Imperial College London

COURSEWORK 1

IMPERIAL COLLEGE LONDON

DEPARTMENT OF MATHEMATICS

Computational Linear Algebra

Author:
Richard Fu (CID: 01852979)

Date: November 3, 2022

1 QUESTION 1

1.1 Part a

The function is located in the cw1/q1.py, and is called **basis_coeffs**. There is also a function construct_A in q1.py that helps construct the least-squares matrix, which is used in **basis_coeffs** and the pytests. The pytests for this function are located in the cw1/test1.py file and is called **test_basis_coeffs**.

1.2 Part b

The function that is used to produce each of the Figures 1, 2, 3, 4 is located in the q1.py file in the cw1 directory, and is called **draw_plots**. Below the function is a script that calls the draw_plots function which can be un-commented and run to actually see the plots in a new window.

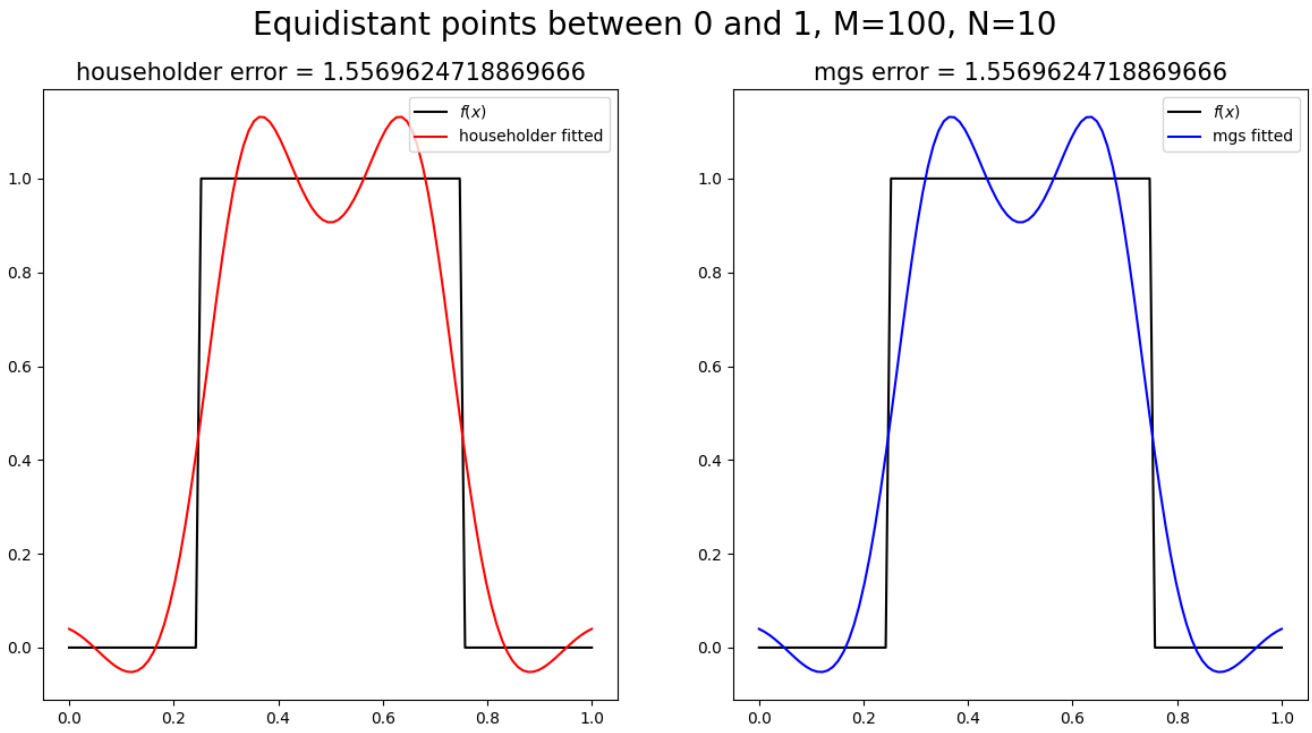

Figure 1

