PROJECT REPORT

OPERATION RESEARCH 1
(LINEAR PROGRAMMING)

Jasmeet Singh

#V00843228

Table of Contents

Introduction

This report contains the code snapshots of the implementation of the Primal Simplex, Dual Simplex method, Criss Cross Method for linear programming. It also includes the snapshot for the test file and the output file snapshot corresponding to the test. All the implementation has been done in python programming language. I have used numpy framework for storing my vectors, matrices, computing inverse of matrices and other basic mathematic computations.

Since it was not clear from the project assignment pdf, what to include in the project report, I have included the screenshots corresponding to the code I have written. I have uploaded by code on the github along with the matrices csv that I have used for testing and development purposes. The code represents my work only. ***I have also uploaded the PDF file generated by Latex code at the end of this report i.e. you will find the latex PDF attached at the end of this report.*** Since, it was not clear in the description, whether we have to make the latex file for Primal or Dual method. I have created tex file and pdf files for both the solvers. The code generates the latex file depending upon the input to the solvers.

All the latex files, Code files, test files, csv files are on the github : https://github.com/jasmeet17/lp_project

Code Snapshots (Primal and Dual Method)

```python
import sys
import numpy as np
from numpy import genfromtxt
import time
from numpy.linalg import inv
import latex_sample
import os


class LinearProgram(object):

    def __init__(self,dict_A,C_n,b):
        if len(sys.argv)!=4:
            pass
            # print "Please pass all the files."
            # self.dataEntered = False

        ### Simplex Dictionary, Vectors and variables
        self.dict_A = dict_A # Dictioanry A
        self.C_n = C_n # Vector C
        self.b = b # Vector b
        self.numberofConstraints = self.dict_A.shape[0] # m
        self.numberOfVaraibles = self.C_n.shape[0]    # n
        self.C_starts = 0
        self.B = np.identity(self.numberofConstraints)   # Identity Matrix (For Basic Vars)
        self.N = self.dict_A                         # Non-Basic columns
        self.Old_C_n = []
        self.dict_A = np.concatenate((self.N, self.B), axis=1) # [N B]

        ### track Basic-Var Columns
        self.Basic =  range(self.N.shape[1]+1, self.N.shape[1] + self.numberofConstraints+1)
        ### track Non Basic-Var Columns
        self.Non_Basic = range(1,self.N.shape[1]+1)
        # keep copy on Non_Basic for
        self.old_Non_Basic = self.Non_Basic
        # keep copy on Basic for
        self.old_Basic = self.Basic

        ### Associated Dual Dictionary
        #self.Z_n = -1 * self.C_n

        self.X_n = np.zeros(len(self.Non_Basic))
        self.X_b = inv(self.B).dot(self.b) - (inv(self.B).dot(self.N)).dot(self.X_n)

        self.C_b = np.zeros(len(self.Basic))
        self.Z_n = (np.transpose(inv(self.B).dot(self.N))).dot(self.C_b) -1 * self.C_n
```

```python
        self.dataEntered = True
        self.latex_text = latex_sample.getInitialMatrices(self.dict_A, self.Basic, self.Non_Basic, self.B, self.N, self.X_b, self.Z_n)

    # Since the problem in hand is not dual/primal feasible
    # applies the phase 1
    def phaseOne(self):

        old_NBasic=[]
        old_Basic=[]

        # keep the copy of old Objective Fucntion
        old_C_n = self.C_n

        # make new primal objective function
        self.C_n =  np.empty(self.numberOfVaraibles)
        self.C_n.fill(-1)

        # Update Z_n according to the new C_n
        self.Z_n = (np.transpose(inv(self.B).dot(self.N))).dot(self.C_b) -1 * self.C_n

        # get the initial non basic indexes array
        for x in self.old_Non_Basic:
            old_NBasic.append(x)

        # get the initial basic indexes array
        for x in self.old_Basic:
            old_Basic.append(x)

        # Now we have new primal function and the problem,
        # becomes Dual Feasible.
        self.preformDualSimplex()
        # print 'Now the problem is Primal Feasible, apply Primal Simplex'

        # "the new A is  values are:"
        temp_A = -(inv(self.B)).dot(self.N)

        # number of terms in the Objective funtion
        n_terms = temp_A.shape[1] + 1

        # new Objective Funciton
        sum_array = np.zeros(n_terms)

        for i in range(len(old_NBasic)):
            temp_array = []
```

```python
            temp_array = []
            if old_NBasic[i] in self.Basic:
                t = self.Basic.index(old_NBasic[i])
                temp_array = old_C_n[i] * np.append([self.X_b[t]],temp_A[t])
            else:
                temp_array = np.zeros(n_terms)
                temp_array[i+1] = old_C_n[i]
            sum_array += temp_array

        for i in range(len(self.C_n)):
            self.C_n[i] = sum_array[i+1]

        self.C_starts = sum_array[0]
        self.Z_n = -1 * self.C_n

        # Now we got the updated Objective function after
        # applying the phase 1, now apply primal simplex
        self.Old_C_n = old_C_n

        flag, value =  self.preformPrimalSimplex()
        return flag,value

    ### performs the Simplex method
    def preformPrimalSimplex(self):
        iteration = 1
        if not self.isVectorPositive(self.X_b):
            # print "X_b is < 0 Initial solution is not primal feasible."
            self.latex_text += latex_sample.getInitialCondition(False,'x','B','Primal')
            flag, value =self.phaseOne()
            return flag, value
        else :
            # print 'X_b >= 0 Initial solution is primal feasible.'
            self.latex_text += latex_sample.getInitialCondition(True,'x','B','Primal')

        if self.isVectorPositive(self.Z_n):
            # 'Z_n >=0 Current solution is optimal.'
            self.latex_text += latex_sample.firstStepPrimalDual(iteration,False,'z','N')
            self.printObjectiveFunction(self.Old_C_n, self.C_n)
            # print "Objective Function Value : %s" % self.getObjectiveValue(self.Old_C_n, self.C_n,self.X_b, self.Basic, self.numberOfVaraibles)
            return
        else:
            ### until theres some negative in Z_n
            # STEP 1: Check for optimality
            while not self.isVectorPositive(self.Z_n):

