Exercise F1: Over the course of the semester, several times we haf used a two-word phrase to describe a factorization. in fact there are 15 possible 2WPs which can be formed from a column-A adjective and a column-B noun form this table,

Figure 1: 2WP Table

column A	*column B*
triangular	triangularization
orthogonal	orthogonalization
unitary	diagonalization
	tridiagonalization
	hessenbergization

a Consider only the textbook Lectures which we covered, namely Lectures 1-17 and 20-27. For square matrices, which distinct factorization do use, or easily could use, one of these 15 possible 2WPs? For each factorization, state the matrix factorization, its name and/or algorithm reference in the textbook, and it's 2WP.

Solution:

1 Gram-Schmidt ($A = QR$) (Triangular Orthogonalization)
The Gram-Schmidt algorithm produces a QR factorization by repeatedly applying triangular matrices to the right of A to form an orthogonal matrix Q, as we saw in Lecture 8.

2 Householder ($A = QR$) (Orthogonal Triagularization)
The Householder Algorithm produces a QR factorization by repeatedly applying orthogonal matrices (Householder reflectors) the the left of A to form a triangular matrix, as we saw in Lecture 10.

3 Gaussian Elimination ($A = LU$) (Triangular Triagularization)
The Gaussian elimination algorithm produces an LU factorization by repeatedly applying lower triangular matrices (row operations) to the left of A to produce an upper triangular matrix, as we saw in Lecture 20.

4 Cholesky ($A = R^*R$) (Triangular Triagularization)
Cholesky factorization only succeeds when A is a hermitian positive definite matrix, and it can be thought of as applying Gaussian Elimination on a symmetric matrix with an extra factorization at each iteration which enables $L_i^* = U_i$. This factorization is described in detail in Lecture 23 and since it is a form of Gaussian Elimination can be described as Triangular Triagularization

5 Shur Decomposition ($A = QTQ^*$) (Unitary Triagularization)
The Shur Decomposition is obtained by applying unitary matrices constructed by normalized eigenvectors, as we saw in lecture 24 (Theorem 24.9). Interestingly since the Shur Decomposition is an eigenvalue revealing factorization, that is to say for matrices of a certain size it must be computed iteratively (as we did in one of the homework assignments).

6 Unitary Diagonalization ($A = Q\Lambda Q^*$)
Another eigenvalue revealing factorization, which can be thought of as Shur Decomposition when A is a normal matrix, as we saw in Lecture 24.

7 Hessenberg Form ($A = QHQ^*$) (Unitary Hessenbergization)
Hessenberg form is achieved by applying modified householder reflectors (leave the ith column/row unchanged) to the left and right sides of A, as we saw in lecture 26.

8 Tridiagonal Form ($A = Q\hat{T}Q^*$) (Unitary Tridiagonalization)
Reduction to Tridiagonal Form is achieved when Hessenberg Algorithm is applied to a hermitian matrix A, as we saw in Lecture 26.

b The adjectives 'orthogonal' and 'unitary' are essentially synonyms. However the textbook systematically uses "unitary" for certain kind of factorization. Explain, in a few sentences, the distinct purpose of the 'unitary' factorization.

Solution:
The textbook defines an 'orthogonal' matrix as the real number equivalent to being unitary. To elaborate, when a matrix $A \in \mathbb{C}^{m x m}$ has the property that $A^* = A^{-1}$ we say that it is unitary, when a matrix $A \in \mathbb{R}^{m x m}$ we can say it is orthogonal or unitary. The textbook makes sure to describe these matrices as 'unitary' when the factorization is an eigenvalue-revealing factorization since 'even if the input to a matrix eigenvalue problem is real, the output may have to be complex.'(as stated in Theorem 24.1).

c The SVD factorization is not one of the answers in part (A). Why? Invent a good 2WP to describe it.

Solution:
The SVD factorization reduces a matrix A by $A = U\Sigma V^*$ where U and V are unitary, and Σ is a diagonal matrix(with dimensions A). We can rewrite this factorization as $U^*AV = \Sigma$ and then it can be thought of as Unitary Diagonalization. However when we computed the SVD we did so by solving the eigenvector, eigenvalue problem for A^*A (or AA^*) and then used linear systems to solve for leftover unitary factor, either

U or V depending on how the problem was set up. I could see it being difficult to identify a two-word phrase that encapsulates that algorithm.

d Create, implement, and test an algorithm corresponding to a 2WP which is unused. that is invent an algorithm, which is not in the textbook and which is not listed in your part a answer.

