Homework 3

Ramaseshan Parthasarathy, Saurabh Prasad
11/01/17

Problem 1

1. The steps to show that the inequality holds is shown below:

   $\|A\| = max_{x \neq 0} \frac{\|Ax\|}{\|x\|} \geq \frac{\|Ay\|}{\|y\|} \Rightarrow \|Ay\| = \|A\| \|y\|, \; y \in \mathbb{R}^n (y \neq 0)$
   $\|Ax\| \leq \|A\| \|x\| \Rightarrow \left\|T^k x\right\| \leq \left\|T^k\right\| \|x\| \Rightarrow \left\|T^k\right\| \leq \|T\|^k$
   $\therefore \left\|\mathbf{T^k x}\right\| \leq \|\mathbf{T}\|^{\mathbf{k}} \|\mathbf{x}\|$

2. The proof is rather straight forward:

   $$Ax = b$$
   $$(D - L - U)x \; = \; b$$
   $$Dx = (L + U)x \; + \; b$$
   $$x = D^{-1}(L + U)x \; + \; D^{-1}b$$
   $$x^{(k)} = D^{-1}(L + U)x^{(k-1)} \; + \; D^{-1}b$$
   $$x^{(k+1)} = D^{-1}(L + U)x^{(k)} \; + \; D^{-1}b$$
   $$\mathbf{x^{(k+1)}} = \mathbf{T x^{(k)}} \; + \; \mathbf{c}$$

3. Using the previous part, the following can be shown:

   $$x^* = Tx^* + c$$
   $$x^{(k+1)} \; - \; x^* = Tx^{(k)} + c - (Tx^* + c)$$
   $$e^{(k+1)} = T(x^{(k)} - x^*)$$
   $$\mathbf{e^{(k+1)}} = \mathbf{T e^{(k)}}$$

4. Note that the fact that A is *diagonal dominant*, meaning every diagonal entry is greater than the sum of all other entries in its row

   $\|T\|_\infty = \underset{i}{max} \sum_{i \neq j}^{n} \frac{a_{ij}}{a_{ii}} < 1$ `since` $|a_{ii}| > \sum_{i \neq j}^{n} |a_{ij}|$ `and` $D_{ij}^{-1} = \frac{1}{a_{ii}}$

5. Combining steps 1 - 4 would use the following steps to arrive at the final inequality:

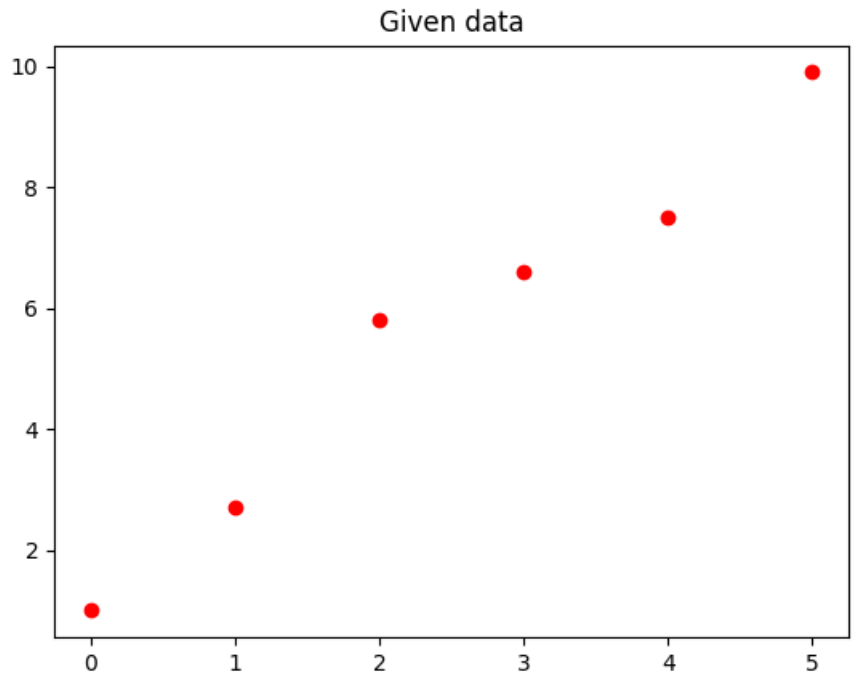
   $e^{(k+1)} = x^{(k+1)} - x^*$ `and` $Tx^k = x^{(k+1)}$ (1)
   $\left\|x^{(k+1)} - x^*\right\| = \left\|Tx^k - Tx^*\right\| = \left\|T(x^k - x^*)\right\| \leq \|T\| \left\|e^k\right\| (2)$
   $\left\|T^k\right\| \leq \|T\|^k \; (3)$
   `knowing (3), apply equation (2) k times to get:` $\left\|e^{(k)}\right\|_\infty \leq \|T\|_\infty^k \left\|e^{(0)}\right\|_\infty$

