UNIVERSITY OF VICTORIA

Programming Project

Disha Garg
V00847833

DEPARTMENT OF COMPUTER SCIENCE

CSC-545

Contents

Question 1: Primal revised simplex method (Pedagogical implementation).

Code:

Fetching the data from the csv:

```
class Project:
    """Question1 and 2."""

    def __init__(self, parent=None):
        """The start point."""
        parser = argparse.ArgumentParser(description='Linear Program Solver.')
        parser.add_argument('A_csv', help='The csv for A matrix.')
        parser.add_argument('b_csv', help='The csv for b vector.')
        parser.add_argument('c_csv', help='The csv for c vector.')
        args = parser.parse_args()

        # Fetch the data for A matrix
        data_a = csv.reader(open(args.A_csv, 'rb'))
        a = []
        for row in data_a:
            a.append(row)
        rows_a = len(a)
        cols_a = len(row)
        a = np.array([a]).reshape(rows_a, cols_a).astype(np.float)

        # Fetch the data for b vector
        data_b = csv.reader(open(args.b_csv, 'rb'))
        self.b = []
        for row in data_b:
            self.b.append(row)
        rows_b = len(self.b)
        cols_b = len(row)
        self.b = np.array([self.b]).reshape(rows_b, cols_b).astype(np.float)

        # Fetch the data for c vector
        data_c = csv.reader(open(args.c_csv, 'rb'))
        self.c = []
        for row in data_c:
            self.c.append(row)
        rows_c = len(self.c)
        cols_c = len(row)
        self.c = np.array([self.c]).reshape(rows_c, cols_c).astype(np.float)
```

Arranging the data into matrices and vectors like Nu, Beta, matrix_N, matrix_B and sending the data to the solvers, according to the feasibility/infeasibility of the problem.

```python
        self.Nu = np.arange(1, rows_c + 1)
        self.Beta = np.arange(len(self.Nu) + 1, rows_b + len(self.Nu) + 1)

        split_a = np.hsplit(a, [rows_c, rows_b + rows_c])
        matrix_N = split_a[0]
        matrix_B = split_a[1]
        rows_N, cols_N = np.shape(matrix_N)
        rows_B, cols_B = np.shape(matrix_B)

        # Check for matrices consistency
        if cols_N == rows_c and rows_a == rows_b == rows_B == cols_B == rows_N:
            print("Everything is consistent!")
        else:
            print("[Error]: Data is inconsistent! Check the CSVs.")
            sys.exit()

        x_starB = self.b
        z_star_n = -1 * self.c
        objective = 0

        # For Primal Infeasibility
        if min(x_starB) < 0.0 and min(z_star_n) >= 0.0:
            print "[Error]: The problem is Primal Infeasible."
            print "So, performing Dual Simplex Method..."
            pt2.dual_simplex_solver(x_starB, matrix_B, matrix_N, z_star_n, self.b, self.c, self.Beta, self.Nu, objective)
            sys.exit()

        # For Dual Infeasibility
        if min(z_star_n) < 0.0 and min(x_starB) >= 0.0:
            print "[Error]: The problem is Dual Infeasible."
            print "So, performing Primal Simplex Method..."
            pt1.primal_simplex_solver(z_star_n, matrix_B, matrix_N, x_starB, self.b, self.c, self.Beta, self.Nu, objective)
            sys.exit()

        # For Dual and Primal Infeasibility
        if min(z_star_n) < 0.0 and min(x_starB) < 0.0:
            print "The problem is Dual and Primal Infeasible."
            pt3.primal_dual_simplex_solver(z_star_n, matrix_B, matrix_N, x_starB, self.b, self.c, self.Beta, self.Nu, objective)
```

Now, **The Primal Solver** has been implemented as:

```python
def choose_smaller_subscript(itemindex):
    """Helper Function to choose the smaller subscript."""
    if len(itemindex[0]) > 1:
        itemindex = tuple(np.asarray([[itemindex[0][0]], [itemindex[1][0]]]))
    return itemindex


def primal_simplex_solver(z_starN, matrix_B, matrix_N, x_star_b, b, c, Beta, Nu, optimal_value):
    """Primal Problem solver."""
    # Step 1: compute the optimal solution till zN < 0, if zN >= 0 then stop
    while np.min(z_starN) < 0.0:
        # Step 2: Pick an index j in Nu for which min(z*j) < 0 (entering variable).
        itemindex_j = np.where(z_starN == np.min(z_starN))
        itemindex_j = choose_smaller_subscript(itemindex_j)
        j = Nu[itemindex_j[0]].squeeze()

        # Step 3: Compute Primal Step Direction delta_x_b
        # Initialize e_j and e_i
        e_j = np.zeros(z_starN.shape)
        e_i = np.zeros(b.shape)
        e_j[itemindex_j[0]] = 1
        mult = np.dot(linalg.inv(matrix_B), matrix_N)
        delta_x_b = np.dot(mult, e_j)

        # Suppress any divide by zero warnings
        import warnings

        def fxn():
            warnings.warn("deprecated", DeprecationWarning)
        with warnings.catch_warnings():
            warnings.simplefilter("ignore")
            temp = np.array(delta_x_b / x_star_b)
            fxn()

        # Step 4: Pick the largest t >= 0 for which every component of x*B remains nonnegative (Primal Step Length).
        if any(np.isnan(ob) for ob in temp):
            index = np.where(np.isnan(temp))
            temp[index] = 0
```

```
    # Step 4: Pick the largest t >= 0 for which every component of x*B remains nonnegative (Primal Step Length).
    if any(np.isnan(ob) for ob in temp):
        index = np.where(np.isnan(temp))
        temp[index] = 0
    if np.max(temp).squeeze() <= 0:
        print "[Error]: The primal is Unbounded"
        sys.exit()
    t = np.reciprocal(np.max(temp).squeeze())

    # Step 5: The leaving variable is chosen with the max ratio.
    itemindex = np.where(temp == np.max(temp))
    itemindex = choose_smaller_subscript(itemindex)
    i = Beta[itemindex[0]].squeeze()

    # Step 6: Compute Dual Step Direction delta_zN.
    e_i[itemindex[0]] = 1
    mult = np.transpose(np.dot(linalg.inv(matrix_B), matrix_N))
    delta_z_Nu = -1 * (np.dot(mult, e_i))

    # Step 7: Compute Dual Step Length.
    z_star_j = z_starN[itemindex_j[0]].squeeze()
    delta_z_j = delta_z_Nu[itemindex_j[0]].squeeze()
    s = z_star_j / delta_z_j

    # Step 8: Update Current Primal and Dual Solutions.
    # Check Degeneracy: if the ratio is infinite then no updation of x*B.
    if any(ob == float('inf') for ob in temp):
        x_star_b = x_star_b
    else:
        x_star_b = x_star_b - np.dot(t, delta_x_b)
        x_star_b[itemindex[0]] = t
    z_starN = z_starN - np.dot(s, delta_z_Nu)
    z_starN[itemindex_j[0]] = s

    # Step 9: Update Basis.
    Beta[itemindex[0]] = j
    Nu[itemindex_j[0]] = i
    matrix_N[:, itemindex_j[0]], matrix_B[:, itemindex[0]] = matrix_B[:, itemindex[0]], matrix_N[:, itemindex_j[0]]
```

```
    # Step 9: Update Basis.
    Beta[itemindex[0]] = j
    Nu[itemindex_j[0]] = i
    matrix_N[:, itemindex_j[0]], matrix_B[:, itemindex[0]] = matrix_B[:, itemindex[0]], matrix_N[:, itemindex_j[0]]

# Objective Function Comptation [c_B]'*[B^-1]*b
optimal_value = 0
print "\t[Optimal Solution found]"
for i in range(len(c)):
    if i + 1 in Beta:
        index = np.where(Beta == i + 1)
        optimal_value += c[i].squeeze() * x_star_b[index].squeeze()
    else:
        optimal_value += c[i] * 0
print "Optimal Solution: ", optimal_value
print "x*B: \n", x_star_b
print "z*N: \n", z_starN
print "B: \n", matrix_B
print "N: \n", matrix_N
print "Beta: ", Beta
print "Nu: ", Nu
return z_starN, x_star_b, matrix_B, matrix_N, Nu, Beta, optimal_value
```

Here, I am using a helper function to choose smaller subscript in case we have ties in choosing the entering and the leaving variables.

Unboundedness is handled by exiting with an error response to the user.

Also, degeneracy is handled by not updating x*B.

Example 1:

Inputs:

The Inputs consist of the matrices/vectors A; b; c specifying the problem in separate csv files.

my_A.csv

	A	B	C	D	E
1	1	-1	1	0	0
2	2	-1	0	1	0
3	0	1	0	0	1

My_b.csv

	A
1	1
2	3
3	5

My_c.csv

	A
1	4
2	3

Running the code as:

python consolidation.py my_A.csv my_b.csv my_c.csv

Output:

During each iteration we have explicit representations (that is separate matrices/vectors) B, N, Beta, Nu, x^*_B, z^*_N, B^{-1}, e_j , e_i, Delta_X_B, and Delta_Z_N as well as the scalars s and t. The code outputs the final optimal solution and associated value of the objective function.

The output for this example looks like:

```
E:\3rd Term + TA\OR1\Project>python consolidation.py csv\my_A.csv csv\my_b.csv csv\my_c.csv
Everything is consistent!
[Error]: The problem is Dual Infeasible.
So, performing Primal Simplex Method...
        [Optimal Solution found]
Optimal Solution:  31.0
x*B:
[[ 4.]
 [ 5.]
 [ 2.]]
z*N:
[[ 5.]
 [ 2.]]
B:
[[ 1. -1.  1.]
 [ 2. -1.  0.]
 [ 0.  1.  0.]]
N:
[[ 0.  0.]
 [ 0.  1.]
 [ 1.  0.]]
Beta:  [1 2 3]
Nu:  [5 4]
```

Example 2:

Inputs:

The Inputs consist of the matrices/vectors A; b; c specifying the problem in separate csv files.

a.csv

	A	B	C	D	E	F
1	2	3	1	1	0	0
2	4	1	2	0	1	0
3	3	4	2	0	0	1

b.csv

	A
1	5
2	11
3	8

c.csv

	A
1	5
2	4
3	3

Running the code as:

python consolidation.py a.csv b.csv c.csv

Output:

During each iteration we have explicit representations (that is separate matrices/vectors) B, N, Beta, Nu, x^*_B, z^*_N, B^{-1}, e_j , e_i, Delta_X_B, and Delta_Z_N as well as the scalars s and t. The code outputs the final optimal solution and associated value of the objective function.

The output for this example looks like:

```
E:\3rd Term + TA\OR1\Project>python consolidation.py csv\a.csv csv\b.csv csv\c.csv
Everything is consistent!
[Error]: The problem is Dual Infeasible.
So, performing Primal Simplex Method...
        [Optimal Solution found]
Optimal Solution:  [ 13.]
x*B:
[[ 2.]
 [ 1.]
 [ 1.]]
z*N:
[[ 1.]
 [ 3.]
 [ 1.]]
B:
[[ 2.  0.  1.]
 [ 4.  1.  2.]
 [ 3.  0.  2.]]
N:
[[ 1.  3.  0.]
 [ 0.  1.  0.]
 [ 0.  4.  1.]]
Beta:  [1 5 3]
Nu:  [4 2 6]
```

Question 2:

a) Dual revised simplex method (Pedagogical implementation).

Code:

Fetching the data from the csv and sending to the solver is the same as before (reading the csv as command line arguments and then converting them to matrices and vectors and sending them to Primal, Dual and Two Phase method implementation.

Now, **The Dual Solver** has been implemented as:

```
def choose_smaller_subscript(itemindex):
    """Helper Function to choose the smaller subscript."""
    if len(itemindex[0]) > 1:
        itemindex = tuple(np.asarray([[itemindex[0][0]], [itemindex[1][0]]]))
    return itemindex


def dual_simplex_solver(x_star_b, matrix_B, matrix_N, z_starN, b, c, Beta, Nu, optimal_value):
    """Dual Problem solver."""
    # Step 1: compute the optimal solution till xB < 0, if xB >= 0 then stop
    while np.min(x_star_b) < 0.0:
        # Step 2: Pick an index i in Beta for which min(x*B) < 0 (entering variable).
        itemindex_i = np.where(x_star_b == np.min(x_star_b))
        itemindex_i = choose_smaller_subscript(itemindex_i)
        i = Beta[itemindex_i[0]].squeeze()

        # Step 3: Compute Dual Step Direction delta_z_n
        # Initialize e_i and e_j
        e_i = np.zeros(x_star_b.shape)
        e_j = np.zeros(c.shape)
        e_i[itemindex_i[0]] = 1

        mult = -1 * (np.transpose(np.dot(linalg.inv(matrix_B), matrix_N)))
        delta_z_n = (np.dot(mult, e_i))

        # Suppress any divide by zero warnings
        import warnings

        def fxn():
            warnings.warn("deprecated", DeprecationWarning)
        with warnings.catch_warnings():
            warnings.simplefilter("ignore")
            temp = np.array(delta_z_n / z_starN)
            fxn()
        # Step 4: Pick the largest s >= 0 for which every component of z*N remains nonnegative (Dual Step Length).
        if any(np.isnan(ob) for ob in temp):
            index = np.where(np.isnan(temp))
```

```
    # Step 5: The leaving variable is chosen with the max ratio.
    s = np.reciprocal(np.max(temp).squeeze())
    if s < 0:
        print "[ERROR]: The dual is unbounded"
        sys.exit()
    itemindex = np.where(temp == np.max(temp))
    itemindex = choose_smaller_subscript(itemindex)
    j = Nu[itemindex[0]].squeeze()    # step 5 done

    # Step 6: Compute Primal Step Direction delta_zN.
    e_j[itemindex[0]] = 1
    mult = np.dot(linalg.inv(matrix_B), matrix_N)
    delta_x_Beta = np.dot(mult, e_j)

    # Step 7: Compute Primal Step Length.
    x_star_i = x_star_b[itemindex_i[0]].squeeze()
    delta_x_i = delta_x_Beta[itemindex_i[0]].squeeze()
    t = x_star_i / delta_x_i

    # Step 8: Update Current Primal and Dual Solutions.
    # Check Degeneracy: if the ratio is infinite then no updation of x*B.
    if any(ob == float('inf') for ob in temp):
        z_starN = z_starN
    else:
        z_starN = z_starN - np.dot(s, delta_z_n)
        z_starN[itemindex[0]] = s
    x_star_b = x_star_b - np.dot(t, delta_x_Beta)
    x_star_b[itemindex_i[0]] = t

    # Step 9: Update Basis.
    Nu[itemindex[0]] = i
    Beta[itemindex_i[0]] = j
    matrix_N[:, itemindex[0]], matrix_B[:, itemindex_i[0]] = matrix_B[:, itemindex_i[0]], matrix_N[:, itemindex[0]]

# Objective Function Comptation [c_B]'*[B^-1]*b
optimal_value = 0
print "\t[Optimal Solution found]"
```

```
    matrix_N[:, itemindex[0]], matrix_B[:, itemindex_i[0]] = matrix_B[:, itemindex_i[0]], matrix_N[:, itemindex[0]]

# Objective Function Comptation [c_B]'*[B^-1]*b
optimal_value = 0
print "\t[Optimal Solution found]"
for i in range(len(c)):
    if i + 1 in Beta:
        index = np.where(Beta == i + 1)
        optimal_value += c[i].squeeze() * x_star_b[index].squeeze()
    else:
        optimal_value += c[i] * 0
print "Optimal Solution: ", optimal_value
print "x*B: \n", x_star_b
print "z*N: \n", z_starN
print "B: \n", matrix_B
print "N: \n", matrix_N
print "Beta: ", Beta
print "Nu: ", Nu
return z_starN, x_star_b, matrix_B, matrix_N, Nu, Beta, optimal_value
```

Similarly, I am using the helper function here to choose smaller subscripts in case we have ties in choosing the entering and the leaving variables.

Unboundedness is handled by exiting with an error response to the user.

And, degeneracy is handled by not updating x*B. (Same as in Primal)

Example 1:

Inputs:

The Inputs consist of the matrices/vectors A; b; c specifying the problem in separate csv files.

my_A2.csv

	A	B	C	D	E
1	-2	-1	1	0	0
2	-2	4	0	1	0
3	-1	3	0	0	1
4					

my_b2.csv

	A
1	4
2	-8
3	-7
4	

my_c2.csv

	A
1	-1
2	-1
3	

Running the code as:

python consolidation.py my_A2.csv my_b2.csv my_c2.csv

Output:

During each iteration we have explicit representations (that is separate matrices/vectors) B, N, Beta, Nu, x^*_B, z^*_N, B^{-1}, e_j , e_i, Delta_X_B, and Delta_Z_N as well as the scalars s and t. The code outputs the final optimal solution and associated value of the objective function.

The output looks like:

```
E:\3rd Term + TA\OR1\Project>python consolidation.py csv\my_A2.csv csv\my_b2.csv csv\my_c2.csv
Everything is consistent!
[Error]: The problem is Primal Infeasible.
So, performing Dual Simplex Method...
        [Optimal Solution found]
Optimal Solution:  [-7.]
x*B:
[[ 18.]
 [  7.]
 [  6.]]
z*N:
[[ 1.]
 [ 4.]]
B:
[[ 1. -2.  0.]
 [ 0. -2.  1.]
 [ 0. -1.  0.]]
N:
[[ 0. -1.]
 [ 0.  4.]
 [ 1.  3.]]
Beta:  [3 1 4]
Nu:  [5 2]
```

Example 2:

Inputs:

The Inputs consist of the matrices/vectors A; b; c specifying the problem in separate csv files.

my_A6.csv

	A	B	C	D	E
1	-1	1	1	0	0
2	-1	-2	0	1	0
3	0	1	0	0	1

my_b6.csv

	A
1	-1
2	-2
3	1

my_c6.csv

	A
1	-2
2	-1
3	

Running the code as:

python consolidation.py my_A6.csv my_b6.csv my_c6.csv

Output:

During each iteration we have explicit representations (that is separate matrices/vectors) B, N, Beta, Nu, x^*_B, z^*_N, B^{-1}, e_j , e_i, Delta_X_B, and Delta_Z_N as well as the scalars s and t. The code outputs the final optimal solution and associated value of the objective function.

The output looks like:

```
E:\3rd Term + TA\OR1\Project>python consolidation.py csv\my_A6.csv csv\my_b6.csv csv\my_c6.csv
Everything is consistent!
[Error]: The problem is Primal Infeasible.
So, performing Dual Simplex Method...
        [Optimal Solution found]
Optimal Solution:  -3.0
x*B:
[[ 1.33333333]
 [ 0.33333333]
 [ 0.66666667]]
z*N:
[[ 1.]
 [ 1.]]
B:
[[-1.  1.  0.]
 [-1. -2.  0.]
 [ 0.  1.  1.]]
N:
[[ 1.  0.]
 [ 0.  1.]
 [ 0.  0.]]
Beta:  [1 2 5]
Nu:  [3 4]
```

b) Two Phase simplex method (Pedagogical implementation).

Code:

Fetching the data from the csv and sending to the solver is the same as before (reading the csv as command line arguments and then converting them to matrices and vectors and sending them to Primal, Dual and Two Phase method implementation.

Now, **The Two Phase method** implementation looks like:

```
def primal_dual_simplex_solver(z_starN, matrix_B, matrix_N, x_star_b, b, c, Beta, Nu, optimal_value):
    """Primal Problem solver."""
    # Step 1: Convert z*N to nonnegative to make it Dual Feasible
    temp_c = -1 * np.ones((len(c), 1))
    z_starN = -1 * temp_c
    z_starN, x_star_b, matrix_B, matrix_N, Nu, Beta, sol = pt2.dual_simplex_solver(x_star_b, matrix_B, matrix_N

    A_matrix = -1 * np.dot(linalg.inv(matrix_B), matrix_N)

    row_matrix_a = []
    indices_rows_matrix_a = []
    # First Loop to pull x*B and corresponding A matrix row for the decision variables
    # and multiply them to their coefficients in the objective function.
    for i in range(len(c)):
        if i + 1 in Beta:
            index = np.where(Beta == i + 1)
            optimal_value += c[i] * x_star_b[index].squeeze()
            row_matrix_a.append(c[i] * (A_matrix[index, :].squeeze()))
            indices_rows_matrix_a.append(Nu)
    indices_rows_matrix_a = np.asarray(indices_rows_matrix_a)
    array_row_matrix_a = np.asarray(row_matrix_a)
    summed_array_to_substitute = []
    # Second Loop to check if we have multiple rows pulled out from matrix_a. Then check the
    # coefficients for the same decision variables and then add them.
    for i in indices_rows_matrix_a[0, :]:
        temp = 0
        item = np.where(indices_rows_matrix_a == i)
        temp += np.sum(array_row_matrix_a[item].squeeze())
        if array_row_matrix_a[item].size != 0:
            summed_array_to_substitute.append(temp)

    # Now we got the summed array for the rows we substituted in original objective
    # So, Third Loop is to check and sum if we have multiple coefficients for the same
    # decision variables in this new objective function.
    summed_array_to_substitute = np.asarray(summed_array_to_substitute)
    for i in range(1, len(Nu) + 1):
        item = np.where(Nu == i)
        temp_sum = summed_array_to_substitute[item].squeeze() + c[item].squeeze()

        if temp_sum.size != 0:
            temp_c[i - 1] = summed_array_to_substitute[item].squeeze() + c[item].squeeze()
        else:
            temp_c[i - 1] = summed_array_to_substitute[i - 1].squeeze()
    z_starN = -1 * temp_c
    if np.min(z_starN) >= 0.0:
        print "\t[Optimal Solution found]"
        print "Optimal Solution is: ", optimal_value
        print "z*N: \n", z_starN
    else:
        pt1.primal_simplex_solver(z_starN, matrix_B, matrix_N, x_star_b, b, c, Beta, Nu, optimal_value)
    return z_starN, x_star_b, matrix_B, matrix_N, Nu, Beta, optimal_value
```

In this method, we have first converted the objective function into a dual feasible objective as:

$$z^*_N = [1 \quad 1] \quad \text{or} \quad c_N = [-1 \quad -1]$$

We have then called the dual solver and it returns the optimal solution for that modified problem, which is primal feasible now. Then, we have converted the objective back to the original one and then we have fed the matrices to the primal solver to find the optimal solution for the original problem.

Example 1:

Inputs:

The Inputs consist of the matrices/vectors A; b; c specifying the problem in separate csv files.

my_A4.csv

	A	B	C	D
1	2	1	1	0
2	-2	2	0	1
3				

my_b4.csv

	A
1	4
2	-2
3	

my_c4.csv

	A
1	1
2	1
3	

Running the code as:

python consolidation.py my_A4.csv my_b4.csv my_c4.csv

Output:

During each iteration we have explicit representations (that is separate matrices/vectors) B, N, Beta, Nu, x^*_B, z^*_N, B^{-1}, e_j , e_i, Delta_X_B, and Delta_Z_N as well as the scalars s and t. The code outputs the final optimal solution and associated value of the objective function.

The output looks like:

```
E:\3rd Term + TA\OR1\Project>python consolidation.py csv\my_A4.csv csv\my_b4.csv csv\my_c4.csv
Everything is consistent!
The problem is Dual and Primal Infeasible.
        [Optimal Solution found]
Optimal Solution:  [-1.]
x*B:
[[ 2.]
 [ 1.]]
z*N:
[[ 0.5]
 [ 2. ]]
B:
[[ 1.  2.]
 [ 0. -2.]]
N:
[[ 0.  1.]
 [ 1.  2.]]
Beta:  [3 1]
Nu:  [4 2]
        [Optimal Solution found]
Optimal Solution:  2.33333333333
x*B:
[[ 0.66666667]
 [ 1.66666667]]
z*N:
[[ 0.16666667]
 [ 0.66666667]]
B:
[[ 1.  2.]
 [ 2. -2.]]
N:
[[ 0.  1.]
 [ 1.  0.]]
Beta:  [2 1]
Nu:  [4 3]
```

Here, we do not have two optimal solutions (although it says so). But, we have made a function call to the Dual Solver in first phase and the Primal Solver in the next phase. Thus, the second optimal solution is actually, the optimal solution for this problem.

Example 2:

Inputs:

The Inputs consist of the matrices/vectors A; b; c specifying the problem in separate csv files.

my_A5.csv

	A	B	C	D	E
1	1	-1	1	0	0
2	-1	-1	0	1	0
3	2	1	0	0	1

my_b5.csv

	A
1	-1
2	-3
3	4

my_c5.csv

	A
1	3
2	1
3	

Running the code as:

python consolidation.py my_A5.csv my_b5.csv my_c5.csv

Output:

During each iteration we have explicit representations (that is separate matrices/vectors) B, N, Beta, Nu, x^*_B, z^*_N, B^{-1}, e_j , e_i, Delta_X_B, and Delta_Z_N as well as the scalars s and t. The code outputs the final optimal solution and associated value of the objective function.

The output looks like:

```
E:\3rd Term + TA\OR1\Project>python consolidation.py csv\my_A5.csv csv\my_b5.csv csv\my_c5.csv
Everything is consistent!
The problem is Dual and Primal Infeasible.
        [Optimal Solution found]
Optimal Solution:  -3.0
x*B:
[[ 2.]
 [ 1.]
 [ 0.]]
z*N:
[[ 1.]
 [ 0.]]
B:
[[-1.  1.  0.]
 [-1. -1.  0.]
 [ 1.  2.  1.]]
N:
[[ 0.  1.]
 [ 1.  0.]
 [ 0.  0.]]
Beta:  [2 1 5]
Nu:  [4 3]
        [Optimal Solution found]
Optimal Solution:  5.0
x*B:
[[ 2.]
 [ 1.]
 [ 0.]]
z*N:
[[ 1.33333333]
 [ 0.33333333]]
B:
[[-1.  1.  0.]
 [-1. -1.  1.]
 [ 1.  2.  0.]]
N:
[[ 0.  1.]
 [ 0.  0.]
 [ 1.  0.]]
Beta:  [2 1 4]
Nu:  [5 3]
```

Here, we do not have two optimal solutions (although it says so). But, we have made a function call to the Dual Solver in first phase and the Primal Solver in the next phase. Thus, the second optimal solution is actually, the optimal solution for this problem.

c) Unit Testing

Unit Test No. 1:

The Tests are serially implemented, meaning they are written in such a way that the second one runs only if the first test passes.

Test to check if the matrices supplied to the solvers are consistent. Then check if the problem is Dual Infeasible and Primal feasible. And, only if it is Dual Infeasible and Primal Feasible, it must run the primal solver and test its solution, if it matches to the optimal solution or not:

```
class UnitTest1(unittest.TestCase):
    """First test case."""

    # preparing to test
    def setUp(self):
        """Setting up for the test."""
        # Primal feasible problem
        self.a = np.asarray([[1, 4, 0, 1, 0, 0], [3, -1, 1, 0, 1, 0]]).astype(np.float)
        self.b = np.asarray([[1], [3]]).astype(np.float)
        self.c = np.asarray([[4], [1], [3]]).astype(np.float)

        # Optimal Solution to verify
        self.optimal_solution = 10.0
        self.x_b_solution = np.asarray([[0.25], [3.25]]).astype(np.float)
        self.z_n_solution = np.asarray([[1], [6], [3]]).astype(np.float)
        self.b_solution = np.asarray([[4, 0], [-1, 1]]).astype(np.float)
        self.n_solution = np.asarray([[1, 1, 0], [0, 3, 1]]).astype(np.float)
        self.beta_solution = [2, 3]
        self.nu_solution = [4, 1, 5]

        rows_b, cols_b = np.shape(self.b)
        rows_c, cols_c = np.shape(self.c)
        self.Nu = np.arange(1, rows_c + 1)
        self.Beta = np.arange(len(self.Nu) + 1, rows_b + len(self.Nu) + 1)

        split_a = np.hsplit(self.a, [rows_c, rows_b + rows_c])
        self.matrix_N = split_a[0]
        self.matrix_B = split_a[1]
        self.x_starB = self.b
        self.z_star_n = -1 * self.c
        self.objective = 0

    def test_1_consistency(self):
        """Check for matrices consistency."""
        rows_a, cols_a = np.shape(self.a)
        rows_b, cols_b = np.shape(self.b)
        rows_c, cols_c = np.shape(self.c)
```

```python
        rows_N, cols_N = np.shape(self.matrix_N)
        rows_B, cols_B = np.shape(self.matrix_B)
        self.assertEqual(cols_N, rows_c)
        self.assertEqual(rows_a, rows_b)
        self.assertEqual(rows_b, rows_B)
        self.assertEqual(rows_B, cols_B)
        self.assertEqual(rows_N, rows_B)
        print("Everything is consistent!")

    def test_2_primal_feasibility(self):
        """Check for feasibility."""
        self.assertLess(min(self.z_star_n), 0.0, "Sorry, the problem is not Dual Infeasible.")
        self.assertGreaterEqual(min(self.x_starB), 0.0, "Sorry, the problem is not Primal Feasible.")

    def test_3_primal_solution(self):
        """Check for Primal Optimal Solution."""
        print "Solving by Primal Simplex."
        z_starN1, x_star_b1, matrix_B1, matrix_N1, Nu1, Beta1, opt_value1 = pt1.primal_simplex_solver(self.z_s
                                                                                                   self.mat
                                                                                                   self.c,
        xb = np.dot(linalg.inv(matrix_B1), self.b)
        self.assertEqual(xb.all(), x_star_b1.all(), msg="x*b != inv(B) * b")
        self.assertGreaterEqual(z_starN1.all(), 0.00000)
        self.assertAlmostEqual(self.optimal_solution, opt_value1, 3)
        self.assertAlmostEqual(self.x_b_solution.all(), x_star_b1.all(), 3)
        self.assertAlmostEqual(self.z_n_solution.all(), z_starN1.all(), 3)
        self.assertAlmostEqual(self.b_solution.all(), matrix_B1.all(), 3)
        self.assertAlmostEqual(self.n_solution.all(), matrix_N1.all(), 3)
        self.assertEqual(all(self.beta_solution), all(Beta1))
        self.assertEqual(all(self.nu_solution), all(Nu1))

    # ending the test
    def tearDown(self):
        """Cleaning up after the test."""
        pass
```

Output:

```
E:\3rd Term + TA\OR1\Project>python unit_test_1.py -v
test_1_consistency (__main__.UnitTest1)
Check for matrices consistency. ... Everything is consistent!
ok
test_2_primal_feasibility (__main__.UnitTest1)
Check for feasibility. ... ok
test_3_primal_solution (__main__.UnitTest1)
Check for Primal Optimal Solution. ... Solving by Primal Simplex.
        [Optimal Solution found]
Optimal Solution:  [ 10.]
x*B:
[[ 0.25]
 [ 3.25]]
z*N:
[[ 1.]
 [ 6.]
 [ 3.]]
B:
[[ 4.  0.]
 [-1.  1.]]
N:
[[ 1.  1.  0.]
 [ 0.  3.  1.]]
Beta:  [2 3]
Nu:  [4 1 5]
ok

----------------------------------------------------------------------
Ran 3 tests in 0.015s

OK
```

Unit Test No. 2:

The Tests are serially implemented.

Test to check if the matrices supplied to the solvers are consistent. Then check if the problem is Primal Infeasible and Dual feasible. And, only if it is Primal Infeasible and Dual feasible, it must run the dual solver and test its solution, if it matches to the optimal solution or not:

```python
class UnitTest2(unittest.TestCase):
    """Second test case."""

    # preparing to test
    def setUp(self):
        """Setting up for the test."""
        # Dual feasible problem
        self.a = np.asarray([[-2, -1, 1, 0, 0], [-2, 4, 0, 1, 0], [-1, 3, 0, 0, 1]]).astype(np.float)
        self.b = np.asarray([[4], [-8], [-7]]).astype(np.float)
        self.c = np.asarray([[-1], [-1]]).astype(np.float)

        # Optimal Solution to verify
        self.optimal_solution = -7.0
        self.x_b_solution = np.asarray([[18], [7], [6]]).astype(np.float)
        self.z_n_solution = np.asarray([[1], [4]]).astype(np.float)
        self.b_solution = np.asarray([[1, -2, 0], [0, -2, 1], [0, -1, 0]]).astype(np.float)
        self.n_solution = np.asarray([[0, -1], [0, 4], [1, 3]]).astype(np.float)
        self.beta_solution = [3, 1, 4]
        self.nu_solution = [5, 2]

        rows_b, cols_b = np.shape(self.b)
        rows_c, cols_c = np.shape(self.c)
        self.Nu = np.arange(1, rows_c + 1)
        self.Beta = np.arange(len(self.Nu) + 1, rows_b + len(self.Nu) + 1)

        split_a = np.hsplit(self.a, [rows_c, rows_b + rows_c])
        self.matrix_N = split_a[0]
        self.matrix_B = split_a[1]
        self.x_starB = self.b
        self.z_star_n = -1 * self.c
        self.objective = 0

    def test_1_consistency(self):
        """Check for matrices consistency."""
        rows_a, cols_a = np.shape(self.a)
        rows_b, cols_b = np.shape(self.b)
        rows_c, cols_c = np.shape(self.c)
```

```python
        self.assertEqual(cols_N, rows_c)
        self.assertEqual(rows_a, rows_b)
        self.assertEqual(rows_b, rows_B)
        self.assertEqual(rows_B, cols_B)
        self.assertEqual(rows_N, rows_B)
        print("Everything is consistent!")

    def test_2_dual_feasibility(self):
        """Check for feasibility."""
        self.assertLess(min(self.x_starB), 0.0, "Sorry, the problem is not Primal Infeasible.")
        self.assertGreaterEqual(min(self.z_star_n), 0.0, "Sorry, the problem is not Dual Feasible.")

    def test_2_dual_solution(self):
        """Check for Dual Optimal Solution."""
        print "Solving by Dual Simplex."
        z_starN1, x_star_b1, matrix_B1, matrix_N1, Nu1, Beta1, opt_value1 = pt2.dual_simplex_solver(self.x_starB

        xb = np.dot(linalg.inv(matrix_B1), self.b)
        self.assertEqual(xb.all(), x_star_b1.all(), msg="x*b != inv(B) * b")
        self.assertGreaterEqual(z_starN1.all(), 0.00000)
        self.assertAlmostEqual(self.optimal_solution, opt_value1, 3)
        self.assertAlmostEqual(self.x_b_solution.all(), x_star_b1.all(), 3)
        self.assertAlmostEqual(self.z_n_solution.all(), z_starN1.all(), 3)
        self.assertAlmostEqual(self.b_solution.all(), matrix_B1.all(), 3)
        self.assertAlmostEqual(self.n_solution.all(), matrix_N1.all(), 3)
        self.assertEqual(all(self.beta_solution), all(Beta1))
        self.assertEqual(all(self.nu_solution), all(Nu1))

    # ending the test
    def tearDown(self):
        """Cleaning up after the test."""
        pass
```

Output:

```
E:\3rd Term + TA\OR1\Project>python unit_test_2.py -v
test_1_consistency (__main__.UnitTest2)
Check for matrices consistency. ... Everything is consistent!
ok
test_2_dual_feasibility (__main__.UnitTest2)
Check for feasibility. ... ok
test_2_dual_solution (__main__.UnitTest2)
Check for Dual Optimal Solution. ... Solving by Dual Simplex.
        [Optimal Solution found]
Optimal Solution:  [-7.]
x*B:
[[ 18.]
 [  7.]
 [  6.]]
z*N:
[[ 1.]
 [ 4.]]
B:
[[ 1. -2.  0.]
 [ 0. -2.  1.]
 [ 0. -1.  0.]]
N:
[[ 0. -1.]
 [ 0.  4.]
 [ 1.  3.]]
Beta:  [3 1 4]
Nu:  [5 2]
ok

-------------------------------------------------------------------
Ran 3 tests in 0.015s

OK
```

Unit Test No. 3:

The Tests are serially implemented.

Test to check if the matrices supplied to the solvers are consistent. Then check if the problem is Primal Infeasible and Dual Infeasible. And, only if it is Primal Infeasible and Dual Infeasible, it must run the two-phase solver and test its solution, if it matches to the optimal solution or not:

```python
class UnitTest3(unittest.TestCase):
    """Third test case."""

    # preparing to test
    def setUp(self):
        """Setting up for the test."""
        # Infeasible problem
        self.a = np.asarray([[1, -1, 1, 0, 0], [-1, -1, 0, 1, 0], [2, 1, 0, 0, 1]]).astype(np.float)
        self.b = np.asarray([[-1], [-3], [4]]).astype(np.float)
        self.c = np.asarray([[3], [1]]).astype(np.float)

        # Optimal Solution to verify
        self.optimal_solution = 5.0
        self.x_b_solution = np.asarray([[2], [1], [0]]).astype(np.float)
        self.z_n_solution = np.asarray([[1.3333], [0.3333]]).astype(np.float)
        self.b_solution = np.asarray([[-1, 1, 0], [-1, -1, 1], [1, 2, 0]]).astype(np.float)
        self.n_solution = np.asarray([[0, 1], [0, 0], [1, 0]]).astype(np.float)
        self.beta_solution = [2, 1, 4]
        self.nu_solution = [5, 3]

        rows_b, cols_b = np.shape(self.b)
        rows_c, cols_c = np.shape(self.c)
        self.Nu = np.arange(1, rows_c + 1)
        self.Beta = np.arange(len(self.Nu) + 1, rows_b + len(self.Nu) + 1)

        split_a = np.hsplit(self.a, [rows_c, rows_b + rows_c])
        self.matrix_N = split_a[0]
        self.matrix_B = split_a[1]
        self.x_starB = self.b
        self.z_star_n = -1 * self.c
        self.objective = 0

    def test_1_consistency(self):
        """Check for matrices consistency."""
        rows_a, cols_a = np.shape(self.a)
        rows_b, cols_b = np.shape(self.b)
        rows_c, cols_c = np.shape(self.c)
```

```
    rows_N, cols_N = np.shape(self.matrix_N)
    rows_B, cols_B = np.shape(self.matrix_B)
    self.assertEqual(cols_N, rows_c)
    self.assertEqual(rows_a, rows_b)
    self.assertEqual(rows_b, rows_B)
    self.assertEqual(rows_B, cols_B)
    self.assertEqual(rows_N, rows_B)
    print("Everything is consistent!")

def test_2_two_phase_feasibility(self):
    """Check for feasibility."""
    self.assertLess(min(self.z_star_n), 0.0, "Sorry, the problem is not Dual Infeasible.")
    self.assertLess(min(self.x_starB), 0.0, "Sorry, the problem is not Primal Infeasible.")

def test_3_two_phase_solution(self):
    """Check for Two Phase Optimal Solution."""
    print "Solving by Two Phase Simplex."
    z_starN1, x_star_b1, matrix_B1, matrix_N1, Nu1, Beta1, opt_value1 = pt3.primal_dual_simplex_solver(self.z
                                                                                                         self.m
                                                                                                         self.c
    xb = np.dot(linalg.inv(matrix_B1), self.b)
    self.assertEqual(xb.all(), x_star_b1.all(), msg="x*b != inv(B) * b")
    self.assertGreaterEqual(z_starN1.all(), 0.00000)
    self.assertAlmostEqual(self.optimal_solution, opt_value1, 3)
    self.assertAlmostEqual(self.x_b_solution.all(), x_star_b1.all(), 3)
    self.assertAlmostEqual(self.z_n_solution.all(), z_starN1.all(), 3)
    self.assertAlmostEqual(self.b_solution.all(), matrix_B1.all(), 3)
    self.assertAlmostEqual(self.n_solution.all(), matrix_N1.all(), 3)
    self.assertEqual(all(self.beta_solution), all(Beta1))
    self.assertEqual(all(self.nu_solution), all(Nu1))

# ending the test
def tearDown(self):
    """Cleaning up after the test."""
    pass
```

Output:

```
E:\3rd Term + TA\OR1\Project>python unit_test_3.py -v
test_1_consistency (__main__.UnitTest3)
Check for matrices consistency. ... Everything is consistent!
ok
test_2_two_phase_feasibility (__main__.UnitTest3)
Check for feasibility. ... ok
test_3_two_phase_solution (__main__.UnitTest3)
Check for Two Phase Optimal Solution. ... Solving by Two Phase Simplex.
        [Optimal Solution found]
Optimal Solution:  -3.0
x*B:
[[ 2.]
 [ 1.]
 [ 0.]]
z*N:
[[ 1.]
 [ 0.]]
B:
[[-1.  1.  0.]
 [-1. -1.  0.]
 [ 1.  2.  1.]]
N:
[[ 0.  1.]
 [ 1.  0.]
 [ 0.  0.]]
Beta:  [2 1 5]
Nu:  [4 3]
        [Optimal Solution found]
Optimal Solution:  5.0
x*B:
[[ 2.]
 [ 1.]
 [ 0.]]
z*N:
[[ 1.33333333]
 [ 0.33333333]]
B:
[[-1.  1.  0.]
 [-1. -1.  1.]
 [ 1.  2.  0.]]
N:
[[ 0.  1.]
 [ 0.  0.]
 [ 1.  0.]]
Beta:  [2 1 4]
Nu:  [5 3]
ok

----------------------------------------------------------------------
Ran 3 tests in 0.031s

OK
```

Unit Test No. 4:

The Tests are serially implemented.

Test to check “Certificate of Optimality”. First, I took a Primal Feasible problem and converted it into its Dual, changed the signs of inequalities and changed Min to Max. Then, recorded the solutions for Primal and Dual, which are complementary of each other.

```
class UnitTest4(unittest.TestCase):
    """Fourth test case."""

    # preparing to test
    def setUp(self):
        """Setting up for the test."""
        # Primal Problem
        self.a = np.asarray([[1, 4, 0, 1, 0, 0], [3, -1, 1, 0, 1, 0]]).astype(np.float)
        self.b = np.asarray([[1], [3]]).astype(np.float)
        self.c = np.asarray([[4], [1], [3]]).astype(np.float)
        rows_b, cols_b = np.shape(self.b)
        rows_c, cols_c = np.shape(self.c)
        self.Nu = np.arange(1, rows_c + 1)
        self.Beta = np.arange(len(self.Nu) + 1, rows_b + len(self.Nu) + 1)
        split_a = np.hsplit(self.a, [rows_c, rows_b + rows_c])
        self.matrix_N = split_a[0]
        self.matrix_B = split_a[1]
        self.x_starB = self.b
        self.z_star_n = -1 * self.c
        self.objective = 0

        # Converted the Primal problem into Dual
        self.a1 = np.asarray([[-1, -3, 1, 0, 0], [-4, 1, 0, 1, 0], [0, -1, 0, 0, 1]]).astype(np.float)
        self.b1 = np.asarray([[-4], [-1], [-3]]).astype(np.float)
        self.c1 = np.asarray([[-1], [-3]]).astype(np.float)
        rows_b1, cols_b1 = np.shape(self.b1)
        rows_c1, cols_c1 = np.shape(self.c1)
        self.Nu1 = np.arange(1, rows_c1 + 1)
        self.Beta1 = np.arange(len(self.Nu1) + 1, rows_b1 + len(self.Nu1) + 1)
        split_a1 = np.hsplit(self.a1, [rows_c1, rows_b1 + rows_c1])
        self.matrix_N1 = split_a1[0]
        self.matrix_B1 = split_a1[1]
        self.x_starB1 = self.b1
        self.z_star_n1 = -1 * self.c1
        self.objective = 0
```

```
        # Optimal Solution to verify
        self.optimal_solution = 10.0
        self.x_b_solution = np.asarray([[0.25], [3.25]]).astype(np.float)
        self.z_n_solution = np.asarray([[1], [6], [3]]).astype(np.float)

    def test_1_consistency(self):
        """Check for matrices consistency."""
        rows_a, cols_a = np.shape(self.a)
        rows_b, cols_b = np.shape(self.b)
        rows_c, cols_c = np.shape(self.c)
        rows_N, cols_N = np.shape(self.matrix_N)
        rows_B, cols_B = np.shape(self.matrix_B)
        self.assertEqual(cols_N, rows_c)
        self.assertEqual(rows_a, rows_b)
        self.assertEqual(rows_b, rows_B)
        self.assertEqual(rows_B, cols_B)
        self.assertEqual(rows_N, rows_B)
        print("Everything is consistent!")

    def test_2_primal_feasibility(self):
        """Check for feasibility."""
        self.assertLess(min(self.z_star_n), 0.0, "Sorry, the problem is not Dual Infeasible.")
        self.assertGreaterEqual(min(self.x_starB), 0.0, "Sorry, the problem is not Primal Feasible.")

    def test_3_certificate_of_optimality(self):
        """Check for Primal Optimal Solution."""
        print "Solving by Primal Simplex."
        z_starN1, x_star_b1, matrix_B1, matrix_N1, Nu1, Beta1, opt_value1 = pt1.primal_simplex_solver(self.z_star_n,
        self.assertAlmostEqual(self.optimal_solution, opt_value1, 3)
        self.assertAlmostEqual(self.x_b_solution.all(), x_star_b1.all(), 3)
        self.assertAlmostEqual(self.z_n_solution.all(), z_starN1.all(), 3)

    def test_4_certificate_of_optimality(self):
        """Check for Dual Optimal Solution."""
        print "Solving by Dual Simplex."
        z_starN1, x_star_b1, matrix_B1, matrix_N1, Nu1, Beta1, opt_value1 = pt2.dual_simplex_solver(self.x_starB1, se
        self.assertAlmostEqual(self.optimal_solution, -1 * opt_value1, 3)
        self.assertAlmostEqual(self.x_b_solution.all(), x_star_b1.all(), 3)
        self.assertAlmostEqual(self.z_n_solution.all(), z_starN1.all(), 3)

    # ending the test
    def tearDown(self):
        """Cleaning up after the test."""
        pass
```

Output:

```
E:\3rd Term + TA\OR1\Project>python unit_test_4.py -v
test_1_consistency (__main__.UnitTest4)
Check for matrices consistency. ... Everything is consistent!
ok
test_2_primal_feasibility (__main__.UnitTest4)
Check for feasibility. ... ok
test_3_certificate_of_optimality (__main__.UnitTest4)
Check for Primal Optimal Solution. ... Solving by Primal Simplex.
        [Optimal Solution found]
Optimal Solution:  [ 10.]
x*B:
[[ 0.25]
 [ 3.25]]
z*N:
[[ 1.]
 [ 6.]
 [ 3.]]
B:
[[ 4.  0.]
 [-1.  1.]]
N:
[[ 1.  1.  0.]
 [ 0.  3.  1.]]
Beta:  [2 3]
Nu:  [4 1 5]
ok
test_4_certificate_of_optimality (__main__.UnitTest4)
Check for Dual Optimal Solution. ... Solving by Dual Simplex.
        [Optimal Solution found]
Optimal Solution:  -10.0
x*B:
[[ 1.]
 [ 6.]
 [ 3.]]
z*N:
[[ 0.25]
 [ 3.25]]
B:
[[-1.  1. -3.]
 [-4.  0.  1.]
 [ 0.  0. -1.]]
N:
[[ 0.  0.]
 [ 1.  0.]
 [ 0.  1.]]
Beta:  [1 3 2]
Nu:  [4 5]
ok

----------------------------------------------------------------------
Ran 4 tests in 0.037s

OK
```

Unit Test No. 5:

The Tests are serially implemented.

Test to check "Complementary Slackness". First, I took a Primal Feasible problem and converted it into its Dual, changed the signs of inequalities and changed Min to Max. Then, recorded the solutions of Beta and Nu vectors for Primal and Dual, which are complementary of each other.

We know that:

$(x_1, \ldots, x_n, w_1, \ldots, w_m) \dashrightarrow (x_1, \ldots, x_n, x_{n+1}, \ldots, x_{n+m})$. *and*

$(z_1, \ldots, z_n, y_1, \ldots, y_m) \dashrightarrow (z_1, \ldots, z_n, z_{n+1}, \ldots, z_{n+m})$.

Therefore, if we have x1 and x2 as decision variables and x3, x4 and x5 as slacks initially:

After finding optimal solution for primal, Beta = [2, 3] and Nu = [4, 1, 5], which means that we have x2, x3 have non-zero value and x4, x1, x5 have zero value. Then, according to complementary slackness we should have in Dual, y1, z1, y2 as non-zero and z2, z3 as zero, where z1, z2 and z3 are slack and y1, y2 are decision variables. This implies that in Dual, z4, z1, z5 must be non-zero and z2, z3 must be zero.

x2, x3 != 0 and z2 and z3 == 0

x4, x1, x5 == 0 and z4, z1, z5 != 0

```
    """Fifth test case."""
    # preparing to test
    def setUp(self):
        """Setting up for the test."""
        # Primal Problem
        self.a = np.asarray([[1, 4, 0, 1, 0, 0], [3, -1, 1, 0, 1, 0]]).astype(np.float)
        self.b = np.asarray([[1], [3]]).astype(np.float)
        self.c = np.asarray([[4], [1], [3]]).astype(np.float)
        rows_b, cols_b = np.shape(self.b)
        rows_c, cols_c = np.shape(self.c)
        self.Nu = np.arange(1, rows_c + 1)
        self.Beta = np.arange(len(self.Nu) + 1, rows_b + len(self.Nu) + 1)
        split_a = np.hsplit(self.a, [rows_c, rows_b + rows_c])
        self.matrix_N = split_a[0]
        self.matrix_B = split_a[1]
        self.x_starB = self.b
        self.z_star_n = -1 * self.c
        self.objective = 0

        # Converted the Primal problem into Dual
        self.a1 = np.asarray([[-1, -3, 1, 0, 0], [-4, 1, 0, 1, 0], [0, -1, 0, 0, 1]]).astype(np.float)
        self.b1 = np.asarray([[-4], [-1], [-3]]).astype(np.float)
        self.c1 = np.asarray([[-1], [-3]]).astype(np.float)
        rows_b1, cols_b1 = np.shape(self.b1)
        rows_c1, cols_c1 = np.shape(self.c1)
        self.Nu1 = np.arange(1, rows_c1 + 1)
        self.Beta1 = np.arange(len(self.Nu1) + 1, rows_b1 + len(self.Nu1) + 1)
        split_a1 = np.hsplit(self.a1, [rows_c1, rows_b1 + rows_c1])
        self.matrix_N1 = split_a1[0]
        self.matrix_B1 = split_a1[1]
        self.x_starB1 = self.b1
        self.z_star_n1 = -1 * self.c1
        self.objective = 0

        # Complementary Slackness to verify
        self.beta_solution = [2, 3]
        self.nu_solution = [4, 1, 5]
        self.beta_solution_for_dual = [1, 3, 2]
        self.nu_solution_for_dual = [4, 5]
```

```
def test_1_consistency(self):
    """Check for matrices consistency."""
    rows_a, cols_a = np.shape(self.a)
    rows_b, cols_b = np.shape(self.b)
    rows_c, cols_c = np.shape(self.c)

    rows_N, cols_N = np.shape(self.matrix_N)
    rows_B, cols_B = np.shape(self.matrix_B)
    self.assertEqual(cols_N, rows_c)
    self.assertEqual(rows_a, rows_b)
    self.assertEqual(rows_b, rows_B)
    self.assertEqual(rows_B, cols_B)
    self.assertEqual(rows_N, rows_B)
    print("Everything is consistent!")

def test_2_primal_feasibility(self):
    """Check for feasibility."""
    self.assertLess(min(self.z_star_n), 0.0, "Sorry, the problem is not Dual Infeasible.")
    self.assertGreaterEqual(min(self.x_starB), 0.0, "Sorry, the problem is not Primal Feasible.")

def test_3_complementary_slackness(self):
    """Check for Complementary Slackness."""
    print "Solving by Primal Simplex."
    z_starN1, x_star_b1, matrix_B1, matrix_N1, Nu1, Beta1, opt_value1 = pt1.primal_simplex_solver(self.z_st
    self.assertEqual(all(self.beta_solution), all(Beta1))
    self.assertEqual(all(self.nu_solution), all(Nu1))

def test_4_complementary_slackness(self):
    """Check for Complementary Slackness."""
    print "Solving by Dual Simplex."
    z_starN1, x_star_b1, matrix_B1, matrix_N1, Nu1, Beta1, opt_value1 = pt2.dual_simplex_solver(self.x_sta
    self.assertEqual(all(self.beta_solution_for_dual), all(Beta1))
    self.assertEqual(all(self.nu_solution_for_dual), all(Nu1))

# ending the test
def tearDown(self):
    """Cleaning up after the test."""
    pass
```

Output:

```
E:\3rd Term + TA\OR1\Project>python unit_test_5.py -v
test_1_consistency (__main__.UnitTest5)
Check for matrices consistency. ... Everything is consistent!
ok
test_2_primal_feasibility (__main__.UnitTest5)
Check for feasibility. ... ok
test_3_complementary_slackness (__main__.UnitTest5)
Check for Complementary Slackness. ... Solving by Primal Simplex.
        [Optimal Solution found]
Optimal Solution:  [ 10.]
x*B:
[[ 0.25]
 [ 3.25]]
z*N:
[[ 1.]
 [ 6.]
 [ 3.]]
B:
[[ 4.  0.]
 [-1.  1.]]
N:
[[ 1.  1.  0.]
 [ 0.  3.  1.]]
Beta:  [2 3]
Nu:  [4 1 5]
ok
test_4_complementary_slackness (__main__.UnitTest5)
Check for Complementary Slackness. ... Solving by Dual Simplex.
        [Optimal Solution found]
Optimal Solution:  -10.0
x*B:
[[ 1.]
 [ 6.]
 [ 3.]]
z*N:
[[ 0.25]
 [ 3.25]]
B:
[[-1.  1. -3.]
 [-4.  0.  1.]
 [ 0.  0. -1.]]
N:
[[ 0.  0.]
 [ 1.  0.]
 [ 0.  1.]]
Beta:  [1 3 2]
Nu:  [4 5]
ok

----------------------------------------------------------------------
Ran 4 tests in 0.047s

OK
```

Question 3: Latex Output

Code:

I made an extra file for this question latexfile.py, which contains and returns all the templates, writes them into a tex file and then executes the tex file to create a pdf.

Latexfile.py

```
"""Code to bind python with LaTeX."""
import os
from itertools import izip

# Variables for LaTeX templates insertion
initial_template = '''\documentclass [12pt] {article}
\usepackage{amsmath}
\usepackage{url}
\usepackage[super]{nth}
\pagestyle{plain}
\\begin{document}
'''
initial_vector_description = '''The matrix A is given by
\[
%(initial_A)s.
\]
The initial sets of basic and nonbasic indices are
\[
{\mathcal B} = \left \{ %(Beta)s \\right \}  \quad
and    \quad
{\mathcal N} = \left \{ %(Nu)s \\right \}.
\]
Corresponding to these sets, we have the submatrices of A:
\[
B = %(initial_B)s   \quad
N = %(initial_N)s,
\]
and the initial values of the basic variables are given by
\[
 x^*_{\mathcal B} = b = %(initial_xstar_b)s ,
\]
and the initial nonbasic dual variables are simply
\[
z^*_{\mathcal N} = -c_{\mathcal N} = %(initial_zstar_n)s.
\]
'''
method_primal = '''
\section{Primal Simplex Method.}
'''
method_dual = '''
\section{Dual Simplex Method.}
```

```
method_primal_dual = '''
\section{Primal-Dual Simplex Method.}
'''
step1_primal_true = '''
\subsection{ \\nth{%(count_iteration)s} Iteration.}
\\textit{Step 1. } Since \\textit{z}$^*_{\mathcal N}$ has some negative components, the current solution is not optimal.\\
'''
step2_primal_true = '''\\textit{Step 2. } Since \\textit{z}$^*_%(entering_index)s$ = %(entering_value)s and this is the mo
\[
j = %(entering_index)s.
\]
'''
step3_primal_true = '''\\textit{Step 3. }
\[
\Delta X_{\mathcal B} = B^{-1} N e_j = %(matrix_BN)s
%(matrix_ej)s
= %(matrix_delta_xb)s
.
\]
'''
step4_primal_true = '''\\textit{Step 4. }
\[
t = \left ( max \left \{ %(matrix_fraction)s \\right \} \\right )^{-1}   = %(max_ratio)s.
\]'''
step5_primal_true = '''\\textit{Step 5.} Since the ratio that achieved the maximum in Step 4 was the %(max_ratio_number)s
\[
i = %(max_ratio_index)s.
\]
'''
step6_primal_true = '''\\textit{Step 6. }
\[
\Delta z_{\mathcal N} = -\left (B^{-1} N \\right )^T e_i = - %(matrix_BN_T)s
%(matrix_ei)s
= %(matrix_delta_zn)s.
```

```
def initial(file_name, method, initial_A, Beta, Nu, initial_B, initial_N, initial_xstar_b, initial_zstar_n):
    """Initial function to begin the LaTeX document."""
    tex_file = file(file_name, "w+")
    tex_file.writelines(initial_template)
    if method == "primal":
        tex_file.writelines(method_primal)
    elif method == "dual":
        tex_file.writelines(method_dual)
    else:
        tex_file.writelines(method_primal_dual)
    initial_A = display_matrix(initial_A)
    initial_B = display_matrix(initial_B)
    initial_N = display_matrix(initial_N)
    Beta = str(Beta.tolist()).strip('[]')
    Nu = str(Nu.tolist()).strip('[]')
    initial_xstar_b = display_matrix(initial_xstar_b)
    initial_zstar_n = display_matrix(initial_zstar_n)
    string = (initial_vector_description % {'initial_A': initial_A, 'Beta': Beta, 'Nu': Nu, 'initial_B': initial_B,
                                            'initial_zstar_n': initial_zstar_n})
    tex_file.writelines(string)
    return tex_file


def step1_primal(tex_file, count_iteration):
    """Step 1 of Primal, if any element in z_star_n is negative."""
    string = (step1_primal_true % {'count_iteration': count_iteration})
    tex_file.writelines(string)


def step1_primal_over(tex_file, matrix_c, optimal_value, count_iteration):
    """Last Iteration of Primal, if all elements in z_star_n are nonnegative."""
    objective_function = display_objective_function(matrix_c)
    string = (step1_primal_false % {'objective_function': objective_function, 'optimal_value': optimal_value, 'count
    tex_file.writelines(string)


def step2_primal(tex_file, entering_index, entering_value):
    """Step 2 of Primal, display the selection of entering variable in z_star_n."""
```

```python
def step3_primal(tex_file, matrix_BN, matrix_ej, matrix_delta_xb):
    """Step 3 of Primal, display the computation of Primal Step Direction (matrix_delta_xb)."""
    matrix_BN = display_matrix(matrix_BN)
    matrix_ej = display_matrix(matrix_ej)
    matrix_delta_xb = display_matrix(matrix_delta_xb)
    string = (step3_primal_true % {'matrix_BN': matrix_BN, 'matrix_ej': matrix_ej, 'matrix_delta_xb': matrix_delta_xb})
    tex_file.writelines(string)


def step4_primal(tex_file, matrix_delta_xb, matrix_x_star_b, max_ratio):
    """Step 4 of Primal, display the computation of Primal Step Length (max_ratio)."""
    matrix_fraction = display_fractions(matrix_delta_xb, matrix_x_star_b)
    string = (step4_primal_true % {'matrix_fraction': matrix_fraction, 'max_ratio': max_ratio})
    tex_file.writelines(string)


def step5_primal(tex_file, max_ratio_number, max_ratio_index):
    """Step 5 of Primal, display the selection of Leaving Variable (max_ratio_number)."""
    string = (step5_primal_true % {'max_ratio_number': max_ratio_number, 'max_ratio_index': max_ratio_index})
    tex_file.writelines(string)


def step6_primal(tex_file, matrix_BN_T, matrix_ei, matrix_delta_zn):
    """Step 6 of Primal, display the computation of Dual Step Direction deltazN (matrix_delta_zn)."""
    matrix_BN_T = display_matrix(matrix_BN_T)
    matrix_ei = display_matrix(matrix_ei)
    matrix_delta_zn = display_matrix(matrix_delta_zn)
    string = (step6_primal_true % {'matrix_BN_T': matrix_BN_T, 'matrix_ei': matrix_ei, 'matrix_delta_zn': matrix_delta_zn})
    tex_file.writelines(string)


def step7_primal(tex_file, z_star_j, delta_z_j, s):
    """Step 7 of Primal, display the computation of Dual Step Length (s)."""
    fraction_zstar_delta = "\\frac{{ {} }}{{ {} }} = {}".format(z_star_j, delta_z_j, s)
    string = (step7_primal_true % {'fraction_zstar_delta': fraction_zstar_delta})
    tex_file.writelines(string)



def step9_primal(tex_file, Beta, Nu, matrix_B, matrix_N, x_star_b, z_starN):
    """Step 9 of Primal, display the updation of Basis."""
    matrix_xstar_indexedBeta = display_matrix_with_indices(Beta, "x")
    matrix_zstar_indexedNu = display_matrix_with_indices(Nu, "z")
    Beta = str(Beta.tolist()).strip('[]')
    Nu = str(Nu.tolist()).strip('[]')
    matrix_B = display_matrix(matrix_B)
    matrix_N = display_matrix(matrix_N)
    matrix_xstar_final = display_matrix(x_star_b)
    matrix_zstar_final = display_matrix(z_starN)
    string = (step9_primal_true % {'matrix_B': matrix_B, 'matrix_N': matrix_N, 'matrix_xstar_final': matri
                                   'matrix_xstar_indexedBeta': matrix_xstar_indexedBeta, 'matrix_zstar_inde
    tex_file.writelines(string)


def end_document(tex_file, file_name):
    """Last Function to end the document and execute the tex file."""
    last_line = "\end{document}"
    tex_file.writelines(last_line)
    tex_file.close()

    # Command to create a pdf
    os.system("pdflatex {}".format(file_name))

    # Cleaning unnecessary files
    os.system('rm *.dvi *.ps')
    if os.path.isfile(file_name.replace('.tex', '.log')):
        os.system('rm *.log')
    if os.path.isfile(file_name.replace('.tex', '.aux')):
        os.system('rm *.aux')
    if os.path.isfile(file_name.replace('.tex', '.bbl')):
        os.system('rm *.bbl')
    if os.path.isfile(file_name.replace('.tex', '.blg')):
        os.system('rm *.blg')
```

Also, used some helper functions to fetch the templates for Matrices and fractions:

```
def display_matrix_with_indices(arrays, x_or_z):
    """Helper function to display a matrix having x* or z* with indices in LaTeX."""
    temp = []
    temp.append("\\begin{bmatrix}")
    for i in arrays:
        if x_or_z == "x":
            strn = "x^*_" + str(i)
        elif x_or_z == "z":
            strn = "z^*_" + str(i)
        temp.append(strn)
        temp.append('\\\\')
    temp.pop(-1)
    temp.append("\end{bmatrix}")
    string = ' '.join(str(i) for i in temp)
    return string


def display_matrix(matrix):
    """Helper function to display a matrix in LaTeX."""
    temp = []
    temp.append("\\begin{bmatrix}")
    for i in matrix:
        for j in range(len(i) - 1):
            temp.append(i[j])
            temp.append('&')
        temp.append(i[len(i) - 1])
        temp.append('\\\\')
    temp.append("\end{bmatrix}")
    string = ' '.join(str(i) for i in temp)
    return string


def display_fractions(matrix_a, matrix_b):
    """Helper function to display fractions between two matrices in LaTeX."""
    temp = []
    for i, j in izip(matrix_a, matrix_b):
        strn = "\\frac{" + str(float(i)) + "}{" + str(float(j)) + "}"
        temp.append(strn)
        temp.append(",")
```

After this, in Primal solver, I made function calls to the methods defined in the above file like:

```python
from numpy import linalg
import latexfile as tex

file_name = "Primal_Solver.tex"


def choose_smaller_subscript(itemindex):
    """Helper Function to choose the smaller subscript."""
    if len(itemindex[0]) > 1:
        itemindex = tuple(np.asarray([[itemindex[0][0]], [itemindex[1][0]]]))
    return itemindex


def primal_simplex_solver(z_starN, matrix_B, matrix_N, x_star_b, b, c, Beta, Nu, optimal_value):
    """Primal Problem solver."""
    # Step 1: compute the optimal solution till zN < 0, if zN >= 0 then stop
    count_iteration = 0
    count_iteration += 1
    initial_A = np.dot(linalg.inv(matrix_B), matrix_N)
    tex_file = tex.initial(file_name, "primal", initial_A, Beta, Nu, matrix_B, matrix_N, x_star_b, z_starN)
    while np.min(z_starN) < 0.0:
        # Step 2: Pick an index j in Nu for which min(z*j) < 0 (entering variable).
        tex.step1_primal(tex_file, count_iteration)

        itemindex_j = np.where(z_starN == np.min(z_starN))
        itemindex_j = choose_smaller_subscript(itemindex_j)
        j = Nu[itemindex_j[0]].squeeze()    # step 2 done
        tex.step2_primal(tex_file, j, np.min(z_starN))

        # Step 3: Compute Primal Step Direction delta_x_b
        # Initialize e_j and e_i
        e_j = np.zeros(z_starN.shape)
        e_i = np.zeros(b.shape)
        e_j[itemindex_j[0]] = 1
        mult = np.dot(linalg.inv(matrix_B), matrix_N)
        delta_x_b = np.dot(mult, e_j)
        tex.step3_primal(tex_file, mult, e_j, delta_x_b)

        # Suppress any divide by zero warnings
        import warnings
```

```
        # Step 5: The leaving variable is chosen with the max ratio.
        itemindex = np.where(temp == np.max(temp))
        tex.step4_primal(tex_file, delta_x_b, x_star_b, t)
        itemindex = choose_smaller_subscript(itemindex)
        i = Beta[itemindex[0]].squeeze()   # step 5 done
        tex.step5_primal(tex_file, itemindex[0].squeeze() + 1, i)

        # Step 6: Compute Dual Step Direction delta_zN.
        e_i[itemindex[0]] = 1
        mult = np.transpose(np.dot(linalg.inv(matrix_B), matrix_N))
        delta_z_Nu = -1 * (np.dot(mult, e_i))
        tex.step6_primal(tex_file, mult, e_i, delta_z_Nu)

        # Step 7: Compute Dual Step Length.
        z_star_j = z_starN[itemindex_j[0]].squeeze()
        delta_z_j = delta_z_Nu[itemindex_j[0]].squeeze()
        s = z_star_j / delta_z_j
        tex.step7_primal(tex_file, z_star_j, delta_z_j, s)

        # Step 8: Update Current Primal and Dual Solutions.
        # Check Degeneracy: if the ratio is infinite then no updation of x*B.
        if any(ob == float('inf') for ob in temp):
            x_star_b = x_star_b
        else:
            matrix_xstar_b = x_star_b - np.dot(t, delta_x_b)
        matrix_zstar_n = z_starN - np.dot(s, delta_z_Nu)
        tex.step8_primal(tex_file, t, s, j, t, x_star_b, delta_x_b, matrix_xstar_b, i, s, z_starN, delta_z_I
        matrix_zstar_n[itemindex_j[0]] = s
        if any(ob == float('inf') for ob in temp):
            x_star_b = x_star_b
        else:
            matrix_xstar_b[itemindex[0]] = t

        # Step 9: Update Basis.
        Beta[itemindex[0]] = j
        Nu[itemindex_j[0]] = i
        matrix_N[:, itemindex_j[0]], matrix_B[:, itemindex[0]] = matrix_B[:, itemindex[0]], matrix_N[:, iter
        x_star_b = matrix_xstar_b
        z_starN = matrix_zstar_n
        tex.step9_primal(tex_file, Beta, Nu, matrix_B, matrix_N, x_star_b, z_starN)

    # Objective Function Comptation [c_B]'*[B^-1]*b
    optimal_value = 0
    print "\t[Optimal Solution found]"
    for i in range(len(c)):
        if i + 1 in Beta:
            index = np.where(Beta == i + 1)
            optimal_value += c[i].squeeze() * x_star_b[index].squeeze()
        else:
            optimal_value += c[i] * 0
    print "Optimal Solution: ", optimal_value
    print "x*B: \n", x_star_b
    print "z*N: \n", z_starN
    print "B: \n", matrix_B
    print "N: \n", matrix_N
    print "Beta: ", Beta
    print "Nu: ", Nu
    tex.step1_primal_over(tex_file, c, optimal_value, count_iteration)
    tex.end_document(tex_file, file_name)
    return z_starN, x_star_b, matrix_B, matrix_N, Nu, Beta, optimal_value
```

Output:

Following are two outputs of different Linear Programming problems:

1 Primal Simplex Method.

The matrix A is given by

$$\begin{bmatrix} 1.0 & 4.0 & 0.0 \\ 3.0 & -1.0 & 1.0 \end{bmatrix}.$$

The initial sets of basic and nonbasic indices are

$$\mathcal{B} = \{4, 5\} \quad \textit{and} \quad \mathcal{N} = \{1, 2, 3\}.$$

Corresponding to these sets, we have the submatrices of A:

$$B = \begin{bmatrix} 1.0 & 0.0 \\ 0.0 & 1.0 \end{bmatrix} \quad N = \begin{bmatrix} 1.0 & 4.0 & 0.0 \\ 3.0 & -1.0 & 1.0 \end{bmatrix},$$

and the initial values of the basic variables are given by

$$x^*_{\mathcal{B}} = b = \begin{bmatrix} 1.0 \\ 3.0 \end{bmatrix},$$

and the initial nonbasic dual variables are simply

$$z^*_{\mathcal{N}} = -c_{\mathcal{N}} = \begin{bmatrix} -4.0 \\ -1.0 \\ -3.0 \end{bmatrix}.$$

1.1 1^{st} Iteration.

Step 1. Since $z^*_{\mathcal{N}}$ has some negative components, the current solution is not optimal.
Step 2. Since z^*_1 = -4.0 and this is the most negative of the two nonbasic dual variables, we see that the entering index is

$$j = 1.$$

Step 3.

$$\Delta X_{\mathcal{B}} = B^{-1} N e_j = \begin{bmatrix} 1.0 & 4.0 & 0.0 \\ 3.0 & -1.0 & 1.0 \end{bmatrix} \begin{bmatrix} 1.0 \\ 0.0 \\ 0.0 \end{bmatrix} = \begin{bmatrix} 1.0 \\ 3.0 \end{bmatrix}.$$

Step 4.

$$t = \left(max \left\{ \frac{1.0}{1.0}, \frac{3.0}{3.0} \right\} \right)^{-1} = 1.0.$$

Step 5. Since the ratio that achieved the maximum in Step 4 was the 1^{st} ratio and this ratio corresponds to basis index 4, we see that

$$i = 4.$$

Step 6.

$$\Delta z_{\mathcal{N}} = -\left(B^{-1}N\right)^T e_i = -\begin{bmatrix} 1.0 & 3.0 \\ 4.0 & -1.0 \\ 0.0 & 1.0 \end{bmatrix} \begin{bmatrix} 1.0 \\ 0.0 \end{bmatrix} = \begin{bmatrix} -1.0 \\ -4.0 \\ -0.0 \end{bmatrix}.$$

Step 7.

$$s = \frac{z_j^*}{\Delta z_j} = \frac{-4.0}{-1.0} = 4.0.$$

Step 8.

$$x_1^* = 1.0, x_{\mathcal{B}}^* = \begin{bmatrix} 1.0 \\ 3.0 \end{bmatrix} - 1.0 \begin{bmatrix} 1.0 \\ 3.0 \end{bmatrix} = \begin{bmatrix} 0.0 \\ 0.0 \end{bmatrix},$$

$$z_4^* = 4.0, z_{\mathcal{N}}^* = \begin{bmatrix} -4.0 \\ -1.0 \\ -3.0 \end{bmatrix} - 4.0 \begin{bmatrix} -1.0 \\ -4.0 \\ -0.0 \end{bmatrix} = \begin{bmatrix} 0.0 \\ 15.0 \\ -3.0 \end{bmatrix}.$$

Step 9. The new sets of basic and nonbasic indices are

$$\mathcal{B} = \{1, 5\} \quad and \quad \mathcal{N} = \{4, 2, 3\}.$$

Corresponding to these sets, we have the new basic and nonbasic submatrices of A,

$$B = \begin{bmatrix} 1.0 & 0.0 \\ 3.0 & 1.0 \end{bmatrix} \quad N = \begin{bmatrix} 1.0 & 4.0 & 0.0 \\ 0.0 & -1.0 & 1.0 \end{bmatrix},$$

and the new basic primal variables and nonbasic dual variables:

$$x_{\mathcal{B}}^* = \begin{bmatrix} x_1^* \\ x_5^* \end{bmatrix} = \begin{bmatrix} 1.0 \\ 0.0 \end{bmatrix}, \quad z_{\mathcal{N}}^* = \begin{bmatrix} z_4^* \\ z_2^* \\ z_3^* \end{bmatrix} = \begin{bmatrix} 4.0 \\ 15.0 \\ -3.0 \end{bmatrix}.$$

1.2 2^{nd} Iteration.

Step 1. Since $z^*_{\mathcal{N}}$ has some negative components, the current solution is not optimal.
Step 2. Since z^*_3 = -3.0 and this is the most negative of the two nonbasic dual variables, we see that the entering index is

$$j = 3.$$

Step 3.

$$\Delta X_{\mathcal{B}} = B^{-1}Ne_j = \begin{bmatrix} 1.0 & 4.0 & 0.0 \\ -3.0 & -13.0 & 1.0 \end{bmatrix} \begin{bmatrix} 0.0 \\ 0.0 \\ 1.0 \end{bmatrix} = \begin{bmatrix} 0.0 \\ 1.0 \end{bmatrix}.$$

Step 4.

$$t = \left(max \left\{ \frac{0.0}{1.0}, \frac{1.0}{0.0} \right\} \right)^{-1} = 0.0.$$

Step 5. Since the ratio that achieved the maximum in Step 4 was the 2^{nd} ratio and this ratio corresponds to basis index 5, we see that

$$i = 5.$$

Step 6.

$$\Delta z_{\mathcal{N}} = -\left(B^{-1}N\right)^T e_i = -\begin{bmatrix} 1.0 & -3.0 \\ 4.0 & -13.0 \\ 0.0 & 1.0 \end{bmatrix} \begin{bmatrix} 0.0 \\ 1.0 \end{bmatrix} = \begin{bmatrix} 3.0 \\ 13.0 \\ -1.0 \end{bmatrix}.$$

Step 7.

$$s = \frac{z^*_j}{\Delta z_j} = \frac{-3.0}{-1.0} = 3.0.$$

Step 8.

$$x^*_3 = 0.0, x^*_{\mathcal{B}} = \begin{bmatrix} 1.0 \\ 0.0 \end{bmatrix} - 0.0 \begin{bmatrix} 0.0 \\ 1.0 \end{bmatrix} = \begin{bmatrix} 1.0 \\ 0.0 \end{bmatrix},$$

$$z^*_5 = 3.0, z^*_{\mathcal{N}} = \begin{bmatrix} 4.0 \\ 15.0 \\ -3.0 \end{bmatrix} - 3.0 \begin{bmatrix} 3.0 \\ 13.0 \\ -1.0 \end{bmatrix} = \begin{bmatrix} -5.0 \\ -24.0 \\ 0.0 \end{bmatrix}.$$

Step 9. The new sets of basic and nonbasic indices are

$$\mathcal{B} = \{1, 3\} \quad and \quad \mathcal{N} = \{4, 2, 5\}.$$

Corresponding to these sets, we have the new basic and nonbasic submatrices of A,

$$B = \begin{bmatrix} 1.0 & 0.0 \\ 3.0 & 1.0 \end{bmatrix} \quad N = \begin{bmatrix} 1.0 & 4.0 & 0.0 \\ 0.0 & -1.0 & 1.0 \end{bmatrix},$$

and the new basic primal variables and nonbasic dual variables:

$$x^*_{\mathcal{B}} = \begin{bmatrix} x^*_1 \\ x^*_3 \end{bmatrix} = \begin{bmatrix} 1.0 \\ 0.0 \end{bmatrix}, \quad z^*_{\mathcal{N}} = \begin{bmatrix} z^*_4 \\ z^*_2 \\ z^*_5 \end{bmatrix} = \begin{bmatrix} -5.0 \\ -24.0 \\ 3.0 \end{bmatrix}.$$

1.3 3rd Iteration.

Step 1. Since $z^*_{\mathcal{N}}$ has some negative components, the current solution is not optimal.
Step 2. Since z^*_2 = -24.0 and this is the most negative of the two nonbasic dual variables, we see that the entering index is

$$j = 2.$$

Step 3.

$$\Delta X_{\mathcal{B}} = B^{-1} N e_j = \begin{bmatrix} 1.0 & 4.0 & 0.0 \\ -3.0 & -13.0 & 1.0 \end{bmatrix} \begin{bmatrix} 0.0 \\ 1.0 \\ 0.0 \end{bmatrix} = \begin{bmatrix} 4.0 \\ -13.0 \end{bmatrix}.$$

Step 4.

$$t = \left(max \left\{ \frac{4.0}{1.0}, \frac{-13.0}{0.0} \right\} \right)^{-1} = 0.25.$$

Step 5. Since the ratio that achieved the maximum in Step 4 was the 1st ratio and this ratio corresponds to basis index 1, we see that

$$i = 1.$$

Step 6.

$$\Delta z_{\mathcal{N}} = -\left(B^{-1}N\right)^T e_i = -\begin{bmatrix} 1.0 & -3.0 \\ 4.0 & -13.0 \\ 0.0 & 1.0 \end{bmatrix} \begin{bmatrix} 1.0 \\ 0.0 \end{bmatrix} = \begin{bmatrix} -1.0 \\ -4.0 \\ -0.0 \end{bmatrix}.$$

Step 7.

$$s = \frac{z_j^*}{\Delta z_j} = \frac{-24.0}{-4.0} = 6.0.$$

Step 8.

$$x_2^* = 0.25, x_{\mathcal{B}}^* = \begin{bmatrix} 1.0 \\ 0.0 \end{bmatrix} - 0.25 \begin{bmatrix} 4.0 \\ -13.0 \end{bmatrix} = \begin{bmatrix} 0.0 \\ 3.25 \end{bmatrix},$$

$$z_1^* = 6.0, z_{\mathcal{N}}^* = \begin{bmatrix} -5.0 \\ -24.0 \\ 3.0 \end{bmatrix} - 6.0 \begin{bmatrix} -1.0 \\ -4.0 \\ -0.0 \end{bmatrix} = \begin{bmatrix} 1.0 \\ 0.0 \\ 3.0 \end{bmatrix}.$$

Step 9. The new sets of basic and nonbasic indices are

$$\mathcal{B} = \{2, 3\} \quad and \quad \mathcal{N} = \{4, 1, 5\}.$$

Corresponding to these sets, we have the new basic and nonbasic submatrices of A,

$$B = \begin{bmatrix} 4.0 & 0.0 \\ -1.0 & 1.0 \end{bmatrix} \quad N = \begin{bmatrix} 1.0 & 1.0 & 0.0 \\ 0.0 & 3.0 & 1.0 \end{bmatrix},$$

and the new basic primal variables and nonbasic dual variables:

$$x_{\mathcal{B}}^* = \begin{bmatrix} x_2^* \\ x_3^* \end{bmatrix} = \begin{bmatrix} 0.25 \\ 3.25 \end{bmatrix}, \quad z_{\mathcal{N}}^* = \begin{bmatrix} z_4^* \\ z_1^* \\ z_5^* \end{bmatrix} = \begin{bmatrix} 1.0 \\ 6.0 \\ 3.0 \end{bmatrix}.$$

1.4 4th Iteration.

Step 1. Since $z_{\mathcal{N}}^*$ has all nonnegative components, the current solution is optimal. The optimal objective function value is

$$\zeta^* = 4.0x_1^* + 1.0x_2^* + 3.0x_3^* = [10.]$$

1 Primal Simplex Method.

The matrix A is given by

$$\begin{bmatrix} 2.0 & 3.0 & 1.0 \\ 4.0 & 1.0 & 2.0 \\ 3.0 & 4.0 & 2.0 \end{bmatrix}.$$

The initial sets of basic and nonbasic indices are

$$\mathcal{B} = \{4, 5, 6\} \quad \textit{and} \quad \mathcal{N} = \{1, 2, 3\}.$$

Corresponding to these sets, we have the submatrices of A:

$$B = \begin{bmatrix} 1.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 1.0 \end{bmatrix} \qquad N = \begin{bmatrix} 2.0 & 3.0 & 1.0 \\ 4.0 & 1.0 & 2.0 \\ 3.0 & 4.0 & 2.0 \end{bmatrix},$$

and the initial values of the basic variables are given by

$$x^*_{\mathcal{B}} = b = \begin{bmatrix} 5.0 \\ 11.0 \\ 8.0 \end{bmatrix},$$

and the initial nonbasic dual variables are simply

$$z^*_{\mathcal{N}} = -c_{\mathcal{N}} = \begin{bmatrix} -5.0 \\ -4.0 \\ -3.0 \end{bmatrix}.$$

1.1 1^{st} Iteration.

Step 1. Since $z^*_{\mathcal{N}}$ has some negative components, the current solution is not optimal.
Step 2. Since z^*_1 = -5.0 and this is the most negative of the two nonbasic dual variables, we see that the entering index is

$$j = 1.$$

Step 3.

$$\Delta X_{\mathcal{B}} = B^{-1} N e_j = \begin{bmatrix} 2.0 & 3.0 & 1.0 \\ 4.0 & 1.0 & 2.0 \\ 3.0 & 4.0 & 2.0 \end{bmatrix} \begin{bmatrix} 1.0 \\ 0.0 \\ 0.0 \end{bmatrix} = \begin{bmatrix} 2.0 \\ 4.0 \\ 3.0 \end{bmatrix}.$$

Step 4.

$$t = \left(max \left\{ \frac{2.0}{5.0}, \frac{4.0}{11.0}, \frac{3.0}{8.0} \right\} \right)^{-1} = 2.5.$$

Step 5. Since the ratio that achieved the maximum in Step 4 was the 1 ratio and this ratio corresponds to basis index 4, we see that

$$i = 4.$$

Step 6.

$$\Delta z_{\mathcal{N}} = -\left(B^{-1}N\right)^T e_i = -\begin{bmatrix} 2.0 & 4.0 & 3.0 \\ 3.0 & 1.0 & 4.0 \\ 1.0 & 2.0 & 2.0 \end{bmatrix} \begin{bmatrix} 1.0 \\ 0.0 \\ 0.0 \end{bmatrix} = \begin{bmatrix} -2.0 \\ -3.0 \\ -1.0 \end{bmatrix}.$$

Step 7.

$$s = \frac{z_j^*}{\Delta z_j} = \frac{-5.0}{-2.0} = 2.5.$$

Step 8.

$$x_1^* = 2.5, x_{\mathcal{B}}^* = \begin{bmatrix} 5.0 \\ 11.0 \\ 8.0 \end{bmatrix} - 2.5 \begin{bmatrix} 2.0 \\ 4.0 \\ 3.0 \end{bmatrix} = \begin{bmatrix} 0.0 \\ 1.0 \\ 0.5 \end{bmatrix},$$

$$z_4^* = 2.5, z_{\mathcal{N}}^* = \begin{bmatrix} -5.0 \\ -4.0 \\ -3.0 \end{bmatrix} - 2.5 \begin{bmatrix} -2.0 \\ -3.0 \\ -1.0 \end{bmatrix} = \begin{bmatrix} 0.0 \\ 3.5 \\ -0.5 \end{bmatrix}.$$

Step 9. The new sets of basic and nonbasic indices are

$$\mathcal{B} = \{1, 5, 6\} \quad and \quad \mathcal{N} = \{4, 2, 3\}.$$

Corresponding to these sets, we have the new basic and nonbasic submatrices of A,

$$B = \begin{bmatrix} 2.0 & 0.0 & 0.0 \\ 4.0 & 1.0 & 0.0 \\ 3.0 & 0.0 & 1.0 \end{bmatrix} \qquad N = \begin{bmatrix} 1.0 & 3.0 & 1.0 \\ 0.0 & 1.0 & 2.0 \\ 0.0 & 4.0 & 2.0 \end{bmatrix},$$

and the new basic primal variables and nonbasic dual variables:

$$x_{\mathcal{B}}^* = \begin{bmatrix} x_1^* \\ x_5^* \\ x_6^* \end{bmatrix} = \begin{bmatrix} 2.5 \\ 1.0 \\ 0.5 \end{bmatrix}, \quad z_{\mathcal{N}}^* = \begin{bmatrix} z_4^* \\ z_2^* \\ z_3^* \end{bmatrix} = \begin{bmatrix} 2.5 \\ 3.5 \\ -0.5 \end{bmatrix}.$$

1.2 2^{nd} Iteration.

Step 1. Since $z^*_{\mathcal{N}}$ has some negative components, the current solution is not optimal.
Step 2. Since z^*_3 = -0.5 and this is the most negative of the two nonbasic dual variables, we see that the entering index is

$$j = 3.$$

Step 3.

$$\Delta X_{\mathcal{B}} = B^{-1}Ne_j = \begin{bmatrix} 0.5 & 1.5 & 0.5 \\ -2.0 & -5.0 & 0.0 \\ -1.5 & -0.5 & 0.5 \end{bmatrix} \begin{bmatrix} 0.0 \\ 0.0 \\ 1.0 \end{bmatrix} = \begin{bmatrix} 0.5 \\ 0.0 \\ 0.5 \end{bmatrix}.$$

Step 4.

$$t = \left(max \left\{ \frac{0.5}{2.5}, \frac{0.0}{1.0}, \frac{0.5}{0.5} \right\} \right)^{-1} = 1.0.$$

Step 5. Since the ratio that achieved the maximum in Step 4 was the 3 ratio and this ratio corresponds to basis index 6, we see that

$$i = 6.$$

Step 6.

$$\Delta z_{\mathcal{N}} = -\left(B^{-1}N\right)^T e_i = -\begin{bmatrix} 0.5 & -2.0 & -1.5 \\ 1.5 & -5.0 & -0.5 \\ 0.5 & 0.0 & 0.5 \end{bmatrix} \begin{bmatrix} 0.0 \\ 0.0 \\ 1.0 \end{bmatrix} = \begin{bmatrix} 1.5 \\ 0.5 \\ -0.5 \end{bmatrix}.$$

Step 7.

$$s = \frac{z^*_j}{\Delta z_j} = \frac{-0.5}{-0.5} = 1.0.$$

Step 8.

$$x^*_3 = 1.0, x^*_{\mathcal{B}} = \begin{bmatrix} 2.5 \\ 1.0 \\ 0.5 \end{bmatrix} - 1.0 \begin{bmatrix} 0.5 \\ 0.0 \\ 0.5 \end{bmatrix} = \begin{bmatrix} 2.0 \\ 1.0 \\ 0.0 \end{bmatrix},$$

$$z^*_6 = 1.0, z^*_{\mathcal{N}} = \begin{bmatrix} 2.5 \\ 3.5 \\ -0.5 \end{bmatrix} - 1.0 \begin{bmatrix} 1.5 \\ 0.5 \\ -0.5 \end{bmatrix} = \begin{bmatrix} 1.0 \\ 3.0 \\ 0.0 \end{bmatrix}.$$

Step 9. The new sets of basic and nonbasic indices are

$$\mathcal{B} = \{1, 5, 3\} \quad \textit{and} \quad \mathcal{N} = \{4, 2, 6\}\,.$$

Corresponding to these sets, we have the new basic and nonbasic submatrices of A,

$$B = \begin{bmatrix} 2.0 & 0.0 & 1.0 \\ 4.0 & 1.0 & 2.0 \\ 3.0 & 0.0 & 2.0 \end{bmatrix} \qquad N = \begin{bmatrix} 1.0 & 3.0 & 0.0 \\ 0.0 & 1.0 & 0.0 \\ 0.0 & 4.0 & 1.0 \end{bmatrix},$$

and the new basic primal variables and nonbasic dual variables:

$$x^*_{\mathcal{B}} = \begin{bmatrix} x^*_1 \\ x^*_5 \\ x^*_3 \end{bmatrix} = \begin{bmatrix} 2.0 \\ 1.0 \\ 1.0 \end{bmatrix}, \quad z^*_{\mathcal{N}} = \begin{bmatrix} z^*_4 \\ z^*_2 \\ z^*_6 \end{bmatrix} = \begin{bmatrix} 1.0 \\ 3.0 \\ 1.0 \end{bmatrix}.$$

1.3 3^{rd} Iteration.

Step 1. Since $z^*_{\mathcal{N}}$ has all nonnegative components, the current solution is optimal. The optimal objective function value is

$$\zeta^* = 5.0x^*_1 + 4.0x^*_2 + 3.0x^*_3 = [13.]$$

Question 4: Criss-Cross Method Implementation

Code:

Fetching the data from the csv and sending to the solver is the same as before (reading the csv as command line arguments and then converting them to matrices and vectors and sending them to Primal, Dual and Two Phase method implementation.

Now, **The Criss-Cross method** implementation looks like:

```
def choose_smaller_subscript(itemindex):
    """Helper Function to choose the smaller subscript."""
    if len(itemindex[0]) > 1:
        itemindex = tuple(np.asarray([[itemindex[0][0]], [itemindex[1][0]]]))
    return itemindex


def criss_cross_solver(z_starN, matrix_B, matrix_N, x_star_b, b, c, Beta, Nu, optimal_value):
    """Criss Cross solver."""
    # Step 1: compute the optimal solution till zN < 0 and xB < 0, stop if both are positive.
    while np.min(z_starN) < 0.0 or np.min(x_star_b) < 0.0:
        # Step 2: Pick indices in z_starN, x_star_b where the coefficients are negative.
        nonbasis_indices = np.where(z_starN < 0.0)
        basis_indices = np.where(x_star_b < 0.0)
        # Step 3: Pick the index which has smaller subscript for the decision variables (entering variable).
        nonbasic_entering_index = choose_smaller_subscript(nonbasis_indices)
        basic_leaving_index = choose_smaller_subscript(basis_indices)

        # Initialize e_j and e_i
        e_j = np.zeros(z_starN.shape)
        e_i = np.zeros(b.shape)

        # if there is a non-basic infeasible variable
        if len(nonbasic_entering_index[0]) > 0:
            z_starN, matrix_B, matrix_N, x_star_b, Beta, Nu = nb.non_basis(nonbasic_entering_index, e_i, e_j, z_sta
        else:
            z_starN, matrix_B, matrix_N, x_star_b, Beta, Nu = nb.basis(basic_leaving_index, e_i, e_j, z_starN, matr

    # Objective Function Comptation [c_B]'*[B^-1]*b
    optimal_value = 0
    print "\t[Optimal Solution found]"
    for i in range(len(c)):
        if i + 1 in Beta:
            index = np.where(Beta == i + 1)
            optimal_value += c[i].squeeze() * x_star_b[index].squeeze()
        else:
            optimal_value += c[i] * 0
    print "Optimal Solution: ", optimal value
```

```python
def non_basis(entering_index, e_i, e_j, z_starN, matrix_B, matrix_N, x_star_b, Beta, Nu):
    j = Nu[entering_index[0]].squeeze()
    e_j[entering_index[0]] = 1
    # Step 4: Compute Primal Step Direction delta_x_b
    mult = np.dot(linalg.inv(matrix_B), matrix_N)
    delta_x_b = np.dot(mult, e_j)
    # Step 5: The leaving variable is chosen as that positive variable in delta_xb which has smallest subscript.
    column_indices = np.where(delta_x_b > 0.0)
    leaving_index = cs.choose_smaller_subscript(column_indices)
    # Suppress any divide by zero warnings
    import warnings

    def fxn():
        warnings.warn("deprecated", DeprecationWarning)
    with warnings.catch_warnings():
        warnings.simplefilter("ignore")
        temp = np.array(delta_x_b / x_star_b)
        fxn()

    if any(np.isnan(ob) for ob in temp):
        index = np.where(np.isnan(temp))
        temp[index] = 0
    if np.max(temp).squeeze() <= 0:
        print "[Error]: The primal is Unbounded"
        sys.exit()

    # Step 6: Pick t as Primal Step Length.
    t = np.reciprocal(temp[leaving_index[0]].squeeze())
    i = Beta[leaving_index[0]].squeeze()
    e_i[leaving_index[0]] = 1

    # Step 7: Compute Dual Step Direction delta_zNu.
    mult = np.transpose(np.dot(linalg.inv(matrix_B), matrix_N))
    delta_z_Nu = -1 * (np.dot(mult, e_i))

    # Step 8: Pick s as Dual Step Length.
    z_star_j = z_starN[entering_index[0]].squeeze()
    delta_z_j = delta_z_Nu[entering_index[0]].squeeze()

    s = z_star_j / delta_z_j

    # Step 9: Update Current Primal and Dual Solutions.
    # Check Degeneracy: if the ratio is infinite then no updation of x*B.
    if any(ob == float('inf') for ob in temp):
        x_star_b = x_star_b
    else:
        x_star_b = x_star_b - np.dot(t, delta_x_b)
        x_star_b[leaving_index[0]] = t
    z_starN = z_starN - np.dot(s, delta_z_Nu)
    z_starN[entering_index[0]] = s

    # Step 10: Update Basis.
    Beta[leaving_index[0]] = j
    Nu[entering_index[0]] = i
    matrix_N[:, entering_index[0]], matrix_B[:, leaving_index[0]] = matrix_B[:, leaving_index[0]], matrix_N[:, entering_index[0]]
    return z_starN, matrix_B, matrix_N, x_star_b, Beta, Nu


def basis(leaving_index, e_i, e_j, z_starN, matrix_B, matrix_N, x_star_b, Beta, Nu):
    i = Beta[leaving_index[0]].squeeze()
    e_i[leaving_index[0]] = 1
    # Step 4: Compute Primal Step Direction delta_x_b
    mult = np.transpose(np.dot(linalg.inv(matrix_B), matrix_N))
    delta_z_Nu = -1 * (np.dot(mult, e_i))
    # Step 5: The entering variable is chosen as that positive variable in delta_z_Nu which has smallest subscript.
    column_indices = np.where(delta_z_Nu > 0.0)
    entering_index = cs.choose_smaller_subscript(column_indices)
    # Suppress any divide by zero warnings
    import warnings

    def fxn():
        warnings.warn("deprecated", DeprecationWarning)
    with warnings.catch_warnings():
        warnings.simplefilter("ignore")
        temp = np.array(delta_z_Nu / z_starN)
        fxn()
```

```python
if any(np.isnan(ob) for ob in temp):
    index = np.where(np.isnan(temp))
    temp[index] = 0
if np.max(temp).squeeze() <= 0:
    print "[Error]: The dual is Unbounded"
    sys.exit()

# Step 6: Pick s as Dual Step Length.
s = np.reciprocal(temp[entering_index[0]].squeeze())
j = Nu[entering_index[0]].squeeze()
e_j[entering_index[0]] = 1

# Step 7: Compute Primal Step Direction delta_x_b.
mult = np.dot(linalg.inv(matrix_B), matrix_N)
delta_x_b = np.dot(mult, e_j)

# Step 8: Pick s as Primal Step Length.
x_star_i = x_star_b[leaving_index[0]].squeeze()
delta_x_i = delta_x_b[leaving_index[0]].squeeze()
t = x_star_i / delta_x_i

# Step 9: Update Current Primal and Dual Solutions.
# Check Degeneracy: if the ratio is infinite then no updation of x*B.
if t == float('inf'):
    x_star_b = x_star_b
else:
    x_star_b = x_star_b - np.dot(t, delta_x_b)
    x_star_b[leaving_index[0]] = t
z_starN = z_starN - np.dot(s, delta_z_Nu)
z_starN[entering_index[0]] = s

# Step 10: Update Basis.
Beta[leaving_index[0]] = j
Nu[entering_index[0]] = i
matrix_N[:, entering_index[0]], matrix_B[:, leaving_index[0]] = matrix_B[:, leaving_index[0]], matrix_N[:, entering_index[0]]
return z_starN, matrix_B, matrix_N, x_star_b, Beta, Nu
```

Example 1:

Inputs:

The Inputs consist of the matrices/vectors A; b; c specifying the problem in separate csv files.

my_A4.csv

	A	B	C	D
1	2	1	1	0
2	-2	2	0	1
3				

my_b4.csv

	A
1	4
2	-2
3	

my_c4.csv

	A
1	1
2	1
3	

Running the code as:

python criss_cross_consolidation.py my_A4.csv my_b4.csv my_c4.csv

Output:

During each iteration we have explicit representations (that is separate matrices/vectors) B, N, Beta, Nu, x^*_B, z^*_N, B^{-1}, e_j, e_i, Delta_X_B, and Delta_Z_N as well as the scalars s and t. The code outputs the final optimal solution and associated value of the objective function.
The output looks like:

```
E:\3rd Term + TA\OR1\Project\Criss Cross>python criss_cross_consolidation.py ..\csv\my_A4.csv ..\csv\my_b4.csv ..\csv\my_c4.csv
Everything is consistent!
The problem is Dual and Primal Infeasible.
So, performing Criss Cross Method...
        [Optimal Solution found]
Optimal Solution:  2.33333333333
x*B:
[[ 0.66666667]
 [ 1.66666667]]
z*N:
[[ 0.16666667]
 [ 0.66666667]]
B:
[[ 1.  2.]
 [ 2. -2.]]
N:
[[ 0.  1.]
 [ 1.  0.]]
Beta:  [2 1]
Nu:  [4 3]
```

Example 2:

Inputs:

The Inputs consist of the matrices/vectors A; b; c specifying the problem in separate csv files.

my_A.csv

	A	B	C	D	E
1	1	1	1	0	0
2	-1	2	0	1	0
3	1	-3	0	0	1

my_b.csv

	A
1	6
2	-0.5
3	-1

my_c.csv

	A
1	-2
2	3
3	

Running the code as:

python criss_cross_consolidation.py my_A.csv my_b.csv my_c.csv

Output:

During each iteration we have explicit representations (that is separate matrices/vectors) B, N, Beta, Nu, x^*_B, z^*_N, B^{-1}, e_j , e_i, Delta_X_B, and Delta_Z_N as well as the scalars s and t. The code outputs the final optimal solution and associated value of the objective function.
The output looks like:

```
E:\3rd Term + TA\OR1\Project\Criss Cross>python criss_cross_consolidation.py my_A.csv my_b.csv my_c.csv
Everything is consistent!
The problem is Dual and Primal Infeasible.
So, performing Criss Cross Method...
        [Optimal Solution found]
Optimal Solution:  -2.5
x*B:
[[ 1. ]
 [ 3.5]
 [ 1.5]]
z*N:
[[ 3.]
 [ 1.]]
B:
[[ 1.  1.  1.]
 [ 0. -1.  2.]
 [ 0.  1. -3.]]
N:
[[ 0.  0.]
 [ 1.  0.]
 [ 0.  1.]]
Beta:  [3 1 2]
Nu:  [4 5]
```

Unit Testing for Criss-Cross:

This test ensures that the optimal solution computed using Criss-Cross method matches exactly with the one computed by using Two-Phase method.

```python
import project_part3 as pt3
import criss_cross as cs


class UnitTest1(unittest.TestCase):
    """First test case."""

    # preparing to test
    def setUp(self):
        """Setting up for the test."""
        # Infeasible problem
        self.a = np.asarray([[2, 1, 1, 0], [-2, 2, 0, 1]]).astype(np.float)
        self.b = np.asarray([[4], [-2]]).astype(np.float)
        self.c = np.asarray([[1], [1]]).astype(np.float)

        # Optimal Solution to verify
        self.optimal_solution = 2.3333
        self.x_b_solution = np.asarray([[0.6666], [1.6666]]).astype(np.float)
        self.z_n_solution = np.asarray([[0.1666], [0.6666]]).astype(np.float)
        self.b_solution = np.asarray([[1, 2], [2, -2]]).astype(np.float)
        self.n_solution = np.asarray([[0, 1], [1, 0]]).astype(np.float)
        self.beta_solution = [2, 1]
        self.nu_solution = [4, 3]

        rows_b, cols_b = np.shape(self.b)
        rows_c, cols_c = np.shape(self.c)
        self.Nu = np.arange(1, rows_c + 1)
        self.Beta = np.arange(len(self.Nu) + 1, rows_b + len(self.Nu) + 1)

        split_a = np.hsplit(self.a, [rows_c, rows_b + rows_c])
        self.matrix_N = split_a[0]
        self.matrix_B = split_a[1]
        self.x_starB = self.b
        self.z_star_n = -1 * self.c
        self.objective = 0

    def test_1_consistency(self):
```

```
        self.assertEqual(rows_b, rows_B)
        self.assertEqual(rows_B, cols_B)
        self.assertEqual(rows_N, rows_B)
        print("Everything is consistent!")

    def test_2_dual_primal_infeasible(self):
        """Check if the problem is both dual and primal infeasible."""
        self.assertLess(min(self.z_star_n), 0.0)
        self.assertLess(min(self.x_starB), 0.0)
        print "The problem is Dual and Primal Infeasible."

    def test_3_optimal_solution_simplex(self):
        """Check optimal solution from Simplex Method to compare with CrissCross."""
        z_starN1, x_star_b1, matrix_B1, matrix_N1, Nu1, Beta1, opt_value1 = pt3.primal_dual_simplex_solver(self.z_
        # assertAlmostEqual is used for fractions and real numbers
        self.assertAlmostEqual(self.optimal_solution, opt_value1, 3)
        self.assertAlmostEqual(self.x_b_solution.all(), x_star_b1.all(), 3)
        self.assertAlmostEqual(self.z_n_solution.all(), z_starN1.all(), 3)
        self.assertAlmostEqual(self.b_solution.all(), matrix_B1.all(), 3)
        self.assertAlmostEqual(self.n_solution.all(), matrix_N1.all(), 3)
        self.assertEqual(all(self.beta_solution), all(Beta1))
        self.assertEqual(all(self.nu_solution), all(Nu1))

    def test_4_optimal_solution_crisscross(self):
        """Check optimal solution from Criss Cross Method to compare with Simplex."""
        z_starN2, x_star_b2, matrix_B2, matrix_N2, Nu2, Beta2, opt_value2 = cs.criss_cross_solver(self.z_star_n, s
        # assertAlmostEqual is used for fractions and real numbers
        self.assertAlmostEqual(self.optimal_solution, opt_value2, 3)
        self.assertAlmostEqual(self.x_b_solution.all(), x_star_b2.all(), 3)
        self.assertAlmostEqual(self.z_n_solution.all(), z_starN2.all(), 3)
        self.assertAlmostEqual(self.b_solution.all(), matrix_B2.all(), 3)
        self.assertAlmostEqual(self.n_solution.all(), matrix_N2.all(), 3)
        self.assertEqual(all(self.beta_solution), all(Beta2))
        self.assertEqual(all(self.nu_solution), all(Nu2))

    # ending the test
    def tearDown(self):
```

Output:

```
        [Optimal Solution found]
Optimal Solution:  2.33333333333
x*B:
[[ 0.66666667]
 [ 1.66666667]]
z*N:
[[ 0.16666667]
 [ 0.66666667]]
B:
[[ 1.  2.]
 [ 2. -2.]]
N:
[[ 0.  1.]
 [ 1.  0.]]
Beta:  [2 1]
Nu:  [4 3]
.       [Optimal Solution found]
Optimal Solution:  2.33333333333
x*B:
[[ 0.66666667]
 [ 1.66666667]]
z*N:
[[ 0.16666667]
 [ 0.66666667]]
B:
[[ 1.  2.]
 [ 2. -2.]]
N:
[[ 0.  1.]
 [ 1.  0.]]
Beta:  [2 1]
Nu:  [4 3]
.
------------------------------------------
Ran 4 tests in 0.053s

OK

E:\3rd Term + TA\OR1\Project\Criss Cross>
```