Imperial College London

COURSEWORK 3

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Computational Linear Algebra

Author:
Richard Fu (CID: 01852979)

Date: January 12, 2023

1 QUESTION 1

1.1 Part a

The script that plots our $||A_s||$ values against iteration number for matrix A_3 is located in cw3/plotting_scripts/q1_plots.py. To help this, a keyword argument *return_norms* has been added to **cla_utils.pure_QR()** to specify whether to return an array of $||A_s||$ values.
In Figure 1 below the generated plot is shown, and we see that $||A_s||$ converges towards 0 - our matrix A_3 is transformed to an upper triangular matrix by the QR algorithm.

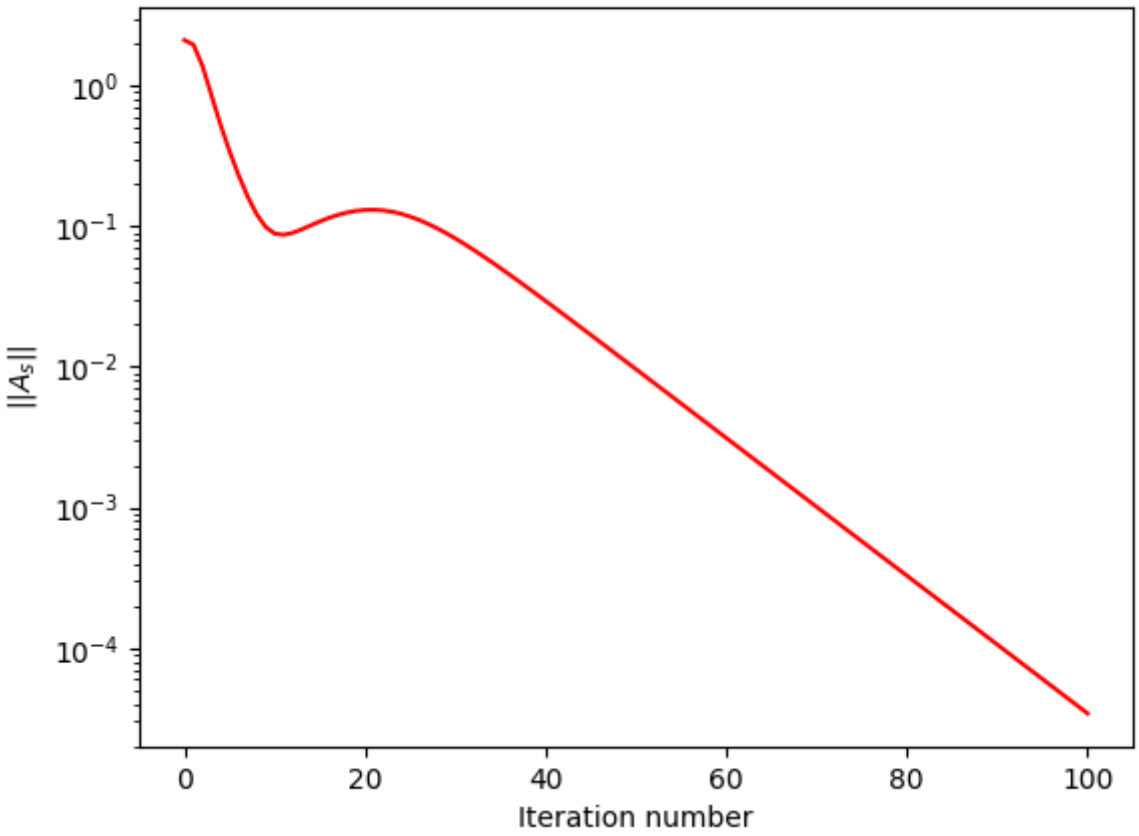

Figure 1

Below this there is some code that verifies the eigenvalues we have obtained are indeed the eigenvalues of A_3, by using **cla_utils.inverse_it()** to find the eigenvector v_i corresponding to each eigenvalue λ_i, and calculating the norm of residual $A_3 v_i - \lambda_i v_i$. We print the norm of the vector containing these residual norms, and we see this is very small, indicating our eigenvalues are indeed those of A_3.

1.2 Part b

To help examine the convergence of the diagonal entries of A_3, for each diagonal entry we plot the errors in its convergence to its associated eigenvalue. This can be seen on the left plot of Figure 2, and it appears that the lower right entry (entry 6,6) converges the quickest, almost instantly, while the upper-left entries (entries 1,1 and 2,2) appear to converge the slowest. The other diagonal entries (entries 3,3, 4,4 and 5,5) appear to converge at a similar rate to each other, slower than the lower-right entry but faster than the upper-left entries.
Furthermore, we also plot the absolute value of the diagonal entries as they converge, and this can be seen on the right plot of Figure 2. It appears that the order in which the diagonal entry converges to its eigenvalue is determined by the absolute value of this eigenvalue. The smaller the magnitude of the eigenvalue the quicker it converges.
Combining the observations of the two plots, we can infer that moving from the lower-right diagonal entry to the upper-left diagonal entry, the convergence speed decreases, whilst the associated eigenvalue magnitude increases.

