MATH 495C. Numerical Linear Algebra
Group 1
Topic: Inverse Power Method

1. Iterative Methods for solving $Ax = b$

This class of methods is used to solve linear systems of n equations and n unknowns.

1.1. **Direct Vs. Iterative.**

When solving a system of linear equations, the most effective methods are characterized by accuracy and speed. Direct methods such as **LU** factorization and **Gaussian elimination**, are efficient for solving relatively small systems in which most of the entries are non-zero. Furthermore, direct methods calculate exact solutions in a finite amount of operations. In contrast, iterative methods excel when dealing with sparse matrices arising from very large systems. Additionally, iterative methods require fewer multiplicative steps and automatically account for errors during the iteration process. On the other hand, iterative methods start with an initial approximation and improve with the number of iterations in an infinite convergence sequence.

1.2. **Drawbacks of Direct Methods.**

If, in the absence of roundoff error, direct methods yield an exact solution within a finite number of steps, then why study iterative methods? There exists a number of drawbacks to direct methods that motivate the use of an alternative strategy:

Computational Complexity/Efficiency: For many large linear systems, it becomes prohibitively expensive to use $\mathcal{O}(n^3)$ direct methods (*i.e.*, Gaussian Elimination).

All-or-none: With direct methods, there is no notion of an early termination of an inexact (yet perfectly satisfactory) solution. Sometimes it may not be necessary to solve the system exactly (we may only need a solution that is within some tolerance of the exact solution), but this is not possible via direct methods since the process must be completed in order to obtain a solution.

Naive: Sometimes, we may have a pretty good idea of an approximate solution (guess) for the solution. Direct methods cannot make good use of such information.

1.3. **Motivation for Iterative Methods.**

Task: Solve the system $A\vec{x} = \vec{b}$.

In practice, inverting A may be very difficult, to the extent that it may be worthwhile to **invert a much "easier" matrix several times**, rather than inverting A directly once.

Iterative methods are often useful for solving

- linear problems involving a large number of variables (potentially millions)
- nonlinear equations (only choice)

1.4. **General Approach to Iterative Methods.**

An **iterative method** is a procedure that generates a sequence of successive approximations x_k to the solution of linear systems $A\vec{x} = \vec{b}$.

The k-th approximation is derived from the previous one, and over the course of repeated iterations the sequence converges to the solution of the system.

Key insight: Rewrite A in terms of simpler matrices with "nice" properties:

$$A = \underbrace{(I - B)}_{\text{"easy to invert"}} \qquad\qquad B : \textit{matrix of the iterative method}$$

Then $Au = c$ is equivalent to

$$(I - B)u = c \Leftrightarrow \boxed{u = Bu + c}$$

1.5. **Convergence of Iterative Methods.** Iterative methods for solving a linear system $Ax = b$ where A is an $n \times n$ matrix and $x, b \in \mathbb{R}^n$ consist of finding some $n \times n$ matrix B and some vector $c \in \mathbb{R}^n$, such that $I - B$ is invertible and the unique solution $\tilde{x}$ of $Ax = b$ is equal to the unique solution $\tilde{u}$ of $u = Bu + c$. Then, starting from any arbitrary vector $u_0 \in \mathbb{R}^n$, we compute the sequence (u_k) given by

$$u_{k+1} = Bu_k + c, k \in \mathbb{N}.$$

We say that the iterative method is *convergent* iff

$$\lim_{k \to \infty} u_k = \tilde{u},$$

for any arbitrary u_0. Given a B matrix, we check whether the iterative method will converge by checking whether $||B|| < 1$ for some matrix norm. To see why this guarantees convergence, consider an iterative scheme $u_{k+1} = Bu_k + c$ that after many iterations has reached a steady state, in which

$$u = Bu + c.$$

If we subtract this equation from the recursion formula and define the **error** of the kth iteration as $e_k = u_k - u$ we obtain

$$u_{k+1} - u = B(u_k - u) \Leftrightarrow e_{k+1} = Be_k.$$

In a convergent scheme the error will tend to zero as k increases. Accordingly, we hope that each e_k is "smaller" than its previous iteration. Using vector norms, we want

$$||e_{k+1}|| < ||e_k||.$$

But from before we know that

$$||e_{k+1}|| = ||Be_k|| \leq ||B|| \cdot ||e_k||.$$

Thus, for the error at the $(k + 1)$ step to be strictly less than that of the k^{th} step, the matrix norm of B must be less than one. That is,

$$||B|| < 1 \Rightarrow ||e_k|| \to 0 \text{ as } k \to \infty$$

which in turn implies that

$$u_k \to u \text{ as } k \to \infty.$$

It is important to note that failure to meet this condition does not imply that the method will not converge: just because a given norm (say, the 2-norm) of B is greater than 1 does not imply that all norms have similar magnitude. Ultimately, it is the value of the spectral radius of B that tells us whether our iterations are leading us towards the right solution.

Theorem 1. *The iterative scheme*

$$u_{k+1} = Bu_k + c$$

converges to the exact solution u if and only if the ***spectral radius*** *of B is strictly less than one. That is, if*

$$\rho(B) < 1.$$

Hence, given a matrix B specified by the type of iterative method we wish to implement, we find its spectral radius

$$\rho(B) = \max\{|\lambda_i|, \ \lambda_i \text{ is an eigenvalue of B}\}$$

and check whether $\rho(B) < 1$. If it does, we are guaranteed that the method will converge to the exact solution of the $Ax = b$ system.

2. Jacobi Method

2.1. **Introduction.** Generally speaking, Jacobi iteration can be described as a method that recursively checks variables within a system of equations and updates each variable assuming each previous value is correct. Over the course of repeated iterations, the method converges to the solutions of the system. Component by component, we notice

$$a_{ii}x_i^{k+1} + \sum_{j \neq i} a_{ij}x_j^k = b_i.$$

2.2. **Formulation.**

Given a matrix A and a column vector b, we wish to solve to system $Ax = b$.

