Final Project

Introduction

For this project, I will be exploring the topic of finding eigenvalues by using the QR algorithm. There are many interesting real world applications of the QR algorithm, including Google's page rank algorithm to traverse between web pages which features a stochastic matrix where each web page is assigned to a row and column, and also in various fields of data science, like in principal component analysis, in which factorization can be utilized to analyze the vibration levels in industrial machines. In the fundamental sense, in order to calculate the eigenvalue of a matrix, one has to compute the characteristic equation, which requires you to calculate the determinant. This can be a relatively expensive computation. Through Gaussian Elimination, you will find the determinant by performing a sequence of row operations. Row reducing an $n \times n$ matrix of integers alone already takes $O(n^3)$. Another issue associated with calculating eigenvalues is that solving the characteristic equation involves computing the roots of a polynomial, which can be a difficult problem. Finding an efficient root finding algorithm becomes part of the problem then. The technique of Newton's method can be used, but the challenge comes from having to choose a set of proper initial points, which could be complex, in order to influence the algorithm to converge to all the roots.

With all of this in mind, how are we to remedy these setbacks for finding the eigenvalues of a matrix? This is where the QR algorithm comes in.

The QR algorithm negates the issue of having to calculate the determinant of the matrix by implementing a Schur decomposition of the original matrix instead. You are also not required to solve for the roots of a polynomial. The process of factoring the original matrix and reverse multiplying the components will in turn convert the original matrix into one where the diagonal entries are the eigenvalues. The reason why the QR algorithm is desired is that if you reduce the original matrix into, say, a Hessenberg matrix, you can achieve a time complexity of $O(n^2)$ operations per iteration as opposed to $O(n^3)$ from before.

Theory

By definition, an eigenvector of an $n \times n$ matrix A is a *nonzero* vector x in R^n where some scalar value λ exists such that $Ax = \lambda x$. The corresponding eigenvalue of A is the scalar multiple λ. We then have the eigenpair, denoted as (λ, x), which is an ordered pair of an eigenvalue and its corresponding eigenvector. In essence, eigenvalues are the *roots* of polynomials of degree n, so there are n eigenvalues for an $n \times n$ matrix (though some roots may be repeated). Thus, the eigenvectors are the diagonals of the reduced matrix. Some issues that we might find when calculating eigenvectors numerically would be the fact that there are infinitely many eigenvalues for the system. Note that if the matrix is above 4×4 dimensions, then the problem of finding eigenvalues and their eigenvectors becomes nearly impossible.

A special case of the finding eigenvalues and eigenvectors problem is when the matrix A is called a symmetric tridiagonal matrix. Let us define what that is.

Definition:

Let A be our $n \times n$ matrix. A is a symmetric tridiagonal matrix if it is in the form

$$\begin{bmatrix} \alpha_1 & \beta_1 & & & \\ \beta_1 & \alpha_2 & \beta_2 & & \\ & \beta_2 & \alpha_3 & \ddots & \\ & & \ddots & \ddots & \beta_{n-1} \\ & & & \beta_{n-1} & \alpha_n \end{bmatrix},$$

where $A_{n,n} = \alpha_n$ and $A_{n+1,n} = A_{n,n+1} = \beta_n$ and the rest of the entries are zeroes.

Definition:

Let A and B be a matrix, then A and B are similar if there exists some kind of transformation from A to B such that

$$A = Q^{-1}BQ, \qquad (1)$$

where Q is an invertible matrix.

This specific transformation is called a similarity transformation. Thus, if A and B are similar, then they will have the same eigenvalues.

Definition:

A square n × n matrix A with real numbers or elements is an orthogonal matrix if its transpose A^T is equal to its inverse matrix I,

i.e. $A^T = A^{-1}$.

Or we can say when the product of a square matrix and its transpose gives an identity matrix, then the square matrix is known as an orthogonal matrix,

i.e. $AA^T = A^TA = I$.

For the QR method, we will be decomposing our matrix A into two separate matrices, Q and R, where Q is an *orthogonal* matrix and R is an *upper triangular* matrix. So,

$$A = QR.$$

Essentially, the QR method will decompose our matrix A further by a number of iterations.

Let A_k be our matrix A during the kth iteration of our QR method. Then A can be decomposed into

$$A_k = Q_kR_k.$$

Now, we are going to have to calculate the matrix for the next iteration A_{k+1}. Since we want to preserve the eigenvalues of the original matrix A throughout our iterations, we can utilize a nice property of orthogonal matrices, which says that

$$Q^{-1} = Q^T.$$

So, we can use the first definition (1) and substitute to decompose our matrix A into

$$A_{k+1} = R_kQ_k = Q_k^{-1}Q_kR_kQ_k = Q_k^{-1}A_kQ_k$$

By using a similarity transformation, matrix A_k and A_{k+1} will have the same eigenvalues. Eventually, by using the QR method, you will converge to the n × n matrix A_k where

$$A_k = R_kQ_k = \begin{matrix} [& \boldsymbol{\lambda} & X & \dots & X \\ & 0 & \boldsymbol{\lambda} & \dots & X \end{matrix}$$

0 0 ….

0 0 …. $\boldsymbol{\lambda}_n$]

All in all, QR decomposition is an expensive method to find the eigenvalues of a matrix, which runs for $O(n^3)$. The off diagonal entries will converge to 0 at a rate of $O(|\boldsymbol{\lambda}_j/\boldsymbol{\lambda}_{j-1}|)$.

The most integral aspect about the QR algorithm is the QR factorization, which will be done using *rotational matrices*.

Definition:

Let $P_{(i,\, j)}$, where $i < j$, denote an orthogonal matrix that is structured identically to the identity matrix except the elements where

$$p_{i,i} = p_{j,j} = \cos\theta \text{ and } p_{i,j} = -p_{j,i} = \sin\theta,$$

for some angle θ.

$P_{(i,\, j)}$ is a rotational matrix.

