AM213A Homework 4 Part 1 (Numerical Problems)

Owen Morehead

March 7, 2022

1 General Guidelines

Under the `part1` directory inside `hw4/code` contains all the fortran code associated with this assignment.
This includes:

- `LinAl.f90` program which contains all the subroutines.
- `Driver_foo.f90` driver program which calls all the subroutines and prints to the screen all the necessary results to this assignment.
- `utility.f90` which holds the variable, $dp = kind(0.e0)$.
- `Makefile` which is the makefile for this program.

The makefile is setup to compile the program such that the command `make LinAl` will make the executable that can then be run with the command, `./LinAl`. This will print all the desired matrices and information as asked in the assignment. It will respectively print to the screen the results for:

- Writing a program to reduce a symmetric matrix to tridiagonal form using Householder matrices for the similarity transformations (Reducing a matrix to **Hessenberg form**).
- Writing a program of the **QR algorithm** (i) **without shifts**, and (ii) **with shifts**, to calculate the eigenvalues of a given matrix.
- Writing a program of the **inverse iteration** to calculate the corresponding eigenvectors of a given matrix given provided eigenvalues.

A note is that all code executions are done using double precision floating point arithmetic. If one wants to change this do single precision, the step to take is in the `utility.f90` file, which is to simply change $dp = kind(0.d0)$ to $dp = kind(0.e0)$.

2 Reducing Symmetric Matrix to Tridiagonal Form

2.1 Algorithm Outline

Many linear algebra algorithms are much more computationally efficient when applied to triangular matrices. Putting a matrix in so-called **Hessenberg form** is an almost triangular matrix that

still makes linear algebra routines much more efficient. As further described below, if the original matrix is normal and nonsingular, the matrix output after this algorithm will be in **Hessenberg form**. If the original matrix is symmetric, the output matrix will be in **tridiagonal** form, which is a matrix that is both an upper and lower Hessenberg matrix. Achieving this matrix transformation can be done in a finite number of steps, for example, using Householder matrices for the similarity transformations as done here.
Putting a matrix in Hessenberg form consists of finding an orthogonal transformation $\mathbf{P}$ such that $\mathbf{A} = \mathbf{P}\mathbf{A}_H\mathbf{P}^T$ where $\mathbf{A}_H$ is of the form,

$$\mathbf{A}_H = \begin{bmatrix} * & * & \cdots & \cdots & * & * \\ * & * & \cdots & \cdots & * & * \\ 0 & * & \cdots & \cdots & * & * \\ 0 & 0 & \ddots & & * & * \\ \vdots & \vdots & \ddots & * & * & * \\ 0 & 0 & 0 & 0 & * & * \end{bmatrix} \tag{1}$$

where everything below the subdiagonal is zero. Otherwise said, this matrix has elements that are equal to 0 if their coefficients ij follows $i > j + 1$. Since $\mathbf{A} = \mathbf{P}\mathbf{A}_H\mathbf{P}^T$ is a similarity transformation, the eigenvalues of $\mathbf{A}_H$ are the same as $\mathbf{A}$. The eigenvectors are different, but they have only gone through a change of base which is easy to reverse if needed. As a result, finding the eigenvalues/eigenvectors of $\mathbf{A}$ is *almost* the same as finding the eigenvalues/eigenvectors of $\mathbf{A}_H$. And as noted, working on the tridiagonal matrix $\mathbf{A}_H$ is much more computationally efficient. In regards to the methods discussed here, the QR algorithm applied to $\mathbf{A}_H$ is much faster than the QR algorithm applied to $\mathbf{A}$. This has to do with the fact that the Hessenberg form is *preserved* by the QR algorithm.
We first note is that matrix formed in QR factorization, $\mathbf{R}$, is upper triangular. We also note that the $\mathbf{Q}$ matrix formed is also in Hessenberg form, as shown by the following,

$$(\mathbf{A}_H)_{ij} = \sum_{k=1}^{m} q_{ik} r_{kj} = \sum_{k=1}^{j} q_{ik} r_{kj}.$$

The only way to guarantee that $(\mathbf{A}_H)_{ij}$ if $i > j + 1$ is to have $q_{ik} = 0$ if $i > k + 1$.
Now we can also show that the reconstruction, $\mathbf{RQ}$ is also in Hessenberg form, since:

$$(\mathbf{RQ})_{ij} = \sum_{k=1}^{m} r_{ik} q_{kj} = \sum_{k=i+1}^{m} r_{ik} q_{kj}.$$

Since $\mathbf{Q}$ is in Hessenberg form, then $q_{kj} = 0$ if $k > j + 1$, and so $(\mathbf{RQ})_{ij} = 0$ if $i > j + 1$. Therefore $\mathbf{RQ}$ is also in Hessenberg form.