Define the matrices D, E and F to be the submatrices of A such that

$$A = D - E - F$$

where D consists of the diagonal entries of A, $-E$ is a lower triangular matrix consisting of the entries of A below the diagonal, and $-F$ is an upper triangular matrix consisting of the entries of A above the diagonal. In other words,

$$\begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} & \cdots & a_{1,n-1} & a_{1,n} \\ a_{2,1} & a_{2,2} & a_{2,3} & \cdots & a_{2,n-1} & a_{2,n} \\ a_{3,1} & a_{3,2} & a_{3,3} & \cdots & a_{3,n-1} & a_{3,n} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n-1,1} & a_{n-1,2} & a_{n-1,3} & \cdots & a_{n-1,n-1} & a_{n-1,n} \\ a_{n,1} & a_{n,2} & a_{n,3} & \cdots & a_{n,n-1} & a_{n,n} \end{pmatrix} = \begin{pmatrix} a_{11} & 0 & 0 & \cdots & 0 & 0 \\ 0 & a_{22} & 0 & \cdots & 0 & 0 \\ 0 & 0 & a_{33} & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & a_{n-1n-1} & 0 \\ 0 & 0 & 0 & \cdots & 0 & a_{nn} \end{pmatrix} +$$

$$\begin{pmatrix} 0 & 0 & 0 & \cdots & 0 & 0 \\ a_{2,1} & 0 & 0 & \cdots & 0 & 0 \\ a_{3,1} & a_{3,2} & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots & \vdots \\ a_{n-1,1} & a_{n-1,2} & a_{n-1,3} & \ddots & 0 & 0 \\ a_{n,1} & a_{n,2} & a_{n,3} & \ddots & a_{n,n-1} & 0 \end{pmatrix} + \begin{pmatrix} 0 & a_{1,2} & a_{1,3} & \cdots & a_{1,n-1} & a_{1,n} \\ 0 & 0 & a_{2,3} & \cdots & a_{2,n-1} & a_{2,n} \\ 0 & 0 & 0 & \ddots & a_{3,n-1} & a_{3,n} \\ \vdots & \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 0 & a_{n-1,n} \\ 0 & 0 & 0 & \cdots & 0 & 0 \end{pmatrix}$$

Substituting in for A, our system becomes

$$(D - E - F)u = b \Leftrightarrow Du = (E + F)u + b$$

which yields the recursion formula

$$Du_{k+1} = (E + F)u_k + b\,, k \geq 0.$$

or

$$u_{k+1} = D^{-1}(E + F)u_k + D^{-1}b\,, k \geq 0,$$

Note: we know that D^{-1} exists since D is a diagonal matrix and by our initial assumption that $a_{ii} \neq 0$ for all $i = 1, \cdots, n$.

This is equivalent to setting $B = D^{-1}(E + F)$ in the original formulation of an iterative method. Then,

$$I - B = I - D^{-1}(E + F) = I - D^{-1}(D - A) = I - (I - A) = A$$

is invertible, as desired.

In equation form, Jacobi's iterative method computes the sequence $(u_k) = (u_1^k, \dots, u_n^k)$ through a system of equations, which appears as follows:

$$\begin{array}{llllllll} u_1^{k+1} & = (& & -a_{12}u_2^k & -a_{13}u_3^k & \cdots & -a_{1n}u_n^k & +b_1) & /a_{11} \\ u_2^{k+1} & = (& -a_{21}u_1^k & & -a_{23}u_3^k & \cdots & -a_{2n}u_n^k & +b_2) & /a_{22} \\ u_3^{k+1} & = (& -a_{31}u_1^k & -a_{32}u_2^k & & \cdots & -a_{3n}u_n^k & +b_3) & /a_{33} \\ \vdots & & \vdots & \vdots & & & & & \\ u_{n-1}^{k+1} & = (& -a_{n-11}u_1^k & \cdots & -a_{n-1n-2}u_{n-2}^k & & -a_{n-1n}u_n^k & +b_{n-1}) & /a_{n-1n-1} \\ u_n^{k+1} & = (& -a_{n1}u_1^k & -a_{n2}u_2^k & \cdots & -a_{nn-1}u_{n-1}^k & & +b_n) & /a_{nn} \end{array}$$

2.3. **Convergence.** We now look at some particular conditions that guarantee us that the Jacobi method will converge. These are to be taken in conjunction with the ones described in section 1.2.

In general, it is not obvious from looking at A whether the scheme will converge or not. However, if A is diagonally dominant we can prove that the Jacobi method converges.

Definition 1. *A* ***diagonally dominant*** *matrix A is an $n \times n$ matrix where, for each row, the absolute value of the diagonal entry is greater than the sum of the absolute values of the remaining entries. That is,*

$$|a_{ii}| \geq \sum_{j=1, j\neq i}^{n} |a_{ij}| \, , \; i = 1, 2, \cdots, n.$$

To illustrate why this, consider the following. For $n = 3$, let

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{21} & a_{22} & a_{33} \end{pmatrix}$$

be a diagonally dominant matrix. We define the matrices

$$D = \begin{pmatrix} a_{11} & 0 & 0 \\ 0 & a_{22} & 0 \\ 0 & 0 & a_{33} \end{pmatrix}, E = \begin{pmatrix} 0 & 0 & 0 \\ -a_{21} & 0 & 0 \\ -a_{21} & -a_{22} & 0 \end{pmatrix}, F = \begin{pmatrix} 0 & -a_{12} & -a_{13} \\ 0 & 0 & -a_{23} \\ 0 & 0 & 0 \end{pmatrix}.$$

A quick calculations shows that

$$B = D^{-1}(E + F) = \begin{pmatrix} 0 & -a_{12}/a_{11} & -a_{13}/a_{11} \\ -a_{21}/a_{22} & 0 & -a_{23}/a_{22} \\ -a_{21}/a_{33} & -a_{32}/a_{33} & 0 \end{pmatrix}$$

Take the infinity norm of B, given by

$$||B||_\infty = \max \left\{ \left(\left|\frac{a_{12}}{a_{11}}\right| + \left|\frac{a_{13}}{a_{11}}\right| \right), \left(\left|\frac{a_{21}}{a_{22}}\right| + \left|\frac{a_{23}}{a_{22}}\right| \right), \left(\left|\frac{a_{31}}{a_{33}}\right| + \left|\frac{a_{32}}{a_{33}}\right| \right) \right\}.$$

But, since A is diagonally dominant,

$$\frac{|a_{12}| + |a_{13}|}{|a_{11}|} < 1 \, , \; \frac{|a_{21}| + |a_{23}|}{|a_{22}|} < 1 \text{ and } \frac{|a_{31}| + |a_{32}|}{|a_{33}|} < 1.$$

This implies that

$$||B||_\infty < 1$$

which, as we have seen before, is a sufficient condition to guarantee that the Jacobi method will converge.

Below is our implementation of the Jacobi method using MATLAB. Our function takes in a given matrix A, vectors b and u_{init}, a number of iteration steps n_{steps} and a desired tolerance *tol*, that when achieved causes the program to stop.

3. Implementation in MATLAB: Jacobi Method

```
function [u,err] = jacobi(A,b,nsteps,uinit,tol)

%JACOBI   solve the linear system Ax = b using the Jacobi Method
%
%     calling sequences:
%             x = jacobi(A,b,nsteps,uinit,tol)
%             jacobi(A,b,nsteps,uinit,tol)
%
%     inputs:
%             A             coefficient matrix for linear system
%                          (matrix must be square)
%             b            right-hand side vector
%             nsteps       number of steps
%             uinit        initialization vector
%             tol          error tolerance for approximate solution
%
%     output:
%             x       solution vector (i.e., vector for which Ax = b)
%
%
%     Based on https://www.cis.upenn.edu/~cis515/cis515-12-sl5.pdf

[nrow,ncol] = size(A);
nb = length(b);

%% Making sure we have the right parameters
if(nrow ~= ncol)
   disp ('jacobi error: Square coefficient matrix required');
   return
end

if (nrow ~= nb)
   disp ('jacobi error: Size of b-vector not compatible with matrix dimension')
   return
end

if(tol <= 0 )
   disp ('jacobi error: Tolerance must be a positive number');
   return
end

if(nsteps < 1 || mod(nsteps,1)~=0 )
   disp ('jacobi error: The number of steps must be an integer greater than 1');
   return
end

if(all(diag(A)) == 0)
   disp ('jacobi error: Entries in the diagonal of A must be non-zero');
   return
end
%% Define relevant matrices and vectors

D = diag(diag(A));
F = -triu(A,1);
E = -tril(A,-1);

% CONVERGENCE
% Method will converge if norm(B) < 1, where B = D\(F+E)

u = zeros(nrow,nsteps);
u(:,1) = uinit;
d = diag(D);

%% Iteration
% D*uk1 = (E+F)*uk + b, k >= 0

err = zeros(1,nsteps);
for i=2:(nsteps+1)
    %u(:,i) = M\(N*u(:,i-1) + b);   % multiplying matrices, slower
    for j=1:nrow
        % vec1 = u(1:end ~= j,i-1);
        % vec2 = A(j,1:end ~= j);
        % sum = dot(vec1,vec2);
        u(j,i) = (b(j) - dot(u(1:end ~= j,i-1),A(j,1:end ~= j)))/d(j); % element-wise multiplication, faster
    end

    diff = u(:,i) - u(:,i-1);   % difference between successive approxs
    err(i-1) = norm(diff, Inf); % infinity norm of diff

    % Terminate loop if error is below tolerance
    if (err(i-1) < tol)
        % Delete zero columns from u and err
        u(:,i+1:end) = [];
        err(i:end) = [];
        disp(sprintf ('Desired tolerance of %d was achieved in less steps than specified', tol))
        return
    end
end
```

4. Introduction to QR Decomposition

Let A be a real $m \times n$ matrix such that $m > n$ and $\text{rank}(A) = n$. Then A can be decomposed into the product

$$A = QR$$

where Q is $m \times n$ and orthogonal ($Q^T Q = I$), and R is an $n \times n$ non-singular upper triangular matrix. This procedure is called **QR decomposition** and is a often referred to as a staple of linear algebra, with applications ranging from finding linear least-squares solutions of overdetermined systems to analyzing multiple-input and multiple-output (MIMO) systems in modern telecommunications.

4.1. **Example.** The 4×3 matrix

$$A = \begin{bmatrix} -1 & -1 & 1 \\ 1 & 3 & 3 \\ -1 & -1 & 5 \\ 1 & 3 & 7 \end{bmatrix}$$

can be written as

$$A = \begin{bmatrix} -1/2 & 1/2 & -1/2 \\ 1/2 & 1/2 & -1/2 \\ -1/2 & 1/2 & 1/2 \\ 1/2 & 1/2 & 1/2 \end{bmatrix} \begin{bmatrix} 2 & 4 & 2 \\ 0 & 2 & 8 \\ 0 & 0 & 4 \end{bmatrix} = \begin{bmatrix} \mathbf{q_1} & \mathbf{q_2} & \mathbf{q_3} \end{bmatrix} \begin{bmatrix} R_{11} & R_{12} & R_{13} \\ 0 & R_{22} & R_{23} \\ 0 & 0 & R_{33} \end{bmatrix} = QR.$$

Note that the columns of Q are orthogonal. That is,

$$q_1 \cdot q_2 = q_2 \cdot q_3 = q_3 \cdot q_1 = 0.$$

In the example above we did not actually compute the QR decomposition of A: there are several ways of doing this, including the Gram-Schmidt process, Givens rotations and **Householder transformations**, each with its pros and cons.

5. Householder Transformation

A fundamental problem to avoid in numerical linear algebra that the Gram-Schmidt algorithm is vulnerable to is the phenomenon known as catastrophic cancellation - a substantially larger increase in relative error than absolute error. Catastrophic cancellation can lead to reduction of the number of significant digits (*loss of significance*) in a numerical calculation using finite-precision arithmetic.

The Householder QR factorization avoids catastrophic cancellations with the successive application of unitary transformations, which inherently preserves length.

Definition 2. *Let $\mathbf{v} \in \mathbb{C}^n$ be a vector such that $||\mathbf{v}||_2 = 1$. Then*

$$P = I - 2\mathbf{v}\mathbf{v}^H$$

*is said to be a **Householder transformation (or reflection)**.*

Theorem 1 summarizes some properties of Householder transformation matrices:

Theorem 2. *Let P be a Householder matrix. Then P has the following properties:*

- *P is **Hermitian**. That is, $P = P^H$, where P^H is the conjugate transpose of P.*
- *P is **involutory** ($P^2 = I$). Reflecting the reflection of a vector results in the original vector.*
- *$P^H P = I$: a reflection is a **unitary** matrix, and thus preserves the norm (this is a consequence of the two points points above).*
- *P has **eigenvalues** $\lambda = \pm 1$.*
- *P has a **determinant** of -1.*

6. Householder QR Factorization

6.1. **Formulation.**

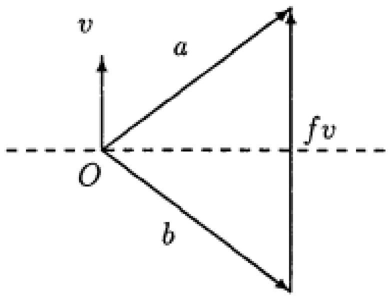

Fig. 1: Householder transformation of a vector.