Thus, our factorization of matrix A will be the premultiplication of A by the rotational matrix $P_{(i,j)}$ where A is of the form

$$A^{(i)} = \begin{bmatrix} a_1 & b_1 & & & & & \\ b_1 & a_2 & b_2 & & & & \\ & b_2 & a_3 & b_3 & & & \\ & & \cdot & \cdot & \cdot & & \\ & & & \cdot & \cdot & \cdot & \\ & & & & \cdot & \cdot & \cdot \\ & & & & b_{n-2} & a_{n-1} & b_{n-1} \\ & & & & & b_{n-1} & a_n \end{bmatrix}$$

and the equation $P_{(i,j)}A$ = factorized(A) is of the form

$$\begin{bmatrix} c_1 & s_1 & & \\ -s_1 & c_1 & & \\ & & 1 & \\ & & & \ddots \end{bmatrix} \begin{bmatrix} a_1 & b_1 & & & \\ b_1 & a_2 & b_2 & & \\ & b_2 & a_3 & b_3 & \\ & & \ddots & \ddots & \ddots \end{bmatrix}$$

$$= \begin{bmatrix} a_1c_1 + b_1s_1 & b_1c_1 + a_2s_1 & b_2s_1 & & \\ -a_1s_1 + b_1c_1 & -b_1s_1 + a_2c_1 & b_2c_1 & & \\ & b_2 & a_3 & b_3 & \\ & & \ddots & \ddots & \ddots \end{bmatrix},$$

where c_j and s_j are the shorthand notation for cosine and sine respectively for the rotational matrix $P_{(j,\, j+1)}$.

The goal is to make the element under each diagonal element equal to 0 in the factored form of A. In each iteration, we can sub in these values of c_j and s_j.

For $j = 2, 3, 4, \ldots, n - 1$, we have

$$c_j = \frac{a_j}{\sqrt{a_j^2 + t^2}} \quad \text{and} \quad s_j = \frac{t}{\sqrt{a_j^2 + t^2}},$$

where t is the old value of element b_j.

By doing this process, knowing these values will help us derive the next iteration of A^i (i.e. A^{i+1}) in the QR method without directly computing the subsequent R^i and Q^i matrices.

Remember that R in the QR factorization is an upper triangular matrix. This triangular matrix $R^{(i)}$ of the matrix $A^{(i)}$ is

$$R^{(i)} = P_{(n-1,n)} \cdots P_{(3,4)} P_{(2,3)} P_{(1,2)} A^{(i)} \qquad (2)$$

If we have that

$$Q^{(i)T} A^{(i)} = R^{(i)}, \qquad (3)$$

then if we substitute (2) into (3),

$$Q^{(i)T} A^{(i)} = P_{(n-1,n)} \cdots P_{(3,4)} P_{(2,3)} P_{(1,2)} A^{(i)}$$
$$Q^{(i)T} = P_{(n-1,n)} \cdots P_{(3,4)} P_{(2,3)} P_{(1,2)}$$
$$Q^{(i)} = P^T_{(1,2)} P^T_{(2,3)} P^T_{(3,4)} \cdots P^T_{(n-1,n)}.$$

It is not necessary to directly compute the matrix $Q^{(i)}$, but it is sufficient to keep track of the s_j and c_j entries for each of the rotation matrices $P_{(j,j+1)}$ based on the relations given by

$$\begin{matrix} i\text{th column of} \\ MP^T_{(i,j)} \end{matrix} = \cos\theta \cdot \begin{matrix} i\text{th column} \\ \text{of } M \end{matrix} + \sin\theta \cdot \begin{matrix} j\text{th column} \\ \text{of } M \end{matrix}$$

and

$$\begin{matrix} j\text{th column of} \\ MP^T_{(i,j)} \end{matrix} = -\sin\theta \cdot \begin{matrix} i\text{th column} \\ \text{of } M \end{matrix} + \cos\theta \cdot \begin{matrix} j\text{th column} \\ \text{of } M \end{matrix}.$$

This leads to the matrix multiplication of RQ, which takes the form

$$\begin{bmatrix} a_1 & b_1 & e_1 & & & \\ & a_2 & b_2 & e_2 & & \\ & & a_3 & b_3 & e_3 & \\ & & & \ddots & \ddots & \ddots \end{bmatrix} \begin{bmatrix} c_1 & -s_1 & & \\ s_1 & c_1 & & \\ & & 1 & \\ & & & \ddots \end{bmatrix}$$
$$= \begin{bmatrix} a_1c_1 + b_1s_1 & -a_1s_1 + b_1c_1 & e_1 & & \\ a_2s_1 & a_2c_1 & b_2 & e_2 & \\ & & a_3 & b_3 & e_3 \\ & & & \ddots & \ddots & \ddots \end{bmatrix}.$$

Lastly, we will set a_j, b_j, and a_{j+1} as follows:

$$a_j = a_jc_j + b_js_j,$$
$$b_j = a_{j+1}s_j,$$
$$a_{j+1} = a_{j+1}c_j$$

for j = 1, 2, 3, n - 1. The e_j term represents the values that we did not save from $R^{(i)}$ during the factorization process. As we can see, it is actually not used in our computations.

For our QR algorithm, the convergence rate of the subdiagonal entries to zero without using a shift are

$$|\lambda_{p+1} / \lambda_p|,$$

where λ_p is the pth largest eigenvalue of the matrix. This can be an issue when eigenvalues have magnitudes that are close to one another.

However, if we were to apply a shift on the original matrix, then we can effectively accelerate the convergence. After integrating a shift σ into the QR algorithm, the convergence rate becomes

$$|\lambda_{p+1} - \sigma| / |\lambda_p - \sigma|.$$

This has it so that if σ is close to an eigenvalue, then convergence of the subdiagonal entry is much more rapid.

We may choose the shift however we wish. Though, the two most commonly used shifts for σ_i are

$$(1)\ \sigma_i = a_n^{(i)}, \text{ or}$$

$$(2)\ \sigma_i = \text{the eigenvalue of } \begin{bmatrix} a_{n-1}^{(i)} & b_{n-1}^{(i)} \\ b_{n-1}^{(i)} & a_n^{(i)} \end{bmatrix} \text{ that is closest to } a_n^{(i)}$$

,

where i is the iteration in the algorithm, $a_n^{(i)}$ denotes the nth diagonal element of n × n matrix A, and $b_{n-1}^{(i)}$ is the n-1th subdiagonal element.

The reason why we apply this shift is to make the $b_{n-1}^{(i+1)}$ subdiagonal element converge to zero the fastest. We will be checking this specific element's value each time at the end of every iteration of the algorithm. If its value is lower than some specified *convergence tolerance*, then the next diagonal element $a_n^{(i+1)}$ is determined to be an eigenvalue of the original matrix $A^{(0)}$. Furthermore, we can calculate the value of that eigenvalue as

$$a_{n-1}^{(i+1)} + \Sigma,$$

where Σ is the total number of shifts accumulated from the first iteration to the current one. This process is repeated until $b_1^{(i+1)}$, in which $a_2^{(i+1)} + \Sigma$ and $a_1^{(i+1)} + \Sigma$ are the last two eigenvalues of $A^{(0)}$.

