Problem Set # 03
Computer Science 117 - S

Ian Nathaniel J. Lapuz
ijlapuz@up.edu.ph

Elaine T. Pajarillo
etpajarillo@up.edu.ph

Junius Wilhelm G. Bueno
University of the Philippines - Baguio
Gov. Pack Road, Baguio City, Baguio 2600
November 3, 2023

1. Consider the following pentadiagonal matrix,

a. Make a Python function block that prints matrix A for different n.

```python
def block(n):
  A = ls.np.zeros((n,n), dtype = float)
  a = ls.np.ones(n, dtype = float)
  b = ls.np.array([2 if i % 2 == 0 else 1 for i in range(n)], dtype = float)
  c = ls.np.array([[3,4,5][i%3] for i in range(n)])
  d = ls.np.array([1 if i % 2 == 0 else 2 for i in range(n)], dtype = float)
  e = ls.np.ones(n, dtype = float)

  for i in range(n):
    A[i, i] = c[i]
    if i > 0:
      A[i, i - 1] = b[i]
      A[i - 1, i] = d[i - 1]
    if i > 1:
      A[i, i - 2] = a[i]
      A[i - 2, i] = e[i]
  return ls.np.matrix(A)

A_matrix = block(150)
print(A_matrix)
```

In this code, we defined the 'block(n)' function to create the Matrix A based on the input value n and the given initial values for a, b, c, d, and e. First, we initialize an n-by-n matrix A with elements set to zero, and inputted the given initial values for variables a, b, c, d, and e from the problem. We created the matrix A by assigning the values of c[i] as the diagonal elements. We apply the conditions to populate the nondiagonal elements: if i>0 or if i is not on the first row or column, we assign the values of b[i] as the upper diagonal elements of the matrix and d[i-1] as the elements on the lower diagonal. Furthermore, if i>1 or i is at least two rows or columns away from the diagonal, we assign a[i] to the elements in the second upper diagonal, and e[i] to the elements in the second lower diagonal. After completing the process, the code will return the Matrix A for any values of n.

b. Include in linearsys.py the column-wise Forward and Backward substitution methods and modify LUSolve to use the column-wise substitution method

```python
def BackwardSubCol(U,b):
  n = len(U)
  U = np.array(U,dtype = float)
  b = np.array(b, dtype = float)
  for j in range(n):
    if U[j,j] == 0:
      raise RuntimeError("No Solution! The Triangular matrix is not singular.")
  for j in range(n-1, 0, -1):
    b[j] = b[j]/U[j,j]
    for i in range(j):
      b[i] = b[i] - (U[i][j] * b[j])
  b[0] = b[0]/U[0][0]
  return b

def ForwardSubCol(L,b):
  n = len(L)
```

```python
    L = np.array(L,dtype=float)
    b = np.array(b,dtype = float)
    for j in range(n):
      if L[j,j] == 0:
        raise RuntimeError("No Solution! The Triangular matrix is not singular.")
    for j in range(n - 1):
      b[j] = b[j]/L[j,j]
      for i in range(j+1, n):
        b[i] = b[i] - (L[i,j] * b[j])
    b[n-1] = b[n-1]/L[n-1, n-1]
    return b


def LUSolve(A,b):
    L , U = GetLU(A)
    y = ForwardSubCol(L,b)
    return BackwardSubCol(U,y)
```

This code contains the Columnwise Forward and Backward substitution methods, this code harbors much resemblance with its row-wise counterpart, with only the difference in the columnwise method iterating through columns first, and then iterating through the rows for each of the columns in order to compute for the solution. The Columnwise Forward Substitution Method will update the elements in order of columns, from the first to the last, while the Columnwise Backward Substitution Method will update the elements in the reverse order, from the last to the first. As for the modification of the LUSolve, we just called the column-wise substitution method instead of the rowise substitution method.

