In [51]:
```
#IMPORT
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
%matplotlib inline

from ipywidgets import interact, interactive, fixed, interact_manual
import ipywidgets as widgets

## Set a seed for the random number generator
np.random.seed(100)
```

The follwoign code was taken from the notes, it is a function that is used by problems 1-3. it has been modified to inculde a count of interchanges

In [52]:
```
def GE_rsp(A):

    c=0 # count of how many interchanges

    m,n = A.shape

    # Scaling vector: absolute maximum elements of each row
    s = np.max(np.abs(A), axis=1)

    # The initial ordering of rows
    p = list(range(n))

     # Start the k-1 passes of Guassian Elimination on A
    for k in range(n-1):

        # Find the pivot element and interchange the rows
        pivot_index = k + np.argmax(np.abs(A[p[k:], k])/s[p[k:]])

        # Interchange element in the permutation vector
        # establish a count of interchanges by the variable c
        if pivot_index !=k:
            temp = p[k]
            p[k]=p[pivot_index]
            p[pivot_index] = temp
            c= c+1

        if np.abs(A[p[k],k]) < 10**(-20):
             sys.exit("ERROR!! Provided matrix is singular.")

        # For the k-th pivot row Perform the Gaussian elimination on the following rows
        for i in range(k+1, n):
            # Find the multiplier
            z = A[p[i],k]/A[p[k],k]

            #Save z in A itself. You can save this in L also
            A[p[i],k] = z

            #Elimination operation: Changes all elements in a row simultaneously
            ##
            A[p[i],k+1:] -= z*A[p[k],k+1:]

    return A, p, c
```

Problem 1: Solve the system Ax=b

In [53]:
```
## modifying GE_srp to solve linear system
## This function uses code given from class and has been modified
def Solve_Axb(A, b):

    m,n = A.shape

    if m !=n:
        sys.exit("This function needs a square matrix as an input.")

    #for solving system
    x = np.zeros(n)

    A, p, c = GE_rsp(A)
```

```
    for k in range(n-1):
        for i in range(k+1, n):
            b[p[i:]] = -b[p[k]]*A[p[i],k]+b[p[i:]] #this scales the b vector the same way it scales A in GE_rsp

    # Below is the code to solve for x vector in Ax=b
    x[p[n-1]] = b[p[n-1]]/A[p[n-1], n-1]
    for j in range(n-2, -1, -1):
        den = sum(A[p[j], j+1:]*x[p[j+1:]])
        x[p[j]] = (b[p[j]]-den)/A[p[j], j]

    return x
```

In [54]:
```
## Solve the following system below
A = np.array([[1, 6, 0], [2,1,0], [0, 2, 1]], dtype=float)
b = np.array([3,1,1], dtype=float)
print("\n Given A: \n ",A)
print("\n Given b: \n", b)
x =Solve_Axb(A, b)
print("\n The system is solved when the x vector is: \n", x)
```

```
Given A:
 [[1. 6. 0.]
[2. 1. 0.]
[0. 2. 1.]]

Given b:
[3. 1. 1.]

The system is solved when the x vector is:
[0.09090909 0.27272727 0.45454545]
```

Problem 2: Find the determinant of a square matrix

In [55]:
```
def det_A(A):

    m,n = A.shape

    if m !=n:
        sys.exit("This function needs a square matrix as an input.")

    A, p, c = GE_rsp(A)

    # below is the code to solve for determinant of A
    U = np.triu(A[p,:])
    dA = 1;
    for i in range(0, n):
        dA = dA*U[i,i]
    if (c % 2) != 0:
        dA = -1*dA

    return dA
```

In [56]:
```
A = np.random.rand(10,10)
print("\n The given matrix A is: \n", A)
dA = det_A(A)
print("\n The determinant of matrix A is: ", dA)
```

```
The given matrix A is:
[[0.54340494 0.27836939 0.42451759 0.84477613 0.00471886 0.12156912
  0.67074908 0.82585276 0.13670659 0.57509333]
[0.89132195 0.20920212 0.18532822 0.10837689 0.21969749 0.97862378
  0.81168315 0.17194101 0.81622475 0.27407375]
[0.43170418 0.94002982 0.81764938 0.33611195 0.17541045 0.37283205
  0.00568851 0.25242635 0.79566251 0.01525497]
[0.59884338 0.60380454 0.10514769 0.38194344 0.03647606 0.89041156
  0.98092086 0.05994199 0.89054594 0.5769015 ]
[0.74247969 0.63018394 0.58184219 0.02043913 0.21002658 0.54468488
  0.76911517 0.25069523 0.28589569 0.85239509]
[0.97500649 0.88485329 0.35950784 0.59885895 0.35479561 0.34019022
  0.17808099 0.23769421 0.04486228 0.50543143]
[0.37625245 0.5928054  0.62994188 0.14260031 0.9338413  0.94637988
  0.60229666 0.38776628 0.363188   0.20434528]
[0.27676506 0.24653588 0.173608   0.96660969 0.9570126  0.59797368
  0.73130075 0.34038522 0.0920556  0.46349802]
[0.50869889 0.08846017 0.52803522 0.99215804 0.39503593 0.33559644
  0.80545054 0.75434899 0.31306644 0.63403668]
```

```
[0.54040458 0.29679375 0.1107879  0.3126403  0.45697913 0.65894007
 0.25425752 0.64110126 0.20012361 0.65762481]]

The determinant of matrix A is:  0.007686324185051754
```