In the figure above, the dotted line represents a plane P perpendicular to a vector v through the origin O. In the Householder Method, for an arbitrary vector a, we wish to find a vector b reflected over P. The difference between a vector a and its reflection b is a vector fv

$$fv = a - b,$$

where $||v|| = 1$. We want to find the reflection of a through P in terms of the unit vector v. Since reflections preserve the norm of a vector,

$$||a|| = ||b||$$

and

$$||a||^2 = ||b||^2.$$

Since we define the scalar product of a and b as $a^T b = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$ and the Euclidean norm $||a||$ as $||a|| = \sqrt{a_1^2 + a_2^2 + \cdots + a_n^2}$, we obtain an equivalent definition for the norm:

$$||a||^2 = a^T a.$$

Now, since a scalar product is symmetric, $a^T b = b^T a$. Knowing this in addition to the previous equalities,

$$\begin{aligned} ||a||^2 &= b^T b \\ &= (a - fv)^T (a - fv) \\ &= (a - fv)^T (a - fv) \\ &= a^T a - f a^T v - f v^T a + f^2 v^T v \\ &= ||a||^2 - 2 f v^T a + f^2 \end{aligned}$$

The equation above can be simplified to find an expression for f, the distance between a and its image b.

$$f = 2v^T a$$

Lastly, by defining b in terms of a and v, we arrive at the reflection matrix H.

$$\begin{aligned} b &= a - vf \\ &= Ia - v(2v^T a) \\ &= (I - 2vv^T)a \end{aligned}$$

Thus, the reflection of a vector a is the vector $b = Ha$, where $H = I - 2vv^T$.

6.2. **Description.** A Householder transformation reflects a given vector about some hyperplane. This operation can be used to calculate the QR factorization of an $m \times n$ matrix where $m \geq n$.

Consider an arbitrary real m-dimensional column vector $\mathbf{x}$ in A such that $||\mathbf{x}|| = |\alpha|$ for a scalar a. Using floating point arithmetic, a should have the opposite sign as the k-th coordinate of $\mathbf{x}$, where x_k is the pivot coordinate of matrix A's upper triangular form after all of its entries are 0.

Now, if we let $e_1 = (1, 0, ..., 0)^T$ be an n-dimensional vector, set

$$\mathbf{u} = \mathbf{x} - \alpha e_1,$$
$$\mathbf{v} = \frac{\mathbf{u}}{||\mathbf{u}||},$$
$$Q = I - 2\mathbf{v}\mathbf{v}^H,$$

where Q is an $m \times m$ Householder transformation such that $Q\mathbf{x} = (\alpha, 0, \cdots, 0)^T$ and $\mathbf{v}$ is a unitary vector in the direction of $\mathbf{u}$.

The algorithm listed above is used to gradually transform an $m \times n$ matrix A to upper triangular form. This is done by sequentially factoring A into a product of Householder matrices Q_i, which we obtain using the columns of matrix A as $\mathbf{x}$ in the equations above. For example, the first step of this algorithm yields the factorization

$$Q_1 A = \begin{bmatrix} \alpha_1 & * & \cdots & * \\ 0 & & & \\ \vdots & & A' & \\ 0 & & & \end{bmatrix}$$

where we denote by A' the $(m-1) \times (n-1)$ matrix obtained by deleting the first row and first column of $Q_1 A$. In order to get the next Householder matrix Q_2', we repeat the same process executed to find Q_1 substituting the smaller A' for A. In order for Q_2' to operate on Q_1 (*i.e.* in order for the matrix product $Q_2' Q_1$ to exist), we expand Q_2' to the upper left corner of the matrix by defining Q_k as follows:

$$Q_k = \begin{bmatrix} I_{k-1} & 0 \\ 0 & Q_k' \end{bmatrix}.$$

After s iterations of this process, where $s = \min(m-1, n)$,

$$R = Q_s \cdots Q_2 Q_1 A$$

becomes an upper triangular matrix. If we allow

$$Q = Q_1^T Q_2^T \cdots Q_s^T,$$

we have that $A = QR$ is a QR decomposition of A.

6.3. **Example of QR Householder Factorization.** Consider the 3×3 matrix

$$A = \begin{bmatrix} \mathbf{a_1} & \mathbf{a_2} & \mathbf{a_3} \end{bmatrix} = \begin{bmatrix} 12 & -51 & 4 \\ 6 & 167 & -68 \\ -4 & 24 & -41 \end{bmatrix}$$

We begin by noting that

$$||\mathbf{a_1}|| = \sqrt{(12)^2 + (6)^2 + (-4)^2} = 14.$$

The first step is then to find a unitary reflection of $\mathbf{a_1}$ onto $||\mathbf{a_1}||\mathbf{e_1} = (14, 0, 0)^T$, which we call $\mathbf{v}$. Now,

$$\mathbf{u} = \mathbf{a_1} - ||\mathbf{a_1}||\mathbf{e_1}$$

and

$$\mathbf{v} = \frac{\mathbf{u}}{||\mathbf{u}||}$$

is a unitary vector. Plugging in for known values,

$$\mathbf{u} = (12, 6, -4)^T - 14(1, 0, 0)^T = (-2, 6, -4)^T$$

and

$$\mathbf{v} = \frac{(-2, 6, -4)^T}{\sqrt{(-2)^2 + (6)^2 + (-4)^2}} = \frac{2}{\sqrt{56}}(-1, 3, -2)^T = \frac{1}{\sqrt{14}}(-1, 3, -2)^T.$$

Our first reflector is

$$Q_1 = I - 2\mathbf{v}\mathbf{v}^H = I - 2\frac{1}{\sqrt{14}}(-1, 3, -2)^T \frac{1}{\sqrt{14}}(-1, 3, -2).$$

Computing the matrix multiplication,

$$\begin{aligned} Q_1 &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} - \frac{2}{\left(\sqrt{14}\right)^2} \begin{bmatrix} -1 \\ 3 \\ -2 \end{bmatrix} \begin{bmatrix} -1 & 3 & -2 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} - \frac{1}{7} \begin{bmatrix} 1 & -3 & 2 \\ -3 & 9 & -6 \\ 2 & -6 & 4 \end{bmatrix} = \begin{bmatrix} 6/7 & 3/7 & 2/7 \\ 3/7 & -2/7 & 6/7 \\ -2/7 & 6/7 & 3/7 \end{bmatrix}. \end{aligned}$$

Multiplying Q_1 by A, we obtain

$$Q_1 A = \begin{bmatrix} 14 & 21 & -14 \\ 0 & -49 & -14 \\ 0 & 168 & -77 \end{bmatrix}$$

which, as desired, has all zeros below the first entry. One more step is required to obtain an upper triangular matrix: we repeat the procedure above for

$$A' = \begin{bmatrix} -49 & -14 \\ 168 & -77 \end{bmatrix},$$

the matrix that results from deleting the first row and column from A. That is, we calculate the vector $\mathbf{u}$, given by

$$\mathbf{u} = \mathbf{a_1'} - ||\mathbf{a_1'}||\mathbf{e_1}$$

where $\mathbf{a_1'}$ is the first column of A'. Now,

$$\mathbf{u} = (-49, 168)^T - \sqrt{(-49)^2 + (168)^2}(1, 0)^T = (-49, 168)^T - 175(1, 0)^T = (-224, 168)^T$$

and

$$\mathbf{v} = \frac{(-224, 168)^T}{\sqrt{(-224)^2 + (168)^2}} = \frac{56}{\sqrt{78400}}(-4, 3)^T = \frac{1}{5}(-4, 3)^T.$$

As before,

$$Q_2 = I - 2\mathbf{v}\mathbf{v}^H = I - 2\frac{1}{5}(-4,3)^T\frac{1}{5}(-4,3).$$

which, in matrix form, equals

$$\begin{aligned} Q_2' &= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} - \frac{2}{25}\begin{bmatrix} -4 \\ 3 \end{bmatrix}\begin{bmatrix} -4 & 3 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} - \frac{2}{25}\begin{bmatrix} 16 & -12 \\ -12 & 9 \end{bmatrix} = \begin{bmatrix} -7/25 & 24/25 \\ 24/25 & 7/25 \end{bmatrix}. \end{aligned}$$

"Augmenting" Q_2' (as described earlier) gives us

$$Q_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -7/25 & 24/25 \\ 0 & 24/25 & 7/25 \end{bmatrix}.$$

The matrix

$$R = Q_2\,Q_1 A = \begin{bmatrix} 14 & 21 & -14 \\ 0 & 175 & -70 \\ 0 & 0 & 35 \end{bmatrix}$$

is upper triangular. Finally,

$$Q = Q_1^T\,Q_2^T = \begin{bmatrix} 6/7 & -69/175 & 58/175 \\ 3/7 & 158/175 & -6/175 \\ -2/7 & 6/35 & 33/35 \end{bmatrix}$$

where, as intended,

$$A = QR = \begin{bmatrix} 6/7 & -69/175 & 58/175 \\ 3/7 & 158/175 & -6/175 \\ -2/7 & 6/35 & 33/35 \end{bmatrix}\begin{bmatrix} 14 & 21 & -14 \\ 0 & 175 & -70 \\ 0 & 0 & 35 \end{bmatrix} = \begin{bmatrix} 12 & -51 & 4 \\ 6 & 167 & -68 \\ -4 & 24 & -41 \end{bmatrix}.$$

7. Implementation in MATLAB: Householder QR Decomposition

```
function [Q, R] = qr_householder(A)
%qr_householder Performs QR decomposition using householder transformation

[m, ~]=size(A);
R = A;          % Start with R=A
Q = eye(m);     % Set Q as the identity matrix

for i = 1:m-1
    x = zeros(m, 1);
    x(i:m, 1) = R(i:m, i);
    alpha = norm(x);
    e_i =  double(1:m == i)';
    u = x - alpha.*e_i;

    % Orthogonal transformation matrix that eliminates one element
    % below the diagonal of the matrix it is post-multiplying:

    l = norm(u);
    if l ~= 0
        v = u/l;
        u = 2*R'*v;
        R = R - v*u';        % Product HR
        Q = Q - 2*Q*(v)*v'; % Product QR
    end
end
end
```

8. Method of Conjugate Directions

8.1. **Conjugacy.** When using steepest descent method, we often find ourselves taking steps along the same direction as earlier iterations. The method of *Conjugate Directions* is designed to avoid this. We choose the search vectors d such that

$$d_{(i)}^T A d_{(j)} = 0.$$

When the above holds, we say that $d_{(i)}$ and $d_{(j)}$ are A**-orthogonal**. Now, consider the i^{th} step along direction $d_{(i)}$. No other step is along a direction that has a component along $d_{(i)}$. We can therefore require that the error of the $(i+1)$ iteration is A-orthogonal to $d_{(i)}$. This condition is equivalent to finding the minimum of f along $d_{(i)}$, as in Steepest Descent. It follows that

$$\begin{aligned} d_{(i)}^T A e_{(i+1)} &= 0, \\ d_{(i)}^T A(e_{(i)} + \alpha_{(i)} d_{(i)}) &= 0, \\ d_{(i)}^T A e_{(i)} + \alpha_{(i)} d_{(i)}^T A d_{(i)} &= 0. \end{aligned}$$

Solving for $\alpha_{(i)}$, we have that

$$\alpha_{(i)} = -\frac{d_{(i)}^T A e_{(i)}}{d_{(i)}^T A d_{(i)}},$$

which we can write in terms of the residual $r_{(i)} = b - Ax_{(i)} = -Ae_{(i)}$:

$$\alpha_{(i)} = \frac{d_{(i)}^T r_{(i)}}{d_{(i)}^T A d_{(i)}}.$$

Since all the terms on the right-hand side of the equation are known, we can calculate α for any given step of the scheme.

We introduced the method of **Conjugate Directions** as a step-up from Steepest Descent - we then expect the method to convergence in a smaller number of iterations. In fact, we will show that is computes x in n steps.

Theorem 3. *The Conjugate Directions method converges to the correct solution in n steps.*

Proof. We start by expressing $e_{(0)}$ as a linear combination of the search direction vectors:

$$e_{(0)} = \sum_{j=0}^{n-1} \delta_j d_{(j)}.$$

In order to find the δ_j weights, we take advantage of the A-orthogonality of the direction vectors and multiply by $d_{(j)}^T A$ on both sides of the equation above:

$$\begin{aligned} d_{(k)}^T A e_{(0)} &= d_{(k)}^T A \left(\sum_{j=0}^{n-1} \delta_j d_{(j)} \right) \\ &= \sum_{j=0}^{n-1} \delta_j \left(d_{(k)}^T A d_{(j)} \right) \\ &= \delta_j \left(d_{(k)}^T A d_{(k)} \right) \qquad \text{(since the direction vectors are } A\text{-orthogonal).} \end{aligned}$$

Rearranging for δ_j,

$$\begin{aligned}
\delta_j &= \frac{d_{(k)}^T A e_{(0)}}{d_{(k)}^T A d_{(k)}} \\
&= \frac{d_{(k)}^T A \left(e_{(0)} + \sum_{i=0}^{k-1} \alpha_{(i)} d_{(i)}\right)}{d_{(k)}^T A d_{(k)}} \qquad \text{(since the direction vectors are } A\text{-orthogonal)} \\
&= \frac{d_{(k)}^T A e_{(k)}}{d_{(k)}^T A d_{(k)}} \qquad \text{(since } x_{(i+1)} = x_{(i)} + \alpha_{(i)} d_{(i)}\text{).}
\end{aligned}$$

Comparing this with our previous result for α, we see that

$$\alpha_{(i)} = -\delta_{(i)}.$$

Therefore, we can write

$$\begin{aligned}
e_{(i)} &= e_{(0)} + \sum_{j=0}^{i-1} \alpha_{(j)} d_{(j)} \\
&= \sum_{j=0}^{n-1} \delta_{(j)} d_{(j)} - \sum_{j=0}^{i-1} \delta_{(j)} d_{(j)} \\
&= \sum_{j=i}^{n-1} \delta_{(j)} d_{(j)}.
\end{aligned}$$

At the i^{th} iteration we "strip" the error vector of its component along the $d_{(i)}$ direction. After n iterations the error is 0, which concludes our proof.

