Purpose: To see how eigenvalues can be calculated by iterative methods that employ QR factorization, and get some understanding of why such methods work. This type of algorithm is used in all professional software for general eigenvalue calculations today, such as MATLAB’s eig function.

Prerequisite: Sections 5.2 and 6.4

MATLAB functions used: qr, *, eye, :, tic, toc, for, eig

qreigdat, qrbasic, qrshift and randint from Lay’s Toolbox

Part I Background

It is not easy to calculate eigenvalues for most matrices. Characteristic polynomials are difficult to compute. Even if you know the characteristic polynomial, algorithms such as Newton’s method for finding zeros cannot be depended upon to produce all the zeros with reasonable speed and accuracy.

Fortunately, numerical analysts have found an entirely different way to calculate eigenvalues of a matrix $A$, using the fact that any matrix similar to $A$ has the same eigenvalues. The idea is to create a sequence of matrices similar to $A$ which converges to an upper triangular matrix; if this can be done then the diagonal entries of the limit matrix are the eigenvalues of $A$. The remarkable discoveries are that the method can be done with great accuracy, and it will converge for almost all matrices. In practice the limit matrix is just block upper triangular, not truly triangular (because only real arithmetic is done), but it is still easy to get the eigenvalues from that. See Note 2 below.

The primary reason that modern implementations of this method are efficient and reliable is that a QR factorization can be used to create each new matrix in the sequence. Each QR factorization can be calculated quickly and accurately; it yields easily a new matrix orthogonally similar to the original matrix; and orthogonal similarities tend to minimize the effect of roundoff error on the eigenvalues.

The calculations in this project can be done without Lay’s Toolbox; see Note 1 below. For more information on the theory, see Note 2 below.

1. (hand) Here you will verify some of the basic matrix properties that underlie this modern method. Suppose $A$ is nxn. Let $A = Q_0 R_0$ be a QR factorization of $A$ and create $A_1 = R_0 Q_0$. Let $A_1 = Q_1 R_1$ be a QR factorization of $A_1$ and create $A_2 = R_1 Q_1$. Explain why the following are true; use your paper and attach:

   (a) $A = Q_0 A_1 Q_0^T$  
      (This is exercise 23, Sec. 5.2)

   (b) $A = (Q_0 Q_1) A_2(Q_0 Q_1)^T$

   (c) $Q_0 Q_1$ is orthogonal  
      (This is exercise 29, Sec. 6.2)

   (d) $A, A_1$ and $A_2$ all have the same eigenvalues.

2. (MATLAB) Type qreigdat to get the following matrices. Then use MATLAB’s eig function to calculate their eigenvalues, and record their eigenvalues below each matrix:

$$A_3 = \begin{bmatrix} 1 & 2 & 8 \\ 7 & -7 & 6 \\ 5 & 7 & -8 \end{bmatrix}, \quad A_4 = \begin{bmatrix} 4 & -2 & 3 & -7 \\ 1 & 2 & 6 & 8 \\ 8 & 5 & 1 & -5 \\ -5 & 8 & -5 & 3 \end{bmatrix}, \quad A_5 = \begin{bmatrix} 2 & 6 & -3 & 4 & -9 \\ -1 & 7 & -4 & -3 & -7 \\ -6 & -6 & 1 & 6 & 5 \\ 9 & 2 & 6 & 2 & -8 \\ -7 & 8 & 6 & -9 & -1 \end{bmatrix}, \quad A_6 = \begin{bmatrix} 0 & 0 & -1 & 4 & -1 \\ -1 & 2 & -5 & 2 & -5 \\ 0 & 4 & 5 & -4 & 3 \\ -1 & 2 & -2 & 3 & 1 \\ -6 & -5 & 2 & 0 & 7 \end{bmatrix}$$

Eigenvalues of each:

Part II The basic QR algorithm

Definition. The basic QR algorithm for eigenvalues is the iterative process begun in question 1, repeated many times: let $A = Q_0 R_0$ be a QR factorization of $A$ and create $A_1 = R_0 Q_0$; let $A_1 = Q_1 R_1$ be a QR factorization of $A_1$ and create $A_2 = R_1 Q_1$; …; having created $A_m$, let $A_m = Q_m R_m$ be a QR factorization of $A_m$ and create $A_{m+1} = R_m Q_m$; etc. Continue until the entries below the diagonal of $A_m$ are sufficiently small (or stop if no convergence is apparent).
3. (MATLAB) For each of the matrices shown above, use the function `qrbasic` to find how many steps of the basic QR algorithm are needed to make the absolute value of every entry below the diagonal smaller than 0.001, and how many seconds this takes. In the first column of the table on page 3, record the number of steps, the time, and the final matrix.

The function `qrbasic` simply does the commands `[Q R] = qr(A), A = R*Q` repeatedly. The program will stop when all entries below the diagonal are smaller than 0.001 (or after 200 steps if that test is never met), and it will report the last matrix and the total number of steps done. Typing `tic, some command, toc` will perform that command and also print the number of seconds required for executing it.