```

ques_1.py UNREGISTERED

criss_cross.py | ques_1.py | latex_sample.py | test.py

```python
        self.latex_text += latex_sample.firstStepPrimalDual(iteration,True,'z','N')

        # STEP 2:
        # Get the least negative number Index( i.e. entering Index)
        # from Non Basic vector
        j = self.Non_Basic[self.Z_n.argmin()]
        self.latex_text += latex_sample.secondStepPrimalDual(self.Z_n[self.Z_n.argmin()],j,'z'
            ,'j')

        # STEP 3: Calculte delata_X_b

        # to create a unit vector, with all element zero except 1
        # np.eye(value,size_of_vector,index_of Value)
        e_j = np.eye(1, len(self.Non_Basic) , self.Non_Basic.index(j))
        self.X_n = e_j[0]
        e_j = np.transpose(e_j)
        delta_X_b = (inv(self.B).dot(self.N)).dot(e_j)
        delta_X_b = np.reshape(delta_X_b,(delta_X_b.shape[0],))
        self.latex_text += latex_sample.thirdStepPrimal(3,inv(self.B).dot(self.N),e_j,
            delta_X_b,'j')

        # STEP 4: Calculate Primal Step Length
        max_val , t_index, infinte_flag = self.primalStepLength(delta_X_b,self.X_b)

        t = 0
        if infinte_flag:
            pass
        elif max_val<=0:
            return -1,'Print problem is unbounded'
        else:
            t = 1/max_val
            self.latex_text += latex_sample.fourthStepPrimalDual(delta_X_b,self.X_b,t,'t')

        # Step 5: Select Leaving Variable
        # max ratio corresponds to index from Basic (Leaving Variable)
        i = self.Basic[t_index]
        self.latex_text += latex_sample.fifthStepPrimal(i,'i')

        # STEP 6: Compute Dual Step Direction
        # to create a unit vector, with all element zero except 1
        # np.eye(value,size_of_vector,index_of Value)
        e_i = np.eye(1, len(self.Basic) , self.Basic.index(i))
        e_i = np.transpose(e_i)

        delta_Z_n = - (np.transpose((inv(self.B)).dot(self.N))).dot(e_i)
        delta_Z_n = np.reshape(delta_Z_n,(delta_Z_n.shape[0],))
```

Line 181, Column 1 Spaces: 4 Python

ques_1.py UNREGISTERED

criss_cross.py | ques_1.py | latex_sample.py | test.py

```python
        delta_Z_n = np.reshape(delta_Z_n,(delta_Z_n.shape[0],))
        self.latex_text += latex_sample.sixthStepPrimal(6,np.transpose((inv(self.B)).dot(self.
            N)),e_i,delta_Z_n,'i')

        # STEP 7: Compute Dual Step Length
        s = self.Z_n[self.Non_Basic.index(j)] / delta_Z_n[self.Non_Basic.index(j)]
        self.latex_text += latex_sample.seventhStepPrimalDual(s,self.Z_n[self.Non_Basic.index(
            j)] , delta_Z_n[self.Non_Basic.index(j)],'s','z',j)

        # STEP 8: Update Current Primal and Dual Solutions
        # if while calculating max ratio we get infinete; we don't update X with t
        if not infinte_flag:
            new_x = t
            old_X_b = self.X_b
            self.X_b = self.X_b - t * delta_X_b

        new_z = s
        old_Z_n = self.Z_n
        self.Z_n = self.Z_n - s * delta_Z_n

        if not infinte_flag:
            self.latex_text += latex_sample.eightStepPrimal(j,i,t,s,old_X_b,old_Z_n,delta_X_b,
                delta_Z_n,self.X_b,self.Z_n)

        # Step 9: Update Basis
        self.Non_Basic[self.Non_Basic.index(j)] = i
        self.Basic[self.Basic.index(i)] = j

        b_columns = self.Basic + np.array([-1.0]*len(self.Basic))
        n_columns = self.Non_Basic + np.array([-1.0]*len(self.Non_Basic))

        self.B = self.dict_A[: , b_columns.astype(np.int64)]
        self.N = self.dict_A[: , n_columns.astype(np.int64)]

        if not infinte_flag:
            self.X_b[self.Basic.index(j)] = new_x

        self.Z_n[self.Non_Basic.index(i)] = new_z

        self.latex_text += latex_sample.ninthStepPrimal(self.Basic,self.Non_Basic,self.B,self.
            N,self.X_b,self.Z_n)

        iteration+=1


    self.latex_text += latex_sample.firstStepPrimalDual(iteration,False,'z','N')
```

Line 181, Column 1 Spaces: 4 Python

ques_1.py

criss_cross.py | ques_1.py | latex_sample.py | test.py

```python
            self.latex_text += latex_sample.firstStepPrimalDual(iteration,False,'z','N')
            # self.printObjectiveFunction(self.Old_C_n, self.C_n)

            # print "Objective Function Value : %s" % self.getObjectiveValue(self.Old_C_n, self.C_n,self.X_b, self.Basic, self.numberOfVaraibles)
            return 0, self.getObjectiveValue(self.Old_C_n, self.C_n,self.X_b, self.Basic, self.numberOfVaraibles)

    """performs Dual Simplex method"""
    def preformDualSimplex(self):
        if not self.isVectorPositive(self.Z_n):
            # print "Z_n is <= 0 "
            # print "Initial solution is not Dual feasible."
            self.latex_text += latex_sample.getInitialCondition(False,'z','N','Dual')
            return
        else :
            # print 'Z_n >= 0'
            # print "Initial solution is Dual feasible."
            self.latex_text += latex_sample.getInitialCondition(True,'z','N','Dual')

        if self.isVectorPositive(self.X_b):
            # print 'X_b >=0'
            # print 'Current solution is optimal.'
            # self.printObjectiveFunction(self.Old_C_n, self.C_n)
            # print "Objective Function Value : %s" % self.getObjectiveValue(self.Old_C_n, self.C_n,self.X_b, self.Basic, self.numberOfVaraibles)
            self.latex_text += latex_sample.firstStepPrimalDual(iteration,False,'x','B')
            pass
        else:
            iteration = 1
            ### until theres some negative in Z_n
            # STEP 1: Check for optimality
            while not self.isVectorPositive(self.X_b):

                self.latex_text += latex_sample.firstStepPrimalDual(iteration,True,'x','B')
                # STEP 2:
                # Get the least negative number Index( i.e. entering Index)
                # from Non Basic vector
                i = self.Basic[self.X_b.argmin()]
                self.latex_text += latex_sample.secondStepPrimalDual(self.X_b[self.X_b.argmin()],i,'x','i')
                # STEP 3: Calculte delata_Z_n

                # to create a unit vector, with all element zero except 1
                # np.eye(value,size_of_vector,index_of Value)
                e_i = np.eye(1, len(self.Basic) , self.Basic.index(i))
```

Line 181, Column 1 | Spaces: 4 | Python

ques_1.py UNREGISTERED

criss_cross.py | ques_1.py | latex_sample.py | test.py

```python
                e_i = np.eye(1, len(self.Basic) , self.Basic.index(i))
                e_i = np.transpose(e_i)
                delta_Z_n = - (np.transpose(inv(self.B).dot(self.N))).dot(e_i)
                delta_Z_n = np.reshape(delta_Z_n,(delta_Z_n.shape[0],))
                self.latex_text += latex_sample.sixthStepPrimal(3,np.transpose((inv(self.B)).dot(self.N)),e_i,delta_Z_n,'i')

                # STEP 4: Calculate Primal Step Length
                max_val , s_index, infinte_flag = self.primalStepLength(delta_Z_n,self.Z_n)

                # if s less than zero problem in unbounded, So STOP.
                if infinte_flag:
                    pass
                elif max_val<=0:
                    return -1,'Print problem is unbounded'
                else:
                    s = 1/max_val
                    self.latex_text += latex_sample.fourthStepPrimalDual(delta_Z_n,self.Z_n,s,'s')

                # Step 5: Select Leaving Variable
                # max ratio corresponds to index from Basic (Leaving Variable)
                j = self.Non_Basic[s_index]
                self.latex_text += latex_sample.fifthStepPrimal(j,'j')


                # STEP 6: Compute Dual Step Direction
                # to create a unit vector, with all element zero except 1
                # np.eye(value,size_of_vector,index_of Value)

                e_j = np.eye(1, len(self.Non_Basic) , self.Non_Basic.index(j))
                e_j = np.transpose(e_j)
                delta_X_b = ((inv(self.B)).dot(self.N)).dot(e_j)
                delta_X_b = np.reshape(delta_X_b,(delta_X_b.shape[0],))
                self.latex_text += latex_sample.thirdStepPrimal(6,inv(self.B).dot(self.N),e_j,delta_X_b,'j')


                # STEP 7: Compute Dual Step Length
                t = self.X_b[self.Basic.index(i)] / delta_X_b[self.Basic.index(i)]
                self.latex_text += latex_sample.seventhStepPrimalDual(t,self.X_b[self.Basic.index(i)], delta_X_b[self.Basic.index(i)],'t','x',j)

                # STEP 8: Update Current Primal and Dual Solutions
                new_x = t
                self.X_b = self.X_b - t * delta_X_b
```

Line 181, Column 1 Spaces: 4 Python

ques_1.py UNREGISTERED

criss_cross.py | ques_1.py | latex_sample.py | test.py

```python
                    self.Z_n = self.Z_n - s * delta_Z_n
                    # self.latex_text += latex_sample.eightStepPrimal(j,i,t,s,old_X_b,old_Z_n,delta_X_b,delta_Z_n,self.X_b,self.Z_n)

                # Step 9: Update Basis
                self.Non_Basic[self.Non_Basic.index(j)] = i
                self.Basic[self.Basic.index(i)] = j

                b_columns = self.Basic + np.array([-1.0]*len(self.Basic))
                n_columns = self.Non_Basic + np.array([-1.0]*len(self.Non_Basic))

                self.B = self.dict_A[: , b_columns.astype(np.int64)]
                self.N = self.dict_A[: , n_columns.astype(np.int64)]

                self.X_b[self.Basic.index(j)] = new_x
                if not infinte_flag:
                    self.Z_n[self.Non_Basic.index(i)] = new_z

                iteration+=1
                self.latex_text += latex_sample.ninthStepPrimal(self.Basic,self.Non_Basic,self.B,self.N,self.X_b,self.Z_n)
                #self.Z_n = self.Z_n * 0

            self.latex_text += latex_sample.firstStepPrimalDual(iteration,False,'x','B')
            # self.printObjectiveFunction(self.Old_C_n, self.C_n)
            # print "Objective Function Value : %s" % self.getObjectiveValue(self.Old_C_n, self.C_n,self.X_b, self.Basic, self.numberOfVaraibles)

            return 0,self.getObjectiveValue(self.Old_C_n, self.C_n,self.X_b, self.Basic, self.numberOfVaraibles)

    ### Calculate Primal Step Length
    ### Divide element by element (also conider 0/0 as 0)
    ### takes the max of the resulted list and return inverse and
    ### index corresponding to max
    def primalStepLength(self,delta_x,delta_x_i):

        temp_list=[]
        infinte_index = -1

        for i in range(delta_x_i.shape[0]):
            if delta_x_i[i]==0:
                if delta_x[i]==0:
                    temp_list.append(0)
                elif delta_x[i]<0:
                    temp_list.append(0)
```

Line 181, Column 1 Spaces: 4 Python

ques_1.py UNREGISTERED

criss_cross.py × | ques_1.py × | latex_sample.py × | test.py ×

```python
                elif delta_x[i]<0:
                    temp_list.append(0)
                elif delta_x[i]>0:
                    infinte_index = i
            else:
                temp_list.append(delta_x[i]/delta_x_i[i])

        if infinte_index!=-1:
            return 0.0, infinte_index , True

        max_val = max(temp_list)

        return float(max_val), temp_list.index(max_val), False


    ### To check all elements in the vectors are positive
    ### checks Greater than or equal to zero and returns True If so
    def isVectorPositive(self,vec):

    # vec_c -> objective function
    # vec_x -> Soultion
    # basic -> index of basic vars
    # n -> number of varaibles in objective function, initially
    # returns the objective Function value
    def getObjectiveValue(self,old_c,vec_c,vec_x,basic_vec,n):

    # Prints the Objective function in the Standard form
    # and the Value obtained by the Solution
    def printObjectiveFunction(self, old_c,vec_c):

    def printAllVariables(self):



dict_A = genfromtxt(sys.argv[1], delimiter=',') # Dictioanry A
C_n = genfromtxt(sys.argv[2], delimiter=',') # Vector C
b = genfromtxt(sys.argv[3], delimiter=',') # Vector b


### create an object of Linear_Prog class
simplex = LinearProgram(dict_A,C_n,b)
# simplex.printAllVariables()
simplex.preformPrimalSimplex()
# simplex.preformDualSimplex()

```

Line 181, Column 1 Spaces: 4 Python

Code Snapshots (Latex File Generator, Python Code)

latex_helper.py

criss_cross.py | ques_1.py | latex_sample.py | latex_helper.py | test.py

```python
import numpy as np
import json

# returns the matrice suitalble for Latex format
def latexMatrice(numpy_array):
    text = ''
    if len(numpy_array.shape)==1:
        for i in range(numpy_array.shape[0]):
            text += str(numpy_array[i]) + " \\\\ "
    else:
        for i in range(numpy_array.shape[0]):
            for j in range(numpy_array.shape[1]):
                if j<numpy_array.shape[1]-1:
                    text += str(numpy_array[i][j]) + " & "
                else:
                    text += str(numpy_array[i][j]) + " \\\\ "

    return text

# return comma seprated array example [3,4,5] -> 3,4,5
def latexCommaSeprated(numpy_array):
    text = json.dumps(numpy_array,',')
    return text[1:-1]

# return output of the form # \frac{1}{1},\frac{2}{0},\frac{0}{5}
def latexRatios(num_array,den_array):
    text = ''
    for i in range(len(num_array)):
        if i < len(num_array) - 1:
            text += '\\frac{%s}{%s},' % (num_array[i],den_array[i])
        else:
            text += '\\frac{%s}{%s}' % (num_array[i],den_array[i])

    return text

# return output of the form \frac{-4}{-1}
def latexFraction(num,den):
    text = '\\frac{%s}{%s}' % (num,den)
    return text


# c contains teh letter of variable
# arr contains teh subscipts
def latexVariablesMatrice(c , arr):
    text = ''
    for i in range(len(arr)):
        text += '%s_{%s}^{*} \\\\' %(c,arr[i])
    return text
```

Line 1, Column 1 | Spaces: 4 | Python

latex_sample.py UNREGISTERED

criss_cross.py × ques_1.py × latex_sample.py ● test.py ×

```python

from latex_helper import latexMatrice
from latex_helper import latexCommaSeprated
from latex_helper import latexRatios
from latex_helper import latexFraction
from latex_helper import latexVariablesMatrice

def getInitialMatrices(matrice_A, b_indices, non_b_indices, matrice_B, matrice_N, matrice_X, matrice_Z ):
    matrice_A = latexMatrice(matrice_A)
    b_indices = latexCommaSeprated(b_indices)
    non_b_indices = latexCommaSeprated(non_b_indices)
    matrice_B = latexMatrice(matrice_B)
    matrice_N = latexMatrice(matrice_N)
    matrice_X = latexMatrice(matrice_X)
    matrice_Z = latexMatrice(matrice_Z)

    text = (initial_matrices % {'matrice_A':matrice_A, 'b_indices':b_indices, 'non_b_indices':non_b_indices,
    return text

# return string based on whether intial condition met (i.e. true else false)
def getInitialCondition(bool_value,main_ch,subscript_ch,solver_type):
    if bool_value:
        return (initial_primal_condition_true % {'main_ch':main_ch,'subscript_ch':subscript_ch,'solver_type':
    else:
        return (initial_primal_condition_false % {'main_ch':main_ch,'subscript_ch':subscript_ch, 'solver_type

# bool_value is false in case z_n has some negative; else true and algo stops
# iteration_no is the iteration number fo the algo
def firstStepPrimalDual(iteration_no,bool_value,main_ch,subscript_ch):
    text = ''
    if bool_value:
        text = (step1_primal_dual_condition_true % {'iteration_no':iteration_no, 'main_ch':main_ch,'subscript
    else:
        text = (step1_primal_dual_condition_false % {'iteration_no':iteration_no, 'main_ch':main_ch,'subscrip
    return text


# second step of primal
# argument 1 is the most negative number
# index is the index of the most negative in the nonbasic vector
def secondStepPrimalDual(negative_no, index,main_ch, var_ch):
    text = (step2_primal_dual % {'negative_no':negative_no,'index':index, 'main_ch': main_ch, 'var_ch':var_ch
    return text

# returns the latex string for the third Step of primal method
def thirdStepPrimal(step,matrice_BN, matrice_EJ,matrice_Result,subscript_ch):

    matrice_BN = latexMatrice(matrice_BN)
    matrice_EJ = latexMatrice(matrice_EJ)
    matrice_Result = latexMatrice(matrice_Result)

    text = (step3_primal % {'step':step,'matrice_BN':matrice_BN, 'matrice_EJ':matrice_EJ,'matrice_Result':mat
```

Line 328, Column 40 Spaces: 4 Python

latex_sample.py UNREGISTERED

criss_cross.py × ques_1.py × latex_sample.py × test.py ×

```python
    matrice_BN = latexMatrice(matrice_BN)
    matrice_EJ = latexMatrice(matrice_EJ)
    matrice_Result = latexMatrice(matrice_Result)

    text = (step3_primal % {'step':step,'matrice_BN':matrice_BN, 'matrice_EJ':matrice_EJ,'matrice_Result':mat
    return text

step4_primal = '''\subsection{Step 4.}
\[
t =\Bigg(
max \left\{%(ratio)s\\right\}
\Bigg)^{-1}\ =\ %(value)s
\]
'''
def fourthStepPrimalDual(num_array,den_array,value,var_ch):
    ratio = latexRatios(num_array,den_array)
    text = (step4_primal % {'ratio':ratio,'value':value,'var_ch':var_ch})

    return text

step5_primal = '''\subsection{Step 5.}
\[
In\ step\ 4, \ the\ ratio\ corresponds\ to\ basic\ index\ %(index)s
\]
\[
%(var_ch)s\ = \ %(index)s
\]
'''

def fifthStepPrimal(index,var_ch):
    text = (step5_primal %{'index':index,'var_ch':var_ch})
    return text

step6_primal='''\subsection{Step 6.}
\[
\Delta z_{\mathcal N}= -( B^{-1} N )^{T}e_%(subscript_ch)s = -\
\\begin{bmatrix}
%(matrice_BN)s
\end{bmatrix}
\\begin{bmatrix}
%(matrice_EI)s
\end{bmatrix}
= \\begin{bmatrix}
%(matrice_Result)s
\end{bmatrix}
\]
'''

def sixthStepPrimal(step,matrice_BN,matrice_EI,matrice_Result,subscript_ch):
    matrice_BN = latexMatrice(matrice_BN)
    matrice_EI = latexMatrice(matrice_EI)
    matrice_Result = latexMatrice(matrice_Result)
```

Line 344, Column 15 Spaces: 4 Python

latex_sample.py UNREGISTERED

criss_cross.py | ques_1.py | latex_sample.py | test.py

```python
def sixthStepPrimal(step,matrice_BN,matrice_EI,matrice_Result,subscript_ch):
    matrice_BN = latexMatrice(matrice_BN)
    matrice_EI = latexMatrice(matrice_EI)
    matrice_Result = latexMatrice(matrice_Result)

    text = (step6_primal %{'step':step,'matrice_BN':matrice_BN,'matrice_EI':matrice_EI,'matrice_Result':matric
    return text

step7_primal_dual = '''\subsection{Step 7.}
\[
%(var_ch)s \ =\ \\frac{%(main_ch)s_{%(subscript_ch)s}^{*}}{ \Delta %(main_ch)s_{%(subscript_ch)s}}\ =\ %(rati
\]
'''
def seventhStepPrimalDual(value,num,den,var_ch,main_ch,subscript_ch):
    text = (step7_primal_dual %{'ratio':latexFraction(num,den),'value':value,'var_ch':var_ch,'main_ch':main_c
    return text



step8_primal = '''\subsection{Step 8.}
\[
x_{%(i_index)s}^{*}\ =\ %(i_value)s, \quad x_{\mathcal B}^{*}\ =\
\\begin{bmatrix}
%(matrice_x_old)s
\end{bmatrix}\ -%(i_value)s\
\\begin{bmatrix}
%(matrice_x_delta)s
\end{bmatrix}\ =\
\\begin{bmatrix}
%(matrice_x_new)s
\end{bmatrix}\ ,
\]
\\

\[
z_{%(j_index)s}^{*}\ =\ %(j_value)s, \quad z_{\mathcal N}^{*}\ =\
\\begin{bmatrix}
%(matrice_z_old)s
\end{bmatrix}\ -%(j_value)s\
\\begin{bmatrix}
%(matrice_z_delta)s
\end{bmatrix}\ =\
\\begin{bmatrix}
%(matrice_z_new)s
\end{bmatrix}\ ,
\]
'''

# Following parameters passed
# j,i,old_X_b,old_Z_n,delta_X_b,delta_Z_n,self.X_b,self.Z_n
def eightStepPrimal(j,i,t,s,old_X_b,old_Z_n,delta_X_b,delta_Z_n,new_X_b,new_Z_n):
    matrice_x_old = latexMatrice(old_X_b)
```

Line 344, Column 15 Spaces: 4 Python

latex_sample.py UNREGISTERED

criss_cross.py | ques_1.py | latex_sample.py | test.py

```python
    matrice_var_x = latexVariablesMatrice('x',basic)
    matrice_var_z = latexVariablesMatrice('z',nonbasic)

    basic = latexCommaSeprated(basic)
    nonbasic = latexCommaSeprated(nonbasic)
    matrice_B = latexMatrice(matrice_B)
    matrice_N = latexMatrice(matrice_N)

    matrice_X = latexMatrice(matrice_X)
    matrice_Z = latexMatrice(matrice_Z)

    text = (step9_primal % {'basic':basic,'nonbasic':nonbasic,'matrice_B':matrice_B,'matrice_N':matrice_N,'ma
    return text


final_doc = '''\documentclass [12pt] {article}
\usepackage{amsmath}
\makeatletter
\\renewcommand{\@seccntformat}[1]{}
\makeatother
\usepackage{url}
\usepackage[margin=0.8in]{geometry}
\pagestyle{plain}
\\begin{document}
\section*{Dual Simplex Method Initial Matrices and Vector} %(latex_tex)s \end{document}
'''

def getWholeLatex(latex_text):
    text = (final_doc % {'latex_tex':latex_text})
    return text


objectiveFuntion ='''
\[
\zeta^{*} = %(equation)s
\]
'''
def latexObjectiveFuntion(vec_c,value):
    equation = ''
    for i in range(len(vec_c)):
        if i != len(vec_c)-1:
            equation += "%sx_{%s}^{*}\\ +" % (str(vec_c[i]),str(i+1))
        else:
            # equation += "%sx_{%s}^{*}\\ +" % (str(vec_c[i]),str(i+1))
            equation += "%sx_{%s}^{*}\\ =\\ %s" % (str(vec_c[i]),str(i+1),str(value))

    text = (objectiveFuntion %{'equation':equation})
    return text
```

Line 344, Column 15 | Spaces: 4 | Python

Code Snapshots (Criss Cross Method)

```python


    def callNonBasicSteps(self,j):

        e_j = np.eye(1, len(self.Non_Basic) , self.Non_Basic.index(j))
        e_j = np.transpose(e_j)
        delta_X_b = ((inv(self.B)).dot(self.N)).dot(e_j)

        i = self.findMaxMinSubscript(delta_X_b,self.Basic,True)
        leaving_index = self.Basic.index(i)


        # STEP 4: Calculate Primal Step Length
        max_val = self.primalStepLength(delta_X_b,self.X_b,leaving_index)

        t = 0
        t = 1/max_val

        e_i = np.eye(1, len(self.Basic) , self.Basic.index(i))
        e_i = np.transpose(e_i)

        delta_Z_n = - (np.transpose((inv(self.B)).dot(self.N))).dot(e_i)

        s = self.Z_n[self.Non_Basic.index(j)]/delta_Z_n[self.Non_Basic.index(j)]
        s= float(s[0])

        self.X_b = self.X_b - t * np.transpose(delta_X_b)
        self.X_b = self.X_b[0]
        self.X_b[self.Basic.index(i)]=t

        self.Z_n = self.Z_n - s * np.transpose(delta_Z_n)
        self.Z_n = self.Z_n[0]
        self.Z_n[self.Non_Basic.index(j)]=s


        # Update Basis
        self.Non_Basic[self.Non_Basic.index(j)] = i
        self.Basic[self.Basic.index(i)] = j

        b_columns = self.Basic + np.array([-1.0]*len(self.Basic))
        n_columns = self.Non_Basic + np.array([-1.0]*len(self.Non_Basic))

        self.B = self.dict_A[: , b_columns.astype(np.int64)]
        self.N = self.dict_A[: , n_columns.astype(np.int64)]



    def callBasicSteps(self,i):

        e_i = np.eye(1, len(self.Basic) , self.Basic.index(i))
        e_i = np.transpose(e_i)
        # delta_X_b = ((inv(self.B)).dot(self.N)).dot(e_i)
        delta_Z_n = - (np.transpose((inv(self.B)).dot(self.N))).dot(e_i)

        j = self.findMaxMinSubscript(delta_Z_n,self.Non_Basic,True)
        entering_index = self.Non_Basic.index(j)
```

criss_cross.py UNREGISTERED

criss_cross.py | test.py | ques_1.py | latex_sample.py | latex_helper.py

```python
        j = self.findMaxMinSubscript(delta_Z_n,self.Non_Basic,True)
        entering_index = self.Non_Basic.index(j)
        # STEP 4: Calculate Primal Step Length
        max_val = self.primalStepLength(delta_Z_n,self.Z_n,entering_index)

        s = 0
        s = 1/max_val

        e_j = np.eye(1, len(self.Non_Basic) , self.Non_Basic.index(j))
        e_j = np.transpose(e_j)

        delta_X_b = ((inv(self.B)).dot(self.N)).dot(e_j)

        t = self.X_b[self.Basic.index(i)]/delta_X_b[self.Basic.index(i)]
        t= float(t[0])

        self.X_b = self.X_b - t * np.transpose(delta_X_b)
        self.X_b = self.X_b[0]
        self.X_b[self.Basic.index(i)]=t

        self.Z_n = self.Z_n - s * np.transpose(delta_Z_n)
        self.Z_n = self.Z_n[0]
        self.Z_n[self.Non_Basic.index(j)]=s

        self.Non_Basic[self.Non_Basic.index(j)] = i
        self.Basic[self.Basic.index(i)] = j

        # Update Basis

        b_columns = self.Basic + np.array([-1.0]*len(self.Basic))
        n_columns = self.Non_Basic + np.array([-1.0]*len(self.Non_Basic))

        self.B = self.dict_A[: , b_columns.astype(np.int64)]
        self.N = self.dict_A[: , n_columns.astype(np.int64)]



    def preformCrissCross(self):

        # Step 1 If Z_n>=0 and X_b >=0 Stop
        if self.isVectorPositive(self.Z_n) and self.isVectorPositive(self.X_b):
            simplex = linear.LinearProgram(self.dict_A,self.C_n,self.b)
            flag , value =simplex.preformPrimalSimplex()

        while not self.isVectorPositive(self.Z_n) or not self.isVectorPositive(self.X_b):
            # step 2
            j_z = self.findMaxMinSubscript(self.Z_n,self.Non_Basic,False)

            i_z = self.findMaxMinSubscript(self.X_b,self.Basic,False)

            if not self.isVectorPositive(self.Z_n):
                self.callNonBasicSteps(j_z)
            else:
                self.callBasicSteps(i_z)

        value = self.getObjectiveValue([],self.C_n, self.X_b, self.Basic, self.numberOfVaraibles
```

Line 265, Column 4 Spaces: 4 Python

Code Snapshots (Test file with output on terminal)

test.py — UNREGISTERED

criss_cross.py | test.py | ques_1.py | latex_sample.py | latex_helper.py

```python
import numpy as np
import unittest

import ques_1 as linear
import criss_cross


class Test(unittest.TestCase):
    def setUp(self):
        self.a = np.asarray([[1 , -1], [2, -1], [0, 1]]).astype(np.float)
        self.b = np.asarray([1,3,5]).astype(np.float)
        self.c = np.asarray([4,3]).astype(np.float)

        self.a1 = np.asarray([[-2 , -1], [-2, 4], [-1, 3]]).astype(np.float)
        self.b1 = np.asarray([4,-8,-7]).astype(np.float)
        self.c1 = np.asarray([-1,-1]).astype(np.float)

        self.N1 = np.asarray([[ 0 ,-1],[ 0 , 4],[ 1 , 3]]).astype(np.float)
        self.B1 = np.asarray([[ 1 ,-2 , 0], [ 0 ,-2 , 1], [ 0 ,-1 , 0]]).astype(np.float)

        self.a2 = np.asarray([[-2 , -1], [-2, 4], [-1, 3]]).astype(np.float)
        self.b2 = np.asarray([4,-8,-7]).astype(np.float)
        self.c2 = np.asarray([-1,4]).astype(np.float)

        self.a3 = np.asarray([[1 , 4, 0], [3 , -1, 1]]).astype(np.float)
        self.b3 = np.asarray([1,3]).astype(np.float)
        self.c3 = np.asarray([4, 1, 3]).astype(np.float)

        self.a4 = np.asarray([[1 , -1], [-1, -1], [2 , 1]]).astype(np.float)
        self.b4 = np.asarray([-1,-3, 4]).astype(np.float)
        self.c4 = np.asarray([3, 1]).astype(np.float)

        # Criss Cross
        self.a5 = np.asarray([[1 , 1], [-1, 2], [1 , -3]]).astype(np.float)
        self.b5 = np.asarray([6,-0.5,-1]).astype(np.float)
        self.c5 = np.asarray([-2, 3]).astype(np.float)



    '''Check the Basic and non Baic Indices return are corrent'''
    def test1(self):
        print 'TEST 1---------------------------------------------- TEST 1'
        simplex = linear.LinearProgram(self.a,self.c,self.b)
        flag , value =simplex.preformPrimalSimplex()
        self.assertEqual(simplex.Non_Basic,[5,4])
        self.assertEqual(simplex.Basic,[1, 2, 3])
        print 'Basic and Non Basic Indices return'

    def test2(self):
        print 'TEST 2---------------------------------------------- TEST 2'
        simplex = linear.LinearProgram(self.a,self.c,self.b)
        flag , value =simplex.preformPrimalSimplex()
        self.assertAlmostEqual(value,31.0)
        print 'Objective value returned by Primal simplex is Correct'

    def test3(self):
```

Line 29, Column 74 | Spaces: 4 | Python

test.py UNREGISTERED

criss_cross.py | test.py | ques_1.py | latex_sample.py | latex_helper.py

```python
        dual = linear.LinearProgram(self.a1,self.c1,self.b1)
        flag , value =dual.preformDualSimplex()
        self.assertAlmostEqual(value,-7)
        print 'Objective value returned by Daul simplex is Correct'

    def test4(self):
        print 'TEST 4----------------------------------------------------- TEST 4'
        dual = linear.LinearProgram(self.a1,self.c1,self.b1)
        flag , value = dual.preformDualSimplex()
        self.assertEqual(dual.N.all(),self.N1.all())
        self.assertEqual(dual.B.all(),self.B1.all())
        print 'The matrices B and N returned are corrent for Dual simplex'

    def test5(self):
        print 'TEST 5----------------------------------------------------- TEST 5'
        simplex = linear.LinearProgram(self.a2,self.c2,self.b2)
        flag,value =simplex.preformPrimalSimplex()
        self.assertAlmostEqual(flag,-1)
        print 'The Problem is UnBounded.'

    def test6(self):
        print 'TEST 6----------------------------------------------------- TEST 6'
        simplex = linear.LinearProgram(self.a3,self.c3,self.b3)
        flag,primal_value =simplex.preformPrimalSimplex()

        # new Matrices and vectors for the Dual
        a_dual = -np.transpose(self.a3)
        c_dual = -np.transpose(self.b3)
        b_dual = -np.transpose(self.c3)

        simplex = linear.LinearProgram(a_dual,c_dual,b_dual)
        flag , dual_value = simplex.preformDualSimplex()

        self.assertAlmostEqual(primal_value,-dual_value)

        print 'Duality Thoery Checked'

    def test7(self):
        print 'TEST 7----------------------------------------------------- TEST 7'
        simplex = linear.LinearProgram(self.a4,self.c4,self.b4)
        flag,value =simplex.preformPrimalSimplex()
        self.assertAlmostEqual(flag,0)
        print 'Two Phase Method Problem Passed'

    def test8(self):
        print 'TEST 8----------------------------------------------------- TEST 8'
        cc = criss_cross.CrissCross(self.a5,self.c5,self.b5)
        value =cc.preformCrissCross()
        self.assertAlmostEqual(-2.5,value)
        print 'Value returned by the CrissCross is Correct'



if __name__ == '__main__':
    unittest.main()
```

Line 29, Column 74 Spaces: 4 Python

ques_1 — -bash — 76×24

```
OK
abhi ques_1 $ python test.py
TEST 1---------------------------------------------- TEST 1
Basic and Non Basic Indices return
.TEST 2---------------------------------------------- TEST 2
Objective value returned by Primal simplex is Correct
.TEST 3---------------------------------------------- TEST 3
Objective value returned by Daul simplex is Correct
.TEST 4---------------------------------------------- TEST 4
The matrices B and N returned are corrent for Dual simplex
.TEST 5---------------------------------------------- TEST 5
The Problem is UnBounded.
.TEST 6---------------------------------------------- TEST 6
Duality Thoery Checked
.TEST 7---------------------------------------------- TEST 7
Two Phase Method Problem Passed
.TEST 8---------------------------------------------- TEST 8
Value returned by the CrissCross is Correct
.
----------------------------------------------------------------------
Ran 8 tests in 0.017s

OK
abhi ques_1 $
```

Primal Simplex Method Initial Matrices and Vector

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

$$Initial\ set\ of\ basic\ and\ nonbasic\ indices$$
$$\beta = \{3,4,5\} \quad and \quad \mathcal{N} = \{1,2\}$$

$$Submatrice\ of\ A$$

$$B = \begin{bmatrix} 1.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 1.0 \end{bmatrix} \quad and \quad N = \begin{bmatrix} 1.0 & -1.0 \\ 2.0 & -1.0 \\ 0.0 & 1.0 \end{bmatrix}$$

$$Inital\ values\ of\ the\ basic\ variables\ are\ given\ by$$
$$x_B^* = b = \begin{bmatrix} 1.0 \\ 3.0 \\ 5.0 \end{bmatrix}$$
$$Inital\ values\ of\ the\ nonbasic\ dualvariables\ are\ given\ by$$
$$z_N^* = -c_N = \begin{bmatrix} -4.0 \\ -3.0 \end{bmatrix}$$

$$Since\ x_B^*\ \geq\ 0, the\ initial\ solution\ is\ primal\ feasible.$$

Iteration No 1

Step 1.

$$Since\ z_N^*\ has\ some\ negative\ components,\ the\ current\ solution\ is\ not\ optimal.$$

Step 2.

$$Since\ z_1^*\ =\ -4.0\ and\ this\ is\ the\ most\ negative\ dual\ variables,$$

$$we\ see\ that\ the\ entering\ index\ is\ j\ =\ 1$$

Step 3.

$$\Delta x_{\mathcal{B}} = B^{-1}Ne_j = \begin{bmatrix} 1.0 & -1.0 \\ 2.0 & -1.0 \\ 0.0 & 1.0 \end{bmatrix} \begin{bmatrix} 1.0 \\ 0.0 \end{bmatrix} = \begin{bmatrix} 1.0 \\ 2.0 \\ 0.0 \end{bmatrix}$$

Step 4.

$$t = \left(max \left\{ \frac{1.0}{1.0}, \frac{2.0}{3.0}, \frac{0.0}{5.0} \right\} \right)^{-1} \; = \; 1.0$$

Step 5.

$$\textit{In step 4, the ratio corresponds to basic index } 3$$

$$i \; = \; 3$$

Step 6.