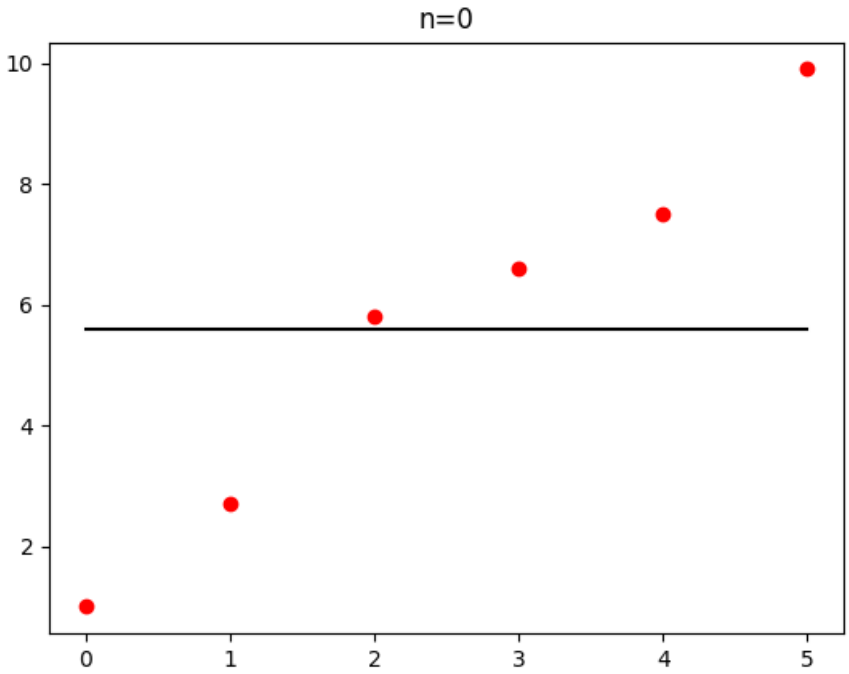
Problem 2

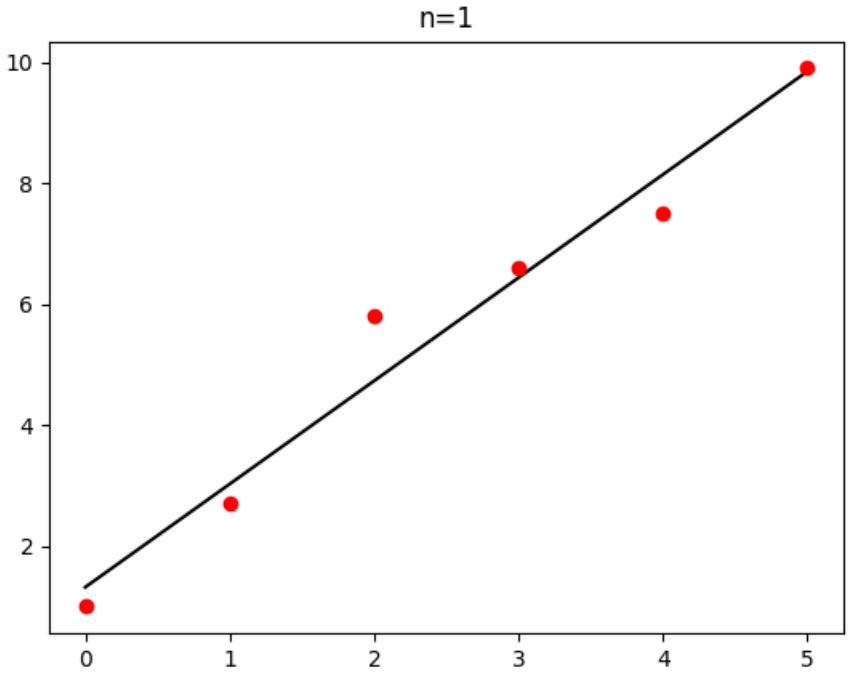
```
import numpy as np
import matplotlib.pyplot as plt

def Gaussian_Elimination(A):
    m=A.shape[0]
    n=A.shape[1]
    if(m!=n):
        print( 'Matrix is not square!');
        return
    for k in range(0,n-1):
        if A[k,k] == 0:
            return
        for i in range(k+1,n):
            A[i,k]=A[i,k]/A[k,k]
        for j in range(k+1,n):
            for i in range(k+1,n):
                A[i,j]-=A[i,k]*A[k,j]

def Back_Substitution(A,b,x):
    m=A.shape[0]
    n=A.shape[1]
    if(m!=n):
        print( 'Matrix is not square!')
        return
    for j in range(n-1,-1,-1):
        if A[j,j] == 0:
            print( 'Matrix is singular!')
            return          # matrix is singular
        x[j] = b[j]/A[j,j]
        for i in range(0,j):
            b[i] = b[i] - A[i,j]*x[j]

def Forward_Substitution(A,b,x):
    m=A.shape[0]
    n=A.shape[1]
    if(m!=n):
        print( 'Matrix is not square!')
        return
    for j in range(0,n):
        if A[j,j] == 0:
            print( 'Matrix is singular!')
            return          # matrix is singular
        x[j] = b[j]/A[j,j]
        for i in range(j+1,n):
