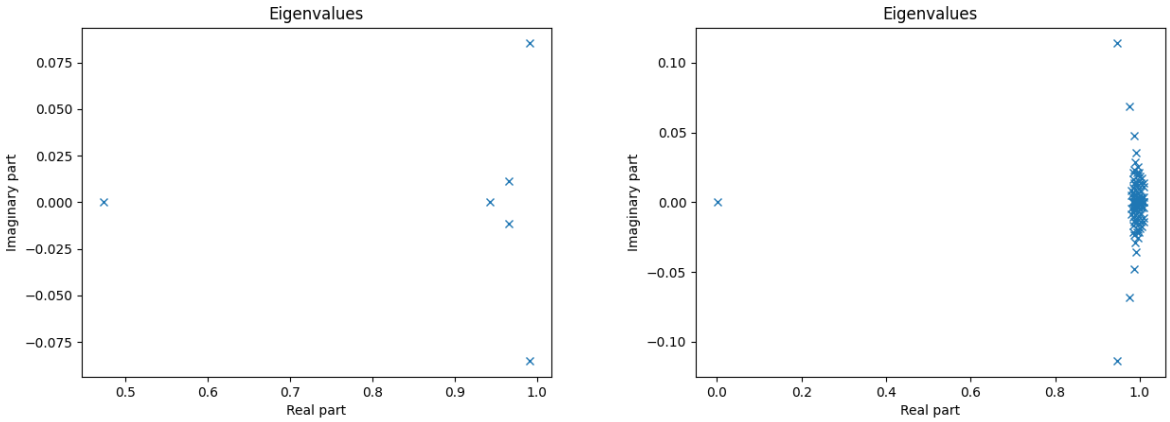

Figure 10: Eigenvalues of $M^{-1}A$ corresponding to plots above

Run `q4.py` to see these plots. The `best_c` function determines the minimum value c so that $M = cU$ satisfies the condition in part b, namely $||I - M^{-1}A|| < 1$. The range of values tested over are between 0 and 10, since these were experimentally deemed to cover the vast majority of cases for the optimal value of c.

Tests verifying the condition in part b is met are contained within the script in the function `run_investigation`. Further tests are implemented in `q4_test`.

Question 5

Part a

Defining

$$U = \begin{pmatrix} p_1 \\ q_1 \\ \vdots \\ p_N \\ q_N \end{pmatrix} \in \mathbb{R}^{2MN}, p_i = \begin{pmatrix} u_1^i \\ u_2^i \\ \vdots \\ u_M^i \end{pmatrix} \in \mathbb{R}^M, q_i = \begin{pmatrix} w_1^i \\ w_2^i \\ \vdots \\ w_M^i \end{pmatrix} \in \mathbb{R}^M$$

we can write the system in the question as

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

where

$$B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \in \mathbb{R}^{2M \times 2M}$$

with

$$B_{12} = \begin{pmatrix} -\Delta t & 0 & 0 & \dots & 0 \\ 0 & -\Delta t & 0 & \dots & 0 \\ 0 & 0 & -\Delta t & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \dots & 0 & 0 & -\Delta t \end{pmatrix} = -\Delta t \cdot I \in \mathbb{R}^{M \times M}, B_{21} = \frac{\Delta t}{(\Delta x)^2} \begin{pmatrix} 2 & -1 & 0 & \dots & -1 \\ -1 & 2 & -1 & \dots & 0 \\ 0 & -1 & 2 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -1 & \dots & 0 & 2 & -1 \end{pmatrix} \in \mathbb{R}^{M \times M}$$

and

$$r = \begin{pmatrix} u_1^0 + \frac{\Delta t}{2} w_1^0 \\ u_2^0 + \frac{\Delta t}{2} w_2^0 \\ \vdots \\ u_M^0 + \frac{\Delta t}{2} w_M^0 \\ w_1^0 + \frac{\Delta t}{2(\Delta x)^2}(u_M^0 - 2u_1^0 + u_2^0) \\ w_2^0 + \frac{\Delta t}{2(\Delta x)^2}(u_1^0 - 2u_2^0 + u_3^0) \\ \vdots \\ w_M^0 + \frac{\Delta t}{2(\Delta x)^2}(u_{M-1}^0 - 2u_M^0 + u_1^0) \end{pmatrix} \in \mathbb{R}^{2M}$$