Specifically, to perform the calculations for $A_3$, type `tic, qrbasic(A3, 0.001), toc` Record the results on page 3. Then repeat this calculation for $A_4$, $A_5$ and $A_6$.

Part III. Improving the basic QR algorithm by shifting and deflating

It is true that the basic algorithm can fail to converge for some matrices, and even when it does converge it can be extremely slow. There are simple modifications which greatly speed it up and can also make it converge for more matrices.

One of these modifications is shifting. The idea is: if $A$ is the original matrix and $A_m$ is the current matrix in the iteration, choose a scalar $c$, then get a QR factorization of the shifted matrix $A_m - cI$, and then undo the shift when you define $A_{m+1}$. If the scalars can be chosen so they get closer and closer to an eigenvalue of $A$, this will dramatically speed up convergence. The next algorithm shows one way to choose the scalars, and also introduces deflating.

Before trying out this new algorithm, answer question 4 where you will see the theoretical effect of shifting and why you can deflate after the last row looks like $[0 \ldots 0 \, x]$.

4. (hand) Let $A$ be $n \times n$, let $I$ denote the $n \times n$ identity matrix, and let $c$ be a constant.

(a) Let $\lambda$ be an eigenvalue of $A$. Explain why $Ax = \lambda x$ is true if and only if $(A - cI)x = (\lambda - c)x$ is true. Use this to explain why the eigenvalues of $A - cI$ are the numbers obtained by subtracting $c$ from each eigenvalue of $A$. (This is the reason that creating $A - cI$ is called "shifting.")

(b) Let $A - cI = QR$ be a QR factorization of $A - cI$ and define $A_1 = RQ + cI$. Show that $A = QA_1Q^T$.

(c) Suppose $A = \begin{bmatrix} B & C \\ O & D \end{bmatrix}$, where $B$ and $D$ are square and $O$ is a zero matrix. Explain why the eigenvalues of $A$ are the eigenvalues of $B$ together with those of $D$. (This is Supplementary Exercise 10 in Chapter 5.)

**Definition.** The QR algorithm for eigenvalues using diagonal shifts is the following iterative process. Repeat the step described in question 4(b), each time choosing the value for the next scalar $c$ to be the last diagonal entry of the previous matrix $A_k$. Stop when the last row of $A_k$ looks like $[s_1 \, s_2 \ldots s_k \, 0]$ where each $s_i$ is very small. Then that last entry $x$ is approximately an eigenvalue of $A$. (By question 4(c), if each $s_i$ were exactly zero, $x$ would be a true eigenvalue of $A$.) Now deflate -- i.e., create a new smaller matrix by discarding the last row and column. Begin the process again on the new matrix. Continue repeating this process until all eigenvalues have been calculated, or until it appears the limit matrix cannot be improved.

5. (MATLAB) Use the function `qrshift` to apply the shift-deflate algorithm just described to the four matrices, $A_3$, $A_4$, $A_5$, $A_6$. Find how many seconds and how many steps are needed to make the absolute value of every entry below the diagonal less than 0.001, then for 0.0001. Record results in the table on page 3.

The function `qrshift` does shifting as described in question 4(b), and it deflates after the entries below the diagonal are less than the tolerance you specify. It will display each deflation, the final matrix and how many iterative steps were done. To perform the calculations for $A_3$, type

`tic, qrshift(A3, 0.001), toc`

Record the results on page 3. (Note: if you want to see the result of each deflation step, type `qrshift(A3, 0.001, 1)` instead – but this adds CPU time so it prevents you from seeing how much faster `qrshift` really is.)

Next type `tic, qrshift(A3, 0.0001), toc`, record results, and repeat these calculations for the other matrices.

6. (hand) Discuss: based on what you have seen, how does the basic QR method compare with the shift-deflate QR method?
Results from Question 3

<table>
<thead>
<tr>
<th>Matrix $A_3$:</th>
<th>Basic QR, tol = 0.001</th>
<th>QR with diagonal shifts, tol = 0.001</th>
<th>QR with diagonal shifts, tol = 0.0001</th>
</tr>
</thead>
<tbody>
<tr>
<td># steps</td>
<td>time = ___</td>
<td># steps ___</td>
<td># steps ___</td>
</tr>
</tbody>
</table>

Matrix after you stop iterations:

********************************************************************************************

<table>
<thead>
<tr>
<th>Matrix $A_4$:</th>
<th>Basic QR, tol = 0.001</th>
<th>QR with diagonal shifts, tol = 0.001</th>
<th>QR with diagonal shifts, tol = 0.0001</th>
</tr>
</thead>
<tbody>
<tr>
<td># steps</td>
<td>time = ___</td>
<td># steps ___</td>
<td># steps ___</td>
</tr>
</tbody>
</table>

Matrix after you stop iterations:

************************************************************* *******************************

<table>
<thead>
<tr>
<th>Matrix $A_5$:</th>
<th>Basic QR, tol = 0.001</th>
<th>QR with diagonal shifts, tol = 0.001</th>
<th>QR with diagonal shifts, tol = 0.0001</th>
</tr>
</thead>
<tbody>
<tr>
<td># steps</td>
<td>time = ___</td>
<td># steps ___</td>
<td># steps ___</td>
</tr>
</tbody>
</table>

