sol_week8

Aman Kumar EE21B013 <ee21b013@smail.iitm.ac.in>

April 16, 2023

1 Assingment 8 solution

1.1 Normal circuit solver functions without optimisation

[1]:
```
# swap function for swapping of two rows and its corresponding value in␣
 ↪solmatrix
def swap(  i,  j,  m,matrix,solmatrix):
    for ele in range(m):
        cc=matrix[i][ele]
        matrix[i][ele]=matrix[j][ele]
        matrix[j][ele]=cc

    temp=solmatrix[i]
    solmatrix[i]=solmatrix[j]
    solmatrix[j]=temp
```

1.1.1 Gaussian function for convering into RREF(Row Reduced Echelon Form).

[2]:
```
# Function to convert the given two matrix (matrix and solmatrix) into gaussian␣
 ↪form(uppper traingular matix and the respective value of solmatrix)

def gaussian(matrix,solmatrix):
    n=len(matrix)
    m=len(matrix[0])  #n and m are the rows and columms of the given matrix
    # print('nodex',n,m)
    minele=min(n,m)
    for i in range (minele):
        if(abs(matrix[i][i])<1e-15): # I have used tolerence level as 1e-15 i.e.
 ↪ considering the value of less than 1e-15 as 0.
            matrix[i][i]=0
            for row in range(i+1,n):
                if(abs(matrix[row][i])>1e-15):

                    swap(i,row,m,matrix,solmatrix) # If digonal entry is zero␣
 ↪than using swap function built earlier swap this row with some other␣
 ↪non-zero digonal element rows
```

```
        # If digonal element is non-zero than change the matrix by doing forward␣
 ↪ellimination
        if(abs(matrix[i][i])>1e-15):
            divisorno=matrix[i][i]
            solmatrix[i]=solmatrix[i]/divisorno  #making the pivot element␣
 ↪1(digonal element).

            for j in range(i,m):
                matrix[i][j]=matrix[i][j]/divisorno  # normalizing every other␣
 ↪element of row correspondingly.

            for row in range(i+1,n):
                if(abs(matrix[row][i])>1e-15):
                    multiplier=matrix[row][i]
                    solmatrix[row]=solmatrix[row]-multiplier*solmatrix[i] ␣
 ↪#Making corresponding changes in solmatrix

                    #Now normalizing the other row below the pivot row
                    matrix[row][i]=0
                    for k in range(i+1,m):
                        matrix[row][k]=matrix[row][k]-multiplier*matrix[i][k]
                # elif(matrix[row][i]<=1e-15):
                #     matrix[row][i]=0
        else :
            return

# a=[[1,1,2,1],[2,2,-2,3],[7,8,9,17],[6,1,1,1]]
# b=[14,6,44,16]
```

1.1.2 solve function used for finding the solution after the matrices reduced to RREF.

[3]:
```
import numpy as np
# Function to solve two linar equation of form Ax=B , where A=matrix and␣
 ↪B=solmatrix which are passed inside the function as an argument
def solve(matrix,solmatrix):
    solution=[]  # Matrix to store the final result(solution) of variables
    solution.clear()

    gaussian(matrix,solmatrix)  #calling the gaussian function to tranform the␣
 ↪given matrix into echleon form (upper trinagular matrix)
                                # and the respective solmatrix
    cnt=0
    n=len(matrix)
    m=len(matrix[0])
    index=0
    for row in matrix:
        index+=1
```

```
        cntNonZeroElement=0
        for ele in row:
            if(ele!=0):
                cntNonZeroElement+=1
        if(cntNonZeroElement>0):
            cnt+=1
        if(cntNonZeroElement==0):
            if(solmatrix[index-1]!=0):
                # print("No solution Exits")
                return "No solution Exits"

     # checkint condition for infinite solution
    if(cnt<m):
        # print('Infinte solution')
        return "Infinite solution"

    #checking condition for no solution
    if(cnt>m):
        # print('No solution')
        return "NO solution"

    # if sol exits then finding the solution using BackPropagation Method from Down to Up
    for row in range(m-1,-1,-1):
        ans=0
        l=len(solution)
        l=l-1

        for col in range(m-1,row,-1):

            if(l<0):
                continue
            ans+=solution[l]*matrix[row][col]
            l-=1
        ans=solmatrix[row]-ans
        solution.insert(0,ans)
    # print(solution)
    return solution  #returning the solution matrix
```

Testing on 100x100 matrix the function written by me and linal.solve and comparing the time taken by both the function to solve 100x100 complex matrices(Because we are usning this solver to solve the circuits having complex variables that's why i am testing it on complex matrices).