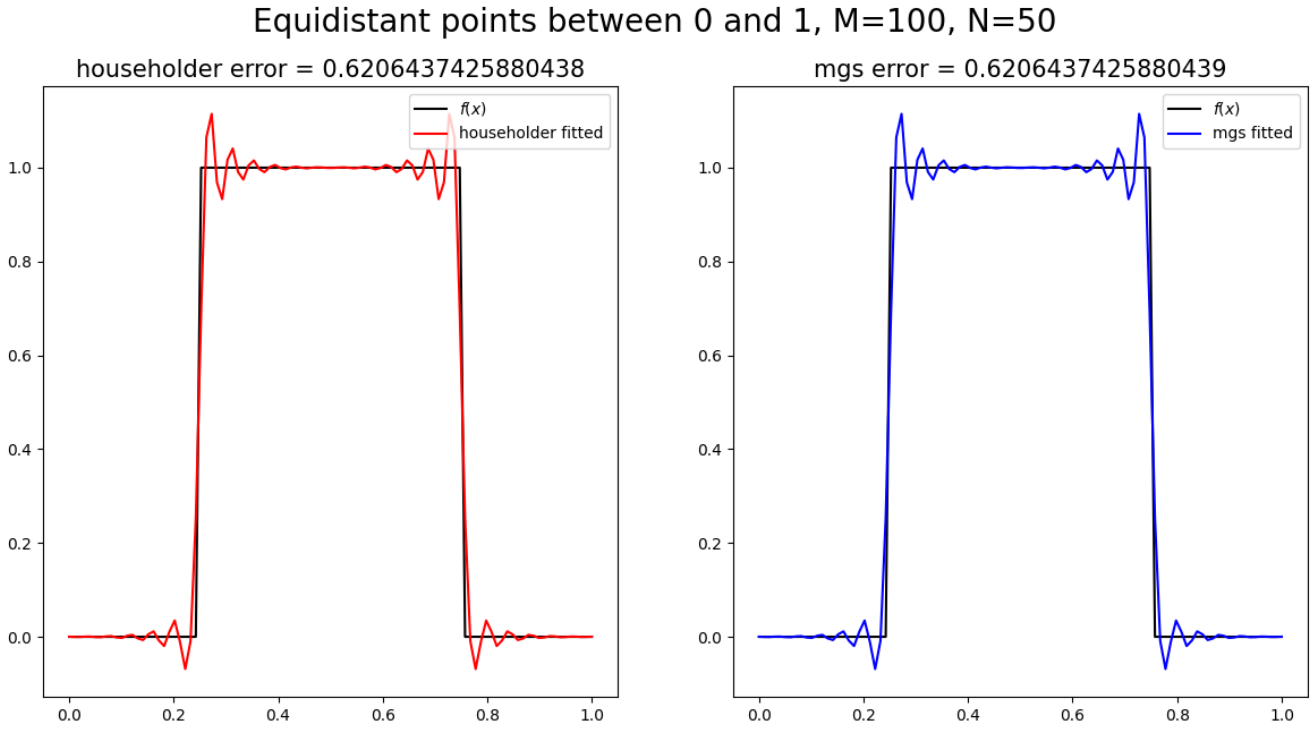

Figure 2

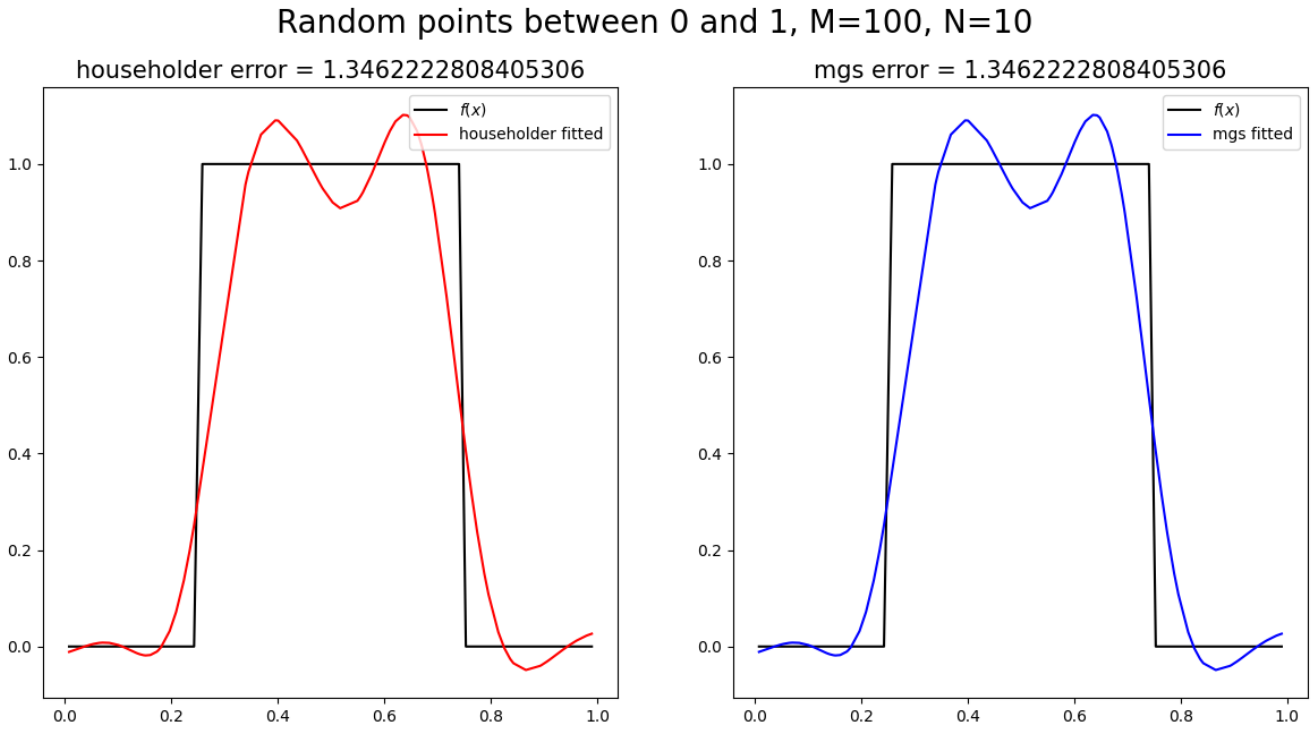

Figure 3

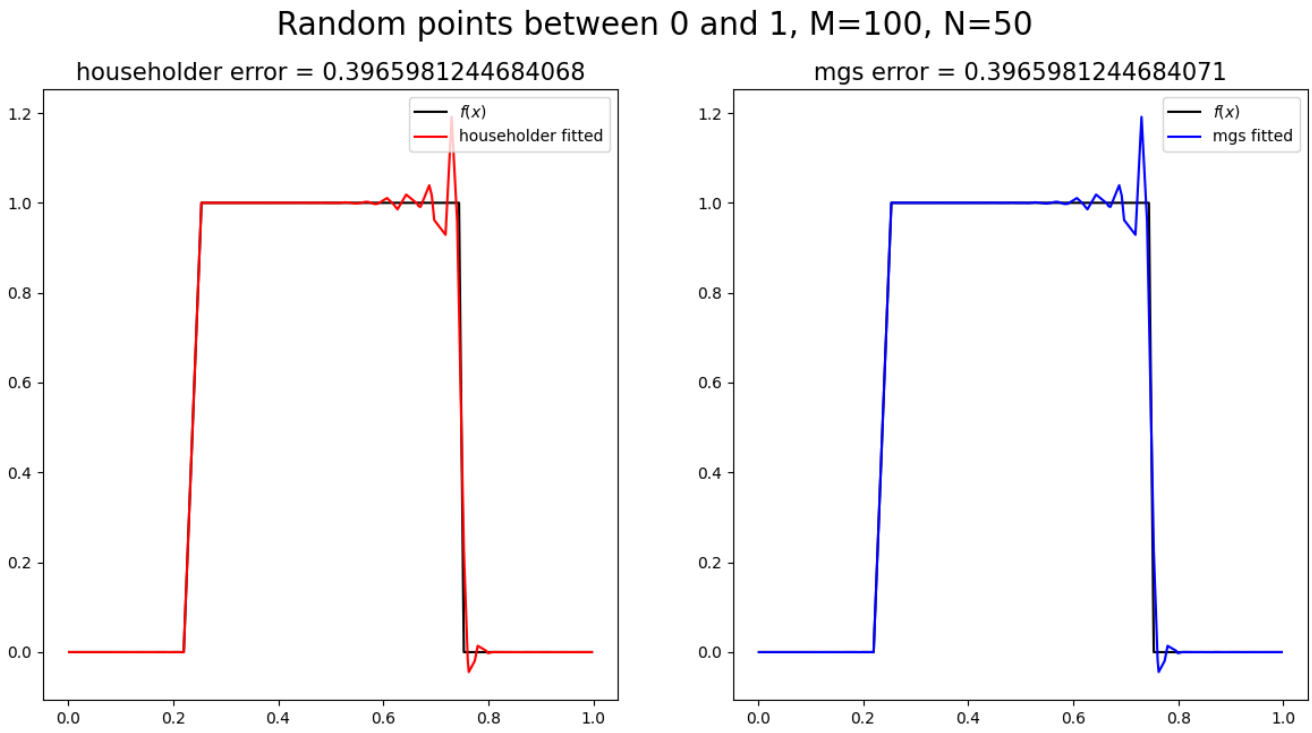

Figure 4

Comparing Figure 1 with Figure 2, and Figure 3 with Figure 4, we can see that by increasing N from 10 to 50, our error between the fitted function values and actual values decreases. We have $M = 100$ observed function values, and so increasing N from 10 to 50 gives us more basis functions to fit our function to, thus reducing the error from under-fitting. The effects of the under-fitting can be seen visually for $N = 10$ in Figure 1 and Figure 3, in which the fitted function does not follow $f(x)$ too closely and appears to be continuous at the discontinuities of the real $f(x)$, which are $x = 0.25$ and $x = 0.75$. For $N = 100$ however, we are able to see that our fitted function fits very well to $f(x)$ and it appears to have detected the discontinuous points, with only some minor oscillatory noise around these points.

Comparing the fit of equidistant points to random points on $[0, 1]$, we see that random points give less error for both $N = 10$ and $N = 50$. This could be due to the fact that random points give us a better chance of less noise around the discontinuities, as these points are more likely to be positioned on the smooth domain of $f(x)$, whereas equidistant points force a fixed number of points to be evaluated near the discontinuity.

We also see that for this function $f(x)$, using Modified Gram-Schmidt to solve the least squares problem performs identically/almost-identically in terms of error to the Householder reflection method. For this function $f(x)$, $M = 100$ and $N = 10, 50$, there seems to be no effects of numerical instability caused by rounding errors in Modified Gram-Schmidt.