Problem 3: Find the inverse of matrix A method: using the [A|I] to [I|A-1] method

In [57]:
```python
    def inv_A(A):

     m,n = A.shape

     if m !=n:
         sys.exit("This function needs a square matrix as an input.")

     A, p, c = GE_rsp(A)

     Ain = np.eye(n) # identity matrix to help look for A

     for k in range(n-1):
         for i in range(k+1, n):
             Ain[p[i], k+1:] -= A[p[i],k]*Ain[p[k],k+1:] # this is for finding the inverse

     F = np.triu(A[p,:])  #upper triangular matrix of A (this matrix is the U in PA=LU)
     for k in range(0, n):  # make sure all the diagonals are 1s
         div = F[k, k]
         Ain[p[k],:] = Ain[p[k],:]/div
         F[k,k:] = F[k,k:]/div

     for k in range (1, n):  #start changing F to an identity matrix and Ain to the inverse
         for i in range (0, k):
             z = F[i, k]
             F[i, k:]-= z*F[k, k:]
             Ain[p[i],:]-= z*Ain[p[k],:]

     Acpy = Ain.copy() #make sure Ain is un-permuted to just be the inverse of A
     for j in range (0, n):
         Acpy[j,:] = Ain[p[j], :]
     Ain = Acpy.copy()

     return Ain
```

In [58]:
```python
A = np.array([[1, 6, 0], [2,1,0], [0, 2, 1]], dtype=float)
print("\n The given matrix A is: \n", A)
Ain = inv_A(A)
print("\n The inverse of matrix A is: \n", Ain)
```

```
The given matrix A is:
[[1. 6. 0.]
[2. 1. 0.]
[0. 2. 1.]]

The inverse of matrix A is:
[[-0.09090909  0.54545455  0.        ]
[ 0.18181818 -0.09090909  0.        ]
[-0.36363636  0.18181818  1.        ]]
```

Problem A: Guass-Seidel Iteration where x_old represents the x(kth) and x represents the x(k+1th)

In [59]:
```python
# You can modify this code to answer the following
'''
Jacobi's iteration method for solving the system of equations Ax=b.
p0 is the initialization for the iteration.
'''
def seidel(A, b, x0, tol, maxIter=100):
    n=len(A)
    x = x0

    for k in range(maxIter):
        x_old = x.copy() # In python assignment is not the same as copy

        # Update every component of iterant p
        for i in range(n):
            sumi = b[i];
            rsum = 0
            lsum = 0
            for j in range(i+1, n):
```

```
                rsum += A[i,j]*x_old[j]
            for j in range(0, i):
                lsum += A[i,j]*x[j]
            x[i] = (1/A[i,i])*(sumi-rsum-lsum)

        diff = 0
        for i in range(n):
            diff += abs(x[i]-x_old[i])
        rel_error = diff/n

        if rel_error < tol:
            print("TOLERANCE MET BEFORE MAX-ITERATION")
            break
    return x
```

In [60]:
```
# Example System
A = np.array([[10, -1, 2, 0],
              [-1, 11, -1, 3],
              [2, -1, 10, -1],
              [0, 3, -1, 8]],dtype=float)
b = np.array([6, 25, -11, 15],dtype=float)
tol= 0.00001
x0 = [0,0,0,0]
```

In [61]:
```
x = seidel(A, b, x0, tol)
print("\n Solving the system: \n", x)
```

```
TOLERANCE MET BEFORE MAX-ITERATION

 Solving the system:
 [1.000000666348162, 2.0000000246073673, -1.0000002091224143, 0.9999999646319354]
```

Problem B: Successive Over-relaxation (SOR) where x_old represents the x(kth) and x represents the x(k+1th) different from the above method due to the w (weight factor)

In [62]:
```
def SOR(A, b, x0, w, tol, maxIter=100):

    n=len(A)
    x = x0

    for k in range(maxIter):
        x_old = x.copy() # In python assignment is not the same as copy

        # Update every component of iterant p
        for i in range(n):
            rsum = 0
            lsum = 0
            for j in range(i+1, n):
                rsum += A[i,j]*x_old[j]
            for j in range(0, i):
                lsum += A[i,j]*x[j]
            x[i] = (1.0-w)*x_old[i] + (w/A[i,i])*(b[i]-lsum-rsum)

        diff = 0
        for i in range(n):
            diff += abs(x[i]-x_old[i])
        rel_error = diff/n

        if rel_error < tol:
            print("TOLERANCE MET BEFORE MAX-ITERATION")
            break
    return x
```

In [63]:
```
# Example System
A = np.array([[10, -1, 2, 0],
              [-1, 11, -1, 3],
              [2, -1, 10, -1],
              [0, 3, -1, 8]],dtype=float)
b = np.array([6, 25, -11, 15],dtype=float)
tol= 0.00001
x0 = [0,0,0,0]
w = 1.5
x = SOR(A, b, x0, w, tol)
print("\n Solving the system: \n", x)
```

```
TOLERANCE MET BEFORE MAX-ITERATION

 Solving the system:
 [1.000001039057121, 2.0000047104803325, -1.0000027864625325, 0.9999981864832086]
```

In [ ]: