4.1 The LU Decomposition

The Elementary Matrices and \texttt{badgauss}

In the previous chapters we talked a bit about the solving systems of the form $Lx = b$ and $Ux = b$ where $L$ is lower triangular and $U$ is upper triangular. In the exercises for Chapter 2 you were asked to write a program $x=\text{lusolver}(L,U,b)$ which solves $LUx = b$ using \texttt{forsub} and \texttt{backsub}. We now address the problem of representing a matrix $A$ as a product of a lower triangular $L$ and an upper triangular $U$. Recall our old friend \texttt{badgauss}.

function $B=\text{badgauss}(A)$

$m=\text{size}(A,1);$ 
$B=A;$

for $i=1:m-1$

for $j=i+1:m$

$a=-B(j,i)/B(i,i);$ 
$B(j,:)=a*B(i,:)+B(j,:);$ 
end
end

The heart of \texttt{badgauss} is the elementary row operation of type 3:

$B(j,:)=a*B(i,:)+B(j,:);$ 

where $a=-B(j,i)/B(i,i);$ Note also that the index $j$ is greater than $i$ since the loop is

for $j=i+1:m$

As we know from linear algebra, an elementary opreration of type 3 can be viewed as matrix multiplication $EA$ where $E$ is an elementary matrix of type 3. $E$ looks just like the identity matrix, except that $E(j,i)=a$ where $j,i$ and $a$ are as in the MATLAB code above. In particular, $E$ is a lower triangular matrix, moreover the entries on the diagonal are 1’s. We call such a matrix \textit{unit lower triangular}.

\textbf{Theorem.}

1. The product of (unit) lower triangular matrices is (unit) lower triangular.
2. The inverse of a (unit)lower triangular matrix is (unit) lower triangular.

From \texttt{badgauss} we can conclude the following:

$L_k \ldots L_1 A = U$

where $L_k \ldots L_1$ is a sequence of unit lower triangular matrices (in fact elementary matrices of type 3.) The product of these is unit lower triangular and so is the inverse, thus letting $L = (L_k \ldots L_1)^{-1}$, we get $A = LU$. There is an easier and more efficient way to get $L$. It turns out that $L(j,i)=-a$ where $a$ is given in the \texttt{badgauss} code above.

\textit{Row Switching}

The necessity of row switching is seen in \texttt{badgauss} to avoid division by 0. It turns out that it is necessary to perform row interchanges for numerical reasons as well. Consider the following example of Forsythe and Moler.

$$\begin{pmatrix} \text{ulp} & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$$
where \( u = \text{ulp} \). The exact answer is

\[
x = \frac{1}{1 - \text{ulp}} \quad y = \frac{1 - 2u \text{ulp}}{1 - \text{ulp}}
\]

Rounding gives the respectable answer, \( x = 1 \) and \( y = 1 \). See if MATLAB agrees with this answer. Now suppose that we run \texttt{badgauss} on \([A, b]\), then we get

\[
\begin{pmatrix}
\text{ulp} & 1 & 1 \\
0 & -1/\text{ulp} & -1/\text{ulp}
\end{pmatrix}
\]

Which gives the answer \( x = 0 \) and \( y = 1 \) by back substitution. What is the relative error of this answer?

\[
\frac{\| \begin{pmatrix} 1 \\ 1 \end{pmatrix} - \begin{pmatrix} 0 \\ 1 \end{pmatrix} \|_2}{\| \begin{pmatrix} 1 \\ 1 \end{pmatrix} \|_2}
\]

Now switch the first and second rows of \([A, b]\) and run \texttt{badgauss} again. You should get

\[
\begin{pmatrix} 1 & 1 & 2 \\
0 & 1 & 1
\end{pmatrix}
\]

Which gives \( x = 1 \) and \( y = 1 \) by back substitution, an excellent answer.

An approach to correcting this type of error is the \textit{partial pivoting strategy}. Look at the \texttt{badgauss} code on the previous page. On the \( i \)th iteration of the Gaussian Elimination instead of automatically pivoting on \( B(i, i) \), choose \( k \geq i \) where

\[
|B(k, i)| = \max(|B(i : n, i)|)
\]

and pivot on \( B(k, i) \) instead (this will be \( B(j, j) \) after a row switch.) We are faced with the prospect of row interchanges, \textit{i.e.} multiplication by elementary matrices of type 1. This will kill the possibility of \( L \) being lower triangular. The way out of this is by keeping track of all the row switches. A \textit{permutation matrix} is a product of row switching matrices. Permutation matrices are easily identifiable as each row and each column has exactly one 1 and the other entries in the row or column are 0. The PLU Decomposition combines the row switching with the LU Decomposition.