□

8.2. **Gram-Schmidt Conjugation.** In order to find the **A-orthogonal** search directions, denoted as $d_{(i)}$, we use a generation method called the *conjugate Gram-Schmidt process.* The first step when constructing $d_{(i)}$ is to subtract any components of the n linearly independent vectors $u_0, u_1, \cdots, u_{n-1}$ that are not A-orthogonal to the previous d vectors. Simply stated, let $d_{(0)} = u_0$ and for $i > 0$, set

$$d_{(i)} = u_{(i)} + \sum_{k=0}^{i-1} \beta_{ik} d_{(k)},$$

where β_{ik} acts as a coefficient to represent the previous search directions defined for $i > k$. Now, similar to the strategy implemented when finding the value of $\delta_{(k)}$, premultiply both sides of the equation by $Ad_{(j)}$. Now, we see

$$\begin{aligned}
d_{(i)}^T A d_{(j)} &= u_{(i)}^T A d_{(j)} + \sum_{k=0}^{i-1} \beta_{ik} d_{(k)}^T A d_{(j)} \\
0 &= u_{(i)}^T A d_{(j)} + \beta_{ij} d_{(j)}^T A d_{(j)}, \qquad \text{where } i > j \text{ by } A\text{-orthogonality of } d \text{ vectors} \\
\beta_{ij} &= -\frac{u_{(i)}^T A d_{(j)}}{d_{(j)}^T A d_{(j)}}.
\end{aligned}$$

The downfall of using the Gram-Schmidt conjugation process is that all of the previously calculated search vectors must be used in order to construct each new one, requiring a higher operation count ($\mathcal{O}(n^3)$ operations).

9. Implementation in MATLAB: Conjugate Directions Method

10. Method of Conjugate Gradients

The method of **Conjugate Gradients** applies the same procedures as the method of Conjugate Directions with a slight alteration. Instead of using an arbitrary set of n linearly independent vectors $(u_{(0)}, u_{(1)}, \cdots, u_{(n-1)})$, the search directions of the Conjugate Gradient method are found by conjugation of the residuals, that is $u_{(i)} = r_{(i)}$.

The residuals have the advantage of being orthogonal to the previous search directions, ensuring a new, linearly independent search direction every iteration. The largest advantage of this selection, however, stems from the observation that since each new residual is orthogonal to the previous search direction it is also orthogonal to the previous residuals. Now, to parallel what we found in the method of Conjugate Directions,

$$u_{(i)}^T r_{(j)} = 0$$

becomes

$$r_{(i)}^T r_{(j)} = 0$$

where $i \neq j$. Furthermore, since $r_{i+1} = r_i - \alpha_i Ad_{(i)}$, each new residual is a linear combination of the previous residual with $Ad_{(i-1)}$. Since $d_{(i-1)} \in \mathcal{D}$, each new subspace $\mathcal{D}_{(i+1)}$ is given by

$$\mathcal{D}_{(i+1)} = \mathcal{D}_{(i)} \cup A\mathcal{D}_{(i)}.$$

A subspace of this form, created by successive applications of a matrix to a vector, is called a **Krylov subspace**. We then have that

$$\begin{aligned} \mathcal{D}_{(i)} &= \operatorname{span}\left\{d_{(0)}, Ad_{(0)}, A^2 d_{(0)}, \cdots, A^{i-1} d_{(0)}\right\} \\ &= \operatorname{span}\left\{r_{(0)}, Ar_{(0)}, A^2 r_{(0)}, \cdots, A^{i-1} r_{(0)}\right\}. \end{aligned}$$

From the previous section,

$$\beta_{ij} = -\frac{r_{(i)}^T A d_{(j)}}{d_{(j)}^T A d_{(j)}}.$$

Now,

$$\begin{aligned} r_{(j+1)} &= r_{(j)} - \alpha_{(j)} A d_{(j)} \\ \Leftrightarrow r_{(i)}^T r_{(j+1)} &= r_{(i)}^T r_{(j)} - \alpha_{(j)} r_{(i)}^T A d_{(j)} \text{ (dot product with } r_{(i)}^T) \\ \Leftrightarrow \alpha_{(j)} r_{(i)}^T A d_{(j)} &= r_{(i)}^T r_{(j)} - r_{(i)}^T r_{(j+1)}. \end{aligned}$$

The residuals $r_{(i)}$ and $r_{(j)}$ are orthogonal (the dot product is zero) when $i \neq j$. Accounting for this fact, we isolate three cases:

$$r_{(i)}^T A d_{(j)} = \begin{cases} \frac{1}{\alpha_{(i)}} r_{(i)}^T r_{(i)}, & \text{if } i = j \\ -\frac{1}{\alpha_{(i-1)}} r_{(i)}^T r_{(i)}, & \text{if } i = j+1 \\ 0, & \text{otherwise} \end{cases}$$

which reduce to the following two possible expressions for β:

$$\beta_{ij} = \begin{cases} \frac{1}{\alpha_{(i-1)}} \frac{r_{(i)}^T r_{(i)}}{d_{(i-1)}^T A d_{(i-1)}}, & \text{if } i = j \\ 0, & \text{if } i > j+1 \end{cases}$$

Most of the terms are zero, which greatly reduces the number of calculations we must perform. In fact, this reduces the complexity of the algorithm from $\mathcal{O}(n^2)$ to $\mathcal{O}(m^2)$, where m is the number of nonzero entries of the matrix A. A further simplification of the above entails expressing α as

$$\alpha_{(i)} = \frac{d_{(i)}^T r_{(i)}}{d_{(i)}^T A d_{(i)}}$$

which simplifies the formula for β. Denoting $\beta_{(i)} = \beta_{i,i-1}$, we have that

$$\beta_{(i)} = \frac{r_{(i)}^T r_{(i)}}{d_{(i-1)}^T r_{(i)}} = \frac{r_{(i)}^T r_{(i)}}{r_{(i-1)}^T r_{(i)}}$$

since $d_{(i-1)}^T r_{(i)} = r_{(i-1)}^T r_{(i)}$.