2 QUESTION 2

2.1 Part a

The code to find and print the Householder QR is located at the top of the q2.py file in the cw1 directory, and running the script will print it in the console.

Computing the QR decomposition of $\boldsymbol{A}_1$, we get $\boldsymbol{A}_1 = \boldsymbol{QR} \implies \boldsymbol{Q}^T\boldsymbol{A}_1 = \boldsymbol{R}$, where $\boldsymbol{Q}$ orthogonal and $\boldsymbol{R}$ upper triangular.
$\boldsymbol{R} = \boldsymbol{Q}^T\boldsymbol{A}_1 \implies rank(\boldsymbol{R}) = rank(\boldsymbol{Q}^T\boldsymbol{A}_1) = rank(\boldsymbol{A}_1)$, with the last equality true because $\boldsymbol{Q}$ is orthogonal and therefore full-rank, which means $\boldsymbol{Q}^T$ is full-rank and therefore rank is preserved when left-multiplying $\boldsymbol{A}_1$ by $\boldsymbol{Q}^T$. Since $\boldsymbol{R}$ is upper triangular, we can get the $rank(\boldsymbol{R})$ by counting the number of non-zero terms on the diagonal, and this is also the $rank(\boldsymbol{A}_1)$.

The code to return the number of non-zero diagonal terms in $\boldsymbol{R}$ of the QR decomposition is also in cw1/q2.py, and by running the script we see that the number of non-zero diagonal terms is 4, which is indeed $rank(\boldsymbol{A}_1)$

2.2 Part b

The ideas for this proof are taken from "Matrix Computations (3rd Ed.)" (Golub and Van Loan 1996)

Let $\boldsymbol{A} \in \mathbb{R}^{mxn}$, $m \geq n$ and $rank(\boldsymbol{A}) = r$.
At any iteration of transforming $\boldsymbol{A}$ to $\boldsymbol{R}$, the algorithm can be represented by the right multiplication of a permutation matrix $\boldsymbol{P}$, and the left multiplication of a householder reflection $\boldsymbol{H}$, where $\boldsymbol{P}$ is the identity when no columns are swapped.
After the $(k-1)^{th}$ iteration we can write the current iteration of $\boldsymbol{R}$ as:

$$\boldsymbol{R}^{(k-1)} = \boldsymbol{H}_{k-1}...\boldsymbol{H}_1\boldsymbol{A}\boldsymbol{P}_1...\boldsymbol{P}_{k-1}, \tag{1}$$

$$= \begin{bmatrix} \boldsymbol{R}_{11}^{(k-1)} & \boldsymbol{R}_{12}^{(k-1} \\ \boldsymbol{0} & \boldsymbol{R}_{22}^{(k-1)} \end{bmatrix}. \tag{2}$$

where $\boldsymbol{R}_{11}^{(k-1)}$ is $(k-1)\times(k-1)$ upper-triangular, and also invertible since the swapping in the algorithm ensures diagonal entries are non-zero.

At the k^{th} iteration we consider the norms of the columns of $\boldsymbol{R}_{22}^{(k-1)}$ to try and find a possible swap. If the max of these norms is 0, then it follows that all the columns of $\boldsymbol{R}_{22}^{(k-1)}$ are zero and thus $\boldsymbol{R}_{22}^{(k-1)} = \boldsymbol{0}$, and we have $\boldsymbol{R}$ in our desired form, where $\boldsymbol{R}_{11}^{(k-1)}$ is $(k-1)\times(k-1)$, and therefore $k-1 = r$ since it is invertible and therefore full-rank.

This algorithm terminates and returns $\boldsymbol{R}$ once the columns of $\boldsymbol{R}_{22}^{(k-1)}$ are all zero, and if this doesn't happen before the final iteration n, then we know that $rank(\boldsymbol{A}) = n$, and therefore $\boldsymbol{A}$ is full-rank in this case.

2.3 Part c

The code that implements the column swapping has been added to the **householder** function in cla_utils/exercise3.py. To run the householder algorithm with column swapping, set keyword argument "swaps=True" when calling the function. The pytests for this swapping implementation are located in cw1/test2.py and are called **test_householder_swap_1** and **test_householder_swap_2** respectively.

2.4 Part d

The code that implements the column swapping and returns $\boldsymbol{R}'$ given a tolerance of zero, s, has been added to the **householder** function in cla_utils/exercise3.py. To run this version of the householder algorithm, set keyword arguments "swaps=True" as well as "reduced_tol=s" when calling the function.
The pytest for testing this implementation to calculate $rank(\boldsymbol{A}_1)$ is in cw1/test2.py and is called **test_rank_from_householder_swap**.

2.5 Part e

The code used is located in cw1/q2.py, and the functions **create_rd_matrix** and **rd_ranks_and_errors** are used in the script at the bottom of q2.py that plots the graphs in Figure 5.