\textbf{Theorem. The PLU Decomposition.} If \( A \) is an \( n \times n \) matrix then there is a permutation matrix \( P \), a unit lower triangular matrix \( L \) and an upper triangular matrix \( U \) such that

\[
P A = L U
\]

MATLAB has a function which computes the LU Decomposition.

\[
A = \text{ones}(5) + \text{diag}(1:5)
\]

\[
[L, U] = \text{lu}(A),
\]

\[
A - L \cdot U
\]

The lower triangular matrix \( L \) obtained always has 1’s on the main diagonal. This permits a storage method which holds \( U \) on and above the diagonal and \( L \) below the diagonal. The call \( B = \text{lu}(A) \) will return a matrix \( B \) which stores \( L \) and \( U \) in this manner. The call \( [L, U] = \text{lu}(A) \) will return two matrices; with \( U \) an upper triangular matrix though the \( L \) will not necessarily be lower triangular – the rows need to be permuted. But it is the case that \( A = L \cdot U \).

\[
B = \text{lu}(A)
\]

\[
\text{triu}(B) - U
\]
But notice that \( L \) may not be lower triangular

\[
A = \text{magic}(5);
\]

\[
[L, U] = \text{lu}(A)
\]

By permuting the rows of \( L \) we can make it into a lower triangular matrix. MATLAB will do a PLU Decomposition with

\[
[L, U, P] = \text{lu}(A)
\]

We solve a system of equations using the PLU Decomposition in basically the same way as with an LU Decomposition. Suppose that \( PA = LU \) and \( Ax = b \) is to be solved, then \( A = P^{-1}LU \). Now solve \( Ly = Pb \) and \( Ux = y \), then

\[
Ax = P^{-1}LUx = P^{-1}Ly = P^{-1}Pb = b.
\]

### 4.2 The QR Decomposition

The QR Decomposition factors a matrix \( A \) as \( A = QR \) where \( Q \) is orthogonal and \( R \) is upper triangular. If the matrix \( A \) has linearly independent columns, then \( n \geq m \), and using the Gram-Schmidt process, we can find an \( m \times n \) orthogonal matrix using \( Q = \text{qrmsch}(A) \) and an \( n \times n \) upper triangular matrix \( R \). In general given any \( A \) we can find using either Householder reflections or Givens rotations an \( m \times m \) orthogonal matrix \( Q \) and an \( m \times n \) upper triangular matrix \( R \). We will do both versions of the QR decomposition.

**Gram Schmidt QR**

The Gram-Schmidt process is the following:

Given vectors \( v_1, \ldots, v_n \) which are linearly independent,

\[
u_1 = v_1 / \| v_1 \|
\]

for \( i = 2 : n \)

\[
w = v_i
\]

for \( j = 1 : i - 1 \)

\[
w = w - (v_i \cdot u_j)u_j
\]

end

\[
u_i = w / \| w \|
\]

end

We have deliberately written Gram-Schmidt in this pseudo-code rather than MATLAB to highlight the dot product in the inner loop. Now if we solve for \( v_i \) in this we see that

\[
v_i = \| w \| u_i + (v_1 \cdot u_1)u_1 + \ldots + (v_{i-1} \cdot u_{i-1})u_{i-1}
\]

Or in matrix form if we let \( Q = [u_1, \ldots, u_n] \), and define

\[
R(i,j) = \begin{cases} 
  u_i \cdot v_j & \text{if } j > i \\
  \| w \| & \text{if } j = i \\
  0 & \text{if } j < i 
\end{cases}
\]

then it is easy to check that \( A = QR \). Note that if we know \( Q \), then it is easy to find \( R \) by multiplying both sides by \( Q^T \), to get \( R = Q^T A \). This would be an inefficient way to compute \( R \).

Using the Gram-Schmidt QR we can solve the Least Squares problem for \( Ax = b \) by looking at the normal equations: \( A^T Ax = A^T b \),

\[
A^T Ax = R^T Q^T QRx = R^T Rx = R^T Q^T b
\]
And clearly the solution to $Rx = Q^T b$ will solve this equation. Use \texttt{backsub} to solve $Rx = Q^T b$.

**Modified Gram Schmidt**

The Gram Schmidt algorithm is unstable for certain matrices (see the Project) but a rather minor rearrangement of the computation will greatly improve the accuracy. The Modified Gram Schmidt method is the following:

Again assuming $v_1, \ldots, v_n$ are linearly independent.

\begin{verbatim}
for i = 1 : n - 1  
u_i = v_i/\|v_i\|  
for j = i + 1 : n  
v_j = v_j - (v_j \cdot u_i) u_i  
end  
end  
\end{verbatim}

$u_n = v_n/\|v_n\|$

The result of using the modified Gram Schmidt method will produce an $m \times n$ orthogonal matrix $Q$. The upper triangular matrix $R$ comes from the basic projection relation

\[  w_i = \sum_{k=1}^{i-1} (v_i \cdot u_k) u_k \]

\[  u_i = w_i/\|w_i\| \]

Which under the Modified Gram Schmidt arrangement becomes

\[  v_i = \sum_{k=1}^{i-1} (v_i \cdot u_k) u_k + \|v_i\| u_i \]

Thus, $R(i,j) = v_j \cdot u_i$, while $R(i,i) = \|v_i\|$.