[4]:
```
# a=[[1,1,2,1],[2,2,-2,3],[7,8,9,17],[6,1,1,1]]
# b=[14,6,44,16]

# a=np.random.rand(100,100)
```

```
# b=np.random.rand(100)
a= np.random.uniform(-1, 1, size=(10, 10)) + np.random.uniform(-1, 1, size=(10,␣
 ↪10)) * 1j #Generating matrix A(dimension 100x100)
b= np.random.uniform(-1, 1, size=(10)) + np.random.uniform(-1, 1, size=(10)) *␣
 ↪1j  #generating matrix B(dimension 100*1)
# print(b)
print(f'The solution by solver made by me : {solve(a,b)}') #printing the␣
 ↪solution my solver made by me
print(f'The time taken by my solver is : ',end='')
%timeit solve(a,b) # show the time taken by my solver

print(f'The solution of np.linalg.solve is : {np.linalg.solve(a,b)}') #solution␣
 ↪given by np.linalg.solve
print(f'the time taken by np.linalg.solve is : ',end='')
%timeit np.linalg.solve(a,b)# time taken by np.linalg.solve
```

```
The solution by solver made by me : [(0.13283210485834496+0.38072175910275297j),
(-0.7239666092001455-1.5475092013197622j),
(0.5094801857860147-1.5060807956475757j),
(-0.9596254394016394-0.2881645310013812j),
(-0.4836653953302358+1.772892431692164j),
(-1.1590339022788976-0.6770292126874455j),
(-0.22233460922852588-1.2615491556581062j),
(-0.341160159100649+1.378608781060282j),
(1.5544932541604952-0.03159932777111618j),
(-0.961831482562567+0.5587975947174166j)]
The time taken by my solver is : 45 µs ± 448 ns per loop (mean ± std. dev. of 7
runs, 10,000 loops each)
The solution of np.linalg.solve is : [ 0.1328321 +0.38072176j
-0.72396661-1.5475092j   0.50948019-1.5060808j
 -0.95962544-0.28816453j -0.4836654 +1.77289243j -1.1590339 -0.67702921j
 -0.22233461-1.26154916j -0.34116016+1.37860878j  1.55449325-0.03159933j
 -0.96183148+0.55879759j]
the time taken by np.linalg.solve is : 12.9 µs ± 2.48 µs per loop (mean ± std.
dev. of 7 runs, 100,000 loops each)
```