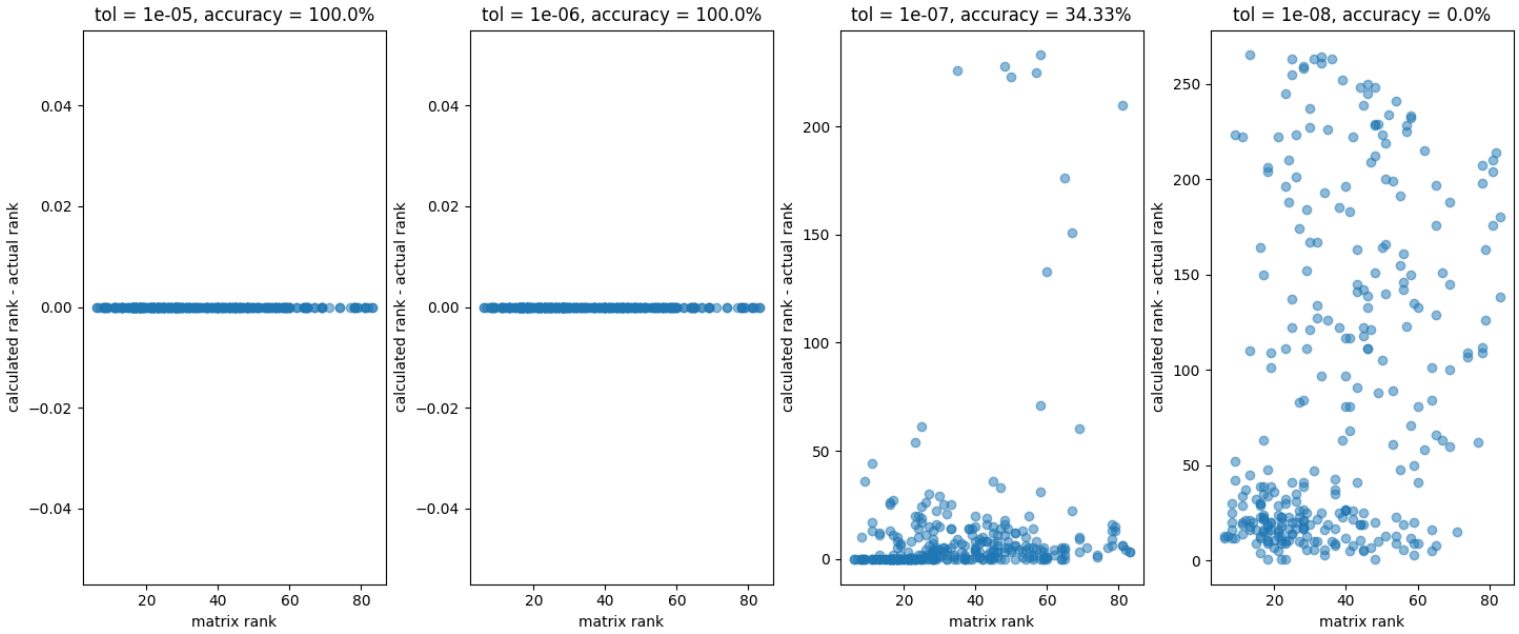

Figure 5: Scatter plots using varying values of reduced_tol, splotting rank error against actual rank, for 300 rank-deficient matrices

From Figure 5 we see that using reduced_tol values of 1e-05 and 1e-06 allowed us 100% accuracy in calculating the rank of the rank-deficient matrices. This is due to the fact that most vector norms we calculate that are supposed to be 0 tend to be fall

under these tolerances, and thus we treat them as 0 appropriately, and the correct sized $\boldsymbol{R}'$ is returned.

However, we can see that as we ever so slightly make the tolerance smaller down to 1e-07, we begin to start overestimating our rank, with only around 34% accuracy, and down to 1e-08, we lose all accuracy in our rank calculations and many ranks are wildly overestimated. This sudden increase in over-estimations is caused by the fact that now vectors we should consider to be 0 in the algorithm are now no longer being treated as 0, because the norms of these vectors just fall above 1e-07. This means that the algorithm does not terminate when it should, returning an $\boldsymbol{R}'$ that is taller, and therefore overestimating the rank.

From this investigation we can see the importance of picking a suitable tolerance in the algorithms success, and that a suitable tolerance to use that is not too small but also not too big is 1e-06.

2.6 Part f

I will use python indexing and numpy slicing conventions in the description below of the modified calculation algorithm i.e. 0^{th} entry is the first entry, $0:3 = \{0, 1, 2\}$

- For the 0^{th} iteration of the householder algorithm we calculate the squared norms of each column of the subject matrix, and store this in an array. Note that householder reflections preserve norm, so the norm of each column is preserved after each householder is applied. We can find the vector with max norm from our squared norm array, as the candidate vector with largest norm will also have the largest square norm. If a column swap is needed remember to also swap the corresponding entries in the squared norm array.

