Homework 1

1 P1: Page 55 Problem 1

Write a function GetMatrixType(A), it should return "singular" if the matrix A is singular, "well conditioned" if the matrix is well conditioned and "ill conditioned" if the matrix is "ill conditioned". Notice that you will need to write your own LUdecomp and LUsolve code, see the requirement in italic above.

By evaluating the determinant, classify the following matrices as singular, ill conditioned, or well conditioned:

(a)

$$\mathbf{A} = \begin{bmatrix} 1.0 & 2.0 & 3.0 \\ 2.0 & 3.0 & 4.0 \\ 3.0 & 4.0 & 5.0 \end{bmatrix} \tag{1.1}$$

$$\mathbf{P} = \begin{bmatrix} 0.0 & 0.0 & 1.0 \\ 0.0 & 1.0 & 0.0 \\ 1.0 & 0.0 & 0.0 \end{bmatrix} \tag{1.2}$$

$$\mathbf{L} = \begin{bmatrix} 1.0 & 0.0 & 0.0 \\ 0.667 & 1.0 & 0.0 \\ 0.333 & 2.0 & 1.0 \end{bmatrix} \tag{1.3}$$

$$\mathbf{U} = \begin{bmatrix} 3.0 & 4.0 & 5.0 \\ 0.0 & 0.333 & 0.667 \\ 0.0 & 0.0 & 0.0 \end{bmatrix} \tag{1.4}$$

$$\mathbf{P^T} \cdot \mathbf{L} \cdot \mathbf{U} = \begin{bmatrix} 1.0 & 2.0 & 3.0 \\ 2.0 & 3.0 & 4.0 \\ 3.0 & 4.0 & 5.0 \end{bmatrix} \tag{1.5}$$

Det: -0.0

Non-singular: False

(b)

$$\mathbf{A} = \begin{bmatrix} 2.11 & -0.8 & 1.72 \\ -1.84 & 3.03 & 1.29 \\ -1.57 & 5.25 & 4.3 \end{bmatrix} \tag{1.6}$$

$$\mathbf{P} = \begin{bmatrix} 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 \\ 0.0 & 1.0 & 0.0 \end{bmatrix} \tag{1.7}$$

$$\mathbf{L} = \begin{bmatrix} 1.0 & 0.0 & 0.0 \\ -0.744 & 1.0 & 0.0 \\ -0.872 & 0.501 & 1.0 \end{bmatrix} \tag{1.8}$$

$$\mathbf{U} = \begin{bmatrix} 2.11 & -0.8 & 1.72 \\ 0.0 & 4.655 & 5.58 \\ 0.0 & 0.0 & -0.006 \end{bmatrix} \tag{1.9}$$

$$\mathbf{P^T} \cdot \mathbf{L} \cdot \mathbf{U} = \begin{bmatrix} 2.11 & -0.8 & 1.72 \\ -1.84 & 3.03 & 1.29 \\ -1.57 & 5.25 & 4.3 \end{bmatrix} \tag{1.10}$$

Det: 0.05886699999999577

Non-singular: True

Ill Conditioned (0.006996162293874287)

(c)

$$\mathbf{A} = \begin{bmatrix} 2.0 & -1.0 & 0.0 \\ -1.0 & 2.0 & -1.0 \\ 0.0 & -1.0 & 2.0 \end{bmatrix} \tag{1.11}$$

$$\mathbf{P} = \begin{bmatrix} 1.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 1.0 \end{bmatrix} \tag{1.12}$$

$$\mathbf{L} = \begin{bmatrix} 1.0 & 0.0 & 0.0 \\ -0.5 & 1.0 & 0.0 \\ 0.0 & -0.667 & 1.0 \end{bmatrix} \tag{1.13}$$

$$\mathbf{U} = \begin{bmatrix} 2.0 & -1.0 & 0.0 \\ 0.0 & 1.5 & -1.0 \\ 0.0 & 0.0 & 1.333 \end{bmatrix} \tag{1.14}$$

$$\mathbf{P^T} \cdot \mathbf{L} \cdot \mathbf{U} = \begin{bmatrix} 2.0 & -1.0 & 0.0 \\ -1.0 & 2.0 & -1.0 \\ 0.0 & -1.0 & 2.0 \end{bmatrix} \tag{1.15}$$