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

Step 7.

$$s \; = \; \frac{z_1^*}{\Delta z_1} \; = \; \frac{-4.0}{-1.0} \; = \; 4.0$$

Step 8.

$$x_1^* \; = \; 1.0, \quad x_{\mathcal{B}}^* \; = \begin{bmatrix} 1.0 \\ 3.0 \\ 5.0 \end{bmatrix} - 1.0 \begin{bmatrix} 1.0 \\ 2.0 \\ 0.0 \end{bmatrix} \; = \begin{bmatrix} 0.0 \\ 1.0 \\ 5.0 \end{bmatrix} \; ,$$

$$z_3^* \; = \; 4.0, \quad z_{\mathcal{N}}^* \; = \begin{bmatrix} -4.0 \\ -3.0 \end{bmatrix} - 4.0 \begin{bmatrix} -1.0 \\ 1.0 \end{bmatrix} \; = \begin{bmatrix} 0.0 \\ -7.0 \end{bmatrix} \; ,$$

Step 9.

$$\textit{New set of basic and nonbasic indices}$$

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

$$\textit{Corresponding new basis and nonbasis submatrices of } A,$$

$$B = \begin{bmatrix} 1.0 & 0.0 & 0.0 \\ 2.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 1.0 \end{bmatrix} \quad \textit{and} \quad N = \begin{bmatrix} 1.0 & -1.0 \\ 0.0 & -1.0 \\ 0.0 & 1.0 \end{bmatrix}$$

$$\textit{New Basic primal variables and nonbasic dual variables}:$$

$$x_{\mathcal{B}}^* \; = \begin{bmatrix} x_1^* \\ x_4^* \\ x_5^* \end{bmatrix} \; = \begin{bmatrix} 1.0 \\ 1.0 \\ 5.0 \end{bmatrix} \quad z_{\mathcal{N}}^* \; = \begin{bmatrix} z_3^* \\ z_2^* \end{bmatrix} \; = \begin{bmatrix} 4.0 \\ -7.0 \end{bmatrix}$$

Iteration No 2

Step 1.

$$Since\ z_N^*\ has\ some\ negative\ components,\ the\ current\ solution\ is\ not\ optimal.$$

Step 2.

$$Since\ z_2^* \ = \ -7.0\ and\ this\ is\ the\ most\ negative\ dual\ variables,$$

$$we\ see\ that\ the\ entering\ index\ is\ j \ = \ 2$$

Step 3.

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

Step 4.

$$t = \left(max \left\{ \frac{-1.0}{1.0}, \frac{1.0}{1.0}, \frac{1.0}{5.0} \right\} \right)^{-1} \ = \ 1.0$$

Step 5.

$$In\ step\ 4,\ the\ ratio\ corresponds\ to\ basic\ index\ 4$$

$$i \ = \ 4$$

Step 6.

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

Step 7.

$$s \ = \ \frac{z_2^*}{\Delta z_2} \ = \ \frac{-7.0}{-1.0} \ = \ 7.0$$

Step 8.

$$x_2^* \ = \ 1.0, \quad x_{\mathcal{B}}^* \ = \begin{bmatrix} 1.0 \\ 1.0 \\ 5.0 \end{bmatrix} - 1.0 \begin{bmatrix} -1.0 \\ 1.0 \\ 1.0 \end{bmatrix} \ = \begin{bmatrix} 2.0 \\ 0.0 \\ 4.0 \end{bmatrix} \ ,$$

$$z_4^* \ = \ 7.0, \quad z_{\mathcal{N}}^* \ = \begin{bmatrix} 4.0 \\ -7.0 \end{bmatrix} - 7.0 \begin{bmatrix} 2.0 \\ -1.0 \end{bmatrix} \ = \begin{bmatrix} -10.0 \\ 0.0 \end{bmatrix} \ ,$$

Step 9.

$$New\ set\ of\ basic\ and\ nonbasic\ indices$$

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

$$Corresponding\ new\ basis\ and\ nonbasis\ submatrices\ of\ A,$$

$$B = \begin{bmatrix} 1.0 & -1.0 & 0.0 \\ 2.0 & -1.0 & 0.0 \\ 0.0 & 1.0 & 1.0 \end{bmatrix} \quad and \quad N = \begin{bmatrix} 1.0 & 0.0 \\ 0.0 & 1.0 \\ 0.0 & 0.0 \end{bmatrix}$$

$$New\ Basic\ primal\ variables\ and\ nonbasic\ dual\ variables:$$

$$x^*_{\mathcal{B}} = \begin{bmatrix} x^*_1 \\ x^*_2 \\ x^*_5 \end{bmatrix} = \begin{bmatrix} 2.0 \\ 1.0 \\ 4.0 \end{bmatrix} \quad z^*_{\mathcal{N}} = \begin{bmatrix} z^*_3 \\ z^*_4 \end{bmatrix} = \begin{bmatrix} -10.0 \\ 7.0 \end{bmatrix}$$

Iteration No 3

Step 1.

$$Since\ z^*_N\ has\ some\ negative\ components,\ the\ current\ solution\ is\ not\ optimal.$$

Step 2.

$$Since\ z^*_3 = -10.0\ and\ this\ is\ the\ most\ negative\ dual\ variables,$$

$$we\ see\ that\ the\ entering\ index\ is\ j = 3$$

Step 3.

$$\Delta x_{\mathcal{B}} = B^{-1}Ne_j = \begin{bmatrix} -1.0 & 1.0 \\ -2.0 & 1.0 \\ 2.0 & -1.0 \end{bmatrix} \begin{bmatrix} 1.0 \\ 0.0 \end{bmatrix} = \begin{bmatrix} -1.0 \\ -2.0 \\ 2.0 \end{bmatrix}$$

Step 4.

$$t = \left(max \left\{ \frac{-1.0}{2.0}, \frac{-2.0}{1.0}, \frac{2.0}{4.0} \right\} \right)^{-1} = 2.0$$

Step 5.

$$In\ step\ 4,\ the\ ratio\ corresponds\ to\ basic\ index\ 5$$

$$i = 5$$

Step 6.

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

Step 7.

$$s \;=\; \frac{z_3^*}{\Delta z_3} \;=\; \frac{-10.0}{-2.0} \;=\; 5.0$$

Step 8.

$$x_3^* \;=\; 2.0, \quad x_{\mathcal{B}}^* \;=\; \begin{bmatrix} 2.0 \\ 1.0 \\ 4.0 \end{bmatrix} \;-\; 2.0 \begin{bmatrix} -1.0 \\ -2.0 \\ 2.0 \end{bmatrix} \;=\; \begin{bmatrix} 4.0 \\ 5.0 \\ 0.0 \end{bmatrix},$$

$$z_5^* \;=\; 5.0, \quad z_{\mathcal{N}}^* \;=\; \begin{bmatrix} -10.0 \\ 7.0 \end{bmatrix} \;-\; 5.0 \begin{bmatrix} -2.0 \\ 1.0 \end{bmatrix} \;=\; \begin{bmatrix} 0.0 \\ 2.0 \end{bmatrix},$$

Step 9.