We follow with the fact that the Hessenberg form of a symmetric matrix is itself symmetric and therefore must be **tridiagonal**. Based on these results, we know that the QR algorithm applied to a tridiagonal matrix returns a tridiagonal matrix, thus increasing the computational speed of this QR algorithm since many of the matrix elements are zeros.

How to actually put $\mathbf{A}$ in Hessenberg form involves applying successive orthogonal transformations to $\mathbf{A}$ to zero out the elements below the subdiagonal until $\mathbf{A}$ is in Hessenberg form (recall that

the matrix $\mathbf{P}$ is orthogonal in $\mathbf{A} = \mathbf{P}\mathbf{A}_H\mathbf{P}^T$). Thus a method to acomplish this is to apply Householder transformations $\mathbf{H}_j$, where each is created to zero out sub-subdiagonal elements instead of subdiagonal ones:

$$\mathbf{A} \to \mathbf{H}_m \dots \mathbf{H}_1\mathbf{A} \equiv \mathbf{P}^T\mathbf{A}$$

Since we want to create a similarity transformation, we have to apply $\mathbf{P}$ on *both* sides of $\mathbf{A}$, which implies,

$$\mathbf{A}_H = \mathbf{P}^T\mathbf{A}\mathbf{P} = \mathbf{H}_m \dots \mathbf{H}_1\mathbf{A}(\mathbf{H}_m \dots \mathbf{H}_1)^T \tag{2}$$

$$= \mathbf{H}_m \dots \mathbf{H}_1\mathbf{A}\mathbf{H}_1^T \dots \mathbf{H}_m^T = \mathbf{H}_m \dots \mathbf{H}_1\mathbf{A}\mathbf{H}_1 \dots \mathbf{H}_m \tag{3}$$

since each $\mathbf{H}_j$ is symmetric. An important thing to note is that because of the form of the Householder transformations, applying $\mathbf{H}_j$ to the right side of $\mathbf{A}$ at each iteration does not repopulate the elements that just got zeroed out by applying $\mathbf{H}_j$ to the left.

So to zero out the sub-subdiagonal elements in the j-th column, we first compute,

$$s_j = \text{sign}(a_{j+1,j})\sqrt{\sum_{k=j+1}^{m} a_{kj}^2}$$

and then create the Householder vector $\mathbf{v}_j = (0, \dots, 0, a_{j+1,j} + s_j, a_{j+2,j}, \dots, a_{mj})^T$ and normalize it. And finally, we create $\mathbf{H}_j = \mathbf{I} - 2\mathbf{v}_j\mathbf{v}_j^T$.
Since the first j elements of $\mathbf{v}_j$ are all zeros, this means that the first j rows and columns of $\mathbf{H}_j$ are equal to the first j rows and columns of the identity matrix. Because of this, applying $\mathbf{H}_j$ to the left of $\mathbf{A}$ only affects rows $j+1$ and up, and applying it to the right only affects columns $j+1$ and up. This therefore leaves the rows and columns we have already worked on as they are. This is the bulk of this algorithm.

In summary, we first compute s_j, then form $\mathbf{v}_j$ and normalize it, then update $\mathbf{A}$ from the left ($\mathbf{A} = \mathbf{A} - 2\mathbf{v}_j\mathbf{v}_j^T\mathbf{A}$), and then update $\mathbf{A}$ from the right ($\mathbf{A} = \mathbf{A} - 2\mathbf{A}\mathbf{v}_j\mathbf{v}_j^T$). The main difference between this algorithm and the Householder algorithm for the QR decomposition is that for this algorithm, we start from the subdiagonal element $a_{j+1,j}$ in the sum involved in s_j, and in the vector $\mathbf{v}_j$ instead of the diagonal element a_{jj}, as well as we now also apply $\mathbf{H}_j$ from the right.