**Givens QR**

A \textit{Givens rotation}, is arrived at by starting with an identity matrix $G = I$ and assigning $G(i,i) = c, G(i,j) = s, G(j,i) = -s$, and $G(j,j) = c$ where $c^2 + s^2 = 1$. Call this matrix \texttt{givrot(n,i,j,c,s)}. This produces a rotation in the $ij$-plane.

**Theorem.** $G = \text{givrot}(n,i,j,c,s)$ is an orthogonal matrix.

The method is based on the fact that a Givens rotation affects only entries in rows $i$ and $j$.

**Theorem.** For any vector $v$ and integers $i < j$ there is a Givens rotation $G$ such that for $w = Gv$, $w(j) = 0$ and $w(k) = v(k)$ for all $k \neq i,j$.

Proof: Since $Gv$ changes only the $i$ and $j$ entries, we get $w(k) = v(k)$ for $k \neq i,j$. To get $w(j) = 0$ we simply need to solve these equations for $c$ and $s$

\[  -sv(i) + cv(j) = 0 \quad c^2 + s^2 = 1. \]

Let

\[  s = \frac{v(j)}{\sqrt{v(i)^2 + v(j)^2}} \quad \text{and} \quad c = \frac{v(i)}{\sqrt{v(i)^2 + v(j)^2}} \]

Suppose that we have

\begin{verbatim}
v=rand(5,1)
\end{verbatim}

and we want to zero out the entry in the third position using the entry in the second position.

\begin{verbatim}
d=sqrt(v(2)^2 + v(3)^2)
s=v(3)/d, c=v(2)/d,
G=eye(5); G(2,2)=c; G(2,3)=s;
\end{verbatim}
You notice that only the entries in the second and third positions have been affected. In this respect, the Givens method is similar to Gaussian Elimination.

Say we want to zero out \( A(j,k) \). We can do this with a Givens rotation \( G \) so that \( GA \) differs from \( A \) only in the \( i \)th and \( j \)th rows for any \( i < j \). Let \( w = A(:,k) \) and choose \( G \) as in the Theorem.

Follow the structure of badgauss, by successively choosing \( n-1 \) Givens rotations zero out below the \((1,1)\) entry. Continue on to the second column by choosing \( n-2 \) Givens matrices until we zero out below \((2,2)\). Finally an upper triangular matrix is obtained. It is inadvisable to actually multiply out these Givens rotations since that will lead to an excessive number of arithmetic operations. Since Givens rotations act on two rows, they can be multiplied much more economically than by full matrix multiplication.

```matlab
A=magic(5); B=A
G*A
B(2,:)=G(2,2)*A(2,:)+G(2,3)*A(3,:)
B(3,:)=G(3,2)*A(2,:)+G(3,3)*A(3,:)
G*A-B
```

**Householder QR**

The Householder matrix \( H = H(w) = I_n - \frac{2}{w^T w}ww^T \) where \( w \) is an \( n \times 1 \) column vector. Observe that \( Hx = x - 2\text{proj}_w(x) \), where \( \text{proj}_w(x) \) is the projection of \( x \) onto \( w \). The Householder matrices are symmetric, orthogonal, and self-inverse.

**Theorem.** Let \( H \) be a Householder matrix, then \( H = HT = H^{-1} \).

We can build a Householder matrix in MATLAB with any vector.

```matlab
v=rand(5,1);
H=eye(5)-(2/(v'*v))*(v*v')
H-H'
H*H
```

Householder matrices can be used in a triangularization method by applying the following theorem. The proof tells how to select the vector \( w \) promised by the conclusion.

**Theorem.** Given vectors \( x \) and \( y \) with \( ||x|| = ||y|| \) there is a vector \( w \) such that for \( H = H(w) \), \( Hx = y \).

Proof: Let \( w = x - y \) and \( H = H(w) \). We want to show that \( Hx = y \). Let \( w_1 = \text{proj}_w(x) \) and \( w_2 = \text{proj}_w(-y) \), then \( w_1 \cdot (x - w_1) = 0 \) and \( w_2 \cdot (-y - w_2) = 0 \). By the Pythagorean Theorem

\[
||x||^2 = ||w_1||^2 + ||x - w_1||^2
\]
\[
||y||^2 = ||-y||^2 = ||w_2||^2 + ||-y - w_2||^2
\]

Now since \( ||x|| = ||y|| \) and \( x \) and \( -y \) are the adjacent sides of a rhombus with \( x - y \) on the diagonal, we get \( ||w_1||^2 = ||w_2||^2 \) and \( ||x - w_1|| = ||-y - w_2|| \). Thus \( x - y = w_1 + w_2 = 2w_1 \) and so \( y = x - 2\text{proj}_w(x) = Hx \). End of proof.

Given a vector \( v \) suppose we want to select \( w \) so that for the Householder matrix \( H = H(w) \)

\[
Hv = \begin{pmatrix} v_1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}
\]
where \( r \) is to be determined. Notice that to satisfy the hypothesis of the theorem we need \( \|v\| = \|re_1\| = |r| \) and so we choose \( w = v - re_1 \). Let’s try this in MATLAB.

```matlab
v=rand(5,1); e1=zeros(5,1); e1(1)=1;
r=norm(v), w=v-r*e1;
H=eye(5)-(2/(w'*w))*w*w'
H*v-r*e1
```

This works just fine unless \( v \) is close to \( re_1 \) and then we admit the possibility of round-off error. In this case we choose \( r = -\|v\| \) and avoid the problem while still retaining the feature that \( |r| = \|v\| = \|re_1\| \).