2 Optimised Cython code

[5]:
```
%load_ext Cython
```

[14]:
```
%%cython

import cython
import numpy as np
cimport numpy as np
@cython.cdivision(True)
```

```
cdef c_swap(int i,  int j,int m, list matrix, list solmatrix):
    cdef complex cc
    cdef complex temp
    for ele in range(m):
        cc=matrix[i][ele]
        matrix[i][ele]=matrix[j][ele]
        matrix[j][ele]=cc

    temp=solmatrix[i]
    solmatrix[i]=solmatrix[j]
    solmatrix[j]=temp

@cython.cdivision(True)
cdef c_gaussian(list matrix, list solmatrix):
    cdef int n
    cdef int m
    n=len(matrix)
    m=len(matrix[0])  #n and m are the rows and columnns of the given matrix
    # print('nodex',n,m)
    cdef int minele
    minele=min(n,m)
    cdef int i
    cdef int row
    cdef int k
    cdef int j
    cdef complex divisorno
    cdef complex multiplier
    for i in range (minele):
        if(abs(matrix[i][i])<1e-15): # I have used tolerence level as 1e-15 i.e.
 ↪ considering the value of less than 1e-15 as 0.
            matrix[i][i]=0

            for row in range(i+1,n):
                if(abs(matrix[row][i])>1e-15):

                    c_swap(i,row,m,matrix,solmatrix) # If digonal entry is zero␣
 ↪than using swap function built earlier swap this row with some other␣
 ↪non-zero digonal element rows

      # If digonal element is non-zero than change the matrix by doing forward␣
 ↪ellimination
       if(abs(matrix[i][i])>1e-15):
           divisorno=matrix[i][i]
           solmatrix[i]=solmatrix[i]/divisorno  #making the pivot element␣
 ↪1(digonal element).
           for j in range(i,m):
```

```
                matrix[i][j]=matrix[i][j]/divisorno  # normalizing every other␣
 ↪element of row correspondingly.
            # cdef int row
            for row in range(i+1,n):
                if(abs(matrix[row][i])>1e-15):
                    multiplier=matrix[row][i]
                    solmatrix[row]=solmatrix[row]-multiplier*solmatrix[i] ␣
 ↪#Making corresponding changes in solmatrix

                    #Now normalizing the other row below the pivot row
                    matrix[row][i]=0
                    for k in range(i+1,m):
                        matrix[row][k]=matrix[row][k]-multiplier*matrix[i][k]
                # elif(matrix[row][i]<=1e-15):
                #     matrix[row][i]=0
        else :
            return

@cython.cdivision(True)
def c_solve(list  matrix, list solmatrix):
    solution=[]  # Matrix to store the final result(solution) of variables
    solution.clear()

    c_gaussian(matrix,solmatrix)  #calling the gaussian function to tranform␣
 ↪the given matrix into echleon form (upper trinagular matrix)
                                # and the respective solmatrix
    cdef int cnt=0
    cdef int n=len(matrix)
    cdef int m=len(matrix[0])
    cdef int index=0
    cdef complex ele
    cdef int cntNonZeroElement
    cdef int row
    for row in range(0,len(matrix)):
        index+=1
        cntNonZeroElement=0
        for ele in matrix[row]:
            if(ele!=0):
                cntNonZeroElement+=1
        if(cntNonZeroElement>0):
            cnt+=1
        if(cntNonZeroElement==0):
            if(solmatrix[index-1]!=0):
                # print("No solution Exits")
                return "No solution Exits"

     # checkint condition for infinite solution
```

```
    if(cnt<m):
        # print('Infinte solution')
        return "Infinite solution"

    #checking condition for no solution
    if(cnt>m):
        # print('No solution')
        return "NO solution"

    # if sol exits then finding the solution using BackPropagation Method from␣
 ↪Down to Up
    # cdef int row
    cdef int l
    cdef complex ans
    cdef int col
    for row in range(m-1,-1,-1):
        ans=0
        l=len(solution)
        l=l-1

        for col in range(m-1,row,-1):

            if(l<0):
                continue
            ans+=solution[l]*matrix[row][col]
            l-=1
        ans=solmatrix[row]-ans
        solution.insert(0,ans)
    # print(solution)
    return solution  #returning the solution matrix
```

2.0.1 Time comparision Using different functions :

- **Without Optimisation**
- **INbuilt Function linalg.solve**
- **With Optimisation**

[7]:
```
print(f'The time taken without optimisation for solving 10x10 complex matrice :␣
 ↪',end='')
%timeit solve(a,b)
print('The time taken by inbuilt linalg.solve to solve same 10x10 matrix is :␣
 ↪',end='')
%timeit np.linalg.solve(a,b)
a=list(a)
b=list(b)
print('the time taken by Optimised solver is : ',end='')
%timeit c_solve(a,b)
```

```
print()
print('the solution of optimised solver is : ')
print(c_solve(a,b))
```

```
The time taken without optimisation for solving 10x10 complex matrice : 44.3 µs
± 621 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
The time taken by inbuilt linalg.solve to solve same 10x10 matrix is : 12.2 µs ±
971 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
the time taken by Optimised solver is : 17.7 µs ± 144 ns per loop (mean ± std.
dev. of 7 runs, 10,000 loops each)

the solution of optimised solver is :
[(0.13283210485834762+0.38072175910274986j),
(-0.7239666092001478-1.547509201319761j),
(0.5094801857860127-1.5060807956475784j),
(-0.9596254394016396-0.2881645310013812j),
(-0.48366539533023567+1.7728924316921653j),
(-1.1590339022788994-0.6770292126874451j),
(-0.222334609228525-1.261549155658107j),
(-0.341160159100649+1.3786087810602825j),
(1.5544932541604952-0.03159932777111618j),
(-0.961831482562567+0.5587975947174166j)]
```