On exit, $\mathbf{A}$ will be in Hessenberg form if $\mathbf{A}$ is any normal nonsingular matrix, and in tridiagonal form if $\mathbf{A}$ is symmetric. There are both upper and lower Hessenberg matrix forms and they can be either unreduced (where the sub-subdiagonal elements are all zero), or not unreduced (where there can be zeros in the sub-subdiagonal rows). A tridiagonal matrix is one that is both lower and upper Hessenberg matrix. As we will see below, the original matrix provied is symmetric, so the output matrix will be tridiagonal, and is also in the not unreduced form since there is a zero in the above/below-subdiagonals.

Also note that if we want to compute the eigenvectors, we will have to create $\mathbf{P}$ on the side and then save it. This would simply be done by initializing $\mathbf{P} = \mathbf{I}$, then add a line, $\mathbf{P} = \mathbf{H}_j\mathbf{P}$, at the end of the algorithm. This can be done efficiently by storing the sub-subdiagonal elements of the Householder vectors in the place of the zeros, and store the subdiagonal elements of these vectors in a separate array.

Ultimately, reducing A to a Hessenberg matrix means that only $\mathcal{O}(n^2)$ operations per iteration are required instead of $\mathcal{O}(n^3)$ in further computing the eigenvalues of a symmetric matrix by means of, say the QR algorithm as what follows in the next section.

2.2 Numerical Results

The (4×4) matrix we wish to reduce to tridiagonal form is:

$$\mathbf{A} = \begin{bmatrix} 5 & 4 & 1 & 1 \\ 4 & 5 & 1 & 1 \\ 1 & 1 & 4 & 2 \\ 1 & 1 & 2 & 4 \end{bmatrix} \tag{4}$$

The resulting tridiagonal form is:

$$\mathbf{A} = \begin{bmatrix} 5.0 & -4.24264 & 3.14 \times 10^{-16} & 4.93 \times 10^{-32} \\ -4.24264 & 6.0 & 1.41421 & 2.22 \times 10^{-16} \\ 3.14 \times 10^{-16} & 1.41421 & 4.9999 & 2.22 \times 10^{-15} \\ 4.93 \times 10^{-32} & 3.33 \times 10^{-16} & 1.11 \times 10^{-15} & 1.9999 \end{bmatrix} \tag{5}$$

and the simplified form would look like:

$$\mathbf{A} = \begin{bmatrix} 5.0 & -4.24264 & 0 & 0 \\ -4.24264 & 6.0 & 1.41421 & 0 \\ 0 & 1.41421 & 5.0 & 0 \\ 0 & 0 & 0 & 2.0 \end{bmatrix} \tag{6}$$

which we see is in *not unreduced* tridiagonal form, with a zero entry in the subdiagonal and superdiagonal. In this matrix form, numerical eigenvalue problems will be cheaper and better conditioned than dealing with the full matrix.

3 QR Algorithm To Calculate Matrix Eigenvalues

3.1 QR Algorithm Without Shift

3.1.1 Algorithm Outline

The QR algorithm is a common method for solving small eigenvalue problems. It is equivalent in all respects to the Simulatneous Iterations algorithm, that which is a combination of the QR algorithm and the Power Iteration algorithm. These algorithms let us find all the eigenvalues and eigenvectors of a given matrix at the same time. This QR algorithm without shifts can be viewed as a stable procedure for computing QR factorizations of the matrix powers $\mathbf{A}, \mathbf{A}^2, \mathbf{A}^3, \ldots$.

The QR algorithm without shifts is as simple as it gets. We first just compute a QR factorization of $\mathbf{A}$ ($\mathbf{QR} = \mathbf{A}$), then recompute the product[2], switching $\mathbf{R}$ and $\mathbf{Q}$ ($\mathbf{A} = \mathbf{RQ}$). Then we repeat this process with the new matrix formed until our error is small enough. We can interpret *until the error is small enough* in a variety of ways. The method implemented here is to compute the eigenvalue estimate at step n, and then again at step $n+1$, and compare these values. Once the difference is smaller than our desired tolerance (e.g. 1×10^{-12}), we will be left with our full precision eigenvalues on the diagonal and hence any further iterations of the algorithm are not necessary. We can thus terminate the loop and are now left with our desired eigenvalues along the diagonal of our newly formed matrix.

