MATH2071: LAB #6: The Eigenvalue Problem

<table>
<thead>
<tr>
<th>Introduction</th>
<th>Exercise 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eigenvalues and eigenvectors</td>
<td>Exercise 2</td>
</tr>
<tr>
<td>The Power Method</td>
<td>Exercise 3</td>
</tr>
<tr>
<td>The Inverse Power Method</td>
<td>Exercise 4</td>
</tr>
<tr>
<td>Finding Several Eigenvectors at Once</td>
<td>Exercise 5</td>
</tr>
<tr>
<td>Using Shifts</td>
<td>Exercise 6</td>
</tr>
<tr>
<td>The QR Method</td>
<td>Exercise 7</td>
</tr>
<tr>
<td>Convergence of the QR Algorithm</td>
<td>Exercise 8</td>
</tr>
<tr>
<td></td>
<td>Exercise 9</td>
</tr>
</tbody>
</table>

1 Introduction

This lab is concerned with several ways to compute eigenvalues and eigenvectors for a real matrix. All methods for computing eigenvalues and eigenvectors are iterative in nature, except for very small matrices. We begin with a short discussion of eigenvalues and eigenvectors, and then go on to the power method and inverse power methods. These are methods for computing a single eigenpair, but they can be modified to find several. We then look at shifting, which is an approach for computing one eigenpair whose eigenvalue is close to a specified value. We then look at the QR method, the most efficient method for computing all the eigenvalues at once, and also its shifted variant.

You may find it convenient to print the pdf version of this lab rather than the web page itself. This lab will take three sessions.

2 Eigenvalues and eigenvectors

For any square matrix $A$, consider the equation $\det(A - \lambda I) = 0$. This is a polynomial equation of degree $n$ in the variable $\lambda$ so there are exactly $n$ complex roots that satisfy the equation. If the matrix $A$ is real, then the complex roots occur in conjugate pairs. The roots are known as “eigenvalues” of $A$. Interesting questions include:

- How can we find one or more of these roots?
- When are the roots distinct?
- When are the roots real?

In textbook examples, the determinant is computed explicitly, the (often cubic) equation is factored exactly, and the roots “fall out.” This is not how a real problem will be solved.

If $\lambda$ is an eigenvalue of $A$, then $A - \lambda I$ is a singular matrix, and therefore there is at least one nonzero vector $x$ with the property that $(A - \lambda I)x = 0$. This equation is usually written $Ax = \lambda x$

Such a vector is called an “eigenvector” for the given eigenvalue. There may be as many as $n$ linearly independent eigenvectors. In some cases, the eigenvectors are, or can be made, into a set of pairwise orthogonal vectors. Interesting questions about eigenvectors include:

- How do we compute an eigenvector?
- When will there be a full set of $N$ independent eigenvectors?
- When will the eigenvectors be orthogonal?
In textbook examples, the singular system \((A - \lambda I)x = 0\) is examined, and by inspection, an eigenvector is determined. This is not how a real problem is solved either.

Some useful facts about the eigenvalues \(\lambda\) of a matrix \(A\):

- \(A^{-1}\) has the same eigenvectors as \(A\), and eigenvalues \(1/\lambda\);
- \(A^n\) has the same eigenvectors as \(A\), and eigenvalues \(\lambda^n\), for integer \(n\);
- \(A + \mu I\) has the same eigenvectors as \(A\), and eigenvalues \(\lambda + \mu\);
- If \(A\) is real and symmetric, all its eigenvalues and eigenvectors are real; and,
- If \(B\) is invertible, then \(B^{-1}AB\) has the same eigenvalues as \(A\), with different but related eigenvectors.

### 3 The Rayleigh Quotient

If a vector \(x\) is an exact eigenvector of a matrix \(A\), then it is easy to determine the value of the associated eigenvalue: simply take the ratio of a component of \((Ax)_k\) and \(x_k\), for any index \(k\) that you like. But suppose \(x\) is only an approximate eigenvector, or worse yet, just a wild guess. We could still compute the ratio of corresponding components for some index, but the value we get will vary from component to component.

We might even try computing all the ratios, and averaging them, or taking the ratios of the norms. However, the preferred method for estimating the value of an eigenvalue uses the so-called “Rayleigh quotient.”

The Rayleigh quotient of a matrix \(A\) and vector \(x\) is

\[
R(A, x) = \frac{x \cdot Ax}{x \cdot x} = \frac{x^H Ax}{x^H x},
\]

where \(x^H\) denotes the Hermitian (complex conjugate transpose) of \(x\). The Matlab prime (as in \(x'\)) actually means the complex conjugate transpose, not just the transpose, so you can use the prime here. (In order to take a transpose, without the complex conjugate, you must use the Matlab function \texttt{transpose} or the prime with a dot as in \(x.'\).) Note that the denominator of the Rayleigh quotient is just the square of the 2-norm of the vector. In this lab we will be mostly computing Rayleigh quotients of unit vectors, in which case the Rayleigh quotient reduces to

\[
R(A, x) = x^H Ax \quad \text{for} \quad ||x|| = 1.
\]

If \(x\) happens to be an eigenvector of the matrix \(A\), the the Rayleigh quotient must equal its eigenvalue. (Plug \(x\) into the formula and you will see why.) When the real vector \(x\) is an approximate eigenvector of \(A\), the Rayleigh quotient is a very accurate estimate of the corresponding eigenvalue. Complex eigenvalues and eigenvectors require a little care because the dot product involves multiplication by the conjugate transpose.