To conclude, the method of Conjugate Gradients is an iterative method summarized by the following equations:

$$d_{(0)} = r_{(0)} = b - Ax_{(0)},$$

$$\alpha_{(i)} = \frac{d_{(i)}^T r_{(i)}}{d_{(i)}^T A d_{(i)}},$$

$$x_{(i+1)} = x_{(i)} + \alpha_{(i)} d_{(i)},$$

$$r_{(i+1)} = r_{(i)} - \alpha_{(i)} A d_{(i)},$$

$$\beta_{(i+1)} = \frac{r_{(i+1)}^T r_{(i+1)}}{r_{(i)}^T r_{(i)}},$$

$$d_{(i+1)} = r_{(i+1)} + \beta_{(i+1)} d_{(i)}.$$

11. Implementation in MATLAB: Conjugate Gradient Method

```
function x = conjgrad(A,b,tol)
% CONJGRAD  Conjugate Gradient Method.
%   X = CONJGRAD(A,B) attemps to solve the system of linear equations Ax=b
%   for x. The N-by-N coefficient matrix A must be symmetric and the right
%   hand side column vector b must have length N.
%
%   X = CONJGRAD(A,B,TOL) specifies the tolerance of the method. The
%   default is 1e-10.
%

if nargin<3
    tol = 1e-10;
end
x = b;
r = b - A*x;
if norm(r) < tol
    return
end
d = r;
delta_new = r'*r;
for i = 1:numel(b);
    q = A*d;
    s = d'*q;
    alpha = (r'*d)/s;
    x = x + alpha*d;
    if mod(i, 50) == 0
        r = b - A*x;
    else
```

```
        r = r - alpha*q;
    end
    if norm(r) < tol
        return
    end
    delta_old = delta_new;
    delta_new = r'*r;
    beta = delta_new/delta_old;
    d = r + beta*d;
end
end
```

12. Power Method

In some engineering applications, we are mainly interested in finding the largest (or smallest) eigenvalue of a given matrix. The power method is an iterative scheme that allows us to find the dominant - that is, the largest in absolute value - eigenvalue of a diagonalizable square matrix.

12.1. **Formulation.**

Let A be an $n \times n$ diagonalizable matrix and suppose its eigenvalues are ordered as

$$|\lambda_1| > |\lambda_2| \geq \cdots \geq |\lambda_n|$$

where λ_1, the so-called **dominant eigenvalue** of A, has multiplicity 1. Given an arbitrary unitary vector $q^{(0)} \in \mathbb{C}^n$, the power method is defined as the following iterative scheme:

$$z^{(k)} = Aq^{(k-1)},$$

$$q^{(k)} = \frac{z^{(k)}}{||z^{(k)}||_2},$$

$$\nu^{(k)} = \left(q^{(k)}\right)^T Aq^{(k)}.$$

Although we will not derive this result, it can be shown that as $k \to \infty$, $q^{(k)}$ aligns itself along the direction of the eigenvector x_1 of A corresponding to λ_1. Also, the Rayleigh quotients ν converge to λ_1.

13. Inverse Power Method

Theorem 4. *Suppose A is a nonsingular square matrix and λ is an eigenvalue of A. Then λ^{-1} is an eigenvalue of A^{-1}, the matrix inverse of A. Also, A and A^{-1} share the same set of eigenvectors.*

Proof. Let A be a nonsingular $n \times n$ matrix and $v \in \mathbb{R}^n$ be an eigenvector of A associated with the eigenvalue $\lambda \neq 0$. Then

$$Av = \lambda v.$$

Multiplying both sides of the equation by the inverse of A, we have that

$$A^{-1}(Av) = A^{-1}(\lambda v) \Leftrightarrow Iv = \lambda A^{-1}v.$$

That is,

$$A^{-1}v = \lambda^{-1}v.$$

Hence v is an eigenvector of A^{-1} associated with the eigenvalue λ^{-1}, which concludes our proof. □

Theorem 5. *Suppose A is a nonsingular square matrix and λ is an eigenvalue of A. Then $\lambda - \mu$ is an eigenvalue of $A - \mu I$. Moreover, if v is an eigenvector of A, it is also an eigenvector of $A - \mu I$.*

Proof. Let A be a nonsingular $n \times n$ matrix and λ one of its eigenvalues. Then

$$Av = \lambda v$$

for some $v \in \mathbb{R}^n$. Now,

$$(A - \mu I)v = Av - \mu(Iv) = \lambda v - \mu v = (\lambda - \mu)v.$$

That is,

$$(A - \mu I)v = (\lambda - \mu)v$$

which implies that $\lambda - \mu$ is an eigenvalue of $A - \mu I$. Also, by definition v is an eigenvector of $A - \mu I$. $\square$

Definition 3. *Let A be an $n \times n$ matrix and $x \in \mathbb{R}^n$. The scalar*

$$r(x) = \frac{x^T A x}{x^T x}$$

is called a ***Rayleigh quotient****.*

Note: If x is an eigenvector of A with the associated eigenvalue λ,

$$r(x) = \frac{x^T A x}{x^T x} = \frac{x^T(\lambda x)}{x^T x} = \lambda.$$

13.1. **Introduction.** The Inverse Power Method, or Inverse Iteration, is an iterative eigenvalue algorithm based on the Power Method. The key difference is that instead of converging to the largest eigenvalue of a matrix A, the inverse power method allows us to compute the eigenvalue of A that is closest to a given number μ. In other words, the inverse power method essentially reverses the iteration step of the power method. Where the power method computes vectors in the sequence

$$z^{(k)} = Aq^{(k-1)},$$

the inverse power method uses the matrix inverse $(A - \mu I)^{-1}$ instead of A,

$$z^{(k)} = (A - \mu I)^{-1} q^{(k-1)}.$$

The iteration above converges to the dominant eigenvalue of the matrix $(A - \mu I)^{-1}$. Given the eigenvalues

$$\lambda_1, \lambda_2, \cdots, \lambda_n$$

of A, the eigenvalues of $(A - \mu I)^{-1}$ are

$$(\lambda_1 - \mu)^{-1}, (\lambda_2 - \mu)^{-1}, \cdots, (\lambda_n - \mu)^{-1}$$

by theorems 4 and 5. The dominant eigenvalue of $(A - \mu I)^{-1}$ corresponds to the smallest of

$$|\lambda_1 - \mu|, |\lambda_2 - \mu|, \cdots, |\lambda_n - \mu|,$$

which gives us the closest eigenvalue of A to μ, just as we stated previously. Just as we assumed that the dominant eigenvalue of A had multiplicity 1 in the power method, here we assume that the eigenvalue closest to μ also has multiplicity 1. It follows that if we wish to find the **smallest** eigenvalue of A, we set $\mu = 0$.