Det: 4.0

Non-singular: True

Well Conditioned (1.0)

(d)

$$\mathbf{A} = \begin{bmatrix} 4.0 & 3.0 & -1.0 \\ 7.0 & -2.0 & 3.0 \\ 5.0 & -18.0 & 13.0 \end{bmatrix} \quad (1.16)$$

$$\mathbf{P} = \begin{bmatrix} 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 1.0 \\ 1.0 & 0.0 & 0.0 \end{bmatrix} \quad (1.17)$$

$$\mathbf{L} = \begin{bmatrix} 1.0 & 0.0 & 0.0 \\ 0.714 & 1.0 & 0.0 \\ 0.571 & -0.25 & 1.0 \end{bmatrix} \quad (1.18)$$

$$\mathbf{U} = \begin{bmatrix} 7.0 & -2.0 & 3.0 \\ 0.0 & -16.571 & 10.857 \\ 0.0 & 0.0 & -0.0 \end{bmatrix} \quad (1.19)$$

$$\mathbf{P^T} \cdot \mathbf{L} \cdot \mathbf{U} = \begin{bmatrix} 4.0 & 3.0 & -1.0 \\ 7.0 & -2.0 & 3.0 \\ 5.0 & -18.0 & 13.0 \end{bmatrix} \quad (1.20)$$

Det: 5.151434834260727e-14

Non-singular: True

Note: Rounding error from numerical computation→**Singular**

Ill Conditioned (2.092627355609377e-15)

2 P2: Page 55 Problem 6

Solve the equations $\mathbf{Ax = b}$ by Gauss elimination

$$\mathbf{A} = \begin{bmatrix} 0.0 & 0.0 & 2.0 & 1.0 & 2.0 \\ 0.0 & 1.0 & 0.0 & 2.0 & -1.0 \\ 1.0 & 2.0 & 0.0 & -2.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & -1.0 & 1.0 \\ 0.0 & 1.0 & -1.0 & 1.0 & -1.0 \end{bmatrix} \quad (2.1)$$

$$\mathbf{B} = \begin{bmatrix} 1.0 \\ 1.0 \\ -4.0 \\ -2.0 \\ -1.0 \end{bmatrix} \quad (2.2)$$

$$\mathbf{P} = \begin{bmatrix} 0.0 & 0.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 & 0.0 \\ 1.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix} \quad (2.3)$$

$$\mathbf{X} = \begin{bmatrix} 2.0 \\ -2.0 \\ 1.0 \\ 1.0 \\ -1.0 \end{bmatrix} \quad (2.4)$$

3 P3: Page 56 Problem 10

Solve the equations $\mathbf{AX = B}$ by Doolittle's decomposition method

$$\mathbf{A} = \begin{bmatrix} 4.0 & -3.0 & 6.0 \\ 8.0 & -3.0 & 10.0 \\ -4.0 & 12.0 & -10.0 \end{bmatrix} \quad (3.1)$$

$$\mathbf{B} = \begin{bmatrix} 1.0 & 0.0 \\ 0.0 & 1.0 \\ 0.0 & 0.0 \end{bmatrix} \quad (3.2)$$

$$\mathbf{P} = \begin{bmatrix} 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 1.0 \\ 1.0 & 0.0 & 0.0 \end{bmatrix} \quad (3.3)$$

$$\mathbf{L} = \begin{bmatrix} 1.0 & 0.0 & 0.0 \\ -0.5 & 1.0 & 0.0 \\ 0.5 & -0.143 & 1.0 \end{bmatrix} \quad (3.4)$$

$$\mathbf{U} = \begin{bmatrix} 8.0 & -3.0 & 10.0 \\ 0.0 & 10.5 & -5.0 \\ 0.0 & 0.0 & 0.286 \end{bmatrix} \quad (3.5)$$

$$\mathbf{P^T} \cdot \mathbf{L} \cdot \mathbf{U} = \begin{bmatrix} 4.0 & -3.0 & 6.0 \\ 8.0 & -3.0 & 10.0 \\ -4.0 & 12.0 & -10.0 \end{bmatrix} \quad (3.6)$$

$$\mathbf{X} = \begin{bmatrix} -3.75 & 1.75 \\ 1.667 & -0.667 \\ 3.5 & -1.5 \end{bmatrix} \tag{3.7}$$