The Rayleigh quotient remains a valuable tool in the complex case, and most of the facts remain true.

If the matrix \(A\) is symmetric, then all eigenvalues are real, and the Rayleigh quotient satisfies the following inequality.

\[
\lambda_{\text{min}} \leq R(A, x) \leq \lambda_{\text{max}}
\]

**Exercise 1:**

(a) Write an m-file called \texttt{rayleigh.m} that computes the Rayleigh quotient. Do not assume that \(x\) is a unit vector. It should have the signature:

```matlab
function r = rayleigh ( A, x )
% r = rayleigh ( A, x )
% comments
% your name and the date
```

2
Be sure to include comments describing the input and output parameters and the purpose of the function.

(b) Copy the m-file `eigen_test.m`. This file contains several test problems. Verify that the matrix you get by calling `A=eigen_test(1)` has eigenvalues 1, -1.5, and 2, and eigenvectors `[1;0;1]`, `[0;1;1]`, and `[1;2;0]`, respectively. That is, verify that $A\mathbf{x} = \lambda \mathbf{x}$ for each eigenvalue $\lambda$ and eigenvector $\mathbf{x}$.

(c) Compute the value of the Rayleigh quotient for the vectors in the following table.

<table>
<thead>
<tr>
<th>x</th>
<th>R(A,x)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>[1; 2; 3]</code></td>
<td>-0.78571</td>
</tr>
<tr>
<td><code>[1; 0; 1]</code></td>
<td></td>
</tr>
<tr>
<td><code>[0; 1; 1]</code></td>
<td></td>
</tr>
<tr>
<td><code>[1; -2; 0]</code></td>
<td></td>
</tr>
<tr>
<td><code>[1; 1; 1]</code></td>
<td></td>
</tr>
<tr>
<td><code>[0; 0; 1]</code></td>
<td></td>
</tr>
</tbody>
</table>

(d) Starting from the vector $\mathbf{x} = [3;2;1]$, we are interested in what happens to the Rayleigh quotient when you look at the sequence of vectors $\mathbf{x}, A\mathbf{x}, A^2\mathbf{x}$, $\ldots$. You should observe the beginnings of convergence to one of the eigenvalues. Which eigenvalue?

<table>
<thead>
<tr>
<th>x</th>
<th>R(A,x)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>[3; 2; 1]</code></td>
<td></td>
</tr>
<tr>
<td>$A\mathbf{x}$</td>
<td></td>
</tr>
<tr>
<td>$A^2\mathbf{x}$</td>
<td></td>
</tr>
<tr>
<td>$A^3\mathbf{x}$</td>
<td></td>
</tr>
<tr>
<td>$A^4\mathbf{x}$</td>
<td></td>
</tr>
<tr>
<td>$A^5\mathbf{x}$</td>
<td></td>
</tr>
<tr>
<td>$A^{10}\mathbf{x}$</td>
<td></td>
</tr>
<tr>
<td>$A^{15}\mathbf{x}$</td>
<td></td>
</tr>
<tr>
<td>$A^{20}\mathbf{x}$</td>
<td></td>
</tr>
<tr>
<td>$A^{25}\mathbf{x}$</td>
<td></td>
</tr>
</tbody>
</table>

The Rayleigh quotient $R(A, \mathbf{x})$ provides a way to approximate eigenvalues from an approximate eigenvector. Now we have to figure out a way to come up with an approximate eigenvector.

4 The Power Method

In many physical and engineering applications, the largest or the smallest eigenvalue associated with a system represents the dominant and most interesting mode of behavior. For a bridge or support column, the largest eigenvalue might reveal the maximum load, and the eigenvector the shape of the object as it begins to fail under this load. For a concert hall, the smallest eigenvalue of the acoustic equation reveals the lowest resonating frequency. For nuclear reactors, the largest eigenvalue determines whether the reactor is subcritical, critical, or supercritical. Hence, a simple means of approximating just one extreme eigenvalue might be enough for some cases.