**Theorem.** Householder QR Factorization. Let \( A \) be any matrix, then there is an orthogonal matrix \( Q \) and an upper triangular matrix \( R \) such that \( A = QR \).

**Proof:** If the first column of \( A \) is all 0’s, go on to the second column, otherwise choose a Householder matrix \( H_1 \) so that

\[
H_1A = \begin{pmatrix}
* & * & \ldots & * \\
0 & * & \ldots & * \\
& \ddots & \ddots & \ddots \\
0 & * & \ldots & *
\end{pmatrix}
\]

Now choose

\[
H_2 = \begin{pmatrix}
1 & 0 & \ldots & 0 \\
0 & \ddots & \ddots & \ddots \\
& \ddots & \ddots & H \\
0 & \ldots & 0 & \ddots 
\end{pmatrix}
\]

where \( H \) is chosen to zero out the column below the \((1,1)\) entry of \((H_1A)(2:n,2:n)\). Continue this process to get

\[H_{n-1} \cdots H_2 H_1 A = R\]

and let \( Q = (H_n \ldots H_1)^T \).

End of proof.

We can do this easily for a \( 3 \times 3 \) matrix in MATLAB.

```matlab
A=rand(3); v1=A(:,1); e1=[1;0;0];
r1=norm(v1); w1=v1-r1*e1;
H1=eye(3)-(2/(w1'*w1))*(w*w');
A1=H1*A
A2=A1(2:3,2:3)
v2=A2(:,1); e2=[1;0];
r2=norm(v2); w2=v2-r2*e2;
H2=eye(2)-(2/(v2'*v2))*(v2*v2');
A3=H2*A2
H3=eye(3); H3(2:3,2:3)=H2
R=H3*H1*A
Q=H1*H3
Q*R-A
```

MATLAB has a function \( qr \) which does this factorization.

```matlab
[Q,R]=qr(A)
Q*R-A
```
4.3 Norms and Condition Numbers

The 2-norm also called the *Euclidean Norm* of a vector is defined to be

\[ \|x\|_2 = \left( \sum_{i=1}^{n} x_i^2 \right)^{1/2} \]

There are other useful norms: The 1-norm

\[ \|x\|_1 = \sum_{i=1}^{n} |x_i| \]

and the \( \infty \)-norm

\[ \|x\|_\infty = \max_{1 \leq i \leq n} |x_i| \]

We have the following relationships:

\[ \|x\|_1 \geq \|x\|_2 \geq \|x\|_\infty \]

MATLAB has a single function `norm` which computes all of these. In MATLAB try

\[ x=1:5; \text{norm}(x,1), \text{norm}(x,2), \text{norm}(x,\text{inf}) \]

A *norm* is defined to satisfy these properties:

1. \( \| \vec{0} \| = 0 \)
2. if \( x \neq \vec{0} \), then \( \|x\| > 0 \)
3. \( \|rx\| = |r| \cdot \|x\| \) for all scalars \( r \).
4. \( \|x + y\| \leq \|x\| + \|y\| \).  

Property (4) is called the *triangle inequality*.

We are interested in getting versions of these norms for matrices. That is, we want to define \( \|A\| \) for an arbitrary matrix. The definition will satisfy the same properties as the vector norms, *viz.*

1. \( \| \vec{0} \| = 0 \)
2. if \( A \neq \vec{0} \), then \( \|A\| > 0 \)
3. \( \|rA\| = |r| \cdot \|A\| \) for all scalars \( r \).
4. \( \|A + B\| \leq \|A\| + \|B\| \).

In addition we want the *consistency property*,

\[ \|AB\| \leq \|A\| \cdot \|B\| . \]

The 1-norm and the \( \infty \)-norm are easy to compute:

\[ \|A\|_1 = \max_{1 \leq j \leq n} \left( \sum_{i=1}^{n} |a_{ij}| \right) = \text{norm}(A,1) \]
\[ \|A\|_\infty = \max_{1 \leq i \leq n} \left( \sum_{j=1}^{n} |a_{ij}| \right) = \text{norm}(A,\text{inf}) \]

The 2-norm is defined on matrices as

\[ \|A\|_2 = \max_{\|x\|_2 = 1} \|Ax\|_2 . \]

The 2-norm is the most important but it is difficult to calculate. MATLAB offers a function `norm` to compute these norms. MATLAB uses the Singular Value Decomposition to compute the 2-norm as \( \|A\|_2 = \sigma_1 \), the largest singular value.

\[ \text{norm}(A) \text{ or norm}(A,2) . \]

The norm of a matrix can be useful in getting quick bounds on a problem. For example, \( |\lambda| \leq \|A\| \) for all eigenvalues \( \lambda \) of \( A \).