where u_i^0 and w_i^0 are defined initial conditions, for $i = 1, \dots, M$.
With

$$C_1 = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ -1 & 1 & 0 & \dots & 0 \\ 0 & -1 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \dots & 0 & -1 & 1 \end{pmatrix} \text{ and } C_2 = \begin{pmatrix} \frac{1}{2} & 0 & 0 & \dots & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & \dots & 0 \\ 0 & \frac{1}{2} & \frac{1}{2} & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \dots & 0 & \frac{1}{2} & \frac{1}{2} \end{pmatrix}$$

we can write the system in the compact notation

$$(C_1 \otimes I + C_2 \otimes B)U = \begin{pmatrix} r \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

Part b

With

$$C_1^{(\alpha)} = \begin{pmatrix} 1 & 0 & 0 & \dots & -\alpha \\ -1 & 1 & 0 & \dots & 0 \\ 0 & -1 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \dots & 0 & -1 & 1 \end{pmatrix} \text{ and } C_2^{(\alpha)} = \begin{pmatrix} \frac{1}{2} & 0 & 0 & \dots & \frac{\alpha}{2} \\ \frac{1}{2} & \frac{1}{2} & 0 & \dots & 0 \\ 0 & \frac{1}{2} & \frac{1}{2} & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \dots & 0 & \frac{1}{2} & \frac{1}{2} \end{pmatrix} = \frac{1}{2} \begin{pmatrix} 1 & 0 & 0 & \dots & \alpha \\ 1 & 1 & 0 & \dots & 0 \\ 0 & 1 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \dots & 0 & 1 & 1 \end{pmatrix}$$

we consider the iterative scheme

$$(C_1^{(\alpha)} \otimes I + C_2^{(\alpha)} \otimes B)U^{k+1} = R = \begin{pmatrix} r + \alpha(-I + B/2) \begin{pmatrix} p_N^k \\ q_N^k \end{pmatrix} \\ 0 \\ \vdots \\ 0 \end{pmatrix}, U^k = \begin{pmatrix} p_1^k \\ q_1^k \\ \vdots \\ p_N^k \\ q_N^k \end{pmatrix} \in \mathbb{R}^{2MN}$$

If this scheme converges, we have $U^{k+1} = U^k$. C_1 differs from $C_1^{(\alpha)}$ by the top-right entry $(-\alpha)$ only, so $C_1 \otimes I$ differs from $C_1^{(\alpha)} \otimes I$ by the top right $2M \times 2M$ entries (-αI), which are multiplied by the bottom $2M$ entries of U^k. It follows that

$$(C_1^{(\alpha)} \otimes I)U^k = (C_1 \otimes I)U^k + \begin{pmatrix} -\alpha I \begin{pmatrix} p_N^k \\ q_N^k \end{pmatrix} \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

and similarly

$$(C_2^{(\alpha)} \otimes B)U^k = (C_2 \otimes B)U^k + \begin{pmatrix} \alpha B/2 \begin{pmatrix} p_N^k \\ q_N^k \end{pmatrix} \\ 0 \\ \vdots \\ 0 \end{pmatrix}.$$

But U^k solves the iterative system, so, by adding the two above expressions, and using $U^{k+1} = U^k$, we have

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

which implies that

$$(C_1 \otimes I + C_2 \otimes B)U^k = \begin{pmatrix} r \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

as required.

Part c

We now verify that

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

is an eigenvector of $C_1^{(\alpha)}$ and $C_2^{(\alpha)}$.
We have that

$$C_1^{(\alpha)} v_j = \begin{pmatrix} 1 & 0 & 0 & \dots & -\alpha \\ -1 & 1 & 0 & \dots & 0 \\ 0 & -1 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \dots & 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ \alpha^{-1/N} e^{\frac{2\pi ij}{N}} \\ \alpha^{-2/N} e^{\frac{4\pi ij}{N}} \\ \vdots \\ \alpha^{-(N-1)/N} e^{\frac{2(N-1)\pi ij}{N}} \end{pmatrix}$$

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

so that the eigenvalue for $C_1^{(\alpha)}$ is $1 - \alpha^{1/N} e^{\frac{2(N-1)\pi ij}{N}}$. Indeed, $(1 - \alpha^{1/N} e^{\frac{2(N-1)\pi ij}{N}})(\alpha^{-1/N} e^{\frac{2\pi ij}{N}}) = \alpha^{-1/N} e^{\frac{2\pi ij}{N}} - \alpha^0 e^{\frac{(2N-2+2)\pi ij}{N}} = \alpha^{-1/N} e^{\frac{2\pi ij}{N}} - e^{2\pi ij} = \alpha^{-1/N} e^{\frac{2\pi ij}{N}} - 1$ as expected.