4 P4: Page 56 Problem 14

$$\mathbf{A} = \begin{bmatrix} 2.0 & -1.0 & 0.0 \\ -1.0 & 2.0 & -1.0 \\ 0.0 & -1.0 & 2.0 \end{bmatrix} \tag{4.1}$$

$$\mathbf{B} = \begin{bmatrix} 1.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 1.0 \end{bmatrix} \tag{4.2}$$

$$\mathbf{X} = \begin{bmatrix} 0.75 & 0.5 & 0.25 \\ 0.5 & 1.0 & 0.5 \\ 0.25 & 0.5 & 0.75 \end{bmatrix} \tag{4.3}$$

$$\mathbf{X} \cdot \mathbf{A} = \begin{bmatrix} 1.0 & 0.0 & -0.0 \\ 0.0 & 1.0 & -0.0 \\ 0.0 & 0.0 & 1.0 \end{bmatrix} \tag{4.4}$$

5 P5: Page 57 Problem 15

Write a code that calls a function that returns the largest n within six significant figures of the solution. That is, you compute the largest n for which the 2-norm between the computed solution and exact solution is still $< 1. \times 10^{-6}$.

$||X_{exact} - X(7)|| = 6.258704933008605\text{e-}08$

Nmax = 7

Error @ 8 = 2.0576776346861646e-06

Error @ 7 = 6.258704933008605e-08

Error @ 6 = 2.1167878221207363e-09

$$\mathbf{H_A} = \begin{bmatrix} 1.0 & 0.5 & 0.33333 & 0.25 & 0.2 & 0.16667 & 0.14286 \\ 0.5 & 0.33333 & 0.25 & 0.2 & 0.16667 & 0.14286 & 0.125 \\ 0.33333 & 0.25 & 0.2 & 0.16667 & 0.14286 & 0.125 & 0.11111 \\ 0.25 & 0.2 & 0.16667 & 0.14286 & 0.125 & 0.11111 & 0.1 \\ 0.2 & 0.16667 & 0.14286 & 0.125 & 0.11111 & 0.1 & 0.09091 \\ 0.16667 & 0.14286 & 0.125 & 0.11111 & 0.1 & 0.09091 & 0.08333 \\ 0.14286 & 0.125 & 0.11111 & 0.1 & 0.09091 & 0.08333 & 0.07692 \end{bmatrix} \tag{5.1}$$

$$\mathbf{H_B} = \begin{bmatrix} 2.59286 \\ 1.71786 \\ 1.32897 \\ 1.09563 \\ 0.93654 \\ 0.81988 \\ 0.73013 \end{bmatrix} \tag{5.2}$$