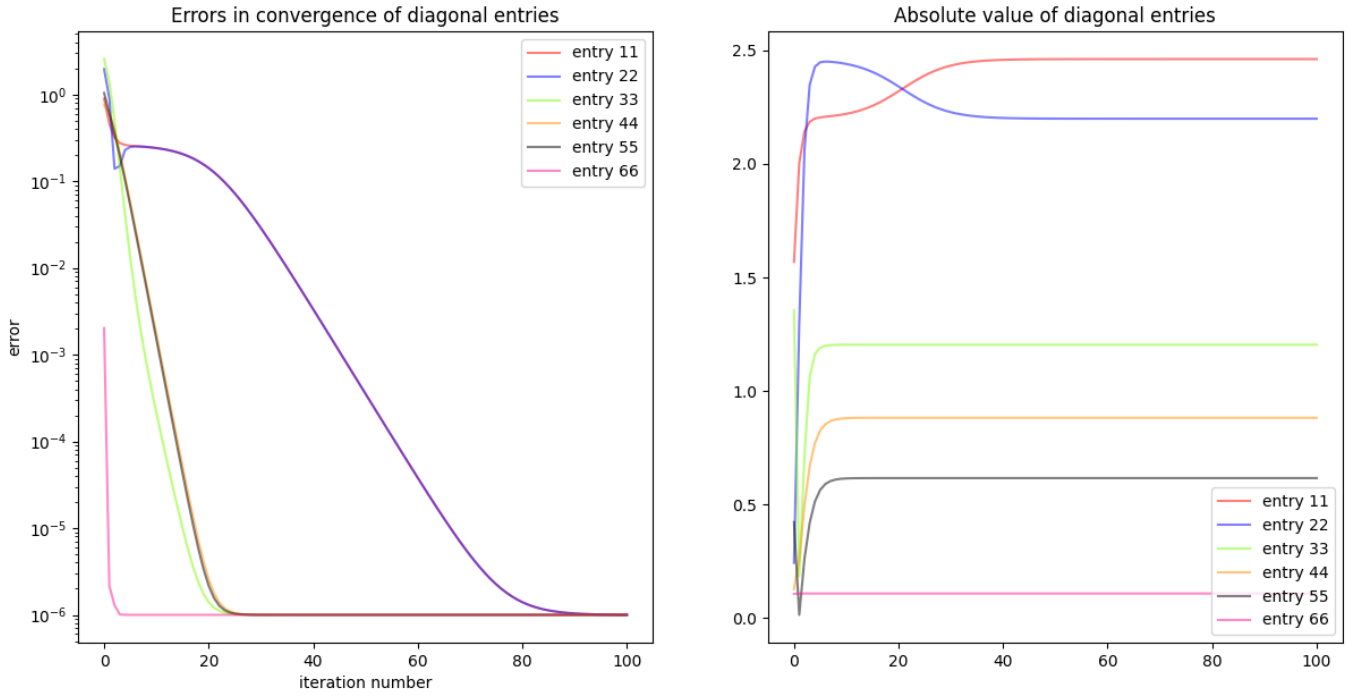

Figure 2

The script that produces Figure 2 is located in cw3/plotting_scripts/q1_plots.py. To help this, a keyword argument *return_iterations* has been added to **cla_utils.pure_QR()** to specify whether to return an array where each row contains the diagonal entries at each iteration.

1.3 Part c

In cw3/q1.py, there is the code that prints the transformed matrix formed from applying **cla_utils.pure_QR()** to A_4, and it is also given in (1).

$$A_4^{(k)} = \begin{bmatrix} -2.4042 & 0.1309 & 0.0031 & 1.6203 & 1.4679 & -0.0818 \\ 0.0000 & 1.613 & 1.3264 & -0.5657 & 0.0662 & 1.4145 \\ -0.0000 & -0.0000 & -0.9980 & -1.1016 & -1.4881 & 0.4347 \\ -0.0000 & 0.0000 & 0.3642 & -0.8004 & -0.9176 & -0.7905 \\ 0.0000 & 0.0000 & 0.0000 & 0.0000 & -0.7388 & -0.5005 \\ -0.0000 & -0.0000 & -0.0000 & 0.0000 & 0.0000 & 0.6155. \end{bmatrix} \tag{1}$$

$A_4^{(k)}$ will never converge to Schur decomposition of A_4 when using **pure_QR()**, because A_4 has complex eigenvalues. The **pure_QR()** algorithm cannot transform A_4 to its Schur decomposition because the QR factorisation at each iteration is real and so the resulting matrix must also be real, therefore it cannot have the complex eigenvalues on its diagonal as its Schur decomposition would.

1.4 Part d

Using the fact given:

$$\det(A') = \prod_{i=1}^{r} \det(R_{ii}), \tag{2}$$

$$\implies \det(A' - \lambda I) = \prod_{i=1}^{r} \det(R_{ii} - \lambda I). \tag{3}$$

Suppose λ is an eigenvalue of $\det(A')$.
$\implies \det(A' - \lambda I) = 0$,
$\implies \prod_{i=1}^{r} \det(R_{ii} - \lambda I)$, using (3)
$\implies \det(R_{ii} - \lambda I) = 0$, for some $i = 1, \ldots, r$.
From this we can see that a method to obtain the eigenvalues of A' involves finding the eigenvalues of R_{ii}, for $i = 1, \ldots, r$.

In order to implement this, I have added a keyword argument *special_criteria* to **cla_utils.pure_QR()** which can be used to alter the stopping criteria of **pure_QR()** so that it stops when a matrix of the form A' is achieved. The corresponding pytest **test_pure_QR_special()** ensures this new criteria is working properly has been added to test/test_test_exercises9.py.
The script that computes the eigenvalues of A_4 is located in cw3/q1.py. We also verify these are indeed eigenvalues by finding the corresponding eigenvector using **cla_utils.inverse_it()**, and again find the norm of residual $A_4 v_i - \lambda_i v_i$. Below we print the norm of the vector containing these residual norms, and again we find that this is small, verifying they indeed are eigenvalues.

We also had to update our **inverse_it()** method to be able to find corresponding eigenvectors for complex eigenvalues, and so we propose the following method to handle this if we pass in a complex shift.
Let our complex eigenvalue shift $\mu = \mu_r + i\mu_i$, with nearest eigenvalue $\lambda = \lambda_r + i\lambda_i$ and corresponding eigenvector $\boldsymbol{v} = \boldsymbol{v}_r + i\boldsymbol{v}_i$:

$$\boldsymbol{A}\boldsymbol{v} = \lambda\boldsymbol{v}, \tag{4}$$
$$\implies \boldsymbol{A}(\boldsymbol{v}_r + i\boldsymbol{v}_i) = (\lambda_r + i\lambda_i)(\boldsymbol{v}_r + i\boldsymbol{v}_i). \tag{5}$$

Taking real and imaginary parts, we get:

$$\boldsymbol{A}\boldsymbol{v}_r + \lambda_i\boldsymbol{v}_i = \lambda_r\boldsymbol{v}_r, \tag{6}$$
$$-\lambda_i\boldsymbol{v}_r + \boldsymbol{A}\boldsymbol{v}_i = \lambda_r\boldsymbol{v}_i. \tag{7}$$

Constructing a matrix $\boldsymbol{B} = \begin{bmatrix} \boldsymbol{A} & \lambda_i\boldsymbol{I} \\ -\lambda_i\boldsymbol{I} & \boldsymbol{A} \end{bmatrix}$ and vector $\boldsymbol{w} = \begin{bmatrix} \boldsymbol{v}_r \\ \boldsymbol{v}_i \end{bmatrix}$. It follows:

$$\boldsymbol{B}\boldsymbol{w} = \lambda_r\boldsymbol{w}. \tag{8}$$

λ_r is an eigenvalue of $\boldsymbol{B}$, and so we can obtain $\boldsymbol{w}$ by calling **inverse_it()** on $\boldsymbol{B}$ with real shift μ_r.
Once we obtain $\boldsymbol{w}$, our eigenvector of $\boldsymbol{B}$, we can recover our eigenvector of $\boldsymbol{A}$ by slicing out $\boldsymbol{v}_r$ and $\boldsymbol{v}_i$, and evaluating $\boldsymbol{v} = \boldsymbol{v}_r + i\boldsymbol{v}_i$.
This method has been implemented directly to **inverse_it()**.