All in all, the QR algorithm is one of the most important algorithms for eigenvalue computation. However, one limitation of the QR algorithm is that it can only be applied to dense or full matrices only, which is a matrix where most elements are nonzero. Another limitation of the algorithm is that there is a transformation step before the algorithm itself, to transform the matrix into the correct form, if the matrix we are evaluating is not already in either one of two forms: the Hessenberg form, which is a variant of triangular matrices, and the Hermitian or symmetric form, which is called a tridiagonal matrix.

Homework Problems

1. Let our symmetric tridiagonal matrix A be

$$\begin{bmatrix} 2 & -1 & 0 \\ -1 & 6 & 1 \\ 0 & 1 & -3 \end{bmatrix}$$

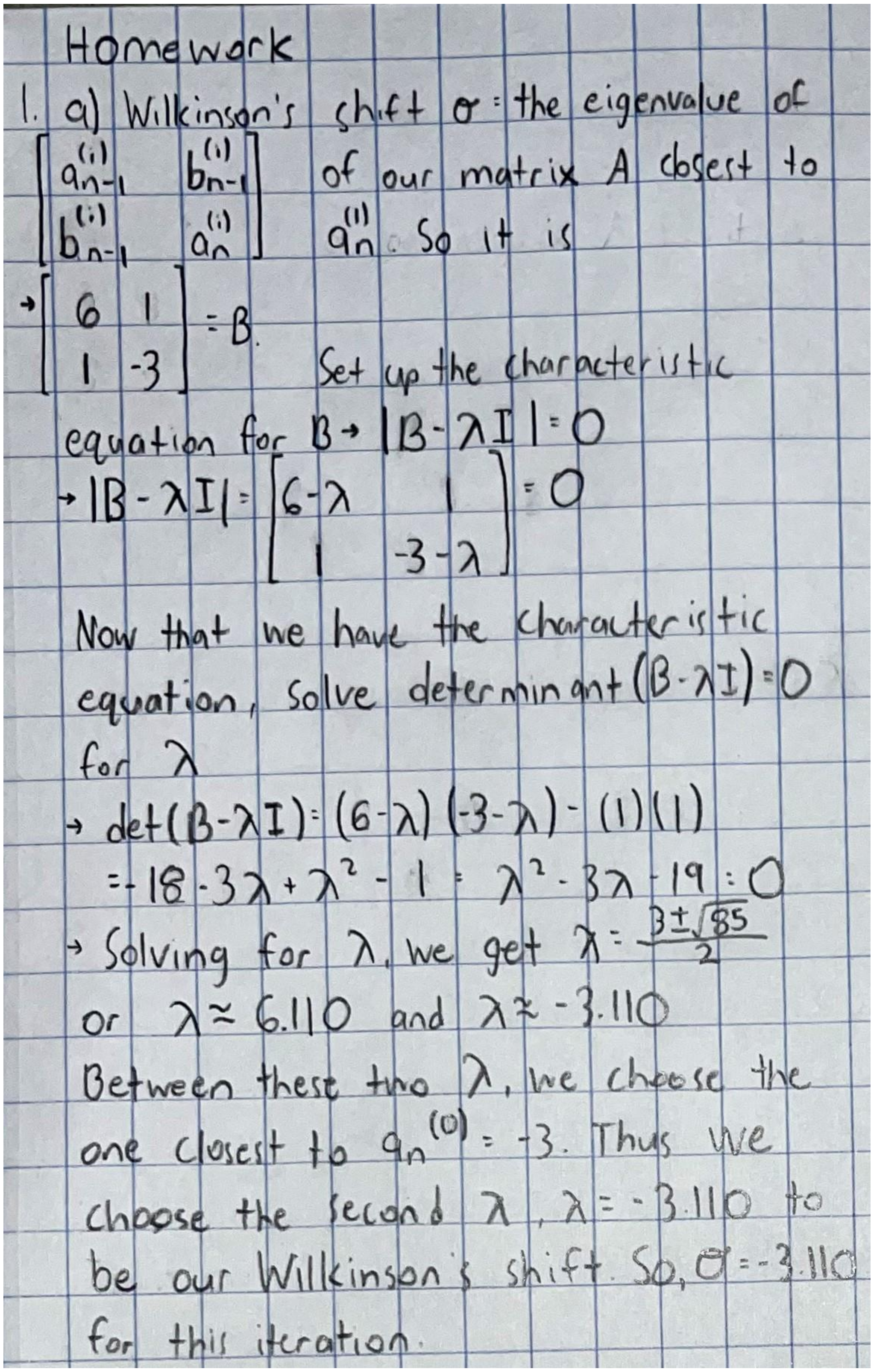
Perform one iteration of the QR algorithm on A to find $A^{(1)}$ using Wilkinson's shift. Let the convergence tolerance TOL be 5 * $10^{\wedge -14}$ and our shift σ to be Wilkinson's shift. In order to do so

   a. Calculate the Wilkinson shift σ.
   b. Find the constants of the factorization of A - σI: a_1, a_2, a_3, b_1, b_2, c_1, c_2, s_1, s_2, where I is the identity matrix and a_i, b_j, corresponds to the entries in matrix A, and cj corresponds to the entries in the rotational matrix R for i = 1, 2, 3 and j = 1, 2.
   c. Then use the results from the factorization of A to compute the product $R^{(0)}Q^{(0)}$.

2. Create a program that implements the QR algorithm to find the eigenvalues of the matrix from problem 1, both without a shift and then with Wilkinson's shift. Compare the number of iterations for each case. Let the convergence tolerance be 5 * $10^{\wedge -14}$.

Solutions

1.
   a.