This algorithm gradually transforms the initial matrix $\mathbf{A}$ into a diagonal matrix containing the eigenvalues in its diagonal, ordered in norm from largest to smallest. In addition, the matrix $\mathbf{Q}$ of

the QR factorization of $\mathbf{A}$ gradually transforms into the matrix that contains all orthogonal and normalized eigenvectors of $\mathbf{A}$. If we want to calcualte the eigenvectors, the extra line of code we want to apply to our algorithm would look like, $\mathbf{V} = \mathbf{VQ}$, where we are the eigenvector matrix the same amount of times we update our eigenvalue matrix.

3.1.2 Numerical Results

The (3×3) matrix we wish to calculate the eigenvalues of is:

$$\mathbf{A} = \begin{bmatrix} 3 & 1 & 0 \\ 1 & 2 & 1 \\ 0 & 1 & 1 \end{bmatrix} \tag{7}$$

and notice that this matrix is already in tridiagonal form, with zeros above the superdiagonal and below the subdiagonal.

The updated matrix $\mathbf{A}$ holding the eigenvalues in its diagonal is:

$$\mathbf{A} = \begin{bmatrix} 3.73205 & 7.447 \times 10^{-7} & 2.0707 \times 10^{-16} \\ 7.447 \times 10^{-7} & 2.0 & -4.0293 \times 10^{-16} \\ -9.778 \times 10^{-42} & 1.0583 \times 10^{-20} & 0.267949 \end{bmatrix} \tag{8}$$

in which we see the eigenvalues of $\mathbf{A}$ are:

- $\lambda_1 = 2 + \sqrt{3} = 3.732050808$
- $\lambda_2 = 2.0$
- $\lambda_3 = 2 - \sqrt{3} = 0.2679491924$

In addition to the correct eigenvalues being calculated, we also notice a difference in the magnitudes of the off-diagonal elements, that which are all close to zero. Essentially, this difference in magnitude is due to the fact that these elements are meant to be zero, but not all relative to the exact same thing. This also has to do with the specific convergence criteria designed for this algorithm. Here, this convergence is chosen to only rely on the diagonal elements as discussed above. Although all the eigenvalues are revealed simultaneously, they are not all revealed with the same convergence rate. Thus as a consequence, some elements of the sub and superdiagonal are processed more than others, and decay relative to different things. We also keep in mind that in double precision arithmetic, 1×10^{-16} is the *relative* precision. Thus, floating point arithmetic is the standard rather than fixed point arithmetic, and so even though some of the off diagonal elements are much larger than others (7×10^{-7} compared to -9×10^{-42}), we cannot directly compare these values since they are meant to be zero *relative* to the magnitude of different values. All in all, the results obtained are satisfactory and indeed the correct eigenvalues are found using this QR algorithm without shift. However, we can speed up the convergence of the algorithm if we introduce *shift* as done below.

3.2 QR Algorithm With Shift

3.2.1 Algorithm Outline

The QR algorithm for finding eigenvalues becomes more practical when we add in what is called *shift*. As previously described, the convergence rate of the QR algorithm is determined by the ration

of the first to next eigenvalues after iterations. If this ration is large, convergence is rapid, but if this ratio is close to one, convergence is much slower. The principle behind the QR algorithm with shifts derives from an important property of the Simultaneous Iterations Algorithm (which is equivalent to the QR algorithm). This principle is described by the Theorem that states:

The Simultaneous Iterations Algorithm applied to an initially ideneity matrix $\mathbf{I}$ is equivalent to an *unshifted simultaneous inverse power iteration* (i.e. a simultaneous power iteration of $\mathbf{A}^{-1}$) on a reverse permutation of the identity matrix $\mathbf{P}$, where $\mathbf{P}$ is a back-to-front identity matrix.