*The Condition Number for Solving \( Ax=b \)*
We are now in a position to define the condition number for solving $Ax = b$ for an invertible matrix $A$. If $\|A\|$ is any norm, then the condition number of $A$ with respect to $\|A\|$ is

$$\text{cond}(A) = \|A\|\|A^{-1}\|.$$  

The MATLAB function `cond(A)` computes the condition number for the 2-norm. The importance of the condition number is found in the next theorem. Our goal is to get an estimate on the relative error which occurs in computing a solution to the equation $Ax = b$.

**Theorem.** Let $A$ be an invertible matrix and $\text{cond}(A)$ be the condition number for some norm $\|A\|$ satisfying the consistency property. Suppose that $x_t$ is the true solution to $Ax = b$, $x_c$ is the computed solution, and $r = Ax_c - b$ is the residual. Then

$$\frac{\|x_c - x_t\|}{\|x_t\|} \leq \text{cond}(A) \frac{\|r\|}{\|b\|}.$$  

Proof: $A(x_c - x_t) = Ax_c - Ax_t = Ax_c - b = r$, so that $x_c - x_t = A^{-1}r$ and by the consistency property $\|x_c - x_t\| \leq \|A^{-1}\||r\|$. This gives

$$\frac{\|x_c - x_t\|}{\|x_t\|} \leq \|A\|\|A^{-1}\| \frac{\|r\|}{\|x_t\|\|A\|}.$$  

Since $Ax_t = b$ we have, by consistency, $\|b\| \leq \|A\|\cdot\|x_t\|$, hence $\frac{1}{\|A\|\|x_t\|} \leq \frac{1}{\|b\|}$ and

$$\frac{\|x_c - x_t\|}{\|x_t\|} \leq \text{cond}(A) \frac{\|r\|}{\|b\|}.$$  

We would like to check our accuracy by computing the norm of the residual $r$. It is possible for the residual to be small while the error in the solution is fairly high. The theorem explains this by appealing to the size of the condition number, which must then be very large. If the condition number is small and the residual is small then we can be confident that the error in the solution is small. The rule of thumb about the condition number is that if $\text{cond}(A) \approx 10^n$, then at most $n$ places of accuracy are lost. In MATLAB where 16 places are held, then we would expect $16 - n$ places of accuracy. This rule is based on the fact (which is not proven here) that when using Gaussian Elimination

$$\frac{\|x_c - x_t\|}{\|x_t\|} \approx \text{cond}(A)/10^{16}$$  

so that if $\text{cond}(A) \approx 10^n$,

$$\frac{\|x_c - x_t\|}{\|x_t\|} \approx 10^{16-n}.$$  

Since the number of accurate digits is

$$-\log_{10}\left(\frac{\|x_c - x_t\|}{\|x_t\|}\right)$$  

the first $16 - n$ places should be correct.

Notice that the Theorem is not specific about which condition number (norm) is being used; any norm which satisfies the consistency property works. The condition number is difficult to compute from its definition, because it is necessary to compute $A^{-1}$, which is as hard as solving the original problem $Ax = b$. While MATLAB does support `cond(A)` which finds the condition number with respect to the 2-norm, this is done at some expense.

**Theorem.** Suppose that $A$ is orthogonal, then

1. $Ax \cdot Ay = x \cdot y$
2. $x \cdot y = 0$ if and only if $Ax \cdot Ay = 0$
(3) \( \|Ax\|_2 = \|x\|_2 \)
(4) \( \|A\|_2 = 1 \)
(5) \( \|AB\|_2 = \|B\|_2 = \|BA\|_2 \)
(6) \( \text{cond}_2(A) = 1 \)

Proof: (1) \( Ax \cdot Ay = x^T A^T Ay = x^T y = x \cdot y \).
(2) Immediate from (1).
(3) \( \|Ax\|_2^2 = Ax \cdot Ax = \|x\|_2^2 \).
(4) \( \|A\|_2 = \max\{\|x\|_1 = 1 \} \|Ax\|_2 = 1 \), by (3).
(5) \( \|AB\|_2 \leq \|A\|_2 \cdot \|B\|_2 \) and \( \|B\|_2 = \|A^T AB\|_2 \leq \|A^T\|_2 \cdot \|AB\|_2 = \|AB\|_2 \). Similarly, for \( BA \), since \( A^T \) is orthogonal.
(6) Since both \( A \) and \( A^T \) are orthogonal, \( \|A\|_2 = \|A^T\|_2 = 1 \).

### 4.4 Eigenvalues

Recall that if \( A \) is an \( n \times n \) matrix, then \( \lambda \) is an eigenvalue with eigenvector \( v \neq 0 \) if \( Av = \lambda v \). We know that the MATLAB function \( \text{eig}(A) \) will produce the eigenvalues, but in the case of very large matrices it is important to have another method available.