Similarly,

$$C_2^{(\alpha)} v_j = \begin{pmatrix} \frac{1}{2} & 0 & 0 & \dots & \frac{\alpha}{2} \\ \frac{1}{2} & \frac{1}{2} & 0 & \dots & 0 \\ 0 & \frac{1}{2} & \frac{1}{2} & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \dots & 0 & \frac{1}{2} & \frac{1}{2} \end{pmatrix} \begin{pmatrix} 1 \\ \alpha^{-1/N} e^{\frac{2\pi ij}{N}} \\ \alpha^{-2/N} e^{\frac{4\pi ij}{N}} \\ \vdots \\ \alpha^{-(N-1)/N} e^{\frac{2(N-1)\pi ij}{N}} \end{pmatrix}$$

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

so that the eigenvalue for $C_2^{(\alpha)}$ is $\frac{1}{2}\left(1+\alpha^{1/N}e^{\frac{2(N-1)\pi ij}{N}}\right)$. Indeed, $\frac{1}{2}\left(1+\alpha^{1/N}e^{\frac{2(N-1)\pi ij}{N}}\right)\alpha^{-1/N}e^{\frac{2\pi ij}{N}} = \frac{1}{2}\left(\alpha^{-1/N}e^{\frac{2\pi ij}{N}}+1\right)$ as expected.

Since we have found N eigenvectors, we know that we have an eigenbasis, so we can diagonalise the system. We define the matrices V and D as

$$V = \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ \alpha^{-1/N} & \alpha^{-1/N}e^{\frac{2\pi i}{N}} & \alpha^{-1/N}e^{\frac{4\pi i}{N}} & \dots & \alpha^{-1/N}e^{\frac{2(N-1)\pi i}{N}} \\ \alpha^{-2/N} & \alpha^{-2/N}e^{\frac{4\pi i}{N}} & \alpha^{-2/N}e^{\frac{8\pi i}{N}} & \dots & \alpha^{-2/N}e^{\frac{4(N-1)\pi i}{N}} \\ \vdots & \vdots & \dots & \vdots & \vdots \\ \alpha^{-(N-1)/N} & \alpha^{-(N-1)/N}e^{\frac{2(N-1)\pi i}{N}} & \alpha^{-(N-1)/N}e^{\frac{4(N-1)\pi i}{N}} & \dots & \alpha^{-(N-1)/N}e^{\frac{2(N-1)^2\pi i}{N}} \end{pmatrix}$$

$$D_1 = \begin{pmatrix} 1-\alpha^{1/N} & 0 & 0 & \dots & 0 \\ 0 & 1-\alpha^{1/N}e^{\frac{2(N-1)\pi i}{N}} & 0 & \dots & 0 \\ 0 & 1 & 1-\alpha^{1/N}e^{\frac{4(N-1)\pi i}{N}} & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1-\alpha^{1/N}e^{\frac{2(N-1)^2)\pi i}{N}} \end{pmatrix}$$

$$D_2 = \frac{1}{2}\begin{pmatrix} 1+\alpha^{1/N} & 0 & 0 & \dots & 0 \\ 0 & 1+\alpha^{1/N}e^{\frac{2(N-1)\pi i}{N}} & 0 & \dots & 0 \\ 0 & 1 & 1+\alpha^{1/N}e^{\frac{4(N-1)\pi ij}{N}} & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1+\alpha^{1/N}e^{\frac{2(N-1)^2\pi i}{N}} \end{pmatrix}$$

Then we may write $C_1^{(\alpha)} = VD_1V^{-1}$ and $C_2^{(\alpha)} = VD_2V^{-1}$, and we have the result $C_1^{(\alpha)} \otimes I + C_2^{(\alpha)} \otimes B = (VD_1V^{-1}) \otimes I + (VD_2V^{-1}) \otimes B$.