1.5 Part e

The function that finds the eigenvalues of real matrices with a similar pattern to $\boldsymbol{A}_4$ is located in cw3/q1.py, and is called **block_matrix_eigs()**. The corresponding pytests are located in cw3/test1.py, and can be run by calling **test_block_matrix_eigs()**.
Below this function, there is a script that calls **block_matrix_eigs()** and prints the eigenvectors of the following example matrices:

$$\boldsymbol{M}_1 = \left[\begin{array}{cc|c} 2 & 5 & 1 \\ -1 & 0 & 2 \\ \hline 0 & 0 & 3 \end{array}\right], \quad \text{numerical eigenvalues } 1{+}2i,\ 1{\text{-}}2i,\ 3. \tag{9}$$

$$\boldsymbol{M}_2 = \left[\begin{array}{cc|cc|c} 0 & 1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 2 & 5 & 1 \\ 0 & 0 & -1 & 0 & 2 \\ \hline 0 & 0 & 0 & 0 & 3 \end{array}\right], \quad \text{numerical eigenvalues } i,\ -i,\ 1{+}2i,\ 1{\text{-}}2i,\ 3. \tag{10}$$

$$M_3 = \left[\begin{array}{cc|cc|cc|cc} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 2 & 5 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 & 2 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 3 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 5 \\ 0 & 0 & 0 & 0 & 0 & -1 & 4 \end{array}\right], \quad \text{numerical eigenvalues } i, -i, 1+2i, 1\text{-}2i, 3, 2+i, 2\text{-}i. \tag{11}$$

Each of these 3 matrices has the structure of 1x1 and 2x2 blocks, and so it can be easily verified by hand that the numerical eigenvalues found are correct by solving for the eigenvalues of each block.
Also, the diagonal blocks of M_1 are also present in M_2 and M_3, so we would expect all three matrices to share the 3 corresponding eigenvalues; we can see they indeed do in our numerical eigenvalue calculations.

1.6 Part f

Performing the $(k+1)^{th}$ procedure:

$$\mu_k = A^{(k)}_{mm}, \tag{12}$$
$$Q^{(k+1)}R^{(k+1)} = A^{(k)} - \mu_k I, \tag{13}$$
$$A^{(k+1)} = R^{(k+1)}Q^{(k+1)} + \mu_k I, \tag{14}$$
$$= Q^{*(k+1)}Q^{(k+1)}R^{(k+1)}Q^{(k+1)} + \mu_k I, \quad \text{since } Q^{(k+1)} \text{ unitary,} \tag{15}$$
$$= Q^{*(k+1)}\left(A^{(k)} - \mu_k I\right)Q^{(k+1)} + \mu_k I, \tag{16}$$
$$= Q^{*(k+1)}A^{(k)}Q^{(k+1)} - \mu_k Q^{*(k+1)}Q^{(k+1)} + \mu_k I, \tag{17}$$
$$= Q^{*(k+1)}A^{(k)}Q^{(k+1)} - \mu_k I + \mu_k I, \quad \text{since } Q^{(k+1)} \text{ unitary,} \tag{18}$$
$$= Q^{*(k+1)}A^{(k)}Q^{(k+1)}. \tag{19}$$

This shows that $A^{(k+1)}$ is similar to $A^{(k)}$ because $Q^{(k+1)}$ is unitary. Eigenvalues are preserved during a similarity transformation, therefore it follows that this procedure preserves the eigenvalues of A.

1.7 Part g

We can apply a similarity transformation to our matrix A before iterating that to transform it to an Upper-Hessenberg matrix, and take that as our $A^{(0)}$. Since A is symmetric, the Hessenberg form is also symmetric, and in particular it must actually be tri-diagonal (banded matrix with bandwidths 1).
Hessenberg form is also preserved during the QR algorithm, and so each $A^{(k)}$ is also tri-diagonal.
When entry $A^{(k)}_{m,m}$ has converged enough, $A^{(k)} - \mu_k I$ is very nearly singular i.e. column m of $A^{(k)} - \mu_k I$ can almost be fully expressed as a linear combination of the previous

$m-1$ columns.

The first $m-1$ columns of $\boldsymbol{A}^{(k)} - \mu_k \boldsymbol{I}$ are given by the span of the first $m-1$ columns of $\boldsymbol{Q}^{(k+1)}$ since $\boldsymbol{R}^{(k+1)}$ is upper triangular. This means that column m of $\boldsymbol{A}^{(k)} - \mu_k \boldsymbol{I}$ can almost be fully expressed as a linear combination of the first $m-1$ columns of $\boldsymbol{Q}^{(k+1)}$, with coefficients given by column m of $\boldsymbol{Q}^{(k+1)}$. It follows that the final entry of this column is very small, i.e. $\boldsymbol{R}^{(k+1)}_{m,m} \approx 0$.

Finally, when evaluating $\boldsymbol{A}^{(k)} = \boldsymbol{R}^{(k+1)}\boldsymbol{Q}^{(k+1)}$, entry $\boldsymbol{A}^{(k+1)}_{m,m-1} = \boldsymbol{R}^{(k+1)}_{m,m} \cdot \boldsymbol{Q}^{(k+1)}_{m,m-1} \approx 0$. By symmetry of the Hessenberg form, $\boldsymbol{A}^{(k+1)}_{m-1,m} \approx 0$. This entry is unaffected by the diagonal addition of μ_k, and so by checking this entry is small, we have a convergence test for the diagonal entry $\boldsymbol{A}^{(k+1)}_{m,m}$ being close to the eigenvalue.

1.8 Part h

The function that implements this modified algorithm for symmetric matrices is called **QR_shift()** and is located in cw3/q1.py. There are corresponding pytests that verify it is working in cw3/test1.py, which can be run by calling **test_QR_shift()**. There is also an automated test for $\boldsymbol{A}_3$ called **test_QR_shift_A3()**.

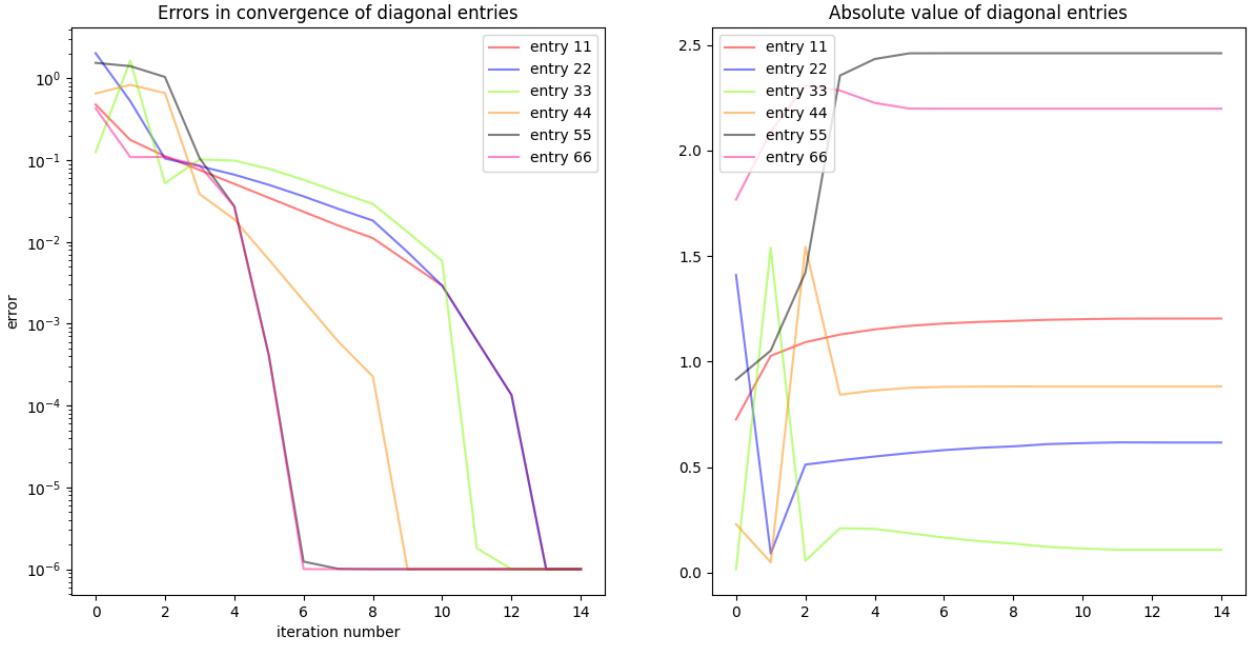

Figure 3

In Figure 3 we have a demonstration of applying **QR_shift()** to the matrix $\boldsymbol{A}_4$. The script that generates this is located in cw3/q1.py We see the non-smooth convergence to the eigenvalues due to changing shift used at each iteration, and further-

more we see that the iterations required for convergence is a lot less than that when using **pure_QR()** in Figure 2 (14 iterations vs 100 iterations).

1.9 Part i

The script that generates Figure 4 is located in cw3/plotting_scripts/q1_plots.py. It plots the number of iterations it takes for all the eigenvalues of a symmetric matrix to converge to 3 decimal places across varying large m, when using both QR algorithms.

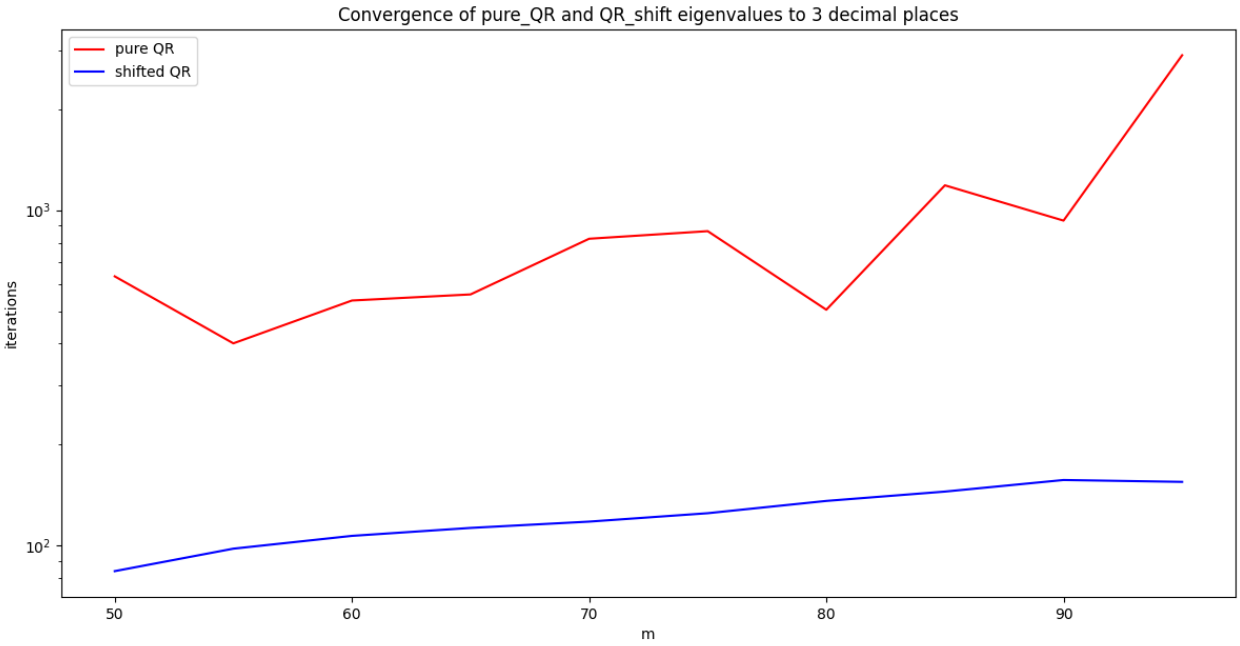

Figure 4

We can see that for these large symmetric matrices, clearly the shifted QR outperforms the pure QR in terms of the iterations required for all the eigenvalues to converge, which is expected since the shifts were implemented to improve convergence rate.

1.10 Part j

For a matrix $A^{(k)}$ of dimension m, the operation count for one QR iteration is $\mathcal{O}(m^3)$. After deflating j times i.e. reduction in matrix dimension j times, the operation count for one QR iteration on this dimension $m-j$ matrix is now $\mathcal{O}((m-j)^3)$.
This suggests that for large matrices where m is large, the deflation strategy can significantly reduce the operation count, and therefore accelerate the procedure.

1.11 Part k

When we apply our implementation to $\boldsymbol{A}_4$, we see that our algorithm will not converge because the stopping criteria is never met.
The issue is again that $\boldsymbol{A}_4$ has complex eigenvalues, and since all our matrix operations involve real matrices, it is impossible for our stopping condition to be satisfied, since this would involve complex numbers on the diagonal.

One way we could fix this is to alter our convergence criteria again so that it supports the block structure that the QR algorithm converged to for matrices with complex eigenvalues.

2 QUESTION 2

2.1 Part a

In cw3/q2.py there is a function **get_callback()** that appends the errors in solution x at each iteration of GMRES to cw3/callback.dat.
The keyword argument *callback* has been added to **cla_utils.GMRES()**, and the pytest **test_GMRES_callback()** has been added to test/test_exercises10.py which verifies this feature is working.

2.2 Part b

Let $\lambda^{(i)}$ be an eigenvalue of a matrix, with corresponding eigenvector $\boldsymbol{u}^{(i)}$ of given form. We find a formula for $\lambda^{(i)}$:

$$\boldsymbol{A}\boldsymbol{u}^{(l)} = \lambda^l \boldsymbol{u}^{(l)}, \tag{20}$$

$$\implies \begin{bmatrix} -2 & 1 & 0 & \dots & 0 \\ 1 & -2 & \ddots & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & -2 & 1 \\ 0 & \dots & 0 & 1 & -2 \end{bmatrix} \begin{bmatrix} \sin\left(\frac{l\pi}{m+1}\right) \\ \sin\left(\frac{2l\pi}{m+1}\right) \\ \vdots \\ \sin\left(\frac{(m-1)l\pi}{m+1}\right) \\ \sin\left(\frac{ml\pi}{m+1}\right) \end{bmatrix} = \lambda^{(l)} \begin{bmatrix} \sin\left(\frac{l\pi}{m+1}\right) \\ \sin\left(\frac{2l\pi}{m+1}\right) \\ \vdots \\ \sin\left(\frac{(m-1)l\pi}{m+1}\right) \\ \sin\left(\frac{ml\pi}{m+1}\right) \end{bmatrix}. \tag{21}$$

$$\implies \sin\left(\frac{(k-1)l\pi}{m+1}\right) - 2\sin\left(\frac{kl\pi}{m+1}\right) + \sin\left(\frac{(k+1)l\pi}{m+1}\right) = \lambda^{(i)} \sin\left(\frac{kl\pi}{m+1}\right), \quad k = 1, \dots, m. \tag{22}$$

Rearranging the relation for $k = 1$:

$$\lambda^{(l)} = \frac{\sin\left(\frac{2l\pi}{m+1}\right)}{\sin\left(\frac{l\pi}{m+1}\right)} - 2, \tag{23}$$

$$= \frac{2\sin\left(\frac{l\pi}{m+1}\right)\cos\left(\frac{l\pi}{m+1}\right)}{\sin\left(\frac{l\pi}{m+1}\right)} - 2, \tag{24}$$

$$= 2\left(\cos\left(\frac{l\pi}{m+1}\right) - 1\right), \quad l = 1, \ldots, m. \tag{25}$$

2.3 Part c

In cw3/q2.py, there is a helper function **get_A()** which builds $\boldsymbol{A}$ as described in the question. The script that creates Figure 5 is located in cw3/plotting_scripts/q2_plots.py.