The “power method” tries to determine the largest magnitude eigenvalue, and the corresponding eigenvector, of a matrix, by computing (scaled) vectors in the sequence:

$$x^{(k+1)} = A x^{(k)}$$

**Note:** The superscript in the above equation does not refer to a power of $x$, or to a component. The use of superscript distinguishes it from the $k^{th}$ component, $x_k$, and the parentheses distinguish it from an exponent. It is simply the $k^{th}$ vector in a sequence. When you write Matlab code, you will typically denote
both $x^{(k)}$ and $x^{(k+1)}$ with the same Matlab name $x$, and overwrite it each iterate. In the signature lines
specified below, $x_{old}$ is used to denote $x^{(k)}$ and $x$ is used to denote $x^{(k+1)}$ and all subsequent iterates.
If we include the scaling, and an estimate of the eigenvalue, the power method can be described in the
following way.

1. Starting with any nonzero vector $x$, divide by its length to make a unit vector called $x^{(0)}$,
2. $x^{(k+1)} = (Ax^{(k)})/||Ax^{(k)}||$
3. Compute the Rayleigh quotient of the iterate (unit vector) $r^{(k+1)} = (x^{(k+1)} \cdot Ax^{(k+1)})$
4. If not done, return to Step 2.

Exercise 2: Write an m-file that takes a specified number of steps of the power method. Your file
should have the signature:

function [r, x, R, X] = power_method ( A, xold, nsteps )
% [r, x, R, X] = power_method ( A, xold, nsteps )
% comments

% your name and the date
. .
for k=1:nsteps
. .
. r= ??? %Rayleigh quotient
. R(k)=r;% save Rayleigh quotient on each step
. X(k)=x(1); % save one eigenvector component on each step
end

The history variables R and X will be used to illustrate the progress that the power method is making
and have no mathematical importance.

Note: It is usually the case that when writing code such as this that requires the Rayleigh quotient,
I would ask that you use rayleigh.m, but in this case the Rayleigh quotient is so simple that it is
probably better to just copy the code for it into power_method.m.

Using your power method code, try to determine the largest eigenvalue of the matrix eigen_test(1),
starting from the vector $[1;1;1]$. Be sure to make this starting vector into a unit vector if you don't
use rayleigh.m. (I have included the result for step 1 to help you check your code out.)

<table>
<thead>
<tr>
<th>Step</th>
<th>Rayleigh q.</th>
<th>$x(1)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>1.0</td>
</tr>
<tr>
<td>1</td>
<td>0.72353</td>
<td>0.0</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
In addition, plot the progress of the method with the following commands:

```matlab
plot(R); title 'Rayleigh quotient vs. k'
plot(X); title 'Eigenvector component vs. k'
```

Please send me these two plots with your summary.

**Exercise 3:**

(a) Revise your `power_method.m` to a new function named `power_method1.m` with the signature:

```matlab
function [r, x, R, X] = power_method1 ( A, xold , tol )
% [r, x, R, X] = power_method1 ( A, xold , tol )
% comments
% your name and the date
	nsteps=10000; % maximum allowable number of steps
```

To determine when to stop iterating, add code to effect all three of the following criteria:

\[
\begin{align*}
    k & > 1 \\
    |r(k) - r(k-1)| & < \epsilon |r(k)| \quad \text{and} \\
    |(|x_1|^k) - |x_1|^{(k-1)}| & < \epsilon |x_1|^k
\end{align*}
\]

where the Matlab variable `tol` represents \( \epsilon \). Look carefully at the absolute values in the last expression above! The second expression says that \( r \) is accurate to a relative error of \( \epsilon \), and the third says that only the absolute value of the first component of the eigenvector is accurate to a relative error of \( \epsilon \). The denominator of the relative error expression multiplies both sides of the inequality so that numerical errors do not cause trouble when \( r \) or \( x_1 \) is small. (This is not a very good stopping criterion, but it will do for now.)

(b) Using your new power method code, try to determine the dominant eigenvalue, and corresponding eigenvector, of the `eigen_test` matrices by repeatedly taking power method steps. Use a starting vector of all 1’s, and a tolerance of 1.e-8. **Hint:** The number of iterations is given by the length of \( R \) or \( X \). If the number of iterations is equal to 1000 (the value of `nsteps`) then you know it did not converge!

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Rayleigh q.</th>
<th>x(1)</th>
<th>no. iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

These matrices were chosen to illustrate different iterative behavior. Some converge quickly and some slowly, some oscillate and some don’t. One does not converge. Generally speaking, the eigenvalue converges more rapidly than the eigenvector. You can check the eigenvalues of these matrices using the `eig` function. Plot the histories of Rayleigh quotient and first eigenvector component for all the cases. Send me the plots for cases 3, 5, and 7.

**Matlab hint:** The `eig` command will show you the eigenvectors as well as the eigenvalues of a matrix \( A \), if you use the command in the form:
The quantities $V$ and $D$ are matrices, storing the $n$ eigenvectors as columns in $V$ and the eigenvalues along the diagonal of $D$.

5 The Inverse Power Method

The inverse power method reverses the iteration step of the power method. We write:

$$Ax^{(k+1)} = x^{(k)}$$

or, equivalently,

$$x^{(k+1)} = A^{-1}x^{(k)}$$

In other words, this looks like we are just doing a power method iteration, but now for the matrix $A^{-1}$. You can immediately conclude that this process will often converge to the eigenvector associated with the largest eigenvalue of $A^{-1}$.