            b[i] = b[i] - A[i,j]*x[j]

def main():
    t=np.array([0.,1.,2.,3.,4.,5.]);
    y=np.array([1.,2.7,5.8,6.6,7.5,9.9]);
    plt.figure(1);
    plt.plot(t,y,'ro')
    plt.title('Given data');
    plt.draw()

    for n in range(6):

        A=np.zeros((6,n+1));

        for r in range(6):
                for c in range(A.shape[1]):
                        A[r,c]=t[r]**(A.shape[1]-1-c);

        x=np.zeros(A.shape[1]);
        y1=np.zeros(A.shape[0]);
        AtA=A.transpose().dot(A)
```

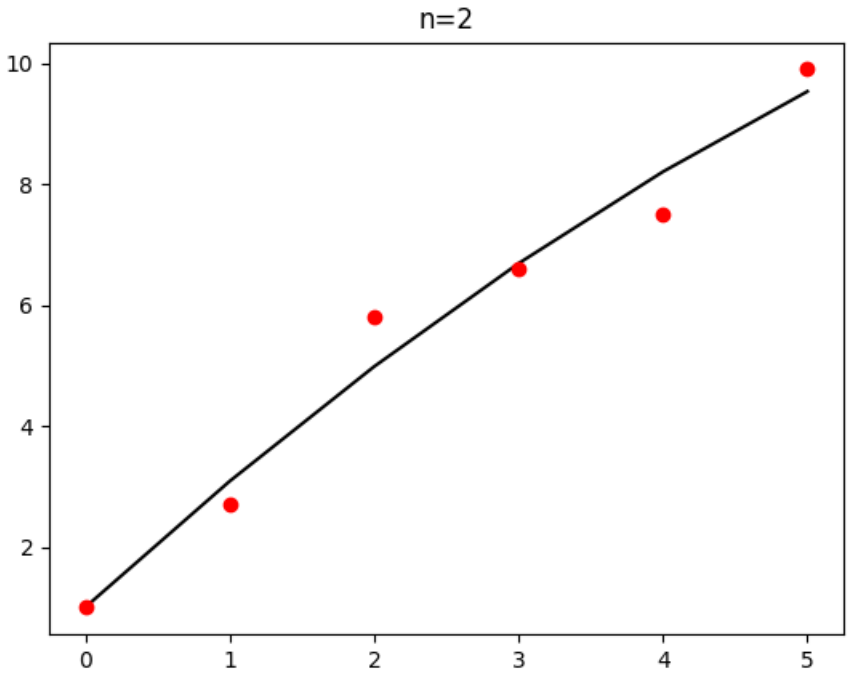
```
        sol=A.transpose().dot(y)

        Gaussian_Elimination(AtA)
        L=np.identity(AtA.shape[0])

        for i in range(1,L.shape[0]):
                for j in range(0,i):
                        L[i,j]=AtA[i,j];

        U=np.zeros((AtA.shape[0],AtA.shape[1]))

        for i in range(U.shape[0]):
                for j in range(U.shape[1]-1,i-1,-1):
                        U[i,j]=AtA[i,j];

        Forward_Substitution(L,sol,y1)
        Back_Substitution(U,y1,x)
        x=np.array(x)

        print('n:',n)
        print(x)

        plt.figure(n+2);
        plt.plot(t,A.dot(x),'k',t,y,'ro')
        plt.title('n='+str(n));
        plt.draw()
    plt.show()

if __name__ == "__main__":
    main()
```