New set of basic and nonbasic indices

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

Corresponding new basis and nonbasis submatrices of A,

$$B = \begin{bmatrix} 1.0 & -1.0 & 1.0 \\ 2.0 & -1.0 & 0.0 \\ 0.0 & 1.0 & 0.0 \end{bmatrix} \quad \textit{and} \quad N = \begin{bmatrix} 0.0 & 0.0 \\ 0.0 & 1.0 \\ 1.0 & 0.0 \end{bmatrix}$$

New Basic primal variables and nonbasic dual variables :

$$x_{\mathcal{B}}^* \;=\; \begin{bmatrix} x_1^* \\ x_2^* \\ x_3^* \end{bmatrix} \;=\; \begin{bmatrix} 4.0 \\ 5.0 \\ 2.0 \end{bmatrix} \qquad z_{\mathcal{N}}^* \;=\; \begin{bmatrix} z_5^* \\ z_4^* \end{bmatrix} \;=\; \begin{bmatrix} 5.0 \\ 2.0 \end{bmatrix}$$

Iteration No 4

Step 1.

Since z_N^* has all nonnegative components, the current solution is optimal.

$$\zeta^* = 4.0x_1^* \; + 3.0x_2^* \;=\; 31.0$$

Dual Simplex Method Initial Matrices and Vector

$$A = \begin{bmatrix} -2.0 & -1.0 & 1.0 & 0.0 & 0.0 \\ -2.0 & 4.0 & 0.0 & 1.0 & 0.0 \\ -1.0 & 3.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}$$

$$Initial\ set\ of\ basic\ and\ nonbasic\ indices$$

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

$$Submatrice\ of\ A$$

$$B = \begin{bmatrix} 1.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 1.0 \end{bmatrix} \quad and \quad N = \begin{bmatrix} -2.0 & -1.0 \\ -2.0 & 4.0 \\ -1.0 & 3.0 \end{bmatrix}$$

$$Inital\ values\ of\ the\ basic\ variables\ are\ given\ by$$

$$x_B^* = b = \begin{bmatrix} 4.0 \\ -8.0 \\ -7.0 \end{bmatrix}$$

$$Inital\ values\ of\ the\ nonbasic\ dualvariables\ are\ given\ by$$

$$z_N^* = -c_N = \begin{bmatrix} 1.0 \\ 1.0 \end{bmatrix}$$

$$Since\ z_N^*\ \geq\ 0, the\ initial\ solution\ is\ Dual\ feasible.$$

Iteration No 1

Step 1.

$$Since\ x_B^*\ has\ some\ negative\ components,\ the\ current\ solution\ is\ not\ optimal.$$

Step 2.

$$Since\ x_4^*\ =\ -8.0\ and\ this\ is\ the\ most\ negative\ dual\ variables,$$

$$we\ see\ that\ the\ entering\ index\ is\ i\ =\ 4$$

Step 3.

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

Step 4.

$$t = \left(max \left\{ \frac{2.0}{1.0}, \frac{-4.0}{1.0} \right\} \right)^{-1} \; = \; 0.5$$

Step 5.

$$In \; step \; 4, \; the \; ratio \; corresponds \; to \; basic \; index \; 1$$
$$j \; = \; 1$$

Step 6.

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

Step 7.

$$t \; = \; \frac{x_1^*}{\Delta x_1} \; = \; \frac{-8.0}{-2.0} \; = \; 4.0$$

Step 9.

$$New \; set \; of \; basic \; and \; nonbasic \; indices$$
$$\beta = \{3, 1, 5\} \quad and \quad \mathcal{N} = \{4, 2\}$$

$$Corresponding \; new \; basis \; and \; nonbasis \; submatrices \; of \; A,$$
$$B = \begin{bmatrix} 1.0 & -2.0 & 0.0 \\ 0.0 & -2.0 & 0.0 \\ 0.0 & -1.0 & 1.0 \end{bmatrix} \quad and \quad N = \begin{bmatrix} 0.0 & -1.0 \\ 1.0 & 4.0 \\ 0.0 & 3.0 \end{bmatrix}$$

$$New \; Basic \; primal \; variables \; and \; nonbasic \; dual \; variables:$$
$$x_{\mathcal{B}}^* \; = \begin{bmatrix} x_3^* \\ x_1^* \\ x_5^* \end{bmatrix} \; = \begin{bmatrix} 12.0 \\ 4.0 \\ -3.0 \end{bmatrix} \quad z_{\mathcal{N}}^* \; = \begin{bmatrix} z_4^* \\ z_2^* \end{bmatrix} \; = \begin{bmatrix} 0.5 \\ 3.0 \end{bmatrix}$$

Iteration No 2

Step 1.

$$Since \; x_B^* \; has \; some \; negative \; components, \; the \; current \; solution \; is \; not \; optimal.$$

Step 2.

$$Since \; x_5^* \; = \; -3.0 \; and \; this \; is \; the \; most \; negative \; dual \; variables,$$

$$we \; see \; that \; the \; entering \; index \; is \; i \; = \; 5$$

Step 3.

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

Step 4.

$$t = \left(max \left\{ \frac{0.5}{0.5}, \frac{-1.0}{3.0} \right\} \right)^{-1} \;=\; 1.0$$

Step 5.

$$In\ step\ 4,\ the\ ratio\ corresponds\ to\ basic\ index\ 4$$

$$j \;=\; 4$$

Step 6.

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

Step 7.

$$t \;=\; \frac{x_4^*}{\Delta x_4} \;=\; \frac{-3.0}{-0.5} \;=\; 6.0$$

Step 9.

$$New\ set\ of\ basic\ and\ nonbasic\ indices$$

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

$$Corresponding\ new\ basis\ and\ nonbasis\ submatrices\ of\ A,$$

$$B = \begin{bmatrix} 1.0 & -2.0 & 0.0 \\ 0.0 & -2.0 & 1.0 \\ 0.0 & -1.0 & 0.0 \end{bmatrix} \quad and \quad N = \begin{bmatrix} 0.0 & -1.0 \\ 0.0 & 4.0 \\ 1.0 & 3.0 \end{bmatrix}$$

$$New\ Basic\ primal\ variables\ and\ nonbasic\ dual\ variables:$$

$$x_{\mathcal{B}}^* \;=\; \begin{bmatrix} x_3^* \\ x_1^* \\ x_4^* \end{bmatrix} \;=\; \begin{bmatrix} 18.0 \\ 7.0 \\ 6.0 \end{bmatrix} \qquad z_{\mathcal{N}}^* \;=\; \begin{bmatrix} z_5^* \\ z_2^* \end{bmatrix} \;=\; \begin{bmatrix} 1.0 \\ 4.0 \end{bmatrix}$$

Iteration No 3

Step 1.

Since x_B^* *has all nonnegative components, the current solution is optimal.*

$$\zeta^* = -1.0x_1^* \; + -1.0x_2^* \; = \; -7.0$$