This means that we can freely choose to study the eigenvalue problem of either $A$ or $A^{-1}$, and easily convert information from one problem to the other. The difference is that the inverse power iteration will find us the biggest eigenvalue of $A^{-1}$, and that’s the eigenvalue of $A$ that’s smallest in magnitude, while the plain power method finds the eigenvalue of $A$ that is largest in magnitude.

The \textbf{inverse power method} iteration is given in the following algorithm.

1. Starting with any nonzero vector $x$, divide by its length to make a unit vector called $x^{(0)}$.
2. Solve $A\hat{x} = x^{(k)}$, and
3. Normalize the iterate by setting $x^{(k+1)} = \hat{x}/||\hat{x}||$;
4. Compute the Rayleigh quotient of the iterate (unit vector) $r^{(k+1)} = (x^{(k+1)} \cdot Ax^{(k+1)})$; and,
5. Return to step 2.

\textbf{Exercise 4}:

(a) Write an m-file that computes the inverse power method, stopping when

$$k > 2$$

$$|r^{(k)} - r^{(k-1)}| < \epsilon |r^{(k)}|,$$

$$||x^{(k)} - x^{(k-1)}|| < \epsilon,$$

where $\epsilon$ denotes a specified relative tolerance. ($x^{(k)}$ is a unit vector so the second condition is a relative condition.)

\textbf{Warning}: The convergence on $x$ is different, and better, than the one we used in \textbf{power method1}. You will have to keep track of the previous iterate to implement it.

Your file should have the signature:

\begin{verbatim}
function [r, x, R, X] = inverse_power ( A, xold, tol)
% [r, x, R, X] = inverse_power ( A, xold, tol)
% comments

% your name and the date
\end{verbatim}
Be sure to include a check on the maximum number of iterations (10000 is a good choice) so that a non-convergent problem will not go on forever.

(b) Using your inverse power method code, determine the smallest eigenvalue of the matrix $A=eigen\textunderscore test(1)$,

$$[r \times R \times]=inverse\textunderscore power(A, [1;1;1], 1.e-8);$$

Please include $r$ and $x$ in your summary. How many steps did it take?

(c) As a debugging check, run

$$[rp \times xp \times Rp \times]=power\textunderscore method(inv(A), [1;1;1], nsteps);$$

where $nsteps$ is the number of steps you just took using $inverse\textunderscore power$. You should find that $r$ is close to $1/rp$, that $x$ and $xp$ are close, and that $X$ and $Xp$ are close. If these relations are not true, you have an error somewhere. Fix it before continuing. **Warning:** the convergence criteria between the two methods is different, so they may take slightly different numbers of iterations. Compare $X$ and $Xp$ only for the common iterations.

**Exercise 5:**

(a) Apply the inverse power method to the matrix $A=eigen\textunderscore test(3)$ starting from the vector of all ones and with a tolerance of $1.e-8$. You will find that it does not converge. Even if you did not print some sort of non-convergence message, you can tell it did not converge because the length of the vectors $R$ and $X$ is precisely the maximum number of iterations you chose. In the following steps, you will discover why it did not converge and fix it.

(b) Plot the Rayleigh quotient history $R$ and the eigenvector history $X$. You should find that $X$ oscillates in a plus/minus fashion, and that is the cause of non-convergence. Include your plot with your summary. **Hint:** you may find the plot hard to interpret. Either zoom in or plot only the final 50 or so values.

(c) If a vector $x$ is an eigenvector of a matrix, then so is $-x$, and with the same eigenvalue. The following code will force the entry of an eigenvector with largest magnitude to have a consistent sign. If you include it in your $inverse\textunderscore power.m$, it will still return an eigenvector, but the largest component of the eigenvector will always be nonnegative and the eigenvector will no longer flip signs from iteration to iteration.

```matlab
% find location of entry with largest magnitude and
% multiply all of x so its entry with largest
% magnitude is a positive number. This code only
% works correctly when x is a real vector.
[absx,location]=max(abs(x));
factor=sign(x(location));
x=factor*x; % makes x(location)>=0
```

(d) Try $inverse\textunderscore power$ again. It should converge very rapidly.

(e) Fill in the following table with the eigenvalue of smallest magnitude. Start from the vector of all ones and use a tolerance of $1.e-8$.

<table>
<thead>
<tr>
<th>Matrix</th>
<th>eigenvalue</th>
<th>x(1)</th>
<th>no. iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Be careful! One of these still may not converge.
6 Finding Several Eigenvectors at Once

It is common to need more than one eigenpair, but not all eigenpairs. For example, finite element models of structures often have upwards of a million degrees of freedom (unknowns) and just as many eigenvalues. Only the lowest few are of practical interest. Fortunately, matrices arising from finite element structural models are typically symmetric and negative definite, so the eigenvalues are real and negative and the eigenvectors are orthogonal to one another. For the remainder of this section, and this section only, we will be assuming that the matrix whose eigenpairs are being sought is symmetric.

If you try to find two eigenvectors by using the power method (or the inverse power method) twice, on two different vectors, you will discover that you get two copies of the same dominant eigenvector. (The eigenvector with eigenvalue of largest magnitude is called “dominant.”) Working smarter helps, though. Because the matrix is assumed symmetric, you know that the dominant eigenvector is orthogonal to the others, so you could use the power method (or the inverse power method) twice, forcing the two vectors to be orthogonal at each step of the way. This will guarantee they do not converge to the same vector. If this approach converges to anything, it probably converges to the dominant and “next-dominant” eigenvectors and their associated eigenvalues.

We considered orthogonalization in Lab 4 and we wrote a function called `gramschmidt.m`. You can use your copy or mine for this lab. You will recall that the `gramschmidt` function regards the vectors as columns of a matrix: the first vector is the first column of a matrix $X$, the second is the second column, etc. (Be warned that this function will sometimes return a matrix with fewer columns than the one it is given. This is because the original columns were not a linearly independent set.)

The power method applied to several vectors can be described in the following algorithm:

1. Start with any linearly independent set of vectors stored as columns of a matrix $V$, use the Gram-Schmidt process to orthonormalize this set and generate a matrix $V^{(0)}$;
2. Generate the next iterates by computing $\hat{V} = AV^{(k)}$;
3. Orthonormalize the iterates $\hat{V}$ using the Gram-Schmidt process and name them $V^{(k+1)}$;
4. Adjust signs of eigenvectors as necessary.
5. Compute the Rayleigh quotient of the iterates as the diagonal entries of the matrix $(V^{(k+1)})^TAV^{(k+1)}$;
6. If converged, exit; otherwise return to step 2.

**Note:** There is no guarantee that the eigenvectors that you find have the largest eigenvalues in magnitude! It is possible that your starting vectors were themselves orthogonal to one or more eigenvectors, and you will never recover eigenvectors that are not spanned by the initial ones. In applications, there are consistency checks that serve to mitigate this problem.

In the following exercise, you will write a Matlab function to carry out this procedure to find several eigenpairs for a symmetric matrix. The initial set of eigenvectors will be represented as columns of a Matlab matrix $V$.

**Exercise 6:**

(a) Begin writing a function mfile named `power_several.m` with the signature

```
function [R, V, k] = power_several ( A, Vold , tol )
% [R, V, k] = power_several ( A, Vold , tol )
% comments
% your name and the date
```
to implement the algorithm described above.
(b) Add comments explaining that \( V \) and \( V\text{old} \) are matrices whose columns represent the eigenvectors and that \( R \) is a diagonal matrix with the eigenvalues on the diagonal.

(c) You can use the following code to make sure the largest entry in each column is positive.

\[
\begin{align*}
[\text{abs}\, V, \text{indices}] &= \text{max(abs(V))}; & \% \text{indices of largest in columns} \\
\text{d} &= \text{diag(\text{sign(V(indices,:)))}}; & \% \text{pick out their signs as +}\, -1 \\
\text{D} &= \text{diag(d)}; & \% \text{diagonal matrix with +}\, -1 \\
\text{V} &= \text{V*D}; & \% \text{Transform V}
\end{align*}
\]

(d) Stop the iterations when both of the following conditions are satisfied.

\[
\begin{align*}
|R^{(k)} - R^{(k-1)}|_{\text{fro}} &< \epsilon| R^{(k)}|_{\text{fro}}, \text{ and} \\
|V^{(k)} - V^{(k-1)}|_{\text{fro}} &< \epsilon| V^{(k)}|_{\text{fro}}.
\end{align*}
\]

(e) Test \texttt{power\_several} for \( A=\text{eigen\_test}(3) \) starting from the vector of all ones, with a tolerance of 1.0e-8. It should be close to the result in Exercise 3.

(f) Run \texttt{power\_several} function for the matrix \( A=\text{eigen\_test}(4) \) with a tolerance of 1.0e-8 starting from the vectors

\[
V = \begin{bmatrix}
1 & 1 & 1 \\
1 & 1 & 1 \\
1 & -1 & 1 \\
1 & -1 & 1 \\
1 & -1 & 1 \\
1 & -1 & 1 \\
\end{bmatrix}
\]

You should immediately recognize that these vectors are not only linearly independent, but also orthogonal. Is the largest eigenvalue this time the same as before?

(g) Further test your results by checking that the two eigenvectors you found are actually eigenvectors and are orthogonal. (Test that \( AV=VR \), where \( V \) is your matrix of eigenvectors and \( R \) is the diagonal matrix of eigenvalues.)

(h) Since this \( A \) is a small \((6 \times 6)\) matrix, you can find all eigenpairs by using \texttt{power\_several} starting from the (obviously linearly independent) vectors

\[
V = \begin{bmatrix}
1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & -1 & 1 \\
1 & 1 & 1 & -1 & -1 & 1 \\
1 & 1 & -1 & -1 & -1 & 1 \\
1 & -1 & -1 & -1 & -1 & 1 \\
1 & -1 & -1 & -1 & -1 & 1 \\
\end{bmatrix}
\]

What are the six eigenvalues?

7 Using Shifts

A major limitation of the power method was that it can only find the eigenvalue of largest magnitude. Similarly, the inverse power iteration seems only able to find the eigenvalue of smallest magnitude. Suppose, however, that you want one (or a few) eigenpairs with eigenvalues that are away from the extremes? Instead of working with the matrix \( A \), let’s work with the matrix \( A+\sigma I \), where we have shifted the matrix by adding \( \sigma \) to every diagonal entry. This shift makes a world of difference.

We already said the inverse power method finds the eigenvalue of smallest magnitude. Denote the smallest eigenvalue of \( A+\sigma I \) by \( \mu \). If \( \mu \) is an eigenvalue of \( A+\sigma I \) then \( A+\sigma I - \mu I = A - (\mu - \sigma)I \) is singular. In other words, \( \lambda = \mu - \sigma \) is an eigenvalue of \( A \), and is the eigenvalue of \( A \) that is closest to \( \sigma \).
In general, the idea of shifting allows us to focus the attention of the inverse power method on seeking the eigenvalue closest to any particular value we care to name. This also means that if we have an estimated eigenvalue, we can speed up convergence by using this estimate in the shift. Although various strategies for adjusting the shift during the iteration are known, and can result in very rapid convergence, we will be using a constant shift.

We are not going to worry here about the details or the expense of shifting and refactorizing the matrix. (A real algorithm for a large problem would be very concerned about this!) Our simple-minded method algorithm for the shifted inverse power method looks like this:

1. Starting with any nonzero vector $x^{(0)}$, and given a fixed shift value $\sigma$:
   2. Solve $(A - \sigma I)\hat{x} = x^{(k)}$, and
   3. Normalize the iterate by setting $x^{(k+1)} = \hat{x}/||\hat{x}||$; and,
   4. Adjust the sign of the eigenvector as necessary.
   5. Estimate the eigenvalue as
   $\begin{align*}
   \lambda^{(k+1)} &= \sigma + (x^{(k+1)} \cdot (A - \sigma I)x^{(k+1)}) \\
   &= (x^{(k+1)} \cdot Ax^{(k+1)})
   \end{align*}$

6. If converged, exit; otherwise return to step 2.

Exercise 7:
Starting from your inverse_power.m, write an m-file for the shifted inverse power method. Your file should have the signature:

```matlab
function [r, x, nsteps] = shifted_inverse ( A, xold, shift, tol)
% [r, x, nsteps] = shifted_inverse ( A, xold, shift, tol)
% comments
% your name and the date
```

(a) Compare your results from inverse_power.m applied to the matrix $A=eigen\_test(2)$ using a relative tolerance of $1.0e^{-8}$ with results from shifted_inverse.m with a shift of 0. The results should be identical and take the same number of steps.

(b) Apply shifted_inverse to the same $A=eigen\_test(2)$ with a shift of 1.0. You should get the same eigenvalue and eigenvector as with a shift of 0. How many iterations did it take this time?

(c) Apply shifted_inverse.m to the matrix $eigen\_test(2)$ with a shift of 4.73, very close to the largest eigenvalue. Start from the vector of all 1’s and use a relative tolerance of $1.0e^{-8}$. You should get a result that agrees with the one you calculated using power\_method but it should take very few steps. (In fact, it gets the right answer on the first step, but convergence detection is not that fast.)

(d) Using your shifted inverse power method code, we are going to search for the “middle” eigenvalue of matrix $eigen\_test(2)$. The power method gives the largest eigenvalue as about 4.73 and the the inverse power method gives the smallest as 1.27. Assume that the middle eigenvalue is near 2.5, start with a vector of all 1’s and use a relative tolerance of $1.0e^{-8}$. What is the eigenvalue and how many steps did it take?

(e) Recapping, what are the three eigenvalues and eigenvectors of $A$? Do they agree with the results of eig?
In the previous exercise, we used the backslash division operator. This is another example where the matrix could be factored first and then solved many times, with a possible time savings.

The value of the shift is at the user’s disposal, and there is no good reason to stick with the initial (guessed) shift. If it is not too expensive to re-factor the matrix \( A - \sigma I \), then the shift can be reset after several iterations. One possible choice for \( \sigma \) would be to set it equal to the current eigenvalue estimate, and do so after every, say, third iteration. The closer the shift to the eigenvalue, the faster the convergence.

8 The QR Method

So far, the methods we have discussed seem suitable for finding one or a few eigenvalues and eigenvectors at a time. In order to find more eigenvalues, we’d have to do a great deal of special programming. Moreover, these methods are surely going to have trouble if the matrix has repeated eigenvalues, distinct eigenvalues of the same magnitude, or complex eigenvalues. The “QR method” is a method that handles these sorts of problems in a uniform way, and computes all the eigenvalues, but not the eigenvectors, at one time.

The QR method can be described in the following way.

- Given a matrix \( A^{(k)} \), construct its QR factorization; and,
- Set \( A^{(k+1)} = RQ \).

The construction of the QR factorization means that \( A^{(k)} = QR \), and the matrix factors are reversed to construct \( A^{(k+1)} \).

The sequence of matrices \( A^{(k)} \) is orthogonally similar to the original matrix \( A \), and hence has the same eigenvalues. This is because \( A^k = QR \), and \( A^{k+1} = RQ = Q^TQRQ = Q^T A^k Q \). Under fairly general conditions, the sequence of matrices \( A^{(k)} \) converges to

- A diagonal matrix if the \( A \) was symmetric.
- An upper triangular matrix if the eigenvalues are all real.
- An “almost” upper triangular matrix, where the main subdiagonal will have nonzero entries only when there is a complex conjugate pair of eigenvalues.

The Matlab command

\[
[ Q, R ] = \text{qr} ( A )
\]

computes the QR factorization (as does our own \texttt{h_factor} routine from Lab 4).

Exercise 8:

(a) Write an m-file for the QR method for a matrix \( A \). Your file should have the signature:

```matlab
function [E,k] = qr_method ( A, tol)
% [E,k] = qr_method ( A, tol)
% comments

% your name and the date

E is a diagonal matrix of eigenvalues of \( A \). Since the matrix \( A^{(k)} \) is supposed to converge, terminate the iteration when

\[
||A^{(k+1)} - A^{(k)}||_{\text{fro}} \leq \epsilon ||A^{(k+1)}||_{\text{fro}}
\]

where \( \epsilon \) is the relative tolerance. Include a check so the number of iterations never exceeds some maximum (10000 is a good choice).
(b) The eigen_test(2) matrix is $3 \times 3$ and we have found its eigenvalues in exercises 3, 4, and 7. Check that you get the same values using qr_method. Use a relative tolerance of 1.0e-8.

(c) It turns out that a “detail” has been left out so far. Try your qr_method function on the matrix $A=eigen_test(1)$. You will find that it takes more than 10000 iterations, and this is many more than it should. In order to see what might be wrong, reduce the maximum number of steps to 50 and print the matrix $A$ each iteration. You can see that the values of $A$ on the diagonal appear to have converged to the correct answer immediately, the values below the diagonal are small and getting smaller, and some values above the diagonal are flipping signs (plus to minus and back again) each iteration. Before continuing, return the maximum number of steps to 10000. Eventually, the values below the diagonal will become exactly zero on the computer, at which point $Q$ becomes the identity and the signs no longer flip each iteration, and convergence is achieved.

(d) The QR factorization is only unique up to multiplication by a diagonal matrix of ±1. This is related to the fact that the columns of $Q$ are orthonormal vectors, and multiplying any one of them by -1 doesn’t change that fact. You have encountered this issue before, and it can be fixed as before. The following code will choose signs in such a way that the largest component of each column of $Q$ is positive.