Although being more computationally expensive than the power method, the inverse iteration has the advantage that it can converge to any eigenvalue of a given matrix. Is it particularly useful in refining an initial estimate μ of an eigenvalue of A.

13.2. **Formulation.** Let A be an $n \times n$ diagonalizable matrix with real entries. Given an arbitrary unitary vector $q^{(0)} \in \mathbb{R}^n$, the inverse power method is constructed in the following way:

```
(1) Initialize $q^{(0)}$ with an arbitrary vector of unitary 2-norm.
(2) For $i = 1, 2, \cdots$
(3)     Solve $(A - \mu I)z^{(k)} = q^{(k-1)}$ for $z^{(k)}$
(4)     $q^{(k)} = z^{(k)}/||z^{(k)}||_2$
(5)     $\sigma^{(k)} = \left(q^{(k)}\right)^T Aq^{(k)}$
```

Computing the inverse of A would be computationally expensive. Thus in step (3) we factor A (using QR or Cholesky factorization, depending on the characteristics of the matrix) and determine $q^{(k-1)}$ by forward and back substitutions. This factorization only needs to be done once.

13.3. **Convergence.** Let λ_m be the eigenvalue of A that is closest to μ, x_m be its associated eigenvector, λ_p the second-closest eigenvalue of A to μ. Then

$$\lim_{k\to\infty} q^{(k)} = x_m$$

and

$$\lim_{k\to\infty} \sigma^{(k)} = \lambda_m.$$

As $q^{(k)}$ approaches the eigenvector x_m, the Rayleigh quotient of $q^{(k)}$ approaches the eigenvalue λ_m.

The speed of convergence of this iterative scheme depends on the ratio between the dominant eigenvalue and second largest eigenvalue (that is, the second-closest to μ). It can be shown that the convergence to the eigenvector is linear, while convergence to the eigenvalue is quadratic. That is,

$$|q^{(k)} - x_m| = \mathcal{O}\left(\left|\frac{\lambda_p}{\lambda_m}\right|^k\right)$$

and

$$|\sigma^{(k)} - \lambda_m| = \mathcal{O}\left(\left|\frac{\lambda_p}{\lambda_m}\right|^{2k}\right).$$

An improvement in the rate of convergence can be achieved by updating the estimate μ for the eigenvalue with the Rayleigh quotient at each iteration. The resulting algorithm converges cubically - the number of correct digits triples in each iteration. However, since the matrix $(A - \mu I)^{-1}$ is continuously changing, we must factor it for every iteration, making this a particularly expensive modification to the original scheme.

13.4. **Worked Example.**

14. Implementation in MATLAB: Inverse Power Method

```
function [s, q, k, err] = inv_power(A, u, nsteps, tol)

%INV_POWER   finds the eigenvalue of A closest to a given number u.
%
%     calling sequences:
%             s = inv_power(A, u, nsteps, qinit, tol)
%             [q, err] = inv_power(A,u , nsteps, qinit,tol)
%
%     inputs:
%             A            coefficient matrix for linear system
%                          (matrix must be square)
%             u            determines the closest eigenvalue
%             nsteps       number of steps
%             tol          error tolerance for approximate solution
%
%     output:
%             q       approximated eigenvector closest to u
%             e       approximated eigenvalue corresponding to q
%
%     Based on http://www4.ncsu.edu/~mtchu/Teaching/Lectures/MA428/By_subjects/18_inverse_power.pdf



%% Making sure we have the right parameters

[nrow, ncol] = size(A);
nu = length(u);
qinit = ones(size(A,1),1)./sqrt(size(A,1));
```

```
if(nrow ~= ncol)
    disp ('inv_power error: Square coefficient matrix required');
    return
end

if (1 ~= nu)
    disp ('inv_power error: Size of b-vector must be 1 (a scalar)')
    return
end

if(tol <= 0)
    disp('inv_power error: Tolerance must be a positive number');
    return
end

if(nsteps < 1 || mod(nsteps,1) ~=0)
    disp('inv_power error: The number of steps must be an integer greater than 1');
    return
end

if(all(diag(A)) == 0)
    disp ('inv_power error: Entries in the diagonal of A must be non-zero');
    return
end

%% Define relevant matrices and vectors

q = zeros(nrow, nsteps);
s = zeros(1, nsteps);
err = zeros(nsteps, 1);
q(:, 1) = qinit;

%% Iteration

for k=2:(nsteps+1)

    % Apply iterative update
    zk = (A - u*eye(nrow))\q(:, k-1);  % z(k) = (A - uI)^(-1) * q^(k-1)
    qk = zk / norm(zk); q(:, k) = qk;  % q(k) = z(k)/||z(k)||
    s(k) = qk'*A*qk;                   % s(k) = q(k)^T * A * q(k)

    % Terminate loop if error is below tolerance
    diff = s(:,k) - s(:,k-1);   % difference between successive approxs
    err(k-1) = norm(diff, 2);   % 2 norm of diff

    if (err(k-1) < tol)
        % Delete zero columns from u and err
        q(:,k+1:end) = [];
        s(:, k+1:end) = [];
        err(k:end)   = [];
       fprintf ('Desired tolerance of %d was achieved in fewer than %d steps \n', tol, nsteps)
        return
    end
end
```

15. References

Jacobi Method

https://www.cis.upenn.edu/ cis515/cis515-12-sl5.pdf

http://www.robots.ox.ac.uk/ sjrob/Teaching/EngComp/linAlg34.pdf

Householder QR

https://en.wikipedia.org/wiki/QR_decomposition

https://www.cs.utexas.edu/users/rvdg/books/HQR.pdf

https://www.inf.ethz.ch/personal/gander/papers/qrneu.pdf

https://www.cs.cornell.edu/ bindel/class/cs6210-f12/notes/lec16.pdf

Conjugate Directions and Conjugate Gradient

http://www.cs.cmu.edu/ quake-papers/painless-conjugate-gradient.pdf

Inverse Power Method

http://www4.ncsu.edu/m̃tchu/Teaching/Lectures/MA428/By_subjects/18_inverse_power.pdf

http://www4.ncsu.edu/ĩpsen/ps/sirev_invit.pdf

http://math.unice.fr/ frapetti/CorsoF/cours4part2.pdf

https://www.mathematik.uni-wuerzburg.de/ borzi/RQGradient_Chapter_10.pdf