Homework

1. a) Wilkinson's shift σ: the eigenvalue of $\begin{bmatrix} a_{n-1}^{(1)} & b_{n-1}^{(1)} \\ b_{n-1}^{(1)} & a_n^{(1)} \end{bmatrix}$ of our matrix A closest to $a_n^{(1)}$. So it is

$\rightarrow \begin{bmatrix} 6 & 1 \\ 1 & -3 \end{bmatrix} = B$. Set up the characteristic equation for $B \rightarrow |B - \lambda I| = 0$

$\rightarrow |B - \lambda I| = \begin{bmatrix} 6-\lambda & 1 \\ 1 & -3-\lambda \end{bmatrix} = 0$

Now that we have the characteristic equation, solve determinant $(B - \lambda I) = 0$ for λ

$\rightarrow \det(B - \lambda I) = (6-\lambda)(-3-\lambda) - (1)(1)$

$= -18 - 3\lambda + \lambda^2 - 1 = \lambda^2 - 3\lambda - 19 = 0$

$\rightarrow$ Solving for λ, we get $\lambda = \frac{3 \pm \sqrt{85}}{2}$ or $\lambda \approx 6.110$ and $\lambda \approx -3.110$

Between these two λ, we choose the one closest to $a_n^{(0)} = -3$. Thus we choose the second λ, $\lambda = -3.110$ to be our Wilkinson's shift. So, $\sigma = -3.110$ for this iteration.

b.

b) The next step is to find the constants that we will use for the factorization of $A-\theta I$. Evaluate as

$$\to A-\theta I = \begin{bmatrix} 2 & -1 & 0 \\ -1 & 6 & 1 \\ 0 & 1 & -3 \end{bmatrix} - \begin{bmatrix} -3.110 & 0 & 0 \\ 0 & -3.110 & 0 \\ 0 & 0 & -3.110 \end{bmatrix}$$

$$= \begin{bmatrix} 5.110 & -1 & 0 \\ -1 & 9.110 & 1 \\ 0 & 1 & 0.110 \end{bmatrix}$$

Now write out the starting values for the constants. For our problem, it is

$\to a_1 = 2,\ a_2 = 6,\ a_3 = -3,\ b_1 = -1,\ b_2 = 1$

To find c_1 and s_1 for our rotational matrix, use the equalities

(1) $c_1 = a_1 / r$ and (2) $s_1 = t/r$ where $r = \sqrt{a_1^{\,2} + t^2}$ and $t = b_1 = -1$. Then

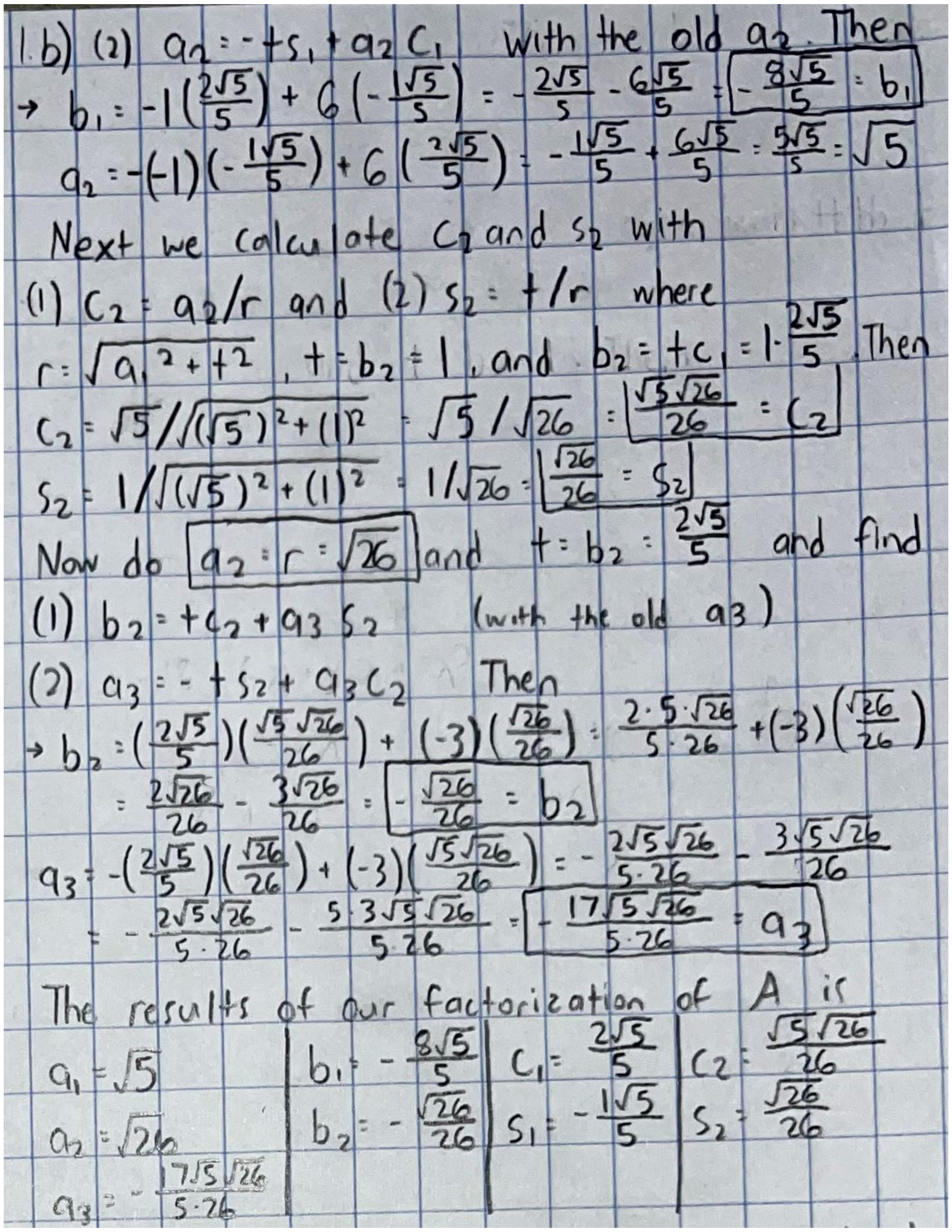
$$\to c_1 = (2)/\sqrt{(2)^2+(-1)^2} = \frac{2}{\sqrt{5}} = \boxed{\frac{2\sqrt{5}}{5} = c_1}$$

$$s_1 = (-1)/\sqrt{(2)^2+(-1)^2} = -\frac{1}{\sqrt{5}} = \boxed{-\frac{1\sqrt{5}}{5} = s_1}$$

Now, we do $\boxed{a_1 = r = \sqrt{5}}$ and $t = b_1 = -1$

With our new values, find

(1) $b_1 = tc_1 + a_2 s_1$ and



1.b) (2) $a_2 = -ts_1 + a_2 c_1$ with the old a_2. Then

$\rightarrow b_1 = -1\left(\frac{2\sqrt5}{5}\right) + 6\left(-\frac{1\sqrt5}{5}\right) = -\frac{2\sqrt5}{5} - \frac{6\sqrt5}{5} = \boxed{-\frac{8\sqrt5}{5} = b_1}$