```
[v,indices]=max(abs(Q)); % indices of largest in columns
d=diag(sign(Q(indices,:))); % pick out their signs as +1
D=diag(d); % diagonal matrix with +-1
Q=Q*D; % Transform Q
R=D*R; % Transform R so still have A=Q*R
```

Add this code just after doing the QR decomposition in your code and try your revised code on $A=eigen_test(1)$. It should converge. How many steps does it take?

(e) Try out the QR iteration for the eigen_test matrices except number 5. Use a relative tolerance of 1.0e-8. Report the results in the following table.

<table>
<thead>
<tr>
<th>Matrix</th>
<th>3 largest Eigenvalues</th>
<th>Number of steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(f) Matrix 5 has 4 distinct eigenvalues of equal norm, and the QR method does not converge. Temporarily reduce the maximum number of iterations to 50 and change qr_method.m to print diag(E) at each iteration to see why it fails. What happens to the iterates? As with the case of changing signs, it is possible to “handle” this case in the code so that it converges, but we will not pursue it here. Return the maximum number of iterations to its large value and make sure nothing is printed each iteration before continuing.

Serious implementations of the QR method save information about each step in order to construct the eigenvectors. The algorithm also uses shifts to try to speed convergence. Some eigenvalues can be determined early in the iteration, which speeds up the process even more. In the following section we consider the question of convergence of the algorithm.

9 Convergence of the QR Algorithm

Throughout this lab we have used a very simple convergence criterion. In general, convergence is a difficult subject and fraught with special cases and exceptions to rules. To give you a flavor of how convergence
might be addressed, we consider the case of the QR algorithm applied to a real, symmetric matrix, \( A \), with distinct eigenvalues, no two of which are the same in magnitude. For such matrices, the following theorem can be shown.

**Theorem** Let \( A \) be a real symmetric matrix of order \( n \) and let its eigenvalues satisfy

\[
|\lambda_1| > |\lambda_2| > \cdots > |\lambda_n| > 0.
\]

Then the iterates \( \{A^{(k)}\} \) of the QR method will converge to a diagonal matrix \( D \) whose entries are the eigenvalues of \( A \). Furthermore,

\[
\|D - A^{(k)}\| \leq C \max_k \left|\frac{\lambda_{k+1}}{\lambda_k}\right|
\]

The theorem points out a method for detecting convergence. Supposing you start out with a matrix \( A \) satisfying the above conditions, and you call the matrix at the \( k \)th iteration \( A^{(k)} \). If \( D^{(k)} \) denotes the matrix consisting only of the diagonal entries of \( A^{(k)} \), then the Frobenius norm of \( A^{(k)} - D^{(k)} \) must go to zero as a power of \( \rho = \max |\lambda_k|/|\lambda_{k+1}| \), and if \( D \) is the diagonal matrix with the true eigenvalues on its diagonal, then \( D^{(k)} \) must converge to \( D \) as a geometric series with ratio \( \rho \). In the following algorithm, the diagonal entries of \( A^{(k)} \) (these are the eigenvalues) will be the only entries considered. The off-diagonal entries converge to zero, but there is no need to make them small so long as the eigenvalues are converged.

The following algorithm includes a convergence estimate as part of the QR algorithm.

1. Define \( A^{(0)} = A \) and \( \epsilon^{(0)} = \text{diag}(A) \).
2. Compute \( Q \) and \( R \) uniquely so that \( A^{(k)} = QR \), set \( A^{(k+1)} = RQ \) and \( \epsilon^{(k+1)} = \text{diag}(A^{(k+1)}) \). (This is one step of the QR algorithm.)
3. Sort the entries of \( \epsilon^{(k+1)} \) in ascending order of absolute value, call them \( \lambda_n \) and compute \( \rho = \max |\lambda_n|/|\lambda_{n+1}| \). (Matlab has a function, `sort`, that sorts a vector in ascending order.)
4. Compute the eigenvalue convergence estimate

\[
\frac{\|\epsilon^{(k+1)} - \epsilon^{(k)}\|}{(1 - \rho)\|\epsilon^{k+1}\|}
\]

Return to step 2 if it exceeds the required tolerance.

You may wonder where the denominator \( (1 - \rho) \) came from. Consider the geometric series

\[
S = 1 + \rho + \rho^2 + \rho^3 + \cdots + \rho^k + \rho^{k+1} + \cdots = \frac{1}{1 - \rho}.
\]

Truncating this series at the \( k \)th term and calling the truncated sum \( S_k \), we have

\[
S - S_k = \rho^{k+1}(1 + \rho + \rho^2 + \ldots) = \frac{\rho^{k+1}}{1 - \rho}.
\]

But, \( \rho^{k+1} = S_{k+2} - S_{k+1} \), so

\[
S - S_k = \frac{S_{k+2} - S_{k+1}}{1 - \rho}.
\]

Thus, the error made in truncating a geometric series is the difference of the two next terms divided by \( (1 - \rho) \).

**Exercise 9:**
(a) Copy your qr_method.m file to qr_convergence.m and change the signature line to

```matlab
function [E, nsteps]=qr_convergence(A,tol)
% [E, nsteps]=qr_convergence(A,tol)
% comments
% your name and the date
Modify the code to implement the above algorithm.
```

(b) Apply your algorithm to eigen_test(8), with a tolerance of 1.0e-8. Include the eigenvalues and the number of iterations in your summary.

(c) Compare your computed eigenvalues with ones computed by eig. How close are your eigenvalues? Is the relative error near the tolerance?

(d) If you decrease the tolerance by a factor of ten to 1.0e-9, Does the relative error decrease by a factor of ten as well? This test indicates the quality of the error estimate.

(e) Apply qr_convergence to A=eigen_test(7) with a tolerance of 1.0e-8. How many iterations does it take?

(f) Use ee=eig(A) to compute “exact” eigenvalues for eigen_test(7). The entries of ee might not be in the same order, so both should be sorted in order to be compared. What is the relative error

\[
\text{norm(sort(e)-sort(ee))/norm(e)}
\]

where e=diag(E) is the vector of eigenvalues from qr_convergence?

(g) How many iterations does qr_method take to reach the (relative tolerance) of 1.0e-8 for the matrix A=eigen_test(7)? Which convergence method yields more accurate results? If your paycheck depended on achieving a particular accuracy, which method would you use?

(h) Convergence based on provable results is very reliable if the assumptions are satisfied. Try to use your qr_convergence function on the nonsymmetric matrix

\[
A = [ .01 \ 1 \\
-1 \ .01 ];
\]

It should fail.