As previously noted, for the QR algorithm without shift, we noticed that the first and last eigenvalues appear to converge faster than the middle ones. This is something we can now deal with by using this QR algorithm with shifts. In essence, we can now shift the inverse iteration on the last column vector to accelerate its convergence greatly, without affecting the convergence of the first eigenvalue/vector by much. This leads to the two steps of this algorithm, that which are the same as without shift, but including the shifted portion. First we choose our shift, $\mu = a_{mm}$, then we compute a QR factorization of $\mathbf{A} - \mu\mathbf{I}$. And lastly, we replace $\mathbf{A}$ by $\mathbf{RQ} + \mu\mathbf{I}$. And as the algorithm without shift, we do these three steps inside a while loop that continues to run until the error is smaller than our provided tolerance. Again, the error here being the difference between eigenvalue estimates at steps n and $n + 1$.

So this algorithm continuously performs one QR factorization of the shifted matrix $\mathbf{A}$, effectively corresponding to one step of shifted inverse simultaneous power iteration. Then, $\mathbf{A}$ gets reconstructed as $\mathbf{RQ} + \mu\mathbf{I}$, thereby canceling the shift. This last step is crucial, otherwise the matrix $\mathbf{A}$ would not converge to our diagonal matrix of the eigenvalues, $\mathbf{D}_\lambda$.

In addition, we note that reconstructing the eigenvectors entails the same step as for the unshifted QR algorithm. Also, as goes for the unshifted algorithm, there are some matrices for which both of these algorithms do not converge. For this reason, other shifting strategies are sometimes used (e.g. the Wilkinson. shift, or multiple shift strategies, etc...). But ultiamtely, many symmetric matrices used in application can be calculated from this QR algorithm.

3.2.2 Numerical Results

The (3×3) matrix we wish to calculate the eigenvalues of is the same as from the above section, calculating the eigenvalues from the QR algorithm without shift.

The updated matrix $\mathbf{A}$ holding the eigenvalues in its diagonal is, again, almost the exact same as the QR algorithm without shift in the above section. Differences in the values only start to appear in the 12th decimal place for the largest eigenvalue. We are left with the updated matrix $\mathbf{A}$:

$$\mathbf{A} = \begin{bmatrix} 3.73205 & -1.1762 \times 10^{-16} & 2.8598 \times 10^{-16} \\ 4.9934 \times 10^{-50} & 1.9999 & 1.0447 \times 10^{-16} \\ -4.9456 \times 10^{-25} & -6.2744 \times 10^{-22} & 0.267949 \end{bmatrix} \tag{9}$$

so we see both methods return the same eigenvalues up to round-off errors:

- $\lambda_1 = 2 + \sqrt{3} = 3.732050808$
- $\lambda_2 = 2.0$

- $\lambda_3 = 2 - \sqrt{3} = 0.2679491924$.

We also find that this algorithm converges in **7 iterations** compared to the **23** it takes to converge the unshifted QR algorithm. It is also to note that as expected for inverse iteration with a shift close to an eigenvalue, the smallest eigenvalue has been determined to the full accuracy. What is now shown in this final matrix here is the step that is taken after determining the last eigenvalue to the full accuracy. What we can then do is reduce the problem to the $m-1 \times m-1$ submatrix for further iterations. And because these diagonal entries are already very close to the eigenvalues, only a few additional iterations are required to obtain full accuracy for these remaining eigenvalues. This now leaves us with the final matrix shown in Equation 9. All of the non-diagonal elements are smaller than 1×10^{-16}, and we are left with the same eigenvalues as calculated from the unshifted QR algorithm, except found in only 7 iterations.

4 Inverse Iteration Method to Calculate Matrix Eigenvectors

4.1 Algorithm Outline

The Inverse Iteration (inverse power method) is an iterative eigenvalue algorithm, allowing one to find an approximate eigenvector when an approximate corresponding eigenvalue is already known. This method addresses a couple of the problems of the Power Iteration. It can find the eigenvectors other than the one corresponding to the fastest-growing eigenvalue and with an accelerated convergence rate. However, we have to have an estimate of the eigenvalue corresponding to the eigenvector we are looking for.