$a_2 = -(-1)\left(-\frac{1\sqrt5}{5}\right) + 6\left(\frac{2\sqrt5}{5}\right) = -\frac{1\sqrt5}{5} + \frac{6\sqrt5}{5} = \frac{5\sqrt5}{5} = \sqrt5$

Next we calculate c_2 and s_2 with

(1) $c_2 = a_2/r$ and (2) $s_2 = t/r$ where

$r = \sqrt{a_1^2 + t^2}$, $t = b_2 = 1$, and $b_2 = tc_1 = 1\cdot\frac{2\sqrt5}{5}$. Then

$c_2 = \sqrt5/\sqrt{(\sqrt5)^2 + (1)^2} = \sqrt5/\sqrt{26} = \boxed{\frac{\sqrt5\sqrt{26}}{26} = c_2}$

$s_2 = 1/\sqrt{(\sqrt5)^2 + (1)^2} = 1/\sqrt{26} = \boxed{\frac{\sqrt{26}}{26} = s_2}$

Now do $\boxed{a_2 = r = \sqrt{26}}$ and $t = b_2 = \frac{2\sqrt5}{5}$ and find

(1) $b_2 = tc_2 + a_3 s_2$ (with the old a_3)

(2) $a_3 = -ts_2 + a_3 c_2$ Then

$\rightarrow b_2 = \left(\frac{2\sqrt5}{5}\right)\left(\frac{\sqrt5\sqrt{26}}{26}\right) + (-3)\left(\frac{\sqrt{26}}{26}\right) = \frac{2\cdot5\cdot\sqrt{26}}{5\cdot26} + (-3)\left(\frac{\sqrt{26}}{26}\right)$

$= \frac{2\sqrt{26}}{26} - \frac{3\sqrt{26}}{26} = \boxed{-\frac{\sqrt{26}}{26} = b_2}$

$a_3 = -\left(\frac{2\sqrt5}{5}\right)\left(\frac{\sqrt{26}}{26}\right) + (-3)\left(\frac{\sqrt5\sqrt{26}}{26}\right) = -\frac{2\sqrt5\sqrt{26}}{5\cdot26} - \frac{3\sqrt5\sqrt{26}}{26}$

$= -\frac{2\sqrt5\sqrt{26}}{5\cdot26} - \frac{5\cdot3\sqrt5\sqrt{26}}{5\cdot26} = \boxed{-\frac{17\sqrt5\sqrt{26}}{5\cdot26} = a_3}$

The results of our factorization of A is

$a_1 = \sqrt5$	$b_1 = -\frac{8\sqrt5}{5}$	$c_1 = \frac{2\sqrt5}{5}$	$c_2 = \frac{\sqrt5\sqrt{26}}{26}$
$a_2 = \sqrt{26}$	$b_2 = -\frac{\sqrt{26}}{26}$	$s_1 = -\frac{1\sqrt5}{5}$	$s_2 = \frac{\sqrt{26}}{26}$
$a_3 = -\frac{17\sqrt5\sqrt{26}}{5\cdot26}$			

c.

c) Now, it is time to calculate the product of $R^{(0)}Q^{(0)}$ of our iteration

First, compute a_1, b_1, and a_2 given by

(1) $a_1 = a_1 c_1 + b_1 s_1$ (where you use values from b))

(2) $b_1 = a_2 s_1$ and

(3) $a_2 = a_2 c_1$ respectively. So,

$a_1 = (\sqrt{5})\left(\frac{2\sqrt{5}}{5}\right) = \boxed{2 = a_1}$

$b_1 = (\sqrt{26})\left(-\frac{1\sqrt{5}}{5}\right) = \boxed{-\frac{\sqrt{26}\sqrt{5}}{5} = b_1}$

$a_2 = (\sqrt{26})\left(\frac{2\sqrt{5}}{5}\right) = \frac{2\sqrt{26}\sqrt{5}}{5}$

Then, compute a_2, b_2, a_3 given by

(1) $a_2 = a_2 c_2 + b_2 s_2$ (where you use values we computed)

(2) $b_2 = a_3 s_2$

(3) $a_3 = a_3 c_2$ respectively. So,

$a_2 = \left(\frac{2\sqrt{26}\sqrt{5}}{5}\right)\left(\frac{\sqrt{5}\sqrt{26}}{26}\right) = \boxed{2 = a_2}$

$b_2 = \left(-\frac{17\sqrt{5}\sqrt{26}}{5\cdot 26}\right)\left(\frac{\sqrt{26}}{26}\right) = \boxed{-\frac{17\sqrt{5}}{5\cdot 26} = b_2}$

$a_3 = \left(-\frac{17\sqrt{5}\sqrt{26}}{5\cdot 26}\right)\left(\frac{\sqrt{5}\sqrt{26}}{26}\right) = \boxed{-17/26 = a_3}$

Now, just fit in the constants in the correct slot, where $R^{(0)}Q^{(0)} = \begin{bmatrix} a_1 & b_1 & 0 \\ b_1 & a_2 & b_2 \\ 0 & b_2 & a_3 \end{bmatrix} = A^{(1)}$

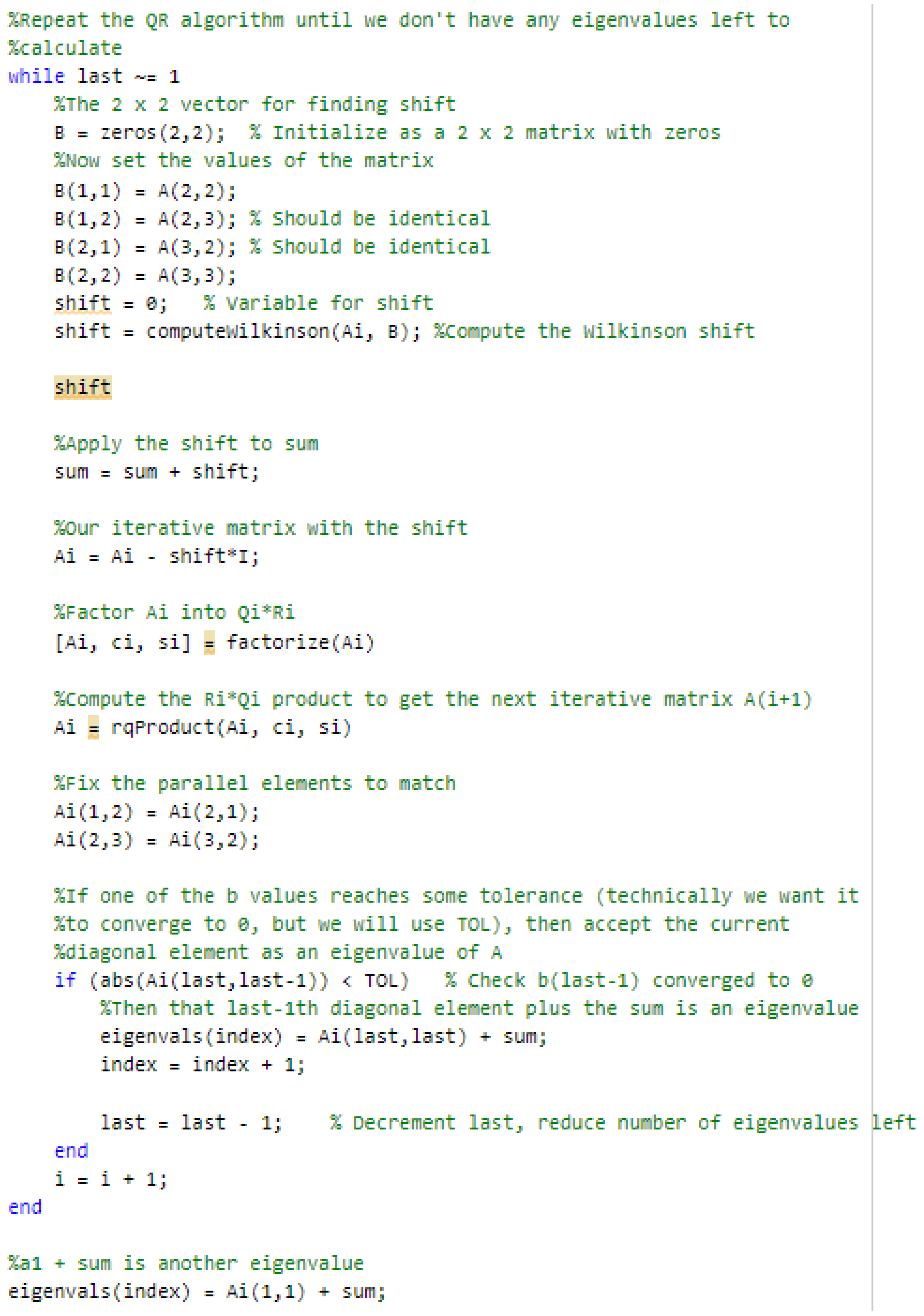
$\rightarrow A^{(1)} = R^{(0)} Q^{(0)} = \begin{bmatrix} 2 & -2.28 & 0 \\ -2.28 & 2 & -0.292 \\ 0 & -0.292 & 0.654 \end{bmatrix}$

2.

First, do the QR algorithm without shift

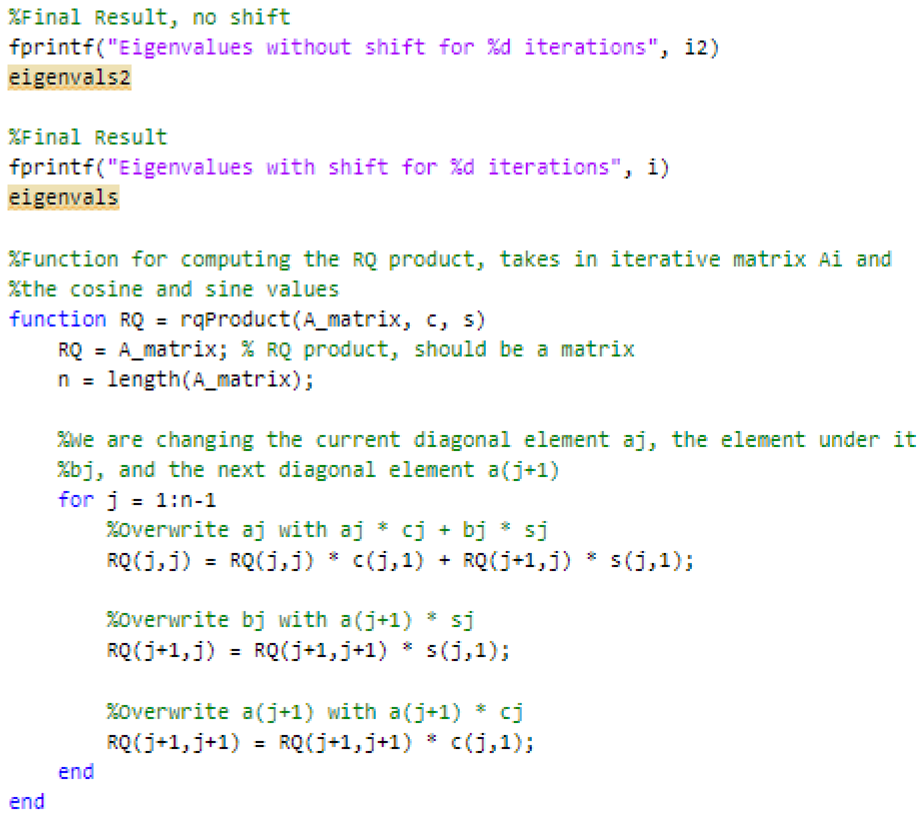
```
%2. Code QR algorithm with Wilkinson's shift
n = 3;   % Set the number of eigenvalues/vectors we will find, corresponds to dimension of matrix
sum = 0; % Initialize a sigma sum variable that calculates the accumulation of the shifts
last = n; % the stopping criteria, n - last is the number of eigenvalues that have been determined
last2 = n;
i2 = 1;
i = 1;   % Variable for the number of iterations, will not be the stopping criteria
%The original matrix for our problem
A = [2, -1, 0;
     -1, 6, 1;
     0, 1, -3]
Ai = zeros(3,3); % Our iterative version of matrix A
Ai = A;
Di = A; % Our iterative matrix without the shift
I = eye(n);    % Identity matrix
eigenvals = zeros(n,1);    % A resultant vector of the eigenvalues of A
eigenvals2 = zeros(n,1);    % A resultant vector of the eigenvalues of D
index = 1;    % Index for eigenvalue
index2 = 1;
TOL = 5 * 10^(-14);    % Let our tolerance be 5 * 10^-14

%Repeat the QR algorithm until we don't have any eigenvalues left to
%calculate, this is the algorithm without the Wilkinson shift
while last2 ~= 1
    %Factor Di into Qi*Ri, no shift
    [Di, ci2, si2] = factorize(Di)

    %Compute the Ri*Qi product to get the next iterative matrix A(i+1), no
    %shift
    Di = rqProduct(Di, ci2, si2)

    %Fix the parallel elements to match
    Di(1,2) = Di(2,1);
    Di(1,2) = Di(2,1);

    %If one of the b values reaches some tolerance (technically we want it
    %to converge to 0, but we will use TOL), then accept the current, no shift
    if (abs(Di(last2,last2-1)) < TOL)   % Check b(last-1) converged to 0
        %Then that last-1th diagonal element plus the sum is an eigenvalue
        eigenvals2(index2) = Di(last2,last2);
        index2 = index2 + 1;

        last2 = last2 - 1;    % Decrement last, reduce number of eigenvalues left
    end
    i2 = i2 + 1;
end

%a1 + sum is another eigenvalue, for no shift
eigenvals2(index2) = Di(1,1);
```

Now do the algorithm with Wilkinson’s shift



```
%Repeat the QR algorithm until we don't have any eigenvalues left to
%calculate
while last ~= 1
    %The 2 x 2 vector for finding shift
    B = zeros(2,2);  % Initialize as a 2 x 2 matrix with zeros
    %Now set the values of the matrix
    B(1,1) = A(2,2);
    B(1,2) = A(2,3); % Should be identical
    B(2,1) = A(3,2); % Should be identical
    B(2,2) = A(3,3);
    shift = 0;   % Variable for shift
    shift = computeWilkinson(Ai, B); %Compute the Wilkinson shift

    shift

    %Apply the shift to sum
    sum = sum + shift;

    %Our iterative matrix with the shift
    Ai = Ai - shift*I;

    %Factor Ai into Qi*Ri
    [Ai, ci, si] = factorize(Ai)

    %Compute the Ri*Qi product to get the next iterative matrix A(i+1)
    Ai = rqProduct(Ai, ci, si)

    %Fix the parallel elements to match
    Ai(1,2) = Ai(2,1);
    Ai(2,3) = Ai(3,2);

    %If one of the b values reaches some tolerance (technically we want it
    %to converge to 0, but we will use TOL), then accept the current
    %diagonal element as an eigenvalue of A
    if (abs(Ai(last,last-1)) < TOL)   % Check b(last-1) converged to 0
        %Then that last-1th diagonal element plus the sum is an eigenvalue
        eigenvals(index) = Ai(last,last) + sum;
        index = index + 1;

        last = last - 1;    % Decrement last, reduce number of eigenvalues left
    end
    i = i + 1;
end

%a1 + sum is another eigenvalue
eigenvals(index) = Ai(1,1) + sum;
```

Print the eigenvalues along with the number of iterations for each version
The function finds the product of RQ after the matrix A is factorized

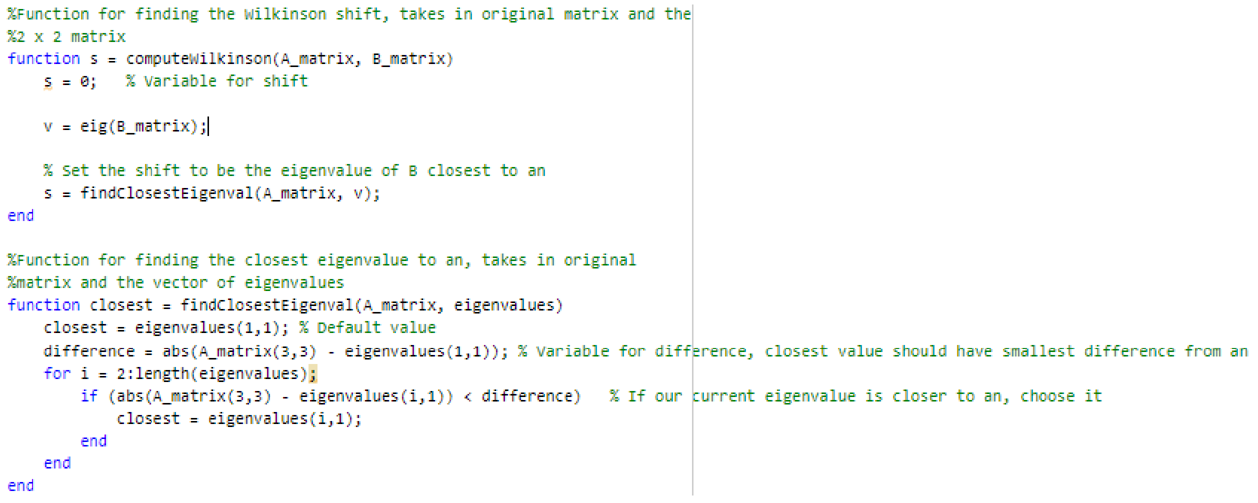


```
%Final Result, no shift
fprintf("Eigenvalues without shift for %d iterations", i2)
eigenvals2

%Final Result
fprintf("Eigenvalues with shift for %d iterations", i)
eigenvals

%Function for computing the RQ product, takes in iterative matrix Ai and
%the cosine and sine values
function RQ = rqProduct(A_matrix, c, s)
    RQ = A_matrix; % RQ product, should be a matrix
    n = length(A_matrix);

    %We are changing the current diagonal element aj, the element under it
    %bj, and the next diagonal element a(j+1)
    for j = 1:n-1
        %Overwrite aj with aj * cj + bj * sj
        RQ(j,j) = RQ(j,j) * c(j,1) + RQ(j+1,j) * s(j,1);

        %Overwrite bj with a(j+1) * sj
        RQ(j+1,j) = RQ(j+1,j+1) * s(j,1);

        %Overwrite a(j+1) with a(j+1) * cj
        RQ(j+1,j+1) = RQ(j+1,j+1) * c(j,1);
    end
end
```

This function will factorize A into Q and R

```
%Function for factorizing a matrix A by finding the constants associated
%with the RQ product
function [A_factorized, c, s] = factorize(A_matrix)
    A_factorized = A_matrix; % Factored matrix
    n = length(A_matrix);    % Number of diagonal elements
    c = zeros(n-1,1); % Array of cosine elements
    s = zeros(n-1,1); % Array of sine elements
    t = A_matrix(1,2);   % Save b1 in a temporary variable t

    %Run for n - 1 because you want to clear the entries above/below each
    % diagonal element, where the nth diagonal element has nothing under it
    for j = 1:n-1
        %aj is the jth diagonal element
        r = sqrt(A_factorized(j,j)^2 + t^2);   % calculate r, the denominator

        %Compute cj, the cosine element in the rotational matrix R
        c(j,1) = A_factorized(j,j) / r;
        %Compute sj, the sine element in the rotational matrix R
        s(j,1) = t / r;

        %Overwite our current jth diagonal element aj with r
        A_factorized(j,j) = r;

        %Save the current b in t temporarily for the next iteration
        t = A_factorized(j+1,j); % bj should be the element under our jth diagonal element

        %Overwrite bj with t * cj + a(j+1) * sj
        A_factorized(j+1,j) = t * c(j,1) + A_factorized(j+1,j+1) * s(j,1);

        %Overwrite a(j+1) with -t * sj + a(j+1) * cj
        A_factorized(j+1,j+1) = -t * s(j,1) + A_factorized(j+1,j+1) * c(j,1);

        %We are going to check the b element, make sure we are not at the
        %n-1th diagonal element so that there is another b to check ahead
        if(j ~= n-1)
            %Save the next b element b(j+1) in our temporary variable t
            t = A_factorized(j+2,j+1);

            %Overwrite b(j+1) with t * cj
            A_factorized(j+2,j+1) = t * c(j,1);
        end
    end
end
```

The first function computes the Wilkinson shift, and the second is a helper function for determining the closest eigenvalue



```
%Function for finding the Wilkinson shift, takes in original matrix and the
%2 x 2 matrix
function s = computeWilkinson(A_matrix, B_matrix)
    s = 0;   % Variable for shift

    v = eig(B_matrix);

    % Set the shift to be the eigenvalue of B closest to an
    s = findClosestEigenval(A_matrix, v);
end

%Function for finding the closest eigenvalue to an, takes in original
%matrix and the vector of eigenvalues
function closest = findClosestEigenval(A_matrix, eigenvalues)
    closest = eigenvalues(1,1); % Default value
    difference = abs(A_matrix(3,3) - eigenvalues(1,1)); % Variable for difference, closest value should have smallest difference from an
    for i = 2:length(eigenvalues);
        if (abs(A_matrix(3,3) - eigenvalues(i,1)) < difference)   % If our current eigenvalue is closer to an, choose it
            closest = eigenvalues(i,1);
        end
    end
end
```

Output

```
   -0.0000    0.4211   -0.0000
         0    0.0000   -4.4656


ci =

     1
     1


si =

   1.0e-13 *

   -0.1339
   -0.0000


Ai =

    4.9842   -0.0000         0
   -0.0000    0.4211   -0.0000
         0    0.0000   -4.4656

Eigenvalues without shift for 65 iterations
eigenvals2 =

    1.7745
   -3.1122
    6.3376

Eigenvalues with shift for 48 iterations
eigenvals =

   -3.1122
    1.7745
    6.3376
```

Homework Narrative Reflection

For the first problem, I wanted to give the student a good understanding of the QR algorithm, and what better way to do it than actually implementing the algorithm by hand yourself. This question is meant to help the student understand the application aspect more, rather than the theory. This problem will be giving them a hands-on implementation by asking them to perform an iteration of it. However, this algorithm can get complicated and tedious if done by hand, for there are a lot of terms that you have to keep track of. That is why I only asked for one iteration of the algorithm, and not to find all of the eigenvalues or eigenvectors of the given matrix. I decided to split this problem up into separate steps to better help the student and lead them along the algorithm. I chose to ask them to do a Wilkinson's shift, because I wanted them to learn the more preferred variant of the QR algorithm. So, for this question, I think that the work is reasonably achievable.

Now, for the second problem, I wanted to have the student work on the same matrix to at least let them confirm parts of their answer to the first question. Then, we will add onto that by asking to code the rest of the QR algorithm. The coding portion was a little more involved than the hand written one. I still thought that it wouldn't be an issue to also ask to implement the algorithm without the shift, since that would be a small change that could be easily implemented in code without too much added work. Plus, the student will be able to see the strength of having a shift in the first place, and hopefully the difference that it makes without having one. Since a coded program can easily execute an algorithm, this coding problem allows the student to examine the theory aspect of the QR algorithm.

I opted out from asking to find the eigenvectors of the matrix, because it is a standard process that is not necessarily unique to the QR algorithm. The second question was already getting on the lengthy side, so I didn't ask them to code a program to find eigenvectors either.

Conclusion

Overall, I enjoyed learning about the QR algorithm. The QR algorithm provides some advantages, such as reducing the number of operations per iteration. Also, the QR algorithm provides more complete answers than other ones. For example, a limitation of another algorithm, the power method, is that it converges to the eigenvector of the *largest* eigenvalue of the matrix for almost all chosen initial vectors. This is not the case for the QR algorithm.

We can further improve the convergence speed by shifting our original matrix. In our case, we used Wilkinson's shift. Without some kind of shift, the convergence for the QR algorithm can be painfully slow and may possibly require thousands of iterations.

Bibliography:

Bradie, Brian. *A Friendly Introduction to Numerical Analysis*. Pearson Prentice Hall, 2006.

Used for limitation →
https://people.inf.ethz.ch/arbenz/ewp/Lnotes/chapter4.pdf

For the image of tridiagonal symmetric matrix →
https://math.stackexchange.com/questions/3986722/since-a-symmetric-tridiagonal-matrix-contains-only-two-distinct-vectors

https://www2.math.upenn.edu/~deturck/m320/text/part8.pdf

Good source → https://pythonnumericalmethods.berkeley.edu/notebooks/chapter15.03-The-QR-Method.html#:~:text=The%20QR%20method%20is%20a%20way%20to%20decompose%20a%20matrix,%3DQTQ%3DI.

https://lpsa.swarthmore.edu/MtrxVibe/EigMat/MatrixEigen.html

https://math.nyu.edu/~stadler/num1/material/num1_eigenvalues

https://www.quora.com/How-many-eigenvalues-does-an-n-x-n-matrix-have

http://madrury.github.io/jekyll/update/statistics/2017/10/04/qr-algorithm.html

Help with deciding a topic:
https://math.stackexchange.com/questions/1555357/numerical-methods-for-finding-eigenvalues-of-large-matrices