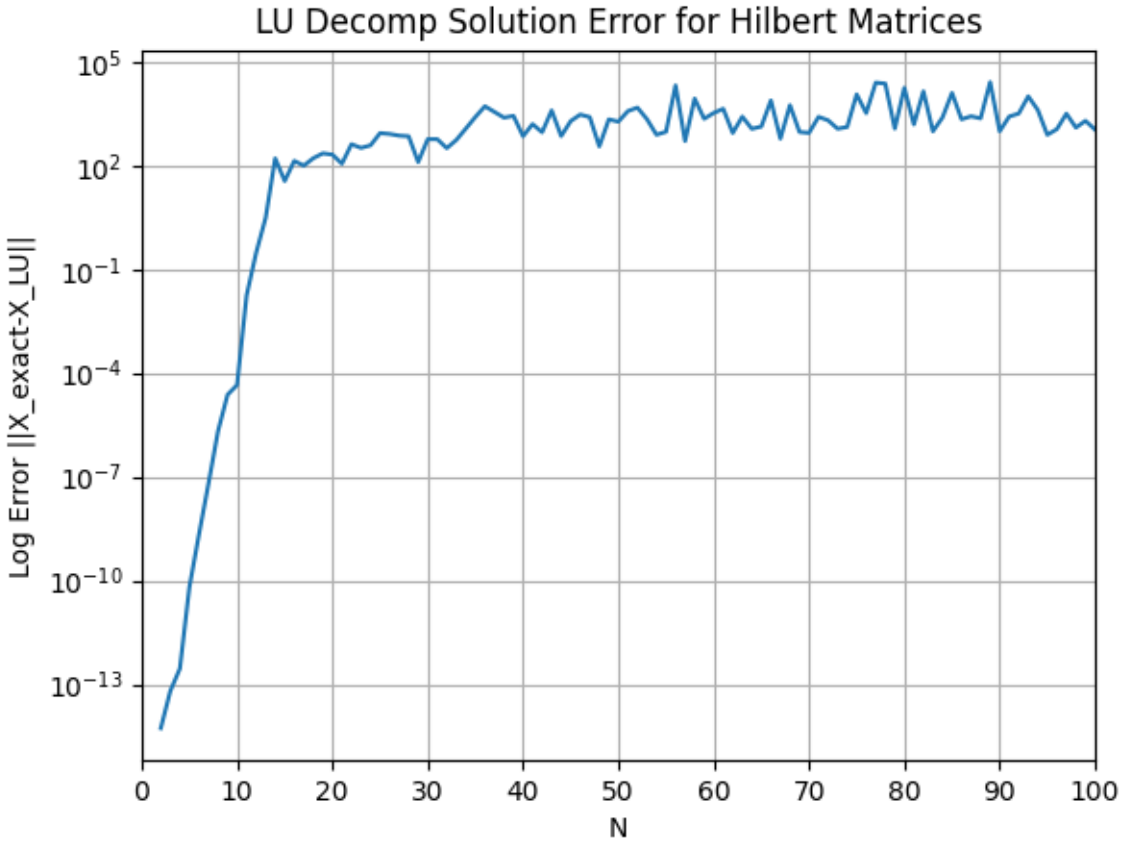

Figure 1: LU Decomposition Solution Errors for the Hilbert matrix using a log scale.

$$\mathbf{X_{LU}} = \begin{bmatrix} 1.0 \\ 1.0000000006 \\ 0.9999999941 \\ 1.0000000231 \\ 0.9999999573 \\ 1.0000000371 \\ 0.9999999877 \end{bmatrix} \tag{5.3}$$

6 P6: Write a couple paragraphs in which you explain what grade you will aim for and how you plan to achieve your goal. What study strategies and timeline will you put in place?

I plan on achieving an A in this class. I am generally a highly motivated student so I complete all of my work usually several days ahead of the due date. However, this class does seem like a significant amount of work so if I am struggling with time management I will put less work into this class and

aim for an A-.

This semester I plan on trying to read the textbook for each of my classes before lecture as I have less time than previous semesters. I think this will help me complete homework and case studies at a faster pace which will help me with time management. Another (short-term) strategy I am planning on implementing is going through a numpy tutorial so I have a better grasp on the built-in functions.

As for a timeline, I am not certain yet. I spent more time than I'd like to admit on this homework and I believe that it was because I don't know the numpy library yet, and I didn't read the textbook which had significant portions of code which I had to reinvent. In the future I will use the textbook more which I hope will limit the work in this class to less than six hours per week.

Homework 1

hw1.py

```
import numpy as np
from math import *
import matplotlib
import matplotlib.pyplot as plt
from np2latex import *

#Return the determinant of A
def det(A):
    nrows = len(A)
    ncols = len(A[0])
    if nrows != ncols:
        print("Matrix is not square! ({} x {}".format(nrows, \
            ncols))
        return None
    #End case for recursion
    if nrows == 2:
        return A[0][0]*A[1][1]-A[0][1]*A[1][0]
    else:
        s=0
        for k in range(len(A)):
            s+=(-1)**k * A[0][k]*det(np.delete(np.delete(A, 0, \
                0), k, 1))
        return s

#Euclidean norm (2norm) of a matrix
def norm(A):
    return sqrt(np.sum(A**2))

def swapRows(a, i, j):
    #for column vecs disguised as row vecs
    if len(a.shape) == 1:
        a[i],a[j] = a[j],a[i]
    #for matrices and col vecs
    else:
        a[[i,j],:] = a[[j,i],:]

def swapCols(a, i, j):
    a[:,[i,j]] = a[:,[j,i]]

def transpose(A):
    #row to col vec
```

```
    if len(A.shape) == 1:
        return A.copy().reshape(A.shape[0],1)
    nr,nc = A.shape
    AT = np.zeros([nc,nr])
    for i in range(nr):
        for j in range(nc):
            #interchange indices
            AT[j,i]=A[i,j]
    return AT

def LUpivot(A):
    n = len(A)
    A = A.copy()
    P = np.identity(n)
    for col in range(n):
        #find largest element in the column and
        #swap its row with cur row
        #prevent swapping back by col: limit on row
        k = np.argmax(np.abs(A[col:, col]))+col
        swapRows(P, k, col)
        swapRows(A, k, col)
    return P

def LUdecomp(A):
    n = len(A)
    A = A.copy()
    P = LUpivot(A)
    A = np.dot(P,A)

    # Store U & L in A -> A = L+U-I
    for row in range(n-1):#Operating row
        for row2 in range(row+1, n):#Row thats being operated on
            if A[row, row] == 0:
                print("Diagonal contains 0 but col is not \
                    reduced")
                return None
            elif A[row2, row] != 0:
                #find L scaling factor
                L_term = A[row2,row]/A[row,row]
                # A = A - L * U; Iterate over row to add on "- L * U"
                # terms for each row2
                A[row2,(row+1):] = A[row2,(row+1):] - L_term*\
                    A[row,(row+1):]
                #store U in upper right + diag
                #store L in lower left
```

```
                A[row2, row] = L_term
    return (A, P)

def LUexpand(LU):
    n = len(LU)
    LU = LU.copy()
    L = np.identity(n)
    for col in range(n):
        for row in range(col+1, n):
            #Put L from LU into new matrix
            L[row,col]=LU[row,col]
            #remove L from LU
            LU[row,col]=0
    return (L, LU)

def LUsolve(A, B, LU, P):
    n = len(A)
    LU = LU.copy()
    P.copy()
    #make sure B is col vec
    if len(B.shape) == 1:
        B = transpose(B)
    nBrows, nBcols = B.shape
    #pivot B
    B=np.dot(P,B)
    if np.isin(0, LU.diagonal()):
        print("diagonal has a 0, can't solve")
    for k in range(1, n):
        #xb for multiple RHS x
        for xB in range(nBcols):
            #Solve U|B to make new col vec Y
            B[k,xB] = B[k,xB]-np.dot(LU[k,0:k],B[0:k,xB])
    B[n-1,:] = B[n-1,:]/LU[n-1,n-1]
    for k in range(n-2,-1,-1):
        for xB in range(nBcols):
            #solve L|Y for X
            B[k,xB] = (B[k,xB]-np.dot(LU[k,k+1:n],\
                B[k+1:n,xB]))/LU[k,k]
    return B

def LUinverse(A, LU, P):
    #AX=I-> A^-1*A*X=A^-1*I -> X=A^-1
    return LUsolve(A, np.identity(len(A)), LU, P)

def LUnonsingular(A, LU, P):
```

```
    LU = LU.copy()
    P.copy()
    #find LU determinant
    d = float(det(P)*LU.diagonal().prod())
    if d == 0:#Singular
        return (d, False, False)
    else:#Nonsignular
        #Test conditioning
        cond = abs(d/norm(A))#Well conditioned
        return (d, True, cond)

def gausElim(A, B):
    n = len(A)
    P = LUpivot(A.copy())
    #pivot A so no 0 on diag
    A = np.dot(P, A)
    #Make sure B is col vec
    if len(B.shape) == 1:
        B = transpose(B)
    nBrows, nBcols = B.shape
    B=np.dot(P,B)
    for row in range(n-1):#Row doing the acting
        for row2 in range(row+1, n):#Row being acted on
            if A[row2, row] != 0:
                scale = A[row2, row]/A[row, row]
                #Standard gauss elim
                A[row2, row+1:n] = A[row2, row+1:] - \
                    scale*A[row,row+1:]
                #Do operations on B too!
                B[row2, :] = B[row2, :] - scale*B[row, :]
    for row in range(n-1, -1, -1):
        for xB in range(nBcols):#xB for multi RHS
            #Standard backwards substitution
            B[row,xB] = (B[row,xB]-np.dot(B[row+1:,xB],\
                A[row, row+1:]))/A[row,row]
    return B

#makes hilbert A matrix
def hilbertA(n):
    A = np.zeros([n,n])
    for i in range(n):
        for j in range(i, n):
            A[i,j] = 1/(i+j+1)
            #symmetric on diag
            A[j,i]= A[i,j]
```

```
    return A

#makes hilbert B vector
def hilbertB(A):
    return np.sum(A, 0)

def hilbertSolve(maxn=100,tol=1.0e-6):
    #make large matrix once
    A = hilbertA(maxn)
    B = hilbertB(A)
    #save errors for later plotting
    ns, errors = [], []
    n_max=0
    print("Hilbert Matrix Generated")
    for n in range(maxn, 1, -1):
        (LU,P) = LUdecomp(A)
        X = LUsolve(A, B, LU, P)
        #error = sqrt(X_exact-X_LU)
        error = norm(1-X)
        errors.append(error)
        ns.append(n)
        if error<tol and n>n_max:#find n that works
            print("||X_g-X({})|| = ".format(n), error)
            n_max=n
            X_max=X
        #remove elements as needed -> much faster!
        B = B[:-1] - A[-1, :-1]
        A = A[:-1,:-1]
    return (ns, errors, n_max, X_max)

if __name__ == "__main__":
    np.set_printoptions(precision=5,suppress=True)
    Q1a_A = np.array([[1, 2, 3],\
                      [2, 3, 4],\
                      [3, 4, 5]], dtype=float)
    Q1b_A = np.array([[ 2.11, -0.80, 1.72],\
                      [-1.84,  3.03, 1.29],\
                      [-1.57,  5.25, 4.30]], dtype=float)
    Q1c_A = np.array([[ 2, -1,  0],\
                      [-1,  2, -1],\
                      [ 0, -1,  2]], dtype=float)
    Q1d_A = np.array([[4,   3, -1],\
                 [7,  -2,  3],\
                 [5, -18, 13]], dtype=float)
```

```
# Using my functions
print("-----------------------------------")
for A, part in zip([Q1a_A,Q1b_A,Q1c_A,Q1d_A],\
        ["a","b","c","d"]):
    print("Question 1 P 55",part)
    #print("A = \n",A)
    np2latex(A,"A")
    (LU, P) = LUdecomp(A)