The mixed product property of the tensor product states that $(A \otimes B)(C \otimes D) = AC \otimes BD$. Thus it follows that $(V \otimes I)(D_1 \otimes I)(V^{-1} \otimes I) = (VD_1 \otimes I)(V^{-1} \otimes I) = VD_1V^{-1} \otimes I$. Similarly, $V \otimes I)(D_2 \otimes B)(V^{-1} \otimes I) = (VD_2 \otimes B)(V^{-1} \otimes I) = VD_2V^{-1} \otimes B$. Thus we finally obtain

$$C_1^{(\alpha)} \otimes I + C_2^{(\alpha)} \otimes B = (VD_1V^{-1}) \otimes I + (VD_2V^{-1}) \otimes B = (V \otimes I)(D_1 \otimes I + D_2 \otimes B)(V^{-1} \otimes I)$$

Part d

Define the discrete Fourier transform of N complex numbers $x_0, \dots, x_{N-1}$ as $X_0, \dots, X_{n-1}$ as

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-\frac{2\pi kni}{N}}$$

and the inverse discrete Fourier transform as

$$x_n = \frac{1}{N}\sum_{k=0}^{N-1} X_k \cdot e^{\frac{2\pi kni}{N}}$$

Consider first $Vx = y$ where $x, y \in \mathbb{R}^N$. Then the kth element of y is $y_k = \alpha^{-(k-1)/N}X_k^*$. This corresponds to taking the inverse discrete Fourier transform (up to normalisation) of the elements of the vector x, then multiplying by $\alpha^{-(k-1)/N}$. Equivalently, we can consider the solution y as being obtained via first taking the inverse transform, and then multiplying the vector by a diagonal matrix =

$diag(\alpha^{-0/N}, \alpha^{-1/N}, \ldots, \alpha^{-(N-1)/N}) \in \mathbb{R}^{N\times N}$.

Consider now $(V \otimes I)U$, where $U \in \mathbb{R}^{2MN}$ as before. Instead of summing over all values of the vector U, since $I \in \mathbb{R}^{2M\times 2M}$, the Fourier transforms (up to normalisation) are taken over all elements spaced $2M$ apart in the vector U, ie all N iterates of u_i or w_i. Secondly, instead of each kth element being multiplied by $\alpha^{-(k-1)/N}$, we have each kth time slice being multiplied by $\alpha^{-(k-1)/N}$, which corresponds to multiplying each time slice by the diagonal matrix $\alpha^{-(k-1)/N} I \in \mathbb{R}^{2M}$.

In matrix-vector language, we may write this as

$$(V \otimes I)U = \begin{pmatrix} A_0 & 0 & \ldots & 0 \\ 0 & A_1 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & A_{N-1} \end{pmatrix} \begin{pmatrix} \mathcal{F}_0^{-1} \\ \mathcal{F}_1^{-1} \\ \vdots \\ \mathcal{F}_{N-1}^{-1} \end{pmatrix}$$

where $A_i = diag(\alpha^{\frac{-i}{N}})$ and $\mathcal{F}_i^{-1} \in \mathbb{C}^{2M}$ is the ith element of the discrete inverse Fourier transform of $(u_1^k, u_2^k, \ldots, u_M^k, w_1^k, w_2^k, \ldots, w_M^k)^T$.

The situation is analagous for $(V^{-1} \otimes I)U$ but we invert the operations and their order: multiply by the inverse of the existing diagonal matrices, then take the Fourier transform.

$$(V^{-1} \otimes I)U = \begin{pmatrix} \mathcal{F}_0' \\ \mathcal{F}_1' \\ \vdots \\ \mathcal{F}_{N-1}' \end{pmatrix}$$

where $\mathcal{F}_i'$ is the Fourier transform of $A_i^{-1}U_i$ (up to normalisation), with U_i the ith time slice of U and $A_i^{-1} = diag(\alpha^{\frac{i}{N}})$.