2.1 Explanation of above code and optimisation:

- As this solver code will be used for solving the circuit so i have chosen the datatypes of the matrices to be complex.
- As we can see there are a lot of yellow lines it's because of there is no inbuilt datatype like complex in c and the cython is converting that complex datatype using some inbuilt function or library into c language code.
- It is because of that complex data type my code is running a little bit slow but it is fastr than the python code.
- I have initialised the datatypes of the variable whatever used in the functions.
- I can't use a datatype for array because there is no data type like complex in c.
-

2.1.1 Explanation of the above code and function:

- The above code is a Cython implementation of the Gaussian Elimination method for solving a system of linear equations. The code is divided into three functions: c_swap, c_gaussian, and c_solve.
- The c_swap function takes four arguments: i, j, m, matrix, and solmatrix. Here, i and j are the indices of the rows to be swapped in the matrix list, which is a list of lists representing the coefficients of the linear equations. m is the number of columns in the matrix, and solmatrix is a list representing the constant terms in the equations. The

function swaps the i-th and j-th rows of the matrix list and also swaps the i-th and j-th elements of the solmatrix list.
  - The c_gaussian function takes two arguments: matrix and solmatrix. Here, matrix and solmatrix are the same as in the c_swap function. The function performs Gaussian Elimination on the matrix list to transform it into an upper triangular matrix. It also performs the same operations on the solmatrix list. The function starts with the i-th row and checks if the i-th diagonal element is zero. If it is zero, it calls the c_swap function to swap the i-th row with a non-zero diagonal element row below it. If the diagonal element is non-zero, the function divides the entire row by the diagonal element to make it a leading 1. Then, it subtracts multiples of this row from the rows below it to make all the elements below the leading 1 in that column zero. It repeats this process for each row until the matrix is transformed into an upper triangular matrix.
  - The c_solve function takes two arguments: matrix and solmatrix. Here, matrix and solmatrix are the same as in the c_swap function. The function calls the c_gaussian function to transform the matrix list into an upper triangular matrix and the solmatrix list accordingly. After that, it checks if there is a unique solution, no solution, or infinite solutions. If there is no solution, the function returns "No solution". If there are infinite solutions, it returns "Infinite solution". If there is a unique solution, it uses back-substitution to find the solution of the linear equations and returns it as a list. The back-substitution is performed by starting from the last row of the matrix and solving for the variable corresponding to that row. It uses the previously found values of the variables to solve for the current variable.

3 Explanation of above output

- As we can see from the above i ran all the three function on same 100x100 matrices to find the solution of eq- Ax=B the soution of all the three are same but we can observe:
  - The time taken by without Optimisation is highest **(44.3 micron sec** in the case taken).
  - After that the time taken by Optimised solver **(17.7 micron sec)** is inbetween inbuilt function np.linalg.solve and unoptimised solver
  - The time taken by the np.linalg.solve is the least **(12.2 micron sec)**.