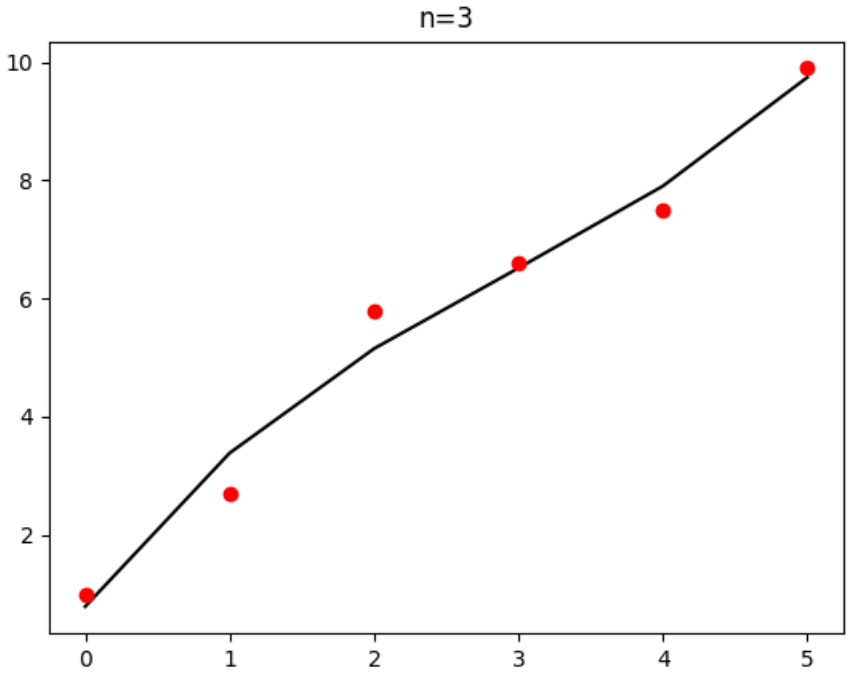
The output plot(s) of the above program have been provided for convenience:

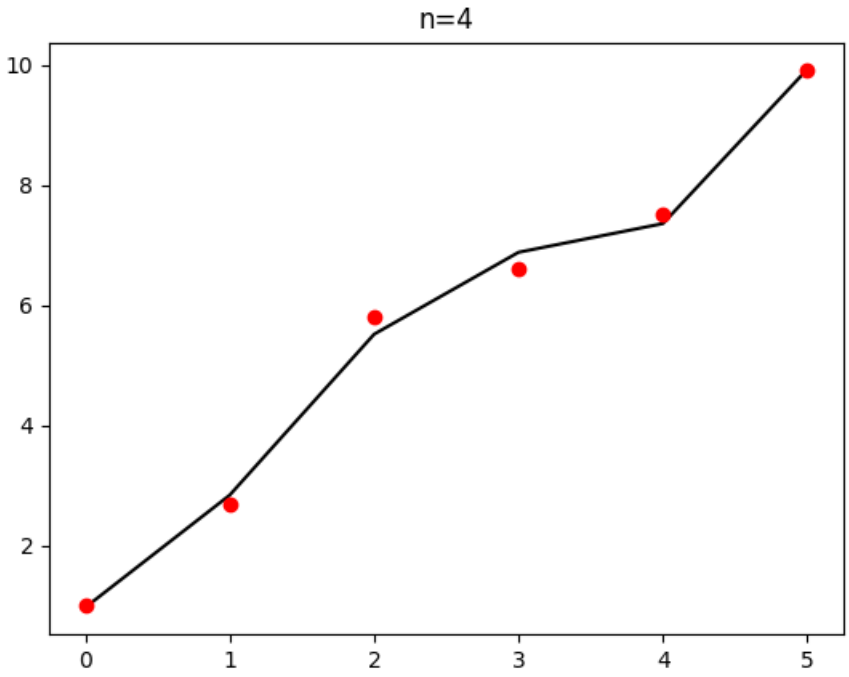


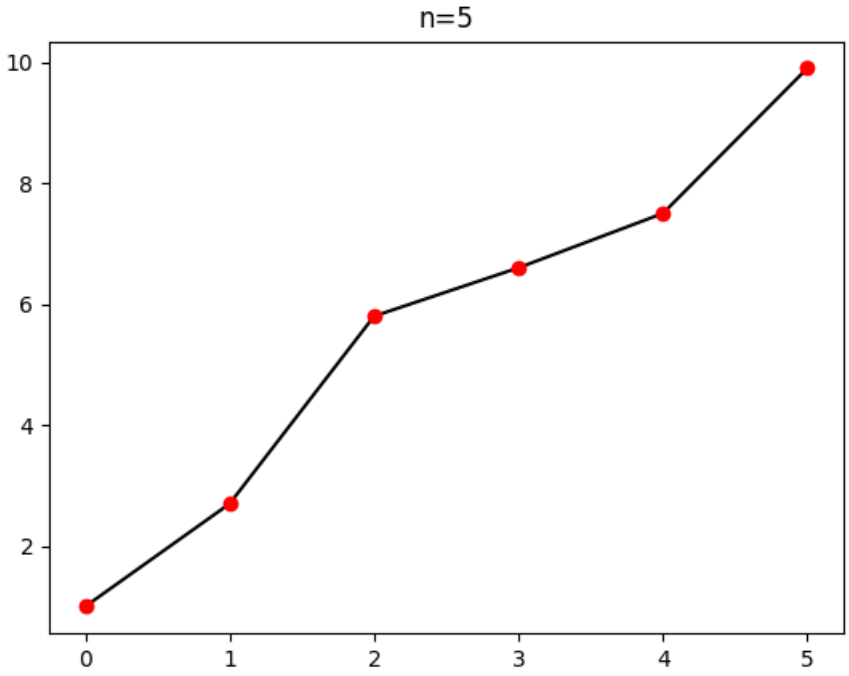

n=0
10
8
6
4
2
0
1
2
3
4
5


n=1
10
8
6
4
2
0
1
2
3
4
5


n=2
10
8
6
4
2
0
1
2
3
4
5


n=3
10
8
6
4
2
0
1
2
3
4
5





From the plots, it can, of course, be seen that polynomial with degree n = 5 displays the best trend for the data.