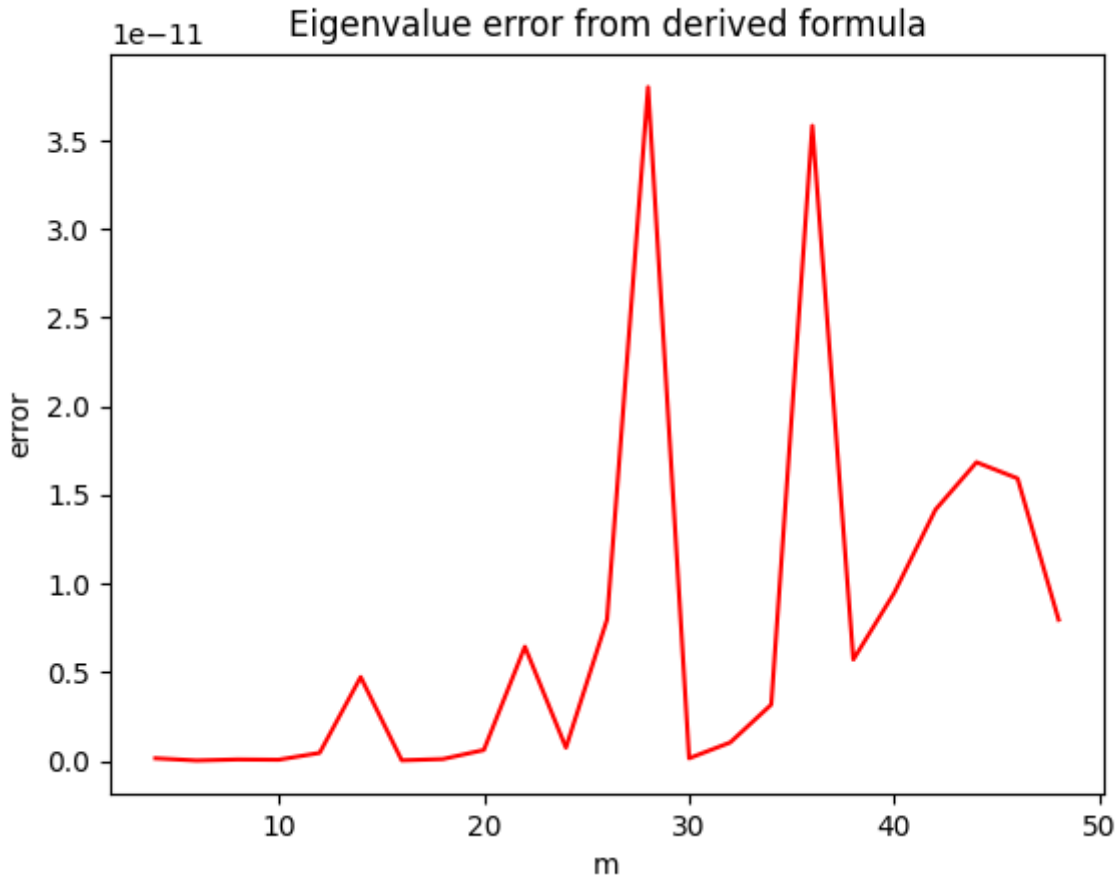

Figure 5

From this figure we can see that our error in eigenvalue when using this formula is very small for all the sizes of m, which suggests that our eigenvalue formula is indeed valid.

2.4 Part d

The script that creates Figure 6 is located in cw3/plotting_scripts/q2_plots.py.

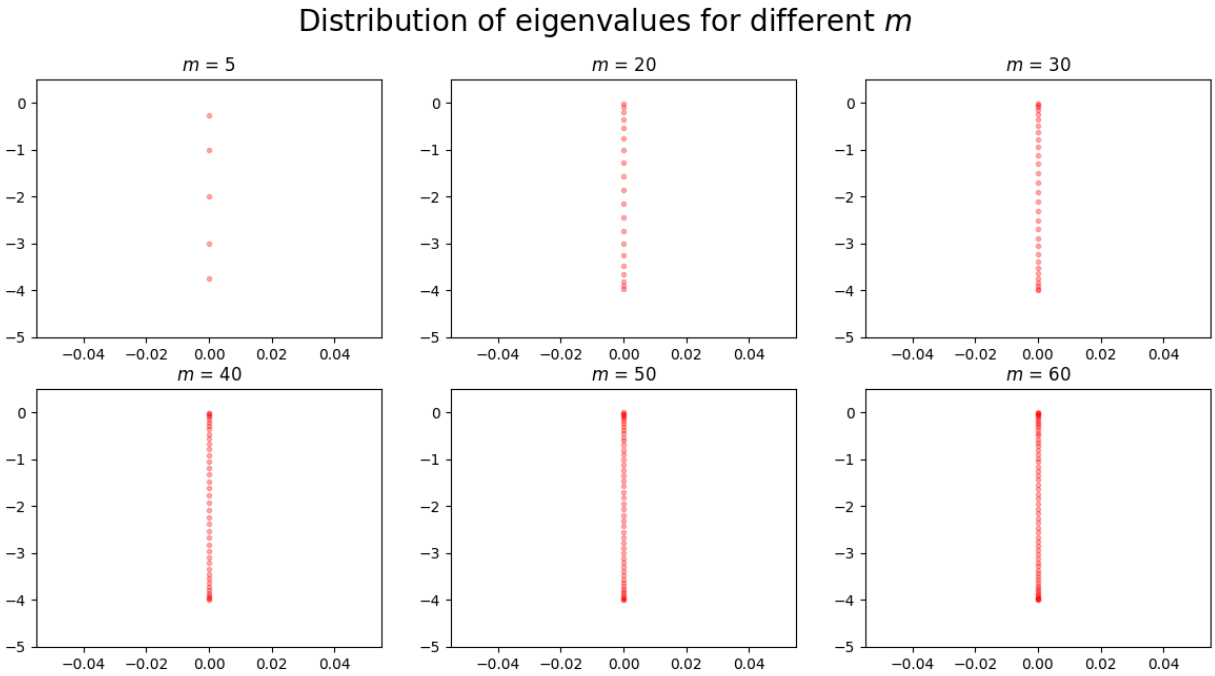

Figure 6

In general, the polynomial $p(x)$ must satisfy $p(0) = 1$, and also $p(x_\lambda)$ must be small for eigenvalues x_λ to make the residual estimate small.
We can see in Figure 6 that as m increases, we get more eigenvalues clustering near 0 and -4, and the space between adjacent eigenvalues seems to decrease.
Therefore for increasingly large m, in order to for our polynomial to go through (0, 1), pass through the cluster near 0, and also pass through 0 near the increasingly closely spaced eigenvalues away from the clusters, the polynomial must have increasingly high degree.
As a result, since the number of iterations for convergence is the same as the polynomial degree, the convergence rate of GMRES gets increasingly slower as m increases.

2.5 Part e

$$\kappa(\boldsymbol{A}) = \|\boldsymbol{A}^{-1}\| \cdot \|A\|, \tag{26}$$

$$= \frac{1}{|\lambda_{min}|} \cdot |\lambda_{max}|, \tag{27}$$

$$= \frac{|2\left(\cos\left(\frac{m\pi}{m+1}\right) - 1\right)|}{|2\left(\cos\left(\frac{\pi}{m+1}\right) - 1\right)|}, \tag{28}$$

$$= \frac{1 - \cos\left(\frac{m\pi}{m+1}\right)}{1 - \cos\left(\frac{\pi}{m+1}\right)}. \tag{29}$$

Let $x = \frac{1}{m}$, and use Taylor expansion around $x = 0$:

$$\kappa(\boldsymbol{A}) = \frac{1 - \cos\left(\frac{m\pi}{m+1}\right)}{1 - \cos\left(\frac{\pi}{m+1}\right)} \tag{30}$$

$$= \frac{1 + \cos\left(\frac{\pi}{m+1}\right)}{1 - \cos\left(\frac{\pi}{m+1}\right)} \tag{31}$$

$$= \frac{1 + \left(1 - \frac{\pi^2}{2(m+1)^2} + \frac{\pi^4}{24(m+1)^4} + \dots\right)}{1 - \left(1 - \frac{\pi^2}{2(m+1)^2} + \frac{\pi^4}{24(m+1)^4} + \dots\right)}, \tag{32}$$

$$= \frac{2 - \frac{\pi^2}{2(m+1)^2} + \frac{\pi^4}{24(m+1)^4} + \dots}{\frac{\pi^2}{2(m+1)^2} - \frac{\pi^4}{24(m+1)^4} + \dots}, \tag{33}$$

$$\approx \frac{2 - \frac{\pi^2}{2(m+1)^2} + \frac{\pi^4}{24(m+1)^4} + \dots}{1 + \mathcal{O}(\frac{1}{(m+1)^2})} \left(\frac{2(m+1)^2}{\pi^2}\right) \tag{34}$$

$$\approx \frac{\frac{4(m+1)^2}{\pi^2} + \mathcal{O}(1)}{1 + \mathcal{O}(\frac{1}{(m+1)^2})} \tag{35}$$

$$\approx \frac{\frac{4}{\pi^2}m^2 + \mathcal{O}(m)}{1 + \mathcal{O}(\frac{1}{(m+1)^2})} \tag{36}$$

$$\approx \frac{4}{\pi^2}m^2 + \mathcal{O}(m) \tag{37}$$

For our given matrix $\boldsymbol{A}$, since $\kappa(\boldsymbol{A}) = \kappa(-\boldsymbol{A})$ and $-\boldsymbol{A}$ is positive definite, the given inequality also holds for $\boldsymbol{A}$ as well as $-\boldsymbol{A}$.

$$\|r_n\| \leq \left(\frac{\kappa^2 - 1}{\kappa^2}\right)^{\frac{n}{2}} \|r_0\| = \left(1 - \frac{1}{\kappa^2}\right)^{\frac{n}{2}} \|r_0\|. \tag{38}$$

From this bound, we see in order for the norm of the residual to converge to 0, n must be sufficiently large so that $\left(1 - \frac{1}{\kappa^2}\right)^{\frac{n}{2}}$ is small. This n depends on the condition number κ. The larger κ is, the closer $\left(1 - \frac{1}{\kappa^2}\right)$ is to 1, and therefore more power iterations $\frac{n}{2}$ are needed for the right hand side to be small. We have shown that κ is $\mathcal{O}(m^2)$, so κ^2 is $\mathcal{O}(m^4)$, and therefore increasing m causes a much larger n to be required for convergence. Hence, the convergence rate will slow down dramatically as m increases.

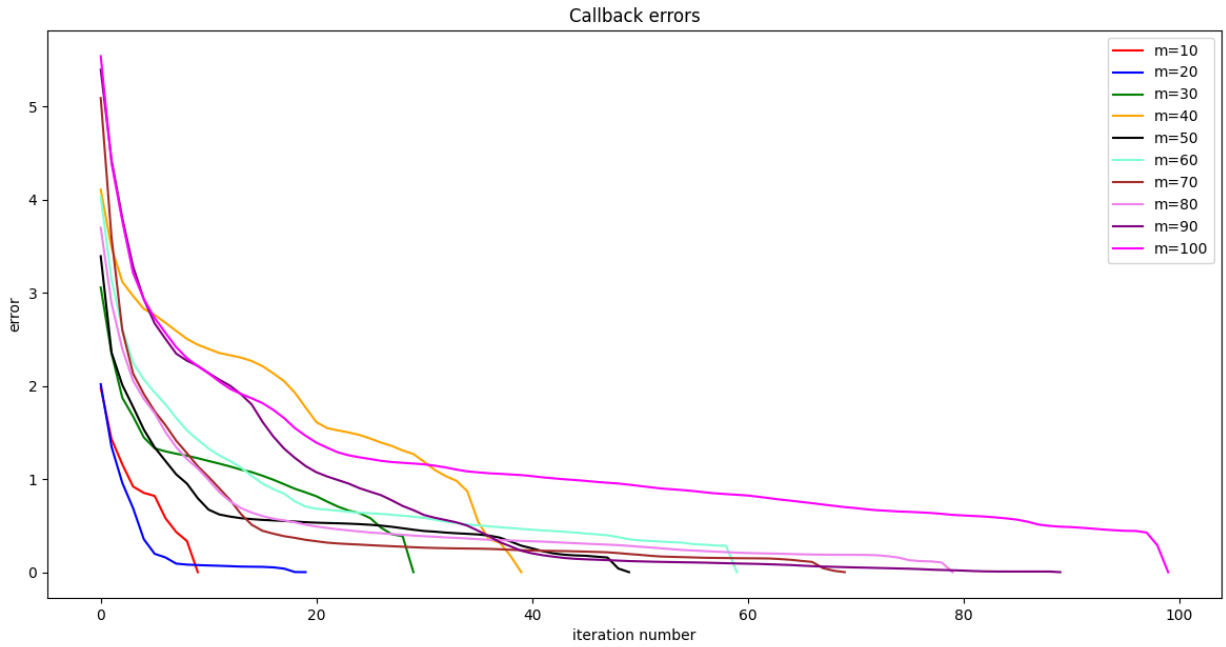

Figure 7

From Figure 7, we can see that as m increases from 10 to 100, our error converges to 0 in a lot more iterations, agreeing with our predictions. The script that creates this plot is located in cw3/plotting_scripts/q2_plots.py.