- Next, 1^{st} iteration, we set the 0^{th} entry in our norm squared array to 0, and subtract the square of the 0^{th} entry in column i for i in $1:n$ from the i^{th} entry in the squared norm array. Here only $(n-1)$ terms are required to be squared. Again we can use the squared norm array to help us locate a potential column swap, and remember to swap the corresponding entries in the squared norm array if a column swap is needed.

- Following this iteratively, by the time we reach the k^{th} iteration, our squared norm array should have 0's in the $0:k^{th}$ entries, and we subtract the square of the $(k-1)^{th}$ entry in column i for i in $k:n$ from the i^{th} entry in the squared norm array. As before, we locate the largest entry in the squared norm to find a potential column swap, and if there is one we swap the columns. Thus, at iteration k, we are only required to square $(n-k)$ terms to know the which norm is greatest of the candidate vectors and locate a potential column swap.

The **householder** function in cla_utils/exercises3 has been updated to use this modification of computing the candidate vector squared norms, and passes all the pytests from before.

3 QUESTION 3

3.1 Part a

First it is useful to introduce permutation matrix $\boldsymbol{P}$ where $\boldsymbol{P}$ is the square matrix with 1's on the anti-diagonal and 0's elsewhere:

$$\boldsymbol{P} = \begin{bmatrix} 0 & \dots & 0 & 1 \\ 0 & \dots & 1 & 0 \\ \vdots & \iddots & \vdots & \vdots \\ 1 & \dots & 0 & 0 \end{bmatrix} \tag{3}$$

Note that:

$$\boldsymbol{P}^2 = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{bmatrix} = \boldsymbol{I}, \quad \boldsymbol{P}^T = \begin{bmatrix} 0 & \dots & 0 & 1 \\ 0 & \dots & 1 & 0 \\ \vdots & \iddots & \vdots & \vdots \\ 1 & \dots & 0 & 0 \end{bmatrix} = \boldsymbol{P} \tag{4}$$

The left-multiplication of $\boldsymbol{P}$ with another square matrix $\boldsymbol{A}$ reverses the rows of $\boldsymbol{A}$, and the right-multiplication of $\boldsymbol{P}$ with another square matrix $\boldsymbol{A}$ reverses the columns of $\boldsymbol{A}$. Following the algorithm described we construct $\hat{\boldsymbol{A}}$ and find the QR decomposition of $\hat{\boldsymbol{A}}^T$:

$$\hat{\boldsymbol{A}} = \boldsymbol{P}\boldsymbol{A}, \tag{5}$$

$$\implies \hat{\boldsymbol{A}}^T = \boldsymbol{A}^T\boldsymbol{P}^T, \tag{6}$$

$$\implies \hat{\boldsymbol{A}}^T = \hat{\boldsymbol{Q}}\hat{\boldsymbol{R}}, \text{ where } \hat{\boldsymbol{Q}} \text{ orthogonal, } \hat{\boldsymbol{R}} \text{ upper-triangular.} \tag{7}$$

To get $\boldsymbol{Q}$, we need to transpose $\hat{\boldsymbol{Q}}$ and reverse the rows.

$$\boldsymbol{Q} = \boldsymbol{P}\hat{\boldsymbol{Q}}^T \tag{8}$$

We verify this $\boldsymbol{Q}$ is indeed orthogonal using properties of $\boldsymbol{P}$ and $\hat{\boldsymbol{Q}}$:

$$\boldsymbol{Q}^T\boldsymbol{Q} = (\boldsymbol{P}\hat{\boldsymbol{Q}}^T)^T(\boldsymbol{P}\hat{\boldsymbol{Q}}^T) = \hat{\boldsymbol{Q}}\boldsymbol{P}^T\boldsymbol{P}\hat{\boldsymbol{Q}}^T, \tag{9}$$

$$= \hat{\boldsymbol{Q}}\boldsymbol{P}^2\hat{\boldsymbol{Q}}^T = \hat{\boldsymbol{Q}}\hat{\boldsymbol{Q}}^T, \tag{10}$$

$$= \boldsymbol{I}, \text{ since } \hat{\boldsymbol{Q}} \text{ orthogonal.} \tag{11}$$

Also, (12)

$$\boldsymbol{Q}\boldsymbol{Q}^T = (\boldsymbol{P}\hat{\boldsymbol{Q}}^T)(\boldsymbol{P}\hat{\boldsymbol{Q}}^T)^T, = \hat{\boldsymbol{Q}}^T\boldsymbol{P}\boldsymbol{P}^T\hat{\boldsymbol{Q}}, \tag{13}$$

$$= \hat{\boldsymbol{Q}}^T \boldsymbol{P}^2 \hat{\boldsymbol{Q}} = \hat{\boldsymbol{Q}}^T \hat{\boldsymbol{Q}}, \tag{14}$$

$$= \boldsymbol{I}, \text{ since } \hat{\boldsymbol{Q}} \text{ orthogonal}, \tag{15}$$

$$\implies \boldsymbol{Q} \text{ orthogonal}. \tag{16}$$

To get $\boldsymbol{R}$, we need to reverse the rows of $\hat{\boldsymbol{R}}$, transpose, and then reverse the rows again.