c. **LU factorization of matrix A and $\tilde{x}$, $\tilde{y} \in \mathbb{R}_n$ that satisfies: $A\tilde{x} = c$, for n = 150**

```
Solution for C:
 [ 0.61485697  0.66524118  0.49018792 -0.25619753  0.8599186   0.8933263
 -0.31079356  0.55123457  0.73457491 -0.25662082  0.79207071  0.89991158
 -0.28952767  0.54404754  0.73264159 -0.25185728  0.79008189  0.89871892
 -0.28797351  0.54412783  0.73227299 -0.25180269  0.7901556   0.89869606
 -0.28798607  0.54413893  0.73227156 -0.25180883  0.79015926  0.89869749
 -0.28798842  0.54413894  0.73227206 -0.25180898  0.7901592   0.89869754
 -0.28798843  0.54413893  0.73227207 -0.25180897  0.79015919  0.89869754
 -0.28798842  0.54413893  0.73227207 -0.25180897  0.79015919  0.89869754
 -0.28798842  0.54413893  0.73227207 -0.25180897  0.79015919  0.89869754
 -0.28798842  0.54413893  0.73227207 -0.25180897  0.79015919  0.89869754
 -0.28798842  0.54413893  0.73227207 -0.25180897  0.79015919  0.89869754
 -0.28798842  0.54413893  0.73227207 -0.25180897  0.79015919  0.89869754
 -0.28798842  0.54413893  0.73227207 -0.25180897  0.79015919  0.89869754
 -0.28798842  0.54413893  0.73227207 -0.25180897  0.79015919  0.89869754
 -0.28798842  0.54413893  0.73227207 -0.25180897  0.79015919  0.89869754
 -0.28798842  0.54413893  0.73227207 -0.25180897  0.79015919  0.89869754
 -0.28798842  0.54413893  0.73227207 -0.25180897  0.79015919  0.89869754
 -0.28798842  0.54413893  0.73227207 -0.25180897  0.79015919  0.89869754
 -0.28798842  0.54413893  0.73227207 -0.25180897  0.79015919  0.89869754
 -0.28798842  0.54413893  0.73227207 -0.25180897  0.79015918  0.89869755
 -0.28798841  0.54413886  0.73227209 -0.25180875  0.79015899  0.89869732
 -0.28798788  0.54413893  0.73227108 -0.25180732  0.79016196  0.89868805
 -0.28799231  0.54418974  0.73224913 -0.25195296  0.79032015  0.89877969
 -0.28840352  0.54454137  0.73278965 -0.25412097  0.78949349  0.90604489
 -0.28856657  0.51153615  0.75258029 -0.16878351  0.67137632  0.89948144]
Relative error for C: 9.483150488903745e-17
```

Solving for the Ax˜ = c, we initially stored the values of c given by the problem in an array of range 150. The resulting array of values saved in CVal to be used in computing for the solution. Then we used LUSolve to compute the solution by utilizing the previously computed A_matrix and CVal to be saved in solution1. Additionally, we also computed the relative error incurred by the solution by getting the norm of the CVal subtracted by the dot product of matrix A and the solution1 divided by the norm of the CVal. Hence, we have arrived at the output above for the computation of the solution x˜ and its corresponding relative error.

d. **LU factorization of matrix A and x˜, y˜ ∈ $\mathbb{R}_n$ that satisfies: Ay˜ = d, for n = 100**

```
Solution for D:
 [ 0.25101212  0.32338026 -0.07641663  0.6083001  -0.12398958  0.17611525
  0.17965952  0.27579418 -0.04301367  0.62707584 -0.14325536  0.1725027
  0.18938074  0.27490455 -0.0447968   0.62809198 -0.1432978   0.17221192
  0.18959213  0.27496197 -0.0448644   0.62807686 -0.14327095  0.17221375
  0.18958074  0.27496388 -0.04486263  0.62807525 -0.14327059  0.17221418
  0.18958031  0.27496382 -0.04486252  0.62807525 -0.14327062  0.17221418
  0.18958032  0.27496382 -0.04486252  0.62807525 -0.14327062  0.17221418
  0.18958032  0.27496382 -0.04486252  0.62807525 -0.14327062  0.17221418
  0.18958032  0.27496382 -0.04486252  0.62807525 -0.14327062  0.17221418
  0.18958032  0.27496382 -0.04486252  0.62807525 -0.14327062  0.17221418
  0.18958032  0.27496382 -0.04486252  0.62807525 -0.14327062  0.17221418
  0.18958032  0.27496382 -0.04486252  0.62807526 -0.14327062  0.17221416
  0.18958032  0.27496392 -0.04486258  0.62807501 -0.14327027  0.17221417
  0.18957946  0.2749655  -0.04486195  0.62806828 -0.14326901  0.17222962
  0.18957179  0.27490906 -0.04481465  0.62819166 -0.1435083   0.17234722
  0.19011733  0.27334587 -0.04488399  0.63391996 -0.14630906  0.16239635
  0.19988395  0.30063943 -0.07877493  0.59271183]

Relative error for D: 1.2677949733248716e-16
```

In solving for the Ay˜ = d, the same process above was performed. First saving the values of d given by the problem in an array of range 100 with the resulting array of values saved in DVal to be used in computing for the solution. Then LUSolve is used to compute for the solution using the previously computed A_matrix and DVal to be saved in solution2. We also computed the relative error that the solution incurred by getting the norm of the DVal subtracted by the dot product of matrix A and the solution2 divided by the norm of the DVal. Hence, we have arrived at the output above for the computation of the solution y˜ and the relative error.

2. Use the Jacobi method to find the solution of the linear system with initial iterate x0 = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1], tolerance 10^{-15}, and maximum iterate of 1000.

In order to use the Jacobi Method to find the solution of the given linear system, we have to first ensure that the main diagonal of our system has only non-zero elements. Putting the given system inside Matrix A with its corresponding vector b our initial inputs are given by:

```python
A = [   [0, 0, 0, -1, 4, 2, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, -1, 4, 2, 0],
        [0, 0, 2, -1, -1, 2, 0, 0, 0, 0],
        [0, 0, 0, 0, -1, 5, 2, 0, 0, 0],
        [4, 2, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, -1, 4, 2, -1, 4, 2, 0],
```

```
        [ 0,  0,  0,  0,  0,  0,  0,  0, -1,  4],
        [ 0,  0,  0,  0,  0,  0, -1,  4,  2,  0],
        [-1,  4,  2,  0,  0,  0,  0,  0,  0,  0],
        [ 0,  0, -1,  4,  2,  0,  0,  0,  0,  0]]
b = [1, 1, 3, 4, 5, 6, 7, 8, 9, 10]
```

From this, we can see that there are elements equal to zero in the main diagonal which prevents the implementation of the Jacobi Method, to resolve this, we need to perform partial pivoting on Matrix A to ensure a main diagonal with all non-zero elements. Below is the code that we used in solving for the solution of the linear system with the additional step of Partial Pivoting the Matrix.

Python

```python
import linearsys as ls

#Original System:
A = [   [0, 0, 0, -1, 4, 2, 0, 0, 0 ,0],
        [0, 0, 0, 0, 0, 0, 0, -1, 4, 2],
        [0, -1, -1, 2, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, -1, 5, 2, 0, 0],
        [4, 2, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, -1, 4, 2, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, -1, 4],
        [0, 0, 0, 0, 0, 0, -1, 4, 2, 0],
        [-1, 4, 2, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, -1, 4, 2, 0, 0, 0, 0, 0]
    ]
b = [1, 1, 3, 4, 5, 6, 7, 8, 9, 10]

def partial_pivot(A, b):
    n = len(A)
    for i in range(n):
        max_row = i
        for j in range(i + 1, n):
            if abs(A[j][i]) > abs(A[max_row][i]):
                max_row = j
        A[i], A[max_row] = A[max_row], A[i]
        b[i], b[max_row] = b[max_row], b[i]
    return A, b

#output the pivoted matrix
partial_pivot(A, b)
print(f"Sorted Matrix:\n{ls.np.matrix(A)}\n")  #hash to hide sorted matrix


#using Jacobi Method
sol, niter, relerr = ls.JacobiSolve(A, b, ls.np.ones(10), 1e-15, 1000)
print("Solution:")
for n in range(10):
    print(f"\tx{n+1} = {sol[n]}")
print(f"Number of Iterations: {niter}")
print(f"Relative Error: {relerr}\n")
```

The code starts with performing partial pivoting, which is used to ensure that the main diagonal of the matrix does not contain zero elements. In this process, a simple for loop checks if the pivot element is greater than the absolute value of the diagonal element at position [i][j]. If it is, rows are swapped to rearrange the matrix until the absolute value of the elements on the diagonal is greater than the absolute value of all the other elements. During the process of this row swapping, the corresponding values of the matrix A in vector b are also rearranged accordingly. Consequently, after completing the partial pivoting, both Matrix A and vector b are rearranged so that the main diagonal of the matrix does not contain any zero elements.