#### The Power Method

We now turn to the problem of finding eigenvalues and eigenvectors. We will look at the easiest method of finding an eigenvalue, the **power method**. Given a square matrix \( A \) and a vector \( v \) consider the sequence given by \( v_0 = v \) and \( v_{n+1} = Av_n \). This sequence of vectors will tend towards an eigenvector, but we need to worry about the terms in the sequence growing toward overflow or underflow. Thus, at each stage we will re-scale the vector using the infinity norm, that is we assign

\[
  v_n = \frac{v_n}{\max(\text{abs}(v))} = \frac{v_n}{\text{norm}(v_n, \text{'inf'})}
\]

Suppose that after re-scaling we arrive at a stage where \( v_n \approx v_{n+1} \) where \( \approx \) means \( \|v_n - v_{n+1}\| < \text{tol} \), and \( \text{tol} \) is some small value we have chosen. Then we have found an eigenvector. This is easy to do in MATLAB.

\[
A = \text{ones}(5)+5*\text{eye}(5) , \quad \text{eig}(A)
\]

We can see the largest eigenvalue. Now use the power method to find it.

\[
v = \text{rand}(5,1) ;
\text{for } i=1:20, y=v; v=A*y; v=v/\text{norm}(v, \text{'inf'}) ; \text{norm}(v-y), \text{pause}, \text{end}
\]

The \( \text{norm} \) has been inserted so that we can watch the vectors converge. The \( \text{pause} \) lets you see the value. To continue the iteration, hit any key. When this is done we will look for the eigenvalue.

\[A*\text{v.} / \text{v}\]

#### Theorem

If \( A \) is diagonalizable with a unique largest (in absolute value) eigenvalue \( \lambda \), then \( v_n \) converges to an eigenvector \( v \) for \( \lambda \).

Proof: Let \( x = c_1 \lambda u + \sum_{i=2}^{n} c_i \lambda_i u_i \), where \( |\lambda| > |\lambda_i| \) for \( i = 2, \ldots, n \). Then \( Ax = c_1 \lambda u + \sum_{i=2}^{n} c_i \lambda_i u_i \), and \( A^m x = c_1 \lambda^m u + \sum_{i=2}^{n} c_i \lambda_i^m u_i \), and now divide both sides by \( \lambda^m \) to get

\[
(1/\lambda^m)A^m x = c_1 u + \sum_{i=2}^{n} c_i (\lambda_i/\lambda)^m u_i .
\]

Since \( |\lambda| > |\lambda_i|, |\lambda/\lambda_i| < 1 \) and so the terms in the sum tend to 0, and thus for large \( m \) we get

\[
A^m x \approx c_1 u .
\]

So for large \( m \), \( A^m x \) and \( A^{m+1} x \) are approximately scalar multiples. After normalization \( A^m x \approx v_m \) and \( A^{m+1} x \approx v_{m+1} \) so that \( v_m \) and \( v_{m+1} \) are positive scalar multiples both of norm value 1, and hence \( v_m \approx v_{m+1} \).
The hypothesis of the theorem that there be a unique largest eigenvalue is needed. Try
\[
A = \text{ones}(5)-5*\text{eye}(5), \ v = \text{rand}(5,1); \ \text{eig}(A)
\]
for \( i=1:20, \ y = v; \ v = A*y; \ v = v/\text{norm}(v,\text{inf})); \)
\[
\text{norm}(v-y), \ \text{pause}, \ \text{end}
\]
In this case you note that \( \text{norm}(v-y,2) \) does not go to 0.

The proof indicates that the starting vector \( x \) is not so critical as long as the coefficient \( c_1 \neq 0 \). Typically we would not know this coefficient, but for an arbitrary \( x \) it is unlikely that \( c_1 = 0 \).

With this approximate eigenvector, \( v \), in hand, we need to find \( \lambda \).

A reasonable approach would be to look at \( Av = \lambda v \).

Since it may not be the case that \( Av \) is an exact multiple of \( v \), this approach is problematic. In fact, viewing \( Av = \lambda v \) as an \( m \times 1 \) system with variable \( \lambda \), this system is likely to be inconsistent. We will use a least squares solution to get the eigenvalue.

**Theorem.** The least squares solution \( \lambda \) to \( Av = \lambda v \) is given by \( \lambda = \frac{v^TAv}{v^Tv} \). The quotient \( \frac{v^TAv}{v^Tv} \) is called the Rayleigh Quotient.

**Proof:** Look at the equation \( v\lambda = Av \) where \( \lambda \) is the variable. The normal equation is \( v^Tv\lambda = v^TAv \), which has the solution \( \lambda = \frac{v^TAv}{v^Tv} \).