$$\boldsymbol{R} = \boldsymbol{P}(\boldsymbol{P}\hat{\boldsymbol{R}})^T = \boldsymbol{P}\hat{\boldsymbol{R}}^T\boldsymbol{P}^T = \boldsymbol{P}\hat{\boldsymbol{R}}^T\boldsymbol{P}. \tag{17}$$

Starting with $\hat{\boldsymbol{R}}$, we tranpose this to get $\hat{\boldsymbol{R}}^T$:

$$\hat{\boldsymbol{R}} = \begin{bmatrix} r_{11} & r_{12} & \dots & r_{1n} \\ 0 & r_{22} & \dots & r_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & r_{nn} \end{bmatrix} \quad \rightarrow \quad \hat{\boldsymbol{R}}^T = \begin{bmatrix} r_{11} & 0 & \dots & 0 \\ r_{12} & r_{22} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ r_{1n} & r_{2n} & \dots & r_{nn} \end{bmatrix} \tag{18}$$

We then use the rules of left and right multiplying by $\boldsymbol{P}$ described previously to find an expression for $\boldsymbol{R}$:

$$\boldsymbol{P}\hat{\boldsymbol{R}}^T = \begin{bmatrix} r_{1n} & r_{2n} & \dots & r_{nn} \\ \vdots & \vdots & \iddots & \vdots \\ r_{12} & r_{22} & \dots & 0 \\ r_{11} & 0 & \dots & 0 \end{bmatrix} \quad \rightarrow \quad \boldsymbol{P}\hat{\boldsymbol{R}}^T\boldsymbol{P} = \begin{bmatrix} r_{nn} & \dots & r_{2n} & r_{1n} \\ \vdots & \ddots & \vdots & \vdots \\ 0 & \dots & r_{22} & r_{12} \\ 0 & \dots & 0 & r_{11} \end{bmatrix} \tag{19}$$

We see that $\boldsymbol{R} = \boldsymbol{P}\hat{\boldsymbol{R}}^T\boldsymbol{P}$ is indeed upper-triangular. Finally, we check $\boldsymbol{A} = \boldsymbol{RQ}$:

$$\boldsymbol{RQ} = (\boldsymbol{P}\hat{\boldsymbol{R}}^T\boldsymbol{P})(\boldsymbol{P}\hat{\boldsymbol{Q}}^T) = \boldsymbol{P}\hat{\boldsymbol{R}}^T\boldsymbol{P}^2\hat{\boldsymbol{Q}}^T = \boldsymbol{P}\hat{\boldsymbol{R}}^T\hat{\boldsymbol{Q}}^T, \tag{20}$$

$$= \boldsymbol{P}(\hat{\boldsymbol{Q}}\hat{\boldsymbol{R}})^T = \boldsymbol{P}(\hat{\boldsymbol{A}}^T)^T = \boldsymbol{P}\hat{\boldsymbol{A}} = \boldsymbol{P}(\boldsymbol{PA}) = \boldsymbol{P}^2\boldsymbol{A}, \tag{21}$$

$$= \boldsymbol{A}. \tag{22}$$

3.2 Part b

A function that computes the RQ decomposition of a matrix $\boldsymbol{A}$ is located in cw1/q3.py and is called **rq_decomposition**. The corresponding pytests are located in cw1/test3.py and are called by running **test_rq**.

3.3 Part c

The following algorithm, which is described in the paper "Generalized QR Factorization and Its Applications" (Anderson, Bai, and Dongarra 1992), can be used to compute the simultaneous factorisation:

- First, since $\boldsymbol{Q}^T\boldsymbol{B} = \boldsymbol{S} \implies \boldsymbol{B} = \boldsymbol{QB}$, we can see that we can get orthogonal $\boldsymbol{Q}$ and $\boldsymbol{S}$ of the form $\begin{bmatrix} \boldsymbol{S}_{11} \\ \boldsymbol{0} \end{bmatrix}$ by calculating the full QR decomposition of $\boldsymbol{B}$.

- Next, since $\boldsymbol{Q}^T\boldsymbol{A}\boldsymbol{U} = \boldsymbol{R} \implies \boldsymbol{Q}^T\boldsymbol{A} = \boldsymbol{R}\boldsymbol{U}^T$, we can use $\boldsymbol{Q}$ as calculated to work out the RQ decomposition of $\boldsymbol{Q}^T\boldsymbol{A} = \tilde{\boldsymbol{R}}\tilde{\boldsymbol{Q}}$, ensuring the full QR decomposition is used in the RQ algorithm.
- Taking $\boldsymbol{R} = \tilde{\boldsymbol{R}}$ and $\boldsymbol{U} = \tilde{\boldsymbol{Q}}^T$ we get $\boldsymbol{R} = \begin{bmatrix}\boldsymbol{0} & \boldsymbol{R}_{11}\end{bmatrix}$ and $\boldsymbol{U}$ orthogonal.