Solution:
Piggybacking off the last problem, since we have an algorithm which solves the eigenvalue problem, namely rqi we can link it together with Householder to produce the SVD. I think it would go like this, first we form A^*A, then we use Householder to find an orthogonal basis for A^*A namely Q. Then we use the columns of Q as initial guesses for rqi to solve the eigenvalue problem. We form a unitary factor, then solve for the columns of the other by setting up the linear system and using householder.

Exercise F2: The producers and I believe we have a great new reality series, Naked and Calculating. (It will do great in the North Korean market ... little competition.) The plan is to have three contestants, each seeking riches, on three islands. Each contestant will have unlimited supply of pencils and paper, plus adequate food and drink, but nothing else. Naturally, there are hidden cameras so we can watch the exciting action! Each island has a problem, and algorithm which the contestant must use:

Island 1: Compute the determinant of an *mxm* matrix using expansion in minors.
Island 2: Solve an *mxm* upper-triangular linear system using back-substitution.
Island 3: Solve an *mxm* linear system using Gaussian elimination.

After finding-out which one is their island, a contestant chooses them value, but then the producers will choose generic values to fill the matrix entries. By choosing m the contestants are gambling that they can compute the correct answer, by hand, in one month of work. In fact, at the end of one month a contestant either gets 2^m US dollars for correctly computing the solution of a problem, or zero dollars if their answer is incorrect. (All entries in the final answer have to be correct to three digits.).

a As a contestant, which island would you choose? Most avoid? Please? Explain?

Solution:
The choice between Island 2, and Island 3 is clear since solving a linear system using Gaussian Elimination requires us to reduce to an upper-triangular system, and then solve that by using back-substitution. Therefore, depending on A Island 2 will always have less than or equal to the amount of work in Island 3. Recall that in Homework 2, Problem 6.b we computed that the number of FLOPs for computing the determinant of an *mxm* matrix was on the order of $O(m!)$ whereas the the text cites the number of FLOPs required for back substitution as $O(m^2)$ (17.2). As a contestant I would go for Island 2.

b For each island, state the specific m you would choose if you are put on that island. You must justify your choices via a quantitative, though necessarilly speculative, explaination.

Solution:
As an estimate I think that typically I am able to perform an arithmetic operation once every 30 seconds. If I was doing this for real I would collect data for each operation and operation length, but for a rough estimate let's say one operation every 30 seconds. Computing an estimate for total operations in a 30 day period we get,

$$\frac{2 \text{ Operations}}{1 \text{ Minute}} \frac{60 \text{ Minute}}{1 \text{ Hour}} \frac{8 \text{ Hour}}{1 \text{ Day}} * 30 = 28,800 \text{ Operations}$$

Island 1 Recall that the number of operations to compute the determinant of an *mxm* matrix by expansion in minors is $O(m!)$. With a rough estimate of 28,800 operations over the 30 day period I would have to choose $m = 7$, since $7! = 5,040$ and $8! = 40,320$. This gives me a huge amount of time to double check my answers but it's unfortunate that increasing m by one more has such a dramatic change in the number of operations.

Island 2 As was stated in the previous problem, the number of operations required to perform back substitution on a *mxm* system is $\sim m^2$. With an estimate of 28,800 operations over the 30 day period we get an m value of

$$m = \sqrt{28800} \approx 169.7.$$

So I would likely choose $m = 169$.

Island 3 The text states that the number of operations required to perform Gaussian elimination(no pivoting) on an *mxm* system is $\sim \frac{2}{3}m^3(20.8)$. With my estimate of 28,800 operations over the 30 day period we get that,

$$m = (\frac{3}{2}28,800)^{\frac{1}{3}} \approx 35.08.$$

So I would likely choose $m = 35$.