Problem 3

```
import numpy as np

A=np.matrix([[.16,.1],[.17,.11],[2.02,1.29]])
b=np.matrix([[.26],[.28],[3.31]])

def Gaussian_Elimination(A):
    m=A.shape[0]
    n=A.shape[1]
    if(m!=n):
```

```
        print( 'Matrix is not square!');
        return
    for k in range(0,n-1):
        if A[k,k] == 0:
            return
        for i in range(k+1,n):
            A[i,k]=A[i,k]/A[k,k]
        for j in range(k+1,n):
            for i in range(k+1,n):
                A[i,j]-=A[i,k]*A[k,j]

def Back_Substitution(A,b,x):
    m=A.shape[0]
    n=A.shape[1]
    if(m!=n):
        print( 'Matrix is not square!')
        return
    for j in range(n-1,-1,-1):
        if A[j,j] == 0:
            print( 'Matrix is singular!')
            return          # matrix is singular
        x[j] = b[j]/A[j,j]
        for i in range(0,j):
            b[i] = b[i] - A[i,j]*x[j]

def Forward_Substitution(A,b,x):
    m=A.shape[0]
    n=A.shape[1]
    if(m!=n):
        print( 'Matrix is not square!')
        return
    for j in range(0,n):
        if A[j,j] == 0:
            print( 'Matrix is singular!')
            return          # matrix is singular
        x[j] = b[j]/A[j,j]
        for i in range(j+1,n):
            b[i] = b[i] - A[i,j]*x[j]

AtA=A.transpose().dot(A)
Atb=A.transpose().dot(b)

Gaussian_Elimination(AtA)

L=np.identity(AtA.shape[0])

for i in range(1,L.shape[0]):
        for j in range(0,i):
                L[i,j]=AtA[i,j];

U=np.zeros((AtA.shape[0],AtA.shape[1]))

for i in range(U.shape[0]):
        for j in range(U.shape[1]-1,i-1,-1):
                U[i,j]=AtA[i,j];

AtbVector=np.array(Atb[:,0])
y=np.zeros(A.shape[1])
x=np.zeros(A.shape[1])
Forward_Substitution(L,AtbVector,y)
Back_Substitution(U,y,x)
print('x:')
print(x)

#Part B
b=np.matrix([[.27],[.25],[3.33]])
AtA=A.transpose().dot(A)
Atb=A.transpose().dot(b)
```

```
Gaussian_Elimination(AtA)

L=np.identity(AtA.shape[0])

for i in range(1,L.shape[0]):
        for j in range(0,i):
                L[i,j]=AtA[i,j];

U=np.zeros((AtA.shape[0],AtA.shape[1]))

for i in range(U.shape[0]):
        for j in range(U.shape[1]-1,i-1,-1):
                U[i,j]=AtA[i,j];

AtbVector=np.array(Atb[:,0])
y=np.zeros(A.shape[1])
x=np.zeros(A.shape[1])
Forward_Substitution(L,AtbVector,y)
Back_Substitution(U,y,x)
print('Part B x:')
print(x)

print('K(A^T*A):')
print(np.linalg.cond(A.transpose().dot(A)))
```

The solutions to the systems in part a and b have been given below:

```
rpartha@rpartha-HP-Notebook:~/Documents/Github/Numerical_Analysis/hw-3$ python3 least_squares.py
x:
[ 1.  1.]
Part B x:
[ 7.00888731 -8.39566299]
K(A^T*A):
1631924.81894
1631924.81894
1204591.14638
1204591.14638
```

The output shows vast difference in sooluttions for small variance in b vector. This could likely be due to the vast difference in the condition numbers when using the L_2 and L_1 = L_{∞} norm. The matrices in either case appear to be well conditioned.

Problem 4

1. The matrix *A* in question is shown below. We can prove that *A* is singular by proving that *A* is not invertible since singular matrices, by definition, do not have inverses.

   $$A = \begin{bmatrix} 0.1 & 0.2 & 0.3 \\ 0.4 & 0.5 & 0.6 \\ 0.7 & 0.8 & 0.9 \end{bmatrix}$$

   The determinant can be computed by expanding from the first row:

   $$0.1(0.45 - 0.48) - 0.2(0.36 - 0.42) + 0.3(0.32 - 0.45) = 0$$

   Knowing that *A* is singular, we can attempt to solve the system *Ax* = *b* below by first setting up the augmented matrix *A**:

   $$A^* = [A|b] = \left[\begin{array}{ccc|c} 0.1 & 0.2 & 0.3 & 0.1 \\ 0.4 & 0.5 & 0.6 & 0.3 \\ 0.7 & 0.8 & 0.9 & 0.5 \end{array}\right]$$

   The following steps outline the row operations that can be performed on this matrix:

$$(1)\ \left[\begin{array}{ccc|c} 0.1 & 0.2 & 0.3 & 0.1 \\ 0.4 & 0.5 & 0.6 & 0.3 \\ 0.7 & 0.8 & 0.9 & 0.5 \end{array}\right] \xrightarrow[R_3 \to R_3 - 7R_1]{R_2 \to R_2 - 4R_1} \left[\begin{array}{ccc|c} 0.1 & 0.2 & 0.3 & 0.1 \\ 0 & -0.3 & -0.6 & -0.1 \\ 0 & -0.6 & -1.2 & -0.2 \end{array}\right]$$

$$(2)\ \left[\begin{array}{ccc|c} 0.1 & 0.2 & 0.3 & 0.1 \\ 0 & -0.3 & -0.6 & -0.1 \\ 0 & -0.6 & -1.2 & -0.2 \end{array}\right] \xrightarrow{R_3 \to R_3 - 2R_2} \left[\begin{array}{ccc|c} 0.1 & 0.2 & 0.3 & 0.1 \\ 0 & -0.3 & -0.6 & -0.1 \\ 0 & 0 & 0 & 0 \end{array}\right]$$