3.4 Part d

A function that computes the simultaneous factorisation of two matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ is located in cw1/q3.py and is called **simultaneous_fact**. The corresponding pytests are located in cw1/test3.py and are called by running **test_simultaneous_fact**.

3.5 Part e

The ideas for the method below to solve this constrained least squares problem are discussed in "Generalized QR Factorization and Its Applications" (Anderson, Bai, and Dongarra 1992).
Since $\boldsymbol{A}$ is an $m \times n$ matrix and $\boldsymbol{B}$ is a $p \times n$ matrix with $m \times n \geq p$, we can find the simultaneous factorisation of the $n \times m$ matrix $\boldsymbol{A}^T$ and $n \times p$ matrix $\boldsymbol{B}^T$.

$$\boldsymbol{Q}^T\boldsymbol{A}^T\boldsymbol{U} = \boldsymbol{R}, \quad \boldsymbol{Q}^T\boldsymbol{B}^T = \boldsymbol{S} \tag{23}$$

We can write:

$$\boldsymbol{R} = \begin{bmatrix}\boldsymbol{0} & \boldsymbol{R}_{11} & \boldsymbol{R}_{12} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{R}_{22}\end{bmatrix}, \quad \boldsymbol{S} = \begin{bmatrix}\boldsymbol{S}_{11} \\ \boldsymbol{0}\end{bmatrix}, \tag{24}$$

where $\boldsymbol{R}_{11}$ is $p \times p$, $\boldsymbol{R}_{12}$ is $p \times (n-p)$, $\boldsymbol{R}_{22}$ is $(n-p) \times (n-p)$, and $\boldsymbol{S}_{11}$ is $p \times p$.

We can also write:

$$\boldsymbol{Q} = \begin{bmatrix}\boldsymbol{Q}_1 & \boldsymbol{Q}_2\end{bmatrix}, \quad \boldsymbol{U} = \begin{bmatrix}\boldsymbol{U}_1 & \boldsymbol{U}_2 & \boldsymbol{U}_3\end{bmatrix}, \tag{25}$$

where $\boldsymbol{Q}_1$ $n \times p$, $\boldsymbol{Q}_2$ $n \times (n-p)$, $\boldsymbol{U}_1$ $m \times (m-n)$, $\boldsymbol{U}_2$ $m \times p$, and $\boldsymbol{U}_3$ $m \times (n-p)$.
Our constrained least-squares problem can be transformed:

$$\min_x ||\boldsymbol{A}x - b||, \text{ such that } \boldsymbol{B}x = d, \tag{26}$$

$$\min_x ||\boldsymbol{U}^T\boldsymbol{A}x - \boldsymbol{U}^T b||, \text{ such that } \boldsymbol{S}^T\boldsymbol{Q}^T x = d, \tag{27}$$

$$\min_x ||\boldsymbol{R}^T\boldsymbol{Q}^T x - \boldsymbol{U}^T b||, \text{ such that } \boldsymbol{S}^T\boldsymbol{Q}^T x = d. \tag{28}$$

Let $y = \begin{bmatrix}y_1 \\ y_2\end{bmatrix} = \boldsymbol{Q}^T x$, and $c = \begin{bmatrix}c_1 \\ c_2 \\ c3\end{bmatrix} = \boldsymbol{U}^T b$,
where y_1 is a p-vector, y_2 is an $(n-p)$-vector, c_1 is an $(m-n)$-vector, c_2 is a p-vector, and c_3 is an $(n-p)$-vector. Our problem now becomes:

$$\min_y ||\boldsymbol{R}^T y - c||, \text{ such that } \boldsymbol{S}^T y = d. \tag{29}$$

The constraint simplifies to $S_{11}^T y_1 = d$, which is solvable by forward-substitution since S_{11}^T is lower-triangular and non-singular from the assumptions in the question.
Using this d we can truncate the problem as an unconstrained least-squares problem:

$$\min_{y_2} \|\boldsymbol{R}_{22}^T y_2 - (c_3 - \boldsymbol{R}_{12}^T y_1)\|. \tag{30}$$

We can solve this new least-squares problem for y_2, and so by appending y_2 to y_1 and using $x = \boldsymbol{Q}y$ we can get our x_{LS} for the original problem.

3.6 Part f

A function that computes the solution x_{LS} to this constrained least-squares problem using the procedure outlined in Part e is located in cw1/q3.py and is called **solve_constrained_ls**. The corresponding pytests are located in cw1/test3.py and are called by running **test_ls_consraint** and **test_ls_error**.

References

Anderson, E, Zhaojun Bai, and J Dongarra (1992). "Generalized QR factorization and its applications". In: **Linear Algebra and its Applications** 162, pp. 243–271.

Golub, Gene H. and Charles F. Van Loan (1996). **Matrix Computations (3rd Ed.)** USA: Johns Hopkins University Press. ISBN: 0801854148.