3.1 Testing the above written optimised and unoptimised code on a netlist file for speed checking

3.1.1 Checking on unoptimised code on week 2 asssignment

[10]:
```
# This function will read the given file and calculate the node voltages and␣
 ↪current through the voltages.It will take only the file name as
# input
import cmath
import time
def Ultimate_MNA(filename):
    import math as mt
    CIRCUIT = '.circuit' # Defining these to easy to handle
    END = '.end'
    AC='.ac'
```

```
   start=-1 #start and end index will store the starting and ending index of␣
↪line of required data.
   end=-1
   is_ac=False
   with open(filename,'r') as mainfile:

       content_of_mainfile=mainfile.readlines() #storing the content of file␣
↪in content_of_mainfile
       for each_line in content_of_mainfile:

           if(CIRCUIT==each_line[:len(CIRCUIT)]):
               start=content_of_mainfile.index(each_line)

           if(END==each_line[:len(END)]):
               end=content_of_mainfile.index(each_line)

       #check for invalid file declearation
       if(start>end):
           print("Invalid declaration of circuit")
           exit()
       if(start==0 and end==0):
           print('No circuit found')
           exit()
       if(start==-1 or end==-1):
           print('Invalid declearation of circuit')
           exit()


       cnt_freq=set()
       womega=0    #frequency of ac source
       for each_line in content_of_mainfile:
           if(len(each_line.split())==0):
               continue
           if(each_line.split()[0]==AC):
               womega=float(each_line.split()[2])  # if there is ac source␣
↪than assigning frequency to womega
               cnt_freq.add(each_line.split()[2])

       #check for if there is more than one frequency source in the file given
       womega=womega*mt.pi*2 #Multiplying through 2*pi.

       # If more than one frequency return because we are not dealing with␣
↪multiple frequency.
       if(len(cnt_freq)>1):
           print("More than one frequency")
           return
       #for counting the no of nodes in the circuit=len(st)
```

```
        # print(womega)
        # return

        st=set()  # for counting of different nodes in the file
        cnt_dc=0  #these cnt_dc and cnt_ac are flags that are checking if there␣
↪is ac or dc or both source together.
        cnt_ac=0
        for index in range(start+1,end):
            list=[word for word in content_of_mainfile[index].split('#')[0].
↪split()]
            if(list[1]!='GND'):
                st.add(list[1])
            if(list[2]!='GND'):
                st.add(list[2])
            if(list[3]=='ac'):
                cnt_ac=1
            if(list[3]=='dc'):
                cnt_dc=1


        # If cnt_ac and cnt_dc both are 1 means It's is also a multiple␣
↪frequency than return from here.
        if(cnt_ac and cnt_dc):
            print('Ac and Dc source together (Multiple Frequency)')
            return
        #If the circuit is ac than execute this if statement it contains the␣
↪ac_analysis.
        if(cnt_ac):

            no_of_node=len(st) #storing the no of unique nodes.

            storage={}  #storage is used to store different types of key value␣
↪pair where key->different element of ckt and value-> is the data related to␣
↪that element
            storage.clear()
            voltage=[] # Storage for different voltage source
            voltage.clear()
            cnt_of_voltage_source=0
            for index in range(start+1,end):
                list=[word for word in content_of_mainfile[index].split('#')[0].
↪split()] #reading the main_file line by line and storing it in list by␣
↪ignoring if any comment is there in that line
                if(list[0][0]=='V'):
                    cnt_of_voltage_source+=1
                    voltage.append(list[0])
                if(list[3]!='ac' and list[3]!='dc'):
```

```
                list[3]=float(list[3])
            storage[list[0]]=list[1:]

        for ele in storage:
            if(storage[ele][0][0]=='n'):
                storage[ele][0]=storage[ele][0][1:]
            if(storage[ele][1][0]=='n'):
                storage[ele][1]=storage[ele][1][1:]

        for ele in storage:
            if(ele[0]=='L'):
                value=womega*storage[ele][2]#womega is the frequency of ac
↪source
                storage[ele][2]=complex(0,value)
            elif(ele[0] =='C'):
                value=1/(womega*storage[ele][2])
                storage[ele][2]=complex(0,-value)

        #making MNAmatrix for solving problem
        size=no_of_node+cnt_of_voltage_source

        cnt=size-cnt_of_voltage_source

        MNAmatrix=[]
        AnsMatrix=[]
        # Initialization of MNAmatrix and AnsMatrix by 0.