Following the partial pivoting of Matrix A, we output the newly rearranged matrix as 'Sorted Matrix.' Then, we proceed to apply the Jacobi Method to solve the linear system. This method can now be used because the initial condition of having all non-zero diagonal elements has been met. We call a function from the 'linearsys.py' file to compute the solution, the number of iterations, and the relative error and just output the results accordingly.

Upon running “Item2Imp.py”, the following results are yielded:

```
Solution:
        x1 = -0.7823189693469743
        x2 = 4.064637938693944
        x3 = -4.0204353620613675
        x4 = 1.52210128831629
        x5 = -0.05442025766326386
        x6 = 1.3698911594846728
        x7 = 0.23300755219902242
        x8 = 2.10242669924478
        x9 = -0.08834962239004884
        x10 = 1.7279125944024878
Number of Iterations: 243
Relative Error: 9.53223792001001e-16
```

3. **Find the solution [a ∗ , b∗ , c∗ , d∗] to the nonlinear system of equations:**

The following Python code “Item3Imp.py” is used to find the solutions of the given nonlinear systems of equations. It imports “linearsys.py” so that we can use the LUSolve() function.

```python
import linearsys as ls

def newton(f,Jf,x,tol,maxit):
        x = ls.np.array(x,float)
        err_newton = tol + 1
        k_newton = 0
        while err_newton > tol and k_newton < maxit:
                dx = ls.LUSolve(Jf(x),-f(x))
                x = x + dx
                err_newton = ls.np.linalg.norm(f(x))
                k_newton += 1
        if err_newton > tol and k_newton == maxit:
                print("Error in tol and / or maxit.")
        return x, err_newton, k_newton
```

This first part of the code contains the function newton(f, Jf, x, tol, maxit), the Python implementation of the Newton Method. It takes in the parameters f, Jf, tolerance, and max iterations. Also, for both items, we let x[0] = a, x[1] = b, x[2] = c, and x[3] = d, respectively, and convert the given functions to the said form for efficiency.

a. $20ac - 8ab + 16c^3 - 4bd = 39$
$12a + 6b + 2c - 2d = 11$
$4a^2 + 2bc - 10c + 2ad^2 = -7$
$-3ad - 2b^2 + 7cd = 16$
with initial iterate x = [1, 1, 1, 1], tolerance 10^{-14}, and maximum iteration of 100.

For item A, note that:

$$f_0 = 20x[0]x[2] - 8x[0]x[1] + 16x[2]^3 - 4x[1]x[3] - 39 = 0$$
$$f_1 = 12x[0] + 6x[1] + 2x[2] - 2x[3] - 11 = 0$$
$$f_2 = 4x[0]^2 + 2x[1]x[2] - 10x[2] + 2x[0]x[3]^2 + 7 = 0$$
$$f_3 = -3x[0]x[3] - 2x[1]^2 + 7x[2]x[3] - 16 = 0$$

Solving for their partial derivates, we get:

/	∂x[0]	∂x[1]	∂x[2]	∂x[3]
∂f_0	$20x[2] - 8x[1]$	$-8x[0] - 4x[3]$	$20x[0] + 48x[2]^2$	$-4x[1]$
∂f_1	12	6	2	-2
∂f_2	$8x[0] + 2x[3]^2$	$2x[2]$	$2x[1] - 10$	$4x[0]x[3]$
∂f_3	$-3x[3]$	$-4x[1]$	$7x[3]$	$-3x[0] + 7x[2]$

Therefore, we have:

$$f_{00} = 20x[2] - 8x[1] \qquad f_{20} = 8x[0] + 2x[3]^2$$
$$f_{01} = -8x[0] - 4x[3] \qquad f_{21} = 2x[2]$$
$$f_{02} = 20x[0] + 48x[2]^2 \qquad f_{22} = 2x[1] - 10$$
$$f_{03} = -4x[1] \qquad f_{23} = 4x[0]x[3]$$
$$f_{10} = 12 \qquad f_{30} = -3x[3]$$
$$f_{11} = 6 \qquad f_{31} = -4x[1]$$
$$f_{12} = 2 \qquad f_{32} = 7x[3]$$
$$f_{13} = -2 \qquad f_{33} = -3x[0] + 7x[2]$$

We, then, hardcode them into “Item3Imp.py” under the function af(x) and aJf(x) so that arrays of the inputted functions are returned. There is function aA(x) contains the same functions but without the b values, which will be used for the computation of the relative error.

```python
#itema
def af(x):
	x = ls.np.array(x, dtype = float)
```

```
        # a = x[0], b = x[1], c = x[2], d = x[3]
        f0 = (20 * x[0] * x[2]) - (8 * x[0] * x[1]) + (16 * x[2]**3) - (4 * x[1] * x[3]) - 39
        f1 = (12 * x[0]) + (6 * x[1]) + (2 * x[2]) - (2 * x[3]) - 11
        f2 = (4 * x[0]**2) + (2 * x[1] * x[2]) - (10 * x[2]) + (2 * x[0] * x[3]**2) + 7
        f3 = (-3 * x[0] * x[3]) - (2 * x[1]**2) + (7 * x[2] * x[3]) -16
        return ls.np.array([f0, f1, f2, f3], float)

def aJf(x):
        f00 = (20 * x[2]) - (8 * x[1])
        f01 = (-8 * x[0]) - (4 * x[3])
        f02 = (20 * x[0]) + (48 * x[2]**2)
        f03 = -4 * x[1]
        f10 = 12
        f11 = 6
        f12 = 2
        f13 = -2
        f20 = (8 * x[0]) + (2 * x[3]**2)
        f21 = 2 * x[2]
        f22 = (2 * x[1]) - 10
        f23 = 4 * x[0] * x[3]
        f30 = -3 * x[3]
        f31 = -4 * x[1]
        f32 = 7 * x[3]
        f33 = (-3 * x[0]) + (7 * x[2])
        return ls.np.array([ [f00, f01, f02, f03],
                             [f10, f11, f12, f13],
                             [f20, f21, f22, f23],
                             [f30, f31, f32, f33]], float)

def aA(x):
        #Function without b for the relative error
        x = ls.np.array(x, dtype = float)
        # a = x[0], b = x[1], c = x[2], d = x[3]
        f0 = (20 * x[0] * x[2]) - (8 * x[0] * x[1]) + (16 * x[2]**3) - (4 * x[1] * x[3])
        f1 = (12 * x[0]) + (6 * x[1]) + (2 * x[2]) - (2 * x[3])
        f2 = (4 * x[0]**2) + (2 * x[1] * x[2]) - (10 * x[2]) + (2 * x[0] * x[3]**2)
        f3 = (-3 * x[0] * x[3]) - (2 * x[1]**2) + (7 * x[2] * x[3])
        return ls.np.array([f0, f1, f2, f3], float)
```

To find the solution of Item A, we pass af(), aJf(), the initial iterate x = [1,1,1,1], the tolerance of 1e-14, and the max iteration of 100 to the newton() function. For the solving of the relative error, we make use of the imported np.linalg.norm(), aA(x), and the separated given solution “gA”.

Python

```
print("Item A:")
itema = newton(af, aJf, [1,1,1,1], 1e-14, 100)
gA = ls.np.array([39, 11, -7, 16], float) #given solution array
print(f" Solution:")
for n in range(4):
        print(f"\t{unk[n%4]} = {itema[0][n]}")
print(f" Error: {(ls.np.linalg.norm(aA(itema[0]) - gA))/ls.np.linalg.norm(gA)}")
print(f" Iterations: {itema[2]}\n")
```

Upon running “Item3Imp.py”, the following results are yielded for item A:

```
Item A:
   Solution:
        a* = 0.2669201878036756
        b* = 1.5623911642242143
        c* = 1.4480821232798882
        d* = 2.2367767427745844
   Error: 8.299304550897164e-17
   Iterations: 7
```

b. $3a - cos(bc) = 0.5$

$a^2 - 81(b + 0.1)^2 + \sin(c) = -1.06$

$e^{-ab} + 20c = \frac{3-10\pi}{3}$

with initial iterate x = [1, 1, 1], tolerance 10^{-14}, and maximum iteration of 100.

For item B, note that:

$$f_0 = 3x[0] - cos(x[1]x[2]) - 0.5 = 0$$

$$f_1 = x[0]^2 - 81(x[1] + 0.1)^2 + sin(x[2]) + 1.06 = 0$$

$$f_2 = e^{-x[0]x[1]} + 20x[2] - \frac{3-10\pi}{3} = 0$$

Solving for their partial derivates, we get:

/	∂x[0]	∂x[1]	∂x[2]
∂f_0	3	$x[1]sin(x[1]x[2])$	$2x[0]$
∂f_1	$2x[0]$	$-162(x[1] + 0.1$	$cos(x[2])$
∂f_2	$-x[1]e^{-x[0]x[1]}$	$-x[0]e^{-x[0]x[1]}$	20

Therefore, we have:

$$f_{00} = 3$$
$$f_{01} = x[2]sin(x[1]x[2])$$
$$f_{02} = x[1]sin(x[1]x[2])$$
$$f_{10} = 2x[0]$$
$$f_{11} = -162(x[1] + 0.1)$$
$$f_{12} = cos(x[2])$$
$$f_{20} = -x[1]e^{-x[0]x[1]}$$
$$f_{21} = -x[0]e^{-x[0]x[1]}$$
$$f_{22} = 20$$

We, again, hardcode them into “Item3Imp.py” under the function bf(x) and bJf(x) so that arrays of the inputted functions are returned. There is function bA(x) contains the same functions but without the b values, which will be used for the computation of the relative error.

```python
#itemb
def bf(x):
      x = ls.np.array(x, dtype = float)
```

```
        # a = x[0], b = x[1], c = x[2]
        f0 = (3 * x[0]) - ls.np.cos(x[1] * x[2]) - 0.5
        f1 = (x[0]**2) - (81 * (x[1] + 0.1)**2) + ls.np.sin(x[2]) + 1.06
        f2 = ls.np.exp(-x[0] * x[1]) + (20 * x[2]) - ((3 - (10 * ls.np.pi))/3)
        return ls.np.array([f0, f1, f2], float)

    def bJf(x):
        f00 = 3
        f01 = x[2] * ls.np.sin(x[1] * x[2])
        f02 = x[1] * ls.np.sin(x[1] * x[2])
        f10 = 2 * x[0]
        f11 = -162 * (x[1] + 0.1)
        f12 = ls.np.cos(x[2])
        f20 = -x[1] * ls.np.exp(-x[0] * x[1])
        f21 = -x[0] * ls.np.exp(-x[0] * x[1])
        f22 = 20
        return ls.np.array([[f00, f01, f02], [f10, f11, f12], [f20, f21, f22]])

    def bA(x):
        #Function without b for the relative error
        x = ls.np.array(x, dtype = float)
        # a = x[0], b = x[1], c = x[2]
        f0 = (3 * x[0]) - ls.np.cos(x[1] * x[2])
        f1 = (x[0]**2) - (81 * (x[1] + 0.1)**2) + ls.np.sin(x[2])
        f2 = ls.np.exp(-x[0] * x[1]) + (20 * x[2])
        return ls.np.array([f0, f1, f2], float)
```

For the solution of Item B, we pass bf(), bJf(), the initial iterate x = [1,1,1], the tolerance of 1e-14, and the max iteration of 100 to the newton() function. For the solving of the relative error, we make use of the imported np.linalg.norm(), bA(x), and the separated given solution “gB”.

```python
print("Item B:")
itemb = newton(bf, bJf, [1,1,1], 1e-14, 100)
gB = ls.np.array([0.5, -1.06, (3 - (10 * ls.np.pi))/3], float) #given solution array
print(" Solution:")
for n in range(3):
    print(f"\t{unk[n%4]} = {itemb[0][n]}")
gB = ls.np.array([0.5, -1.06, (3 - (10 * ls.np.pi))/3], float)
print(f" Error: {(ls.np.linalg.norm(bA(itemb[0]) - gB))/ls.np.linalg.norm(gB)}")
print(f" Iterations: {itemb[2]}\n")
```

Upon running “Item3Imp.py”, the following results are yielded for item B:

```
Item B:
   Solution:
        a* = 0.49999999999999994
        b* = -1.657528818588421e-17
        c* = -0.5235987755982988
   Error: 1.8900455247697116e-16
   Iterations: 8
```