We can conclude, based upon the zero in the last column of *A**, that *A* and *b* are of the same rank. That tells us that the matrix is consistent and has an infinite amount of solutions. We can therefore arrive at a solution by picking an arbitrary x_3 and performing backward substitution. The result is as follows:

$$x = \begin{bmatrix} 1/3 \\ 1/3 \\ 0 \end{bmatrix} + x_3 \begin{bmatrix} 1 \\ -2 \\ 1 \end{bmatrix}$$

2. Using Gaussian Elimination with partial pivoting using exact arithmetic, the process would fail at the second iteration, at which point there are zero entries in the last row of *A**. This process was briefly outlined in the previous part.

3. The code below computes the solution to the matrix as well as the condition number:

```
import numpy as np
from numpy import linalg as linalg

def gauss_elim(A):
    m=A.shape[0]
    n=A.shape[1]
    U = np.zeros((m,n))
    L = np.zeros((m,n))

    if(m!=n):
        print('Not Square Matrix')
        return

    for k in range(0, n-1):
        if A[k,k] == 0:
            return
        for i in range(k+1, n):
            A[i,k] = A[i,k] / A[k,k]
        for j in range (k+1, n):
            for i in range (k+1, n):
                A[i,j] -= A[i,k] * A[k,j]

    L = np.tril(A,0)
    for kk in range(n):
        for i in range(kk+1, n):
            L[kk,kk] = 1
    L[kk,kk]=1

    U = np.triu(A,0)

    return (L,U)

def forward_sub(A,b,x):
    m=A.shape[0]
    n=A.shape[1]
    if(m!=n):
        print 'Matrix is not square!'
        return
    for j in range(0,n):
        #if A[j,j] == 0:
        #       print 'Matrix is singular!'
        #       return          # matrix is singular
        x[j] = b[j]/A[j,j]
        for i in range(j+1,n):
            b[i] = b[i] - A[i,j]*x[j]
```

```
def back_sub(A,b,x):
    m=A.shape[0]
    n=A.shape[1]
    if(m!=n):
        print 'Matrix is not square!'
        return
    for j in range(n-1,-1,-1):
        #if A[j,j] == 0:
        #       print 'Matrix is singular!'
        #       return          # matrix is singular
        x[j] = b[j]/A[j,j]
        for i in range(0,j):
            b[i] = b[i] - A[i,j]*x[j]


def main():
    A = np.matrix([[0.1,0.2,0.3],[0.4,0.5,0.6],[0.7,0.8,0.9]])
    A_inv = np.linalg.inv(A)
    b = np.array([0.1,0.3,0.5])

    k_cond = np.linalg.cond(A)

    x = np.zeros(3)
    y = np.zeros(3)

    L,U = gauss_elim(A)
    forward_sub(L,b,y)
    back_sub(U,y,x)

    print (x)
    print 'condition number: ', k_cond
    print(np.finfo(float).eps)

if __name__ == "__main__":
    main()
```

The output of the above code is as follows:

```
rpartha@rpartha-HP-Notebook:~/Documents/Github/Numerical_Analysis/hw-3$ python solution.py
[ 0.08333333  0.83333333 -0.25      ]
condition number:  2.11189683358e+16
2.22044604925e-16
```

What's interesting to point out is that the solution does match the description answered in part 1. Our small calculations indicated that the system could have one of infinite solutions. The computed solution gave us one such solution in the form of a 3x1 vector and a condition number of 2.11189683358e+16 was estimated. It should be duely noted that machine epislon is typically on the order of 10^{-16} (which can easily be found using np.finfo(float).eps), where as our computation resulted in an order of 10^{+16}.