2.6 Part f

Using the diagonal part of $\boldsymbol{A}$, say this $\boldsymbol{D}$, is a cheap option for a preconditioner because the diagonal system $\boldsymbol{Dx} = \boldsymbol{y}$ is cheap to solve. When we use $\boldsymbol{D}$ as a preconditioner, we consider $\boldsymbol{D}^{-1}\boldsymbol{A} = -\frac{1}{2}\boldsymbol{A}$. Multiplying $\boldsymbol{A}$ by a scalar scales all the eigenvalues of $\boldsymbol{A}$ by the same amount, and so when computing the condition number of $\boldsymbol{D}^{-1}\boldsymbol{A}$, we find it is the same as the condition number of $\boldsymbol{A}$ because the scalars cancel out. Hence, although it is cheap to use, we would not expect to get any improvement in convergence rate when using this preconditioner for GMRES.

3 QUESTION 3

3.1 Part a

Let $\boldsymbol{A} = \boldsymbol{UDV}^*$, where $\boldsymbol{U}, \boldsymbol{V}$ are unitary, and $\boldsymbol{D}$ diagonal with non-negative entries. First consider

$$\boldsymbol{A}^*\boldsymbol{A} = (\boldsymbol{UDV}^*)^*\boldsymbol{UDV}^*, \tag{39}$$

$$= \boldsymbol{VD}^*\boldsymbol{U}^*\boldsymbol{UDV}^*, \tag{40}$$

$$= \boldsymbol{V}\boldsymbol{D}^*\boldsymbol{D}\boldsymbol{V}^*, \quad \text{since } \boldsymbol{U} \text{ is unitary,} \tag{41}$$

$$= \boldsymbol{V}\boldsymbol{D}^2\boldsymbol{V}^*. \tag{42}$$

$\boldsymbol{D}$ is diagonal, and so $\boldsymbol{D}^2$ is also diagonal. We have found a unitary diagonalisation for $\boldsymbol{A}^*\boldsymbol{A}$, and so we can obtain the columns of $\boldsymbol{V}$ by finding the eigenvectors of $\boldsymbol{A}^*\boldsymbol{A}$.

Now consider

$$\boldsymbol{A}\boldsymbol{A}^* = \boldsymbol{U}\boldsymbol{D}\boldsymbol{V}^*(\boldsymbol{U}\boldsymbol{D}\boldsymbol{V}^*)^*, \tag{43}$$

$$= \boldsymbol{U}\boldsymbol{D}\boldsymbol{V}^*\boldsymbol{V}\boldsymbol{D}^*\boldsymbol{U}^*, \tag{44}$$

$$= \boldsymbol{U}\boldsymbol{D}\boldsymbol{D}^*\boldsymbol{U}^*, \quad \text{since } \boldsymbol{V} \text{ is unitary,} \tag{45}$$

$$= \boldsymbol{U}\boldsymbol{D}^2\boldsymbol{U}^*. \tag{46}$$

Again using the argument of unitary diagonalisation, this time for $\boldsymbol{A}\boldsymbol{A}^*$, we can obtain the columns of $\boldsymbol{U}$ by finding the eigenvectors of $\boldsymbol{A}\boldsymbol{A}^*$.

3.2 Part b

$$\boldsymbol{H}\begin{bmatrix}\boldsymbol{V} & \boldsymbol{V}\\ \boldsymbol{U} & -\boldsymbol{U}\end{bmatrix} = \begin{bmatrix}\boldsymbol{0} & \boldsymbol{A}^*\\ \boldsymbol{A} & \boldsymbol{0}\end{bmatrix}\begin{bmatrix}\boldsymbol{V} & \boldsymbol{V}\\ \boldsymbol{U} & -\boldsymbol{U}\end{bmatrix}, \tag{47}$$

$$= \begin{bmatrix}\boldsymbol{A}^*\boldsymbol{U} & -\boldsymbol{A}^*\boldsymbol{U}\\ \boldsymbol{A}\boldsymbol{V} & \boldsymbol{A}\boldsymbol{V}\end{bmatrix}, \tag{48}$$

$$= \begin{bmatrix}(\boldsymbol{U}^*\boldsymbol{A})^* & -(\boldsymbol{U}^*\boldsymbol{A})^*\\ \boldsymbol{A}\boldsymbol{V} & \boldsymbol{A}\boldsymbol{V}\end{bmatrix}, \tag{49}$$

$$= \begin{bmatrix}(\boldsymbol{U}^*\boldsymbol{A}\boldsymbol{V}\boldsymbol{V}^*)^* & -(\boldsymbol{U}^*\boldsymbol{A}\boldsymbol{V}\boldsymbol{V}^*)^*\\ \boldsymbol{U}\boldsymbol{U}^*\boldsymbol{A}\boldsymbol{V} & \boldsymbol{U}\boldsymbol{U}^*\boldsymbol{A}\boldsymbol{V}\end{bmatrix}, \tag{50}$$

$$= \begin{bmatrix}(\boldsymbol{D}\boldsymbol{V})^* & -(\boldsymbol{D}\boldsymbol{V})^*\\ \boldsymbol{U}\boldsymbol{D} & \boldsymbol{U}\boldsymbol{D}\end{bmatrix}, \tag{51}$$

$$= \begin{bmatrix}\boldsymbol{V}\boldsymbol{D} & -\boldsymbol{V}\boldsymbol{D}\\ \boldsymbol{U}\boldsymbol{D} & \boldsymbol{U}\boldsymbol{D}\end{bmatrix}, \tag{52}$$

$$= \begin{bmatrix}\boldsymbol{V} & \boldsymbol{V}\\ \boldsymbol{U} & -\boldsymbol{U}\end{bmatrix}\begin{bmatrix}\boldsymbol{D} & \boldsymbol{0}\\ \boldsymbol{0} & -\boldsymbol{D}\end{bmatrix}. \tag{53}$$

Note:

$$\frac{1}{\sqrt{2}}\begin{bmatrix}\boldsymbol{V} & \boldsymbol{V}\\ \boldsymbol{U} & -\boldsymbol{U}\end{bmatrix}\frac{1}{\sqrt{2}}\begin{bmatrix}\boldsymbol{V} & \boldsymbol{V}\\ \boldsymbol{U} & -\boldsymbol{U}\end{bmatrix}^* = \frac{1}{2}\begin{bmatrix}\boldsymbol{V} & \boldsymbol{V}\\ \boldsymbol{U} & -\boldsymbol{U}\end{bmatrix}\begin{bmatrix}\boldsymbol{V}^* & \boldsymbol{U}^*\\ \boldsymbol{V}^* & -\boldsymbol{U}^*\end{bmatrix}, \tag{54}$$

$$= \begin{bmatrix}\boldsymbol{V}\boldsymbol{V}^* & \boldsymbol{0}\\ \boldsymbol{0} & -\boldsymbol{U}\boldsymbol{U}^*\end{bmatrix}, \tag{55}$$

$$= \boldsymbol{I}. \tag{56}$$

$$\implies \frac{1}{\sqrt{2}}\begin{bmatrix}\boldsymbol{V} & \boldsymbol{V}\\ \boldsymbol{U} & -\boldsymbol{U}\end{bmatrix} \quad \text{is unitary.} \tag{57}$$

Hence:

$$\boldsymbol{H}\frac{1}{\sqrt{2}}\begin{bmatrix}\boldsymbol{V} & \boldsymbol{V}\\ \boldsymbol{U} & -\boldsymbol{U}\end{bmatrix} = \frac{1}{\sqrt{2}}\begin{bmatrix}\boldsymbol{V} & \boldsymbol{V}\\ \boldsymbol{U} & -\boldsymbol{U}\end{bmatrix}, \tag{58}$$

$$\implies \boldsymbol{H} = \frac{1}{\sqrt{2}}\begin{bmatrix}\boldsymbol{V} & \boldsymbol{V}\\ \boldsymbol{U} & -\boldsymbol{U}\end{bmatrix}\begin{bmatrix}\boldsymbol{D} & \boldsymbol{0}\\ \boldsymbol{0} & -\boldsymbol{D}\end{bmatrix}\frac{1}{\sqrt{2}}\begin{bmatrix}\boldsymbol{V} & \boldsymbol{V}\\ \boldsymbol{U} & -\boldsymbol{U}\end{bmatrix}^*. \tag{59}$$

We have found a unitary diagonalisation of $\boldsymbol{H}$. If we find the eigenvectors of $\boldsymbol{H}$, we obtain the matrix $\frac{1}{\sqrt{2}}\begin{bmatrix}\boldsymbol{V} & \boldsymbol{V}\\ \boldsymbol{U} & -\boldsymbol{U}\end{bmatrix}$. By multiplying by $\sqrt{2}$ and slicing out the quarters of this matrix, we can also recover $\boldsymbol{V}$ and $\boldsymbol{U}$.

The algorithm goes as follows:
1) Assemble $\boldsymbol{H}$ and find its eigenvalues using one of our QR algorithms
2) Construct D by placing the non-negative eigenvalues obtained on a m diagonal matrix in descending size.
3) Use our inverse iteration method to find the eigenvectors of $\boldsymbol{H}$
4) Slice out the top left m block of the eigenvector matrix and multiply by $\sqrt{2}$ to obtain $\boldsymbol{V}$.
5) Slice out the bottom left m block of the eigenvector matrix and multiply by $\sqrt{2}$ to obtain $\boldsymbol{V}$.

It is advantageous to use $\boldsymbol{H}$ instead of $\boldsymbol{A}^*\boldsymbol{A}$ and $\boldsymbol{A}\boldsymbol{A}^*$ because the condition number is σ instead of σ^2. To see this, we know the eigenvalues of $\boldsymbol{A}^*\boldsymbol{A}$ and $\boldsymbol{A}\boldsymbol{A}^*$ are the squares of the diagonal entries of $\boldsymbol{D}$, whilst the eigenvalues of $\boldsymbol{H}$ are the diagonal entries of $\boldsymbol{D}$ and their negative values. By definition it follows that the condition number of $\boldsymbol{H}$ is the square root of the condition number of $\boldsymbol{A}^*\boldsymbol{A}$.

3.3 Part c

The function that assembles $\boldsymbol{D}$ as described above is found in cw3/q3.py, and is called **get_D()**. The corresponding pytests are located in cw3/test3.py, and can be called by running **test_get_D()**.

3.4 Part d

The function that computes the eigenvectors corresponding to the numerically obtained eigenvalues is found in cw3/q3.py, and is called **get_evecs()**. The corresponding pytests are located in cw3/test3.py, and can be called by running **test_get_evecs()** and **test_get_evecs_2()**.

3.5 Part e

To help reformulate this problem we will use the decomposition $\boldsymbol{A} = \boldsymbol{U}\boldsymbol{D}\boldsymbol{V}^*$ as defined in 3.1. Let $r = \text{rank}(A)$, d_i be the i^{th} diagonal entry of $\boldsymbol{D}$, $\mathbf{u}_i$ the i^{th} column of

$\boldsymbol{U}$.

$$\min_{\mathbf{x}\in\mathbb{R}^m} \|\boldsymbol{A}\mathbf{x}-\mathbf{b}\|^2 = \min_{\mathbf{x}\in\mathbb{R}^m} \|\boldsymbol{U}^*(\boldsymbol{A}\mathbf{x}-\mathbf{b})\|^2, \quad \text{since unitary matrix } \boldsymbol{U}^* \text{ is norm preserving.} \tag{60}$$

$$= \min_{\mathbf{x}\in\mathbb{R}^m} \|\boldsymbol{U}^*\boldsymbol{A}\boldsymbol{V}\boldsymbol{V}^*\mathbf{x}-\boldsymbol{U}^*\mathbf{b}\|^2, \quad \text{since } \boldsymbol{V} \text{ is unitary} \tag{61}$$

$$= \min_{\mathbf{x}\in\mathbb{R}^m} \|\boldsymbol{D}\boldsymbol{V}^*\mathbf{x}-\boldsymbol{U}^*\mathbf{b}\|^2, \tag{62}$$

$$= \min_{\mathbf{z}\in\mathbb{R}^m} \|\boldsymbol{D}\mathbf{z}-\boldsymbol{U}^*\mathbf{b}\|^2, \quad \boldsymbol{z} = \boldsymbol{V}^*\mathbf{x}. \tag{63}$$

$$\|\boldsymbol{D}\mathbf{z}-\boldsymbol{U}^*\mathbf{b}\|^2 = \sum_{i=1}^{r}(d_i z_i - \mathbf{u}_i^*\mathbf{b})^2 + \sum_{i=r+1}^{m}(\mathbf{u}_i^*\mathbf{b})^2, \quad \text{since } d_i = 0 \text{ for } i > r \tag{64}$$

To minimise (64), we must make the first term on the RHS zero, i.e. $z_i = \frac{\mathbf{u}_i^*\mathbf{b}}{d_i}$ for $i = 1,\ldots,r$. Since $\|\mathbf{x}\| = \|\boldsymbol{V}\mathbf{x}\| = \|\mathbf{z}\|$ by norm preserving property of unitary matrices, minimising $\|\mathbf{x}\|$ is equivalent to minimising $\|\mathbf{z}\|$.

Therefore, we take $z_i = 0$ for $i = r+1,\ldots,m$.
We recover our minimal least squares solution in terms of $\mathbf{x}$ using $\mathbf{x} = \boldsymbol{V}\mathbf{z}$.

Our lower bound for $\min_{\mathbf{x}\in\mathbb{R}^m} \|\boldsymbol{A}\mathbf{x}-\mathbf{b}\|^2$ is given by $\sum_{i=r+1}^{m}(\mathbf{u}_i^*\mathbf{b})^2$.

The function that implements this algorithm is located in cw3/q3.py, and is called **rank_def_ls()**. The corresponding pytests are located in cw3/test3.py, and are run by calling **test_rank_def_ls()**. The tests check that our solution x is orthogonal to the nullspace of $\boldsymbol{A}$. A helper function **create_rd_matrix()** is also located in cw3/q3.py to help form rank-deficient matrices.