Matrix after you stop iterations:

********************************************************************************************

<table>
<thead>
<tr>
<th>Matrix $A_6$:</th>
<th>Basic QR, tol = 0.001</th>
<th>QR with diagonal shifts, tol = 0.001</th>
<th>QR with diagonal shifts, tol = 0.0001</th>
</tr>
</thead>
<tbody>
<tr>
<td># steps</td>
<td>time = ___</td>
<td># steps ___</td>
<td># steps ___</td>
</tr>
</tbody>
</table>

Matrix after you stop iterations:
Note 1. You can do this project without Lay’s Toolbox. First type in the matrices yourself. The following commands will accomplish the basic QR method used in question 3, for an nxn matrix A.

\[ B = A; \text{ bound } = 0.001; \text{ p = 0; num = 200; } \]

while max(max(abs(tril(B,-1)))) > bound  
  \[ [Q R] = qr(B); B = R*Q; \]
  p = p + 1;
  if p > num, break, end  
end  
p, B

The following lines will accomplish the shift-deflate algorithm used in question 5, for an nxn matrix A.

\[ B = A; \text{ bound } = 0.001; \text{ p = 0; num = 20; } \]

for i = n:-1:2  
  B = B(1:i,1:i);  
  while max(max(abs((B(i,1:i -1)))))  > bound  
    \[ [Q R] = qr(B-B(i,i)*eye(i)); B = R*Q + B(i,i)*eye(i) ; \]
    p = p + 1;
    if p > num, break, end  
  end  
A(1:i,1:i) = B;  
end  
p, A

Note 2. Remarks about convergence. There is an excellent discussion of the theory of the QR method in Understanding the QR Algorithm, by D. Watkins, SIAM Review 24 (1982), pp. 427-440. This paper explains the geometric meaning of the algorithm and how it is an extension of the power method. (The power method is presented in Section 5.7 in Lay’s text.) Briefly, the following things are true about the QR method, for an nxn real matrix A.

(a) If the eigenvalues of A all have different magnitudes, then the basic QR algorithm will converge to an upper triangular matrix. To see that the basic QR method can fail if two different eigenvalues have the same magnitude, try it on the following matrices:

\[
\begin{bmatrix}
0 & 1 \\
1 & 0
\end{bmatrix}
\quad \text{and} \quad
\begin{bmatrix}
3 & -2 \\
4 & -3
\end{bmatrix}
\]
each of which has eigenvalues \pm 1.

(b) The matrices used above in this project were chosen so they have only real eigenvalues. However, a general real matrix can have nonreal eigenvalues. In this case, the algorithm described above, which uses only real QR factorizations, cannot possibly converge to an upper triangular matrix (why?). Nevertheless, it is true that a real shift can always be found so that the basic QR method applied to the new matrix will converge to a real block upper triangular matrix whose diagonal blocks are 1x1 or 2x2 matrices; then if you undo the shift, each 1x1 block is an eigenvalue and each 2x2 block easily yields a pair of complex conjugate eigenvalues, of the original matrix.

For example, the following matrices have some complex eigenvalues:

\[
\begin{bmatrix}
0 & 1 \\
0 & 0
\end{bmatrix}
\quad \text{and} \quad
\begin{bmatrix}
0 & 1 & 0 \\
0 & 0 & 1 \\
0 & 0 & 0
\end{bmatrix}
\]
Store the first matrix as \( A \). Calculate \( \text{eig}(A) \) to see what its eigenvalues are. Type \( [Q R] = qr(A), A = R*Q \) and repeat this command several times. You will see cycling. Then apply the basic method to \( A - I \). To do that, type \( \text{qrbasic}(A + \text{eye}(3), 0.0001) \). Now you will see convergence to a block upper triangular matrix with 1x1 and 2x2 blocks as described above. Subtract \( I \) from this limit matrix. Solve the characteristic equation for the 2x2 block, and verify that this gives two complex numbers. These will be the nonreal eigenvalues of \( A \), and the 1x1 block contains the third, real, eigenvalue of \( A \).

(c) The method shown in (b) can be improved by first doing an orthogonal similarity to \( A \) to get a Hessenberg matrix -- one that has zeros below its first subdiagonal. The reason this is better is, if each entry on the first subdiagonal of a Hessenberg matrix is nonzero, then the basic QR algorithm is guaranteed to converge to a block upper triangular matrix. It is quite easy to do an orthogonal similarity to any \( A \) to get a Hessenberg matrix. \(^1\) So all professional software to calculate eigenvalues begins by calculating a Hessenberg matrix which is orthogonally similar to the original \( A \), and then applies the shift-deflate iterative process to this matrix. As soon as the matrix produced after some step has the form

\[
\begin{bmatrix}
B & C \\
O & D
\end{bmatrix}
\]
where the entries of \( O \) are so small they can be treated as true zeros, then the iterative process is done separately on \( B \) and \( D \). Notice this method is a natural for parallel processing.

\(^1\) MATLAB can easily calculate a Hessenberg matrix similar to any \( A \). Try this: \( A = \text{randint}(10), \text{hess}(A) \).