We can use this in MATLAB.
\[
A = \text{ones}(5)+5*\text{ones}(5); \ x = \text{rand}(5,1);
\]
for \( i=1:20, \ y = v; \ v = A*y; \ v = v/\text{norm}(v,\text{inf})); \ end
\]
We have an approximation to an eigenvector with \( v \). The Rayleigh Quotient will approximate the eigenvalue.
\[
r = (v'*A*v)/(v'*v)
\]
Compare with \( \text{eig}(A) \)

**The Inverse Method**

Suppose that \( A \) is invertible and that \( \lambda \neq 0 \) is an eigenvalue of \( A \) with eigenvector \( v \). We have \( Av = \lambda v \), and multiplying by inverses \( A^{-1}v = (1/\lambda)v \). If \( 1/\lambda \) is the largest (in absolute value) eigenvalue of \( A^{-1} \), then \( \lambda \) is the smallest (in absolute value) eigenvalue of \( A \). This method, the inverse method will find the smallest eigenvalue of \( A \) by applying the power method the the inverse of \( A \). Notice that \( v \) is an eigenvector for both \( A \) and \( A^{-1} \). To get an eigenvalue for \( A \) from \( v \) compute the Rayleigh Quotient on \( A \). Try
\[
A = \text{ones}(5)+\text{diag}(1:5); \ B = \text{inv}(A);
\]
for \( i=1:20, \ y = v; \ v = B*y; \ v = v/\text{norm}(v,\text{inf})); \ end
\]
\[
\text{norm}(y-v,2)
\]
\[
r = (v'*A*v)/(v'*v)
\]
\[
\text{min}(`\text{eig}(A)`)
\]

**The Shift Method**

Another variation is the shift method. Here you choose a scalar \( s \) and apply the power method to the shifted matrix \( A - sI \). Notice that if \( v \) is an eigenvector of \( A \) with eigenvalue \( \lambda \), then \( (A - sI)v = Av - sv = (\lambda - s)v \), so that \( \lambda - s \) is an eigenvalue of the shifted matrix and \( v \) is an eigenvector of both \( A \) and \( A - sI \).

**The Inverse Shift Method**

By combining the inverse method with the shift method we get an effective iteration, the inverse shift method. If you choose \( s \) close enough to \( \lambda \), then \( \lambda - s \) is the smallest eigenvalue of the shifted matrix \( A - sI \),
thus applying the power method to the inverse of \( A - sI \) will converge to an eigenvector. The problem is choosing the shift \( s \). Let’s try this in MATLAB, but we will cheat a bit.

\[
A = \text{ones}(3) + \text{diag}(1:3), \quad \text{eig}(A)
\]

Now choose \( s = 2 \) and invert \( B = A - 2 \times \text{eye}(3) \)

\[
B = \text{inv}(A - 2 \times \text{eye}(3)), \quad x = \text{rand}(3,1);
\]

for \( i = 1:20, \) \( x = B \times x; \) \( x = x / \text{norm}(x, \text{inf}); \) end

\[
r = (x^\top A x) / (x^\top x)
\]

\[
\text{eig}(A)
\]

One way to choose a shift is the Rayleigh Quotient \( s = \frac{x^\top Ax}{x^\top x} \) of some vector \( x \). This presupposes that you have vector \( x \), in mind for the eigenvector. A less obvious choice is based on Gershgorin’s Theorem, which suggests using the diagonal entries \( a_{ii} \) as shift values.

Another variation which speeds convergence is to adaptively recalculate the shift after each iteration using the Rayleigh Quotient:

\[
\text{for } i = 1:20, \quad x = B \times x; \quad x = x / \text{norm}(x, \text{inf}); \quad r = x^\top (A \times x) / (x^\top x); \quad B = \text{inv}(A - r \times \text{eye}(n)); \text{ end}
\]

The QR Method

Recall the QR Factorization, \( A = QR \) where \( Q \) is orthogonal and \( R \) is upper triangular. Suppose we let \( A_1 = RQ \), then \( A = QR = QA_1Q^\top \), so \( A \) and \( A_1 \) are similiar and so have the same eigenvalues. By iterating this process, we get a sequence \( A_n \) which frequently (not always) converges to an upper triangular matrix, and the eigenvalues can be read from the diagonal. Try this in MATLAB:

\[
A = [1 \ 2 \ 3; \ 4 \ 5 \ 6; \ 7 \ 8 \ 9]
\]

\[
B = A;
\]

\[
\text{[Q, R]} = \text{qr}(B); \quad B = R \times Q
\]

\[
\text{[Q, R]} = \text{qr}(B); \quad B = R \times Q
\]

\[
\text{[Q, R]} = \text{qr}(B); \quad B = R \times Q
\]

\[
\text{[Q, R]} = \text{qr}(B); \quad B = R \times Q
\]

\[
\text{[Q, R]} = \text{qr}(B); \quad B = R \times Q
\]

You can see the matrices converge to an upper triangular matrix. Compare \( \text{eig}(A) \), \( \text{diag}(B) \)

Not all is rosy with the QR method, as the project exercises will show, but there are ways around the problems. MATLAB’s \text{eig} is based on the QR method.

4.5 The Singular Value Decomposition

The Singular Value Decomposition.

Given any \( m \times n \) real matrix \( A \), there is an \( m \times m \) orthogonal matrix \( U \) and an \( n \times n \) orthogonal matrix \( V \) such that

\[
U^\top AV = S = \begin{pmatrix}
\text{diag}(\sigma_1, \ldots, \sigma_r) & 0 \\
0 & 0
\end{pmatrix}
\]

where \( \sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_r > 0 \).

First note that there are no complex matrices in the conclusion. The \( \sigma \)'s are not eigenvalues of \( A \). They are called the singular values. In the following discussion we will assume these notational conventions

\[
U = [u_1, \ldots, u_m] \quad V = [v_1, \ldots, v_n]
\]

and \( AV = US \).
The Column Space

Look at $A v_1 = \sigma_1 u_1, \ldots, A v_r = \sigma_r u_r$. Thus $\sigma_1 u_1, \ldots, \sigma_r u_r$ are all in the Column Space of $A$. Since $V$ is invertible, $\text{rank}(A) = \text{rank}(AV) = \text{rank}(US) = r$. $\sigma_1 u_1, \ldots, \sigma_r u_r$ are orthogonal, thus $u_1, \ldots, u_r$ form an orthonormal basis for the Column Space of $A$ and $\text{rank}(A) = r$. In MATLAB

```matlab
A = magic(4);
[U, S, V] = svd(A);
```

Here $r = 3$ and so

```matlab
B = U(:, 1:3);
```

is an orthonormal basis for the Column Space of $A$. Compare this with

```matlab
orth(A);
```

The remaining vectors $u_{r+1}, \ldots, u_m$ are a basis for the orthogonal complement of the Column Space of $A$, which is the Null Space of $A^T$. In MATLAB

```matlab
A * V(:, 4);
```

Compare this with

```matlab
null(A);
```

The Null Space

Now $A v_j = \sigma_j u_j = 0$ for $j > r$ and so $v_{r+1}, \ldots, v_n$ are in the Null Space of $A$ and since we know that the dimension of the Null Space is $n - r$, $v_{r+1}, \ldots, v_n$ is an orthonormal basis for the Null Space of $A$. The vectors $v_1, \ldots, v_r$ are a basis for the orthogonal complement of the Null Space of $A$, which is the Column Space of $A^T$. In MATLAB

```matlab
A * V(:, 4);
```

Compare this with

```matlab
null(A);
```

The Eigenvalues of $A^T A$ and $AA^T$

Starting with $AV = US$ and taking the transpose we get $V^T A^T = S^T U^T$, using orthonormality, $A^T U = V S^T$. Looking at the $i^{th}$ column, $A^T u_i = \sigma_i v_i$. Now multiplying by $A$

$$AA^T u_i = A \sigma_i v_i = \sigma_i A v_i = \sigma_i^2 u_i$$

so that $\sigma_i^2$ is an eigenvalue of $AA^T$ and $u_i$ is an eigenvector. Similarly $A^T A v_i = \sigma_i^2 v_i$. Thus $\sigma_1^2, \ldots, \sigma_n^2$ are the eigenvalues of $A^T A$ and $\sigma_1, \ldots, \sigma_n$ are the eigenvalues of $AA^T$. In MATLAB

```matlab
svd(A);
sqrt(eig(A'*A));
sqrt(eig(A*A'));
```

Projections

Recall from above that the projection of $b$ onto the Column Space of $A$ is $p = A x$ where $x$ is the solution to the least squares problem, which is also a solution to the normal equations $A^T A x = A^T b$. Using the singular value decomposition we can write $p = US \cdot (S^T S)^{-1} S^T U^T b$, but, assuming $\text{rank}(A) = n$,

$$S \cdot (S^T S)^{-1} S^T = \begin{pmatrix} I_n & 0 \\ 0 & 0 \end{pmatrix}$$

So we see that the projection formula can be dramatically simplified. If we push on

$$p = (u_1 \cdot b) u_1 + \ldots + (u_n \cdot b) u_n$$

Rank Approximations
Notice that when we take an \( m \times 1 \) vector, like \( u_i \), and an \( n \times 1 \) vector, like \( v_i \), and we form the outer product \( u_i v_i^T \), we get a \( m \times n \) matrix. With that in mind, define the \textit{rank k approximation to A}

\[
A_k = \sigma_1 u_1 v_1^T + \cdots + \sigma_k u_k v_k^T
\]

When we multiply

\[
U^T A_k V = \begin{pmatrix}
\text{diag}(\sigma_1, \ldots, \sigma_k) & 0 \\
0 & 0
\end{pmatrix}
\]

and we notice that \( \text{rank}(A_k) = k \) when \( k \leq r \). Look at \( U^T (A - A_k) V = U^T AV - U^T A_k V \); the singular values \( \sigma_1, \ldots, \sigma_k \) are subtracted out, leaving only \( s_{k+1}, \ldots, s_r \). The matrix \( A_k \) is the rank \( k \) approximation to \( A \).

With the rank \( r \) approximation being \( A_r = A \). Try this in MATLAB:

```matlab
A = magic(4); [U, S, V] = svd(A);
A1 = S(1,1)*U(:,1)*V(:,1)';
rank(A1), svd(A1)
A2 = A1 + S(2,2)*U(:,2)*V(:,2)';
rank(A2), svd(A2)
A3 = A2 + S(3,3)*U(:,3)*V(:,3)';
rank(A3), svd(A3)
A4 = A3 + S(4,4)*U(:,4)*V(:,4)';
rank(A4), svd(A4)
A4 - A
```