This Inverse Iteration algorithm is based on the realization that for any μ that is not exactly equal to one of the eigenvalues, then

- The eigenvectors of $(\mathbf{A} - \mu\mathbf{I})^{-1}$ are the same as the eigenvectors of $\mathbf{A}$
- The eigenvalues of $(\mathbf{A} - \mu\mathbf{I})^{-1}$ corresponding to each eigenvector $\mathbf{v}_i$ are $(\lambda_i - \mu)^{-1}$, where λ_i is the eigenvalue of $\mathbf{A}$ corresponding to $\mathbf{v}_i$

These results are advantageous to this algorithm. Suppose we have a rough estimate μ of any arbitrary eigenvalue λ_i (as opposed to the largest λ_1 for the Power method). Then, by construction, $\frac{1}{\lambda_i - \mu}$ is the largest eigenvalue of $(\mathbf{A} - \mu\mathbf{I})^{-1}$. We can then do a Power Iteration on the matrix $\mathbf{B} = (\mathbf{A} - \mu\mathbf{I})^{-1}$ to find the corresponding eigenvector $\mathbf{v}_i$, which happens to be an eigenvector of both $\mathbf{A}$ and $(\mathbf{A} - \mu\mathbf{I})^{-1}$.

Thus, this Inverse Iteration algorithm is as follows. We first initialize FF approximate (or exact) eigenvalue, μ, then we form $\mathbf{B} = (\mathbf{A} - \mu\mathbf{I})^{-1}$. However, this matrix inverse will be done in multiple steps. We first form the matrix $\mathbf{B}$ but that is not the inverse, $(\mathbf{A} - \mu\mathbf{I})$ and then solve against this by factorizing it as, for example, LU or by Gaussian Elimination. The significance of pulling this factorization step outside a loop is that this matrix will never change (for fixed μ). The factorization factors are independent of $\mathbf{x}$, so we only need to form them once. This step is thus in the form of solving a system of linear equations, $(\mathbf{A} - \mu\mathbf{I})\mathbf{y} = \mathbf{x}$, where we only first perform the factorization. The forward/backward substitution happens inside the following loop.

Next, we perform a while loop that first calculates the new normalized vector, $\mathbf{y} = \frac{\mathbf{Bx}}{||\mathbf{Bx}||}$. As per the factorization that happened outside this loop, all we need to do now is perform a backsubstitution

routine to solve for $\mathbf{y} = (\mathbf{A} - \mu\mathbf{I})^{-1}\mathbf{x}$. This forward/backward (forward if using LU factorization) does depend on $\mathbf{x}$, so it must stay inside the loop.

Then, we calculate the difference between the new and old vectors, $\mathbf{r} = \mathbf{y} - \mathbf{x}$, and lastly, we replace the old vector with the new one, $\mathbf{x} = \mathbf{y}$. This loop will run until $||\mathbf{r}||$ is as small as our desired accuracy. That is, the norm of the difference between the old and new vectors is small enough such that no further iterations are needed and we now have our desired eigenvector results. The only main disadvantage of this algorithm is that it requires a good estimate of a certain eigenvalue to obtain good convergence on its corresponding eigenvector.

4.2 Numerical Results

The (4×4) matrix we wish to calculate the eigenvectors of is:

$$\mathbf{A} = \begin{bmatrix} 2 & 1 & 3 & 4 \\ 1 & -3 & 1 & 5 \\ 3 & 1 & 6 & -2 \\ 4 & 5 & -2 & -1 \end{bmatrix} \tag{10}$$

which has four approximate eigenvalues: $\lambda_1 = 8.0286, \lambda_2 = 7.9329, \lambda_3 = 5.6689, \lambda_4 = 1.5732$.

The resulting four eigenvectors found for each each eigenvalue are as follows (note the last eigenvalue is normalized to 1):

$$\lambda_1 = -8.0286 \quad \mathbf{v}_1 = \begin{bmatrix} -0.38998 \\ -0.97553 \\ 0.29550 \\ 1.0 \end{bmatrix}$$

$$\lambda_2 = 7.9329 \quad \mathbf{v}_2 = \begin{bmatrix} 2.86698 \\ 1.08319 \\ 3.97547 \\ 1.0 \end{bmatrix}$$

$$\lambda_3 = 5.6689 \quad \mathbf{v}_3 = \begin{bmatrix} 0.573647 \\ 0.548946 \\ -0.81482 \\ 1.0 \end{bmatrix}$$

$$\lambda_4 = -1.5732 \quad \mathbf{v}_4 = \begin{bmatrix} -2.608695 \\ 2.366299 \\ 0.98503 \\ 1.0 \end{bmatrix}$$

We can indeed confirm with any online calculator that these are the correct eigenvectors corresponding to each eigenvalue up to round-off errors.