Clearly these are all upper bounds on m as we would have to carve out time to double check computations. It is most definitely the case that using a single value is a poor way to model computations over time since there are various factors such as type of operation, number length, and fatigue that should be taken into consideration.

c For excitement, at the last minute before going on their island, after already having chosen an m value, contestants are told that in fact they can choose their algorithm, and that they can revise their m choice. For one fo the islands this represents a huge improvement in the pay-out. Explain; give a new m value with explanation.

Solution:
Island 1 should compute their determinant by producing the LU factorization of A. Note that if we let $A = LU$, then we get the following,

$$\begin{aligned} det(A) &= det(LU), \\ det(A) &= det(L)det(U), \\ det(A) &= det(LU), \\ det(A) &= \prod_{i=1}^{m} U_{i,i}. \end{aligned}$$

This reduces the FLOPs to $\sim \frac{2}{3}m^3 + (m-1)$, where we get $\sim \frac{2}{3}m^2$ operations to perform the LU factorization and $(m-1)$ operations to evaluate the final product. Now that Island 1 and Island 3 are using the same algorithm I would likely estimate the same $m = 35$ for Island 1 using this new method. I would note however that now Island 1 requires fewer operations than Island 3 since after forming the LU factorization all we need to do is evaluate the diagonal product U which is order m less expensive than performing back substitution.

Exercise F3: Consider the following $A \in \mathbb{R}^{3x4}$,

$$A = \begin{bmatrix} 2 & 3 & 5 & -1 \\ -1 & 7 & 3 & 5 \\ -1 & -1 & 2 & 4 \end{bmatrix}$$

Find the $B \in \mathbb{R}^{3x4}$ of rank 2 which is the closest to A in the 2-norm.

Solution:
Recall from Theorem 5.7 that A can be decomposed to a sum of r rank-one matrices, where r the rank of A,

$$A = \sum_{j=1}^{r} \sigma_j u_j v_j^*.$$

Recall that by Theorem 5.8 the v th partial sum, where $0 \leqslant v \leqslant r$ gives us the rank v, best approximation to A under the 2-norm. The following Matlab script computes B, the second partial sum of A's rank-one decomposition,
Code:

```
%Initialize A
A = [2  3  5 -1;
    -1  7  3  5;
    -1 -1  2  4];

%Compute SVD
[U, S, V] = svd(A);

%Initialize Matrix B
B = zeros(3,4);

%Populate B with 2nd partial sum of A's Rank-One Decomposition.
for i = 1:2
    B = B + S(i,i)*U(:,i)*V(:,i)';
end

% B =
%     1.8390    3.9718    3.9798   -1.3639
%    -0.8435    6.0554    3.9916    5.3537
%    -1.3234    0.9525   -0.0498    3.2689
```

Exercise F4: Expansion in minors may be a terrible algorithm for computing determinants, but it is backwards-stable. Prove this in the $1x1$ and $2x2$ matrix cases, assuming as usual that your computer satisfies axioms (13.5) and (13.7).

Solution:
Let $\tilde{f}$ be the Expansion in minors algorithm for the problem of computing the determinant of a matrix. Consider a $1x1$ matrix A whose sole entry is a_{11}. We know that the determinant of a $1x1$ matrix is simply the single entry, so computing the algorithm $\tilde{f}(A)$, we get

$$\tilde{f}(A) = fl(a_{11}) = a_{11}(1+\epsilon_1) = f(\tilde{A}).$$

Where $\tilde{A}$ is a $1x1$ matrix whose entry is $a_{11}(1+\epsilon_1)$. By axiom (13.5) it follows that,

$$\begin{aligned}
\frac{||\tilde{A}-A||}{||A||} &= \frac{|a_{11}(1+\epsilon_1)-a_{11}|}{|a_{11}|}, \\
&= \frac{|a_{11}((1+\epsilon_1)-1)|}{|a_{11}|}, \\
&= \frac{|a_{11}||\epsilon_1|}{|a_{11}|}, \\
&= \epsilon_1, \\
&= O(\epsilon_{machine}).
\end{aligned}$$

Thus the $1x1$ matrix case of Expansion in minors is backwards stable.

Consider a $2x2$ matrix A with the following entries,

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$$

Applying $\tilde{f}$ to A we get the following,

$$\begin{aligned}
\tilde{f}(A) &= fl(a_{11}) \otimes fl(a_{22}) \ominus fl(a_{21}) \otimes fl(a_{12}) \\
&= (a_{11}a_{22})(1+\epsilon_1) \ominus (a_{21}a_{12})(1+\epsilon_2) \\
&= ((a_{11}a_{22})(1+\epsilon_1) - (a_{21}a_{12})(1+\epsilon_2))(1+\epsilon_3) \\
&= (a_{11}a_{22})(1+\epsilon_1)(1+\epsilon_3) - (a_{21}a_{12})(1+\epsilon_2)(1+\epsilon_3) \\
&= (a_{11}(1+\epsilon_1))(a_{22}(1+\epsilon_3)) - (a_{21}(1+\epsilon_2))(a_{12})(1+\epsilon_3)) \\
&= (\tilde{a}_{11}\tilde{a}_{22}) - (\tilde{a}_{21}\tilde{a}_{12}) \\
&= f(\tilde{A})
\end{aligned}$$

Where $\tilde{A}$ is,

$$\tilde{A} = \begin{bmatrix} a_{11}(1+\epsilon_1) & a_{12}(1+\epsilon_3) \\ a_{21}(1+\epsilon_2) & a_{22}(1+\epsilon_3) \end{bmatrix}.$$

Now note that,

$$\tilde{A} - A = \begin{bmatrix} a_{11}(1+\epsilon_1)-a_{11} & a_{12}(1+\epsilon_3)-a_{12} \\ a_{21}(1+\epsilon_2)-a_{21} & a_{22}(1+\epsilon_3)-a_{22} \end{bmatrix} = \begin{bmatrix} a_{11}\epsilon_1 & a_{12}\epsilon_3 \\ a_{21}\epsilon_2 & a_{22}\epsilon_3 \end{bmatrix}$$

So applying the Frobenius norm we get the following,

$$\begin{aligned}
\frac{||\tilde{A} - A||_f}{||A||_f} &= \frac{((a_{11}\epsilon_1)^2 + (a_{12}\epsilon_3)^2 + (a_{21}\epsilon_2)^2 + (a_{22}\epsilon_3)^2)^{\frac{1}{2}}}{((a_{11})^2 + (a_{12})^2 + (a_{21})^2 + (a_{22})^2)^{\frac{1}{2}}}, \\
&\leqslant \frac{((a_{11}\epsilon_{max})^2 + (a_{12}\epsilon_{max})^2 + (a_{21}\epsilon_{max})^2 + (a_{22}\epsilon_{max})^2)^{\frac{1}{2}}}{((a_{11})^2 + (a_{12})^2 + (a_{21})^2 + (a_{22})^2)^{\frac{1}{2}}}, \\
&= \frac{\left(a_{11}^2\epsilon_{max}^2 + a_{12}^2\epsilon_{max}^2 + a_{21}^2\epsilon_{max}^2 + a_{22}^2\epsilon_{max}^2\right)^{\frac{1}{2}}}{((a_{11})^2 + (a_{12})^2 + (a_{21})^2 + (a_{22})^2)^{\frac{1}{2}}}, \\
&= \frac{((a_{11})^2 + (a_{12})^2 + (a_{21})^2 + (a_{22})^2)^{\frac{1}{2}}\,\epsilon_{max}}{((a_{11})^2 + (a_{12})^2 + (a_{21})^2 + (a_{22})^2)^{\frac{1}{2}}}, \\
&= \epsilon_{max} = O(\epsilon_{machine}).
\end{aligned}$$

Thus the $2x2$ matrix case of Expansion in minors is backwards stable.

Exercise F5: Recall the following ideas about our most trustworthy linear solver: Algorithm 10.1 is Householder triagularization $A = QR$. Code house.m implements this; read and understand it! It outputs a lower triangular matrix W containing the v_k vectors and an upper triangular R, but of course $WR \neq A$. Next, Alg. 10.2 implements the action of Q^* from the vectors stored in W. Alg 16.1 adds back substitution (Alg.17.1) to give a solver for square, nonsingular linear systems $AX = b$. Theorem 16.2 shows that Alg. 16.1 is backwards-stable.
Implement Alg16.1, but do it in-place. In particular, write a Matlab function,

$$x = \text{inhousesolve(A,b)}$$

This function should start by checking that the inputs make sense i.e A is square (mxm) and b is a compatibly sized column vector. After that, the next line of your code should append one row of space to the bottom of the array A, like this,

$$A = [A; \text{zeros(1,m)}];$$

From now on, the only matrix or 2D array in your code is A itself, and A does not change size. You will modify entries of the array A, and the goal is to construct the vector x. Inside your function you will implement Alg 10.1, by modifying the entries of A and to store W and R, implement Alg 10.2 by referring to the entries of array A, and implement Alg 17.1 by referring to other entries of A.

Solution:
Consider the following Matlab function,
Code:

```
function b = inhousesolve(A, b)
% This function takes a square matrix A and a column vector b
% and solves A\b via inplace householder qr.


%Error Checking
[m, n] = size(A);
[brow, bcol] = size(b);
    if m ~= n
        error('This function only accepts square Matrices')
    end

    if (bcol ~= 1) || (brow ~= m)
        error('This function only a mx1 column vector')
    end


%Inplace Computation of R and W
A = [A; zeros(1,m)];
    for k = 1:n
```

```
        %%%%%%%%%%%%%%%%%%% This Block is the same as house.m %%%%%%%%%%%%%%%%%%%
        v = A(k:m,k); % v is (m-k+1) x 1 column vector
        v(1) = sign(v(1)) * norm(v,2) + v(1);
        v = v / norm(v,2);
        A(k:m,k:n) = A(k:m,k:n) - 2 * v * (v' * A(k:m,k:n));
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        A(k+1:m+1,k) = v;% store v in A
    end


%Linear Solve by only referencing A
%Forming Q^*b in place of b
    for k = 1:m
        b(k:m,:) =  b(k:m,:) - 2*A(k+1:m+1,k)*(A(k+1:m+1,k)'* b(k:m,:));
    end

%Solving for Rx = Q^*b via in place backsubstitution.
b(m,1) = b(m,1) / A(m,m);
    for i = m-1:-1:1
      b(i,1) = (b(i,1) - A(i,i+1:m) * b(i+1:m,1) ) / A(i,i);
    end

end
```

Console:

```
>> A = magic(5)

A =

    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9

>> b = [1 ; 2; 3; 4; 5]

b =

     1
     2
     3
     4
     5

>> x1 = inhousesolve(A,b)
```

```
x1 =

    0.0128
    0.0128
    0.1795
    0.0128
    0.0128

>> x = A\b

x =

    0.0128
    0.0128
    0.1795
    0.0128
    0.0128

>> norm(x1 - x)

ans =

   1.1526e-16
```

Exercise 6: a Implement Algorithm 26.1, Householder reduction to Hessenberg form, Specifically build code with the signature,

$$H = \text{hessen(A, stages)}$$

Your code will check that A is square, print the stages (next part) if stages is true, and finally return a Hessenberg matrix H such that $A = QHQ^*$ for some unitary Q. Note that your code can throw-away the vectors v_k; they are not returned.

Solution:
Consider the following Matlab function,
Code:

```
function A = hessen(A, stages)
% This function takes a square matrix A and
% a bool, stages and reduces the matrix A to
% Hessenberg form via algorithm 26.1. The function will display
% each stage of the algorithm when stages is set to true.

% Error Checking
[m, n] = size(A);
    if m ~= n
        error('This function only accepts square Matrices')
    end


% Reduction to Hessenberg Form
    for k = 1:m-2
        if (stages == true) % Display stages
            disp(strcat('This is stage:', int2str(k)))
            disp(A)
        end
        v = A(k+1:m,k);
        v(1) = sign(v(1)) * norm(v,2) + v(1);
        v = v / norm(v,2);
        A(k+1:m, k:m) =  A(k+1:m, k:m) - 2*v*(v'* A(k+1:m, k:m));
        A(1:m,k+1:m) =  A(1:m,k+1:m) - 2*(A(1:m,k+1:m)*v)*v';
    end

    if (stages == true) % Display stages
        disp(strcat('This is stage:', int2str(m-1)))
        disp(A)
    end
end
```

b For a specific 5x5 matrix A of your choice, run the code and show the four stages A, $Q_1^* A Q_1$,$Q_2^* Q_1^* A Q_1 Q_2$, and $H = Q_3^* Q_2^* Q_1^* A Q_1 Q_2 Q_3$. That is, make concrete the cartoons on pages 197-198, 'A Good Idea'.

Solution:
Calling hessen() on a 5x5 magic matrix A we get,
Console:

```
>> A = magic(5)

A =

    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

>> H1 = hessen(A, true)

 This is stage:1
    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9

 This is stage:2
   17.0000  -28.9413   -3.1787   -2.4469    3.5084
  -27.6767   33.9399  -20.5149  -12.9517  -10.1857
    0.0000  -18.8471    8.6867   12.5188   14.0439
    0.0000  -11.5283   11.2626    9.9119   -8.5139
    0.0000  -11.9043   16.1713  -10.9908   -4.5386

 This is stage:3
   17.0000  -28.9413    1.8470   -1.1284    4.8699
  -27.6767   33.9399   26.1875   -0.6996    2.4660
    0.0000   25.0964   20.6871    1.7669    6.3679
    0.0000         0    1.4221    3.6837  -14.0463
    0.0000   -0.0000    5.7909  -17.4796  -10.3107

 This is stage:4
   17.0000  -28.9413    1.8470   -4.4603    2.2572
  -27.6767   33.9399   26.1875   -2.2280    1.2675
    0.0000   25.0964   20.6871   -6.6055   -0.1973
```

```
    0.0000         0      -5.9630    -16.8163    -12.4454
    0.0000   -0.0000      -0.0000     -9.0122     10.1893


H1 =

   17.0000  -28.9413       1.8470     -4.4603      2.2572
  -27.6767   33.9399      26.1875     -2.2280      1.2675
    0.0000   25.0964      20.6871     -6.6055     -0.1973
    0.0000         0      -5.9630    -16.8163    -12.4454
    0.0000   -0.0000      -0.0000     -9.0122     10.1893

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

>> H = hess(A)

H =

   17.0000  -28.9413       1.8470     -4.4603      2.2572
  -27.6767   33.9399      26.1875     -2.2280      1.2675
         0   25.0964      20.6871     -6.6055     -0.1973
         0         0      -5.9630    -16.8163    -12.4454
         0         0            0     -9.0122     10.1893

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

>> norm((H - H1),2)

   3.1283e-14
```

c For the same $5x5$ matrix A, use the built-in eig() to show that the eigenvalues of A and H are the same within rounding error. Now construct a new $4x4$ hermitian matrix S and compute $T = hessen(S)$. Check that T is tridiagonal and Hermitian. Then show the eigenvalues of S and T are the same within rounding error.

Solution:
Generating the eig values of A and H using the built-in eig() we find that they are within rounding error, with the 2-norm of the difference having size on the order of 10^{-14}.
Console:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
>> A = magic(5)

A =
```

```
    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9

>> H = hessen(A, false)

H =

   17.0000  -28.9413    1.8470   -4.4603    2.2572
  -27.6767   33.9399   26.1875   -2.2280    1.2675
    0.0000   25.0964   20.6871   -6.6055   -0.1973
    0.0000         0   -5.9630  -16.8163  -12.4454
    0.0000   -0.0000   -0.0000   -9.0122   10.1893

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
>> eigH = eig(H)

eigH =

   65.0000
  -21.2768
  -13.1263
   21.2768
   13.1263

>> eigA = eig(A)

eigA =

   65.0000
  -21.2768
  -13.1263
   21.2768
   13.1263

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
>> norm(sort(eigA) - sort(eigH))
ans =
   4.7399e-14
```

Computing a 4x4 hermitian matrix S and computing it's tridiagonal decomposition T we get that the eigenvalues, when computed with the built-in eig() function are within rounding error on the order of 10^{-12}.
Console:

```
>> S = magic(4);, S = S'*S

S =

   378   206   212   360
   206   370   368   212
   212   368   370   206
   360   212   206   378

>> T = hessen(S, false)

T =

  378.0000  -465.8111          0          0
 -465.8111   812.7462   234.3551          0
         0   234.3551   302.4527   -14.2396
         0          0   -14.2396     2.8011

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
>> eigT = eig(T)

eigT =

   1.0e+03 *

   1.156000000000001
   0.320000000000000
   0.020000000000000
  -0.000000000000000

>> eigS = eig(S)

eigS =

   1.0e+03 *

  -0.000000000000000
   0.020000000000000
   0.320000000000000
   1.156000000000000

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
>> norm(sort(eigS) - sort(eigT))
ans =
     1.080123035699853e-12
```

Exercise F7: Implement Algorithm 27.3, namely Rayleigh quotient iteration. In particular, write a Matlab code with the signature:

$$[lam, v] = \text{rqi(A, v0)}$$

which returns an eigenvalue *lam* corresponding to the eigenvector v, and which starts the iteration with $v0$. You will need a stopping criterion to avoid a warning when solving the linear system with the matrix $B = A = \lambda^{k-1} I$. I suggest stopping when,

$$rcond(B) < 10 * eps$$

(Using Matlab documentation, explain what this criterion means.) Show your code works by (i) reproduce the iterates λ^0, λ^1, λ^2 in Example 27.1, and (ii) by matching one of the eigenvalues and eigenvectors of a random $20x20$ Hermitian matrix. (For (ii), assume that the built in eig() is exact.)

Solution:
Consider the following Matlab function,
Code:

```
function [lam, v] = rqi(A, v0)
% This function takes a matrix A and vector v0
% and performs rayleigh quotient iteration to produce
% an eigenvector, eigenvalue pair.

%Normalizing the input vector
v0 = v0./norm(v0);

%Initial rayleigh quotient iteration
lam = v0'*A*v0;

%Initializing count to display lambda values each iteration
count = 0;
    while (rcond(A - lam*eye(size(v0,1))) >= 10*eps)
        %^^ Set threshold to exit iterative method
        %We exit the while loop when the reciprocal of the
        %condition number for B is on the order of 10*eps.

        %Displaying lambda value each iteration.
        disp(strcat(strcat('lambda', int2str(count)),':'))
        disp(lam)
        count = count + 1;

        %Using inverse iteration to get new v0
        v0 = (A - lam*eye(size(v0,1)))\v0;
        v0 = v0./norm(v0);
```

```
        %Using v0 to compute new rayleigh quotient
        lam = v0'*A*v0;
    end

v = v0;

end
```

i Reproducing the iterates from example 27.1 we get the following,
Console:

```
>> A = [2 1 1;
        1 3 1;
        1 1 4];

>> v0 = [1 1 1]'./sqrt(3)

v0 =
    0.577350269189626
    0.577350269189626
    0.577350269189626

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
>> [lam, v] = rqi(A, v0)
lambda0:
    4.999999999999999

lambda1:
    5.213114754098361

lambda2:
    5.214319743184032

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
lam =

    5.214319743377535

v =

    0.397112549787007
    0.520657368439594
    0.755789340683778
```

ii Matching a rqi() computed eigenvector, eigenvalue pair from a random $20x20$ hermitian matrix with eig() we get,
Console:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Generate Random 20x20 Hermitian Matrix

>> A = rand(20);
>> A = A'*A;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Apply RQI to some random vector

>> [lam1, V1] = rqi(A, rand(20,1));
lambda0:
   84.407650805741966

lambda1:
       1.073834579054231e+02

lambda2:
       1.108165520987956e+02

lambda3:
       1.108203064510933e+02


lam1 =

       1.108203064510978e+02


V1 =

    0.181862534644477
    0.193233710426479
    0.233176463025641
    0.226401828684026
    0.251699057544143
    0.183505276102680
    0.219110782343502
    0.232763932951180
    0.247942269438621
    0.183309290633680
    0.261696935009576
    0.240106649145914
    0.237944866513763
    0.230473972624308
    0.272365079667538
    0.180031747780668
    0.220641041593823
```

```
    0.197055878767066
    0.228604034627072
    0.217772564767455


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Produce eigenvectors and eigenvalues for A with eig()
>> [Vec, Val] = eig(A);
>> Val = diag(Val);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Printing out the list I saw we generated the 20th
%% eigenvector, eigenvalue pair
>> lam = Val(20)

lam =

      1.108203064510977e+02

>> V = Vec(:,20);



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Checking the error, very small as expected we could
%% likely attribute this to rounding error.
>> abs(lam - lam1)
ans =

      8.526512829121202e-14

>> norm(V - V1)

ans =

      5.933464366067994e-16
```