Part e

To solve

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

first notice that, from part c, we may rewrite this as

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

or equivalently, using the property of the tensor product that $A \otimes B$ is invertible if and only if A and B are both invertible, in which case $(A \otimes B)^{-1} = A^{-1} \otimes B^{-1}$,

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

So we first compute $\hat{R} = (V^{-1} \otimes I)R$. Then

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

Write $\hat{U} = (V^{-1} \otimes I)U$, so $U = (V \otimes I)\hat{U}$. Now

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

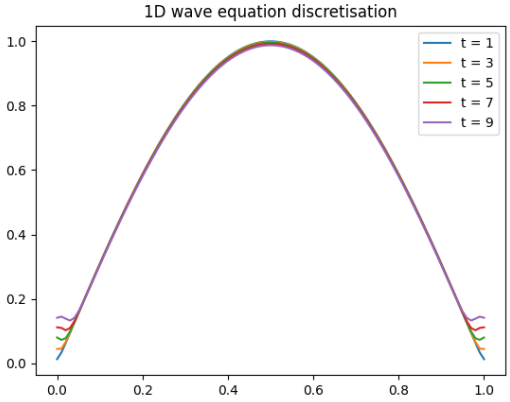

Figure 11: Plot of a few timesteps with initial conditions $u_0(x) = \sin(\pi x)$ as in question 2

Since D_1 and D_2 are diagonal, this system can be separated into its N separate time slices, where we simply need to solve

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

for $k = 1, \dots, N$. Once we have $\hat{U}$, we may simply solve $U = (V \otimes I)\hat{U}$.

Putting this all together, to solve the system, we first compute $\hat{R} = (V^{-1} \otimes I)R$, then solve $(d_{1,k}I + d_{2,k}B)\begin{pmatrix}\hat{p}_k\\ \hat{q}_k\end{pmatrix} = \hat{r}_k$, $k = 1, \dots, N$, and finally solve $U = (V \otimes I)\hat{U}$.

Part f

To eliminate $\hat{p}_k$ we note that

$$\begin{aligned}(d_{1,k}I_{2M} + d_{2,k}B)\begin{pmatrix}\hat{p}_k\\ \hat{q}_k\end{pmatrix} &= \left(d_{1,k}\begin{pmatrix}I_M & 0\\ 0 & I_M\end{pmatrix} + d_{2,k}\begin{pmatrix}0 & B_{12}\\ B_{21} & 0\end{pmatrix}\right)\begin{pmatrix}\hat{p}_k\\ \hat{q}_k\end{pmatrix} = \hat{r}_k = \begin{pmatrix}\hat{r}_{k1}\\ \hat{r}_{k2}\end{pmatrix}\\ &\implies d_{1,k}\hat{p}_k - \Delta t d_{2,k}\hat{q}_k = \hat{r}_{k1}\\ &\implies \hat{p}_k = \frac{\Delta t d_{2,k}\hat{q}_k + \hat{r}_{k1}}{d_{1,k}}\end{aligned}$$

using the first line of the system. Substituting this expression into the second line, we see

$$\begin{aligned}d_{2,k}B_{21}\left(\frac{\Delta t d_{2,k}\hat{q}_k + rk1}{d_{1,k}}\right) + d_{1,k}\hat{q}_k &= r_{k2}\\ \implies d_{2,k}B_{21}\left(\Delta t d_{2,k}\hat{q}_k + rk1\right) + d_{1,k}^2\hat{q}_k &= d_{1,k}r_{k2}\\ \implies \left(d_{2,k}^2\Delta t B_{21} + d_{1,k}^2 I\right)\hat{q}_k &= d_{1,k}r_{k2} - d_{2,k}B_{21}r_{k1}\end{aligned}$$

This is of the same form as that in question 2, so we can use the methods from there to solve this system. We then substitute back to find $\hat{p}_k$ for each time slice before computing U. Figure 11 shows the solution matches what we expect from the theory in question 2.

The code for this section is in `q5.py` with tests contained in `q5_test.py`. Run the script to see the plots. Change the initial conditions by adjusting `p0` and `q0`. There are multiple tests in this file which check the algebraic relationships we have derived in this question.

References

[1] M. S. Bartlett. An inverse matrix adjustment arising in discriminant analysis. *Ann. Math. Statist.*, 22(1):107–111, 03 1951.