    #print("L + U - I = \n",LU)
    np2latex(P,'P')#print("P = \n",P)

    (L,U)=LUexpand(LU)
    np2latex(L,'L')#print("L = \n",L)
    np2latex(U,'U')#print("U = \n",U)
    #print("P^-1 * L * U =\n", np.dot(transpose(P),\
    # np.dot(L,U)))
    #Show that LU works!
    np2latex(np.dot(transpose(P),np.dot(L,U)),\
        'P^T\cdot L\cdot U')
    (d, nonsing,cond) = LUnonsingular(A, LU, P)
    print("Det: ",d)
    print("Non-singular: ",nonsing)
    if nonsing is True:
        print(["Ill","Well"][cond>=0.01],\
            "Conditioned (",cond,")")

    print("-----------------------------------")

print("Question 6 P 55")
Q6_A = np.array([[0, 0, 2, 1, 2],\
                 [0, 1, 0, 2, -1],\
                 [1, 2, 0, -2, 0],\
                 [0, 0, 0, -1, 1],
                 [0, 1, -1, 1, -1]], dtype=float)
Q6_B = np.array([1, 1, -4, -2, -1], dtype=float).reshape(5,1)
np2latex(Q6_A,'A')#print("A =\n", Q6_A)
np2latex(Q6_B,'B')#print("B =\n", Q6_B)
np2latex(LUpivot(Q6_A),'P')#print("P =\n", LUpivot(Q6_A))
Q6_X = gausElim(Q6_A,Q6_B)

np2latex(Q6_X,'X')#print("X =\n", Q6_X)
print("-----------------------------------")

print("Question 10 P 56")
Q10_A = np.array([[ 4, -3,   6],\
```

```
                [ 8, -3,  10],\
                [-4, 12, -10]], dtype=float)
Q10_B = transpose(np.array([[1, 0, 0],[0, 1, 0]], \
    dtype=float))

(Q10_LU,Q10_P) = LUdecomp(Q10_A)
#print("L + U - I = \n",Q10_LU)

(Q10_L,Q10_U)=LUexpand(Q10_LU)
Q10_X = LUsolve(Q10_A, Q10_B, Q10_LU, Q10_P)

np2latex(Q10_A,'A')#print("A = \n",Q10_A)
np2latex(Q10_B,'B')#print("\nB = \n",Q10_B)
np2latex(Q10_P,'P')#print("P = \n",Q10_P)
np2latex(Q10_L,'L')#print("L = \n",Q10_L)
np2latex(Q10_U,'U')#print("U = \n",Q10_U)
np2latex(np.dot(transpose(Q10_P),np.dot(Q10_L,Q10_U)),\
    'P^T\cdot L \cdot U')
#print("P^-1 * L * U =\n", np.dot(transpose(Q10_P),\
# np.dot(Q10_L,Q10_U)))
np2latex(Q10_X,'X')#print("X = \n",Q10_X)

print("-----------------------------------")

print("Question 14 P 56")
Q14_A = np.array([[ 2,-1, 0],\
                                    [-1, 2,-1],\
                                    [ 0,-1, 2]],dtype=float)
Q14_B = np.identity(3).astype(np.float)

Q14_X = gausElim(Q14_A,Q14_B)
np2latex(Q14_A,'A')#print("A =\n", Q14_A)
np2latex(Q14_B,'B')#print("B =\n", Q14_B)
np2latex(Q14_X,'X')#print("X =\n", Q14_X)
#Show that it works
np2latex(np.dot(Q14_X,Q14_A),'X\cdot A')
#print("X*A =\n", np.dot(Q14_X,Q14_A))
print("-----------------------------------")

print("Question 15 P 57")
(ns, errors, nmax, xmax) = hilbertSolve(100)
print("Nmax = ",nmax)
print("Error @ {} = ".format(nmax+1),errors[100-(nmax+1)])
print("Error @ {} = ".format(nmax),errors[100-nmax])
print("Error @ {} = ".format(nmax-1),errors[100-(nmax-1)])
```

```
np2latex(xmax, 'X', 10)#print("Xmax =\n",xmax)
HA = hilbertA(nmax)
np2latex(HA, "H_A",rounding=5)
np2latex(transpose(hilbertB(HA)), "H_B",rounding=5)

#Plot results to show that there isn't a parabolic behavior
#where nmax actually equals 2000
plt.plot(ns, errors)
plt.yscale("log")

plt.xlabel('N')
plt.ylabel('Log Error ||X_exact-X_LU||')
plt.title('LU Decomp Solution Error for Hilbert Matrices')
plt.grid(True)
plt.xlim(0, 100)
plt.xticks(np.arange(0, 101,10))
plt.savefig("figures/hilbert_errors.png")
plt.show()

print("-----------------------------------")
```