        for row in range(size):
            lt=[]
            for col in range(size):
                lt.append(0)
            MNAmatrix.append(lt)
            AnsMatrix.append(0)
        #  Traversing through each of the element in storage and filling up
↪the MNAmatrix and AnsMatrix according to MNA algorithm
        for ele in storage:
            if(storage[ele][0]=='GND'):
                storage[ele][0]='0'

            if(storage[ele][1]=='GND'):
                storage[ele][1]='0'

            if(ele[0]=='R'):
                a=int(storage[ele][0])
                b=int(storage[ele][1])
                if(a!=0 and b!=0):
                    MNAmatrix[a-1][a-1]+=1/float(storage[ele][2])
                    MNAmatrix[a-1][b-1]-=1/float(storage[ele][2])
```

```
                        MNAmatrix[b-1][b-1]+=1/float(storage[ele][2])
                        MNAmatrix[b-1][a-1]-=1/float(storage[ele][2])
                    elif(a==0 and b!=0):
                        MNAmatrix[b-1][b-1]+=1/float(storage[ele][2])
                    elif(a!=0 and b==0):
                        MNAmatrix[a-1][a-1]+=1/float(storage[ele][2])
                    else :
                        continue

                elif(ele[0]=='V'):

                    a=int(storage[ele][0])
                    b=int(storage[ele][1])
                    voltageValue=0

                    voltageValue=-1*complex(float(storage[ele][3])*mt.
↪cos(float(storage[ele][4])*(mt.pi/180)),float(storage[ele][3])*mt.
↪sin(float(storage[ele][4])*(mt.pi/180)))
                    AnsMatrix[cnt]+=voltageValue
                    if(a!=0 and b!=0):

                        MNAmatrix[a-1][cnt]+=1
                        MNAmatrix[cnt][a-1]-=1
                        MNAmatrix[b-1][cnt]-=1
                        MNAmatrix[cnt][b-1]+=1
                    elif(a==0 and b!=0):

                        MNAmatrix[b-1][cnt]-=1
                        MNAmatrix[cnt][b-1]+=1

                    elif(a!=0 and b==0):

                        MNAmatrix[a-1][cnt]+=1
                        MNAmatrix[cnt][a-1]-=1
                    else :
                        continue
                    cnt+=1
                elif(ele[0]=='I'):
                    a=int(storage[ele][0])
                    b=int(storage[ele][1])

                    CurrentValue=0
                    CurrentValue=complex(float(storage[ele][3])*mt.
↪cos(float(storage[ele][4])*(mt.pi/180)),float(storage[ele][3])*mt.
↪sin(float(storage[ele][4])*(mt.pi/180)))
                    if(a>0):
                        AnsMatrix[a-1]=AnsMatrix[a-1]-CurrentValue
```

```
                    if(b>0):
                        AnsMatrix[b-1]=AnsMatrix[b-1]+CurrentValue
                elif(ele[0]=='L'):
                    a=int(storage[ele][0])
                    b=int(storage[ele][1])
                    if(a!=0 and b!=0):
                        MNAmatrix[a-1][a-1]+=1/storage[ele][2]
                        MNAmatrix[a-1][b-1]-=1/storage[ele][2]
                        MNAmatrix[b-1][b-1]+=1/storage[ele][2]
                        MNAmatrix[b-1][a-1]-=1/storage[ele][2]
                    elif(a==0 and b!=0):
                        MNAmatrix[b-1][b-1]+=1/storage[ele][2]
                    elif(a!=0 and b==0):
                        MNAmatrix[a-1][a-1]+=1/storage[ele][2]
                    else :
                        continue
                elif(ele[0]=='C'):
                    a=int(storage[ele][0])
                    b=int(storage[ele][1])
                    if(a!=0 and b!=0):
                        MNAmatrix[a-1][a-1]+=1/storage[ele][2]
                        MNAmatrix[a-1][b-1]-=1/storage[ele][2]
                        MNAmatrix[b-1][b-1]+=1/storage[ele][2]
                        MNAmatrix[b-1][a-1]-=1/storage[ele][2]
                    elif(a==0 and b!=0):
                        MNAmatrix[b-1][b-1]+=1/storage[ele][2]
                    elif(a!=0 and b==0):
                        MNAmatrix[a-1][a-1]+=1/storage[ele][2]
                    else :
                        continue
            # fun_st_time=time.time()
            Final_Solution=solve(MNAmatrix,AnsMatrix)  #solution is stored␣
↪inside Final_Solution array.
            # fun_en_time=time.time()
            # print('Time taken by solver is : ',fun_en_time-fun_st_time)
            # Now Steps for printing the values of voltages through different␣
↪Nodes and Current through Voltages sources.
            vol_iter=0
            i=0
            for ele in Final_Solution:
                if i<no_of_node:
                    print(f'Voltage_Node{i+1} :magnitude:  {abs(ele):>10}␣
↪,phase : {cmath.phase(ele)*(180/cmath.pi):>10}')
                    i+=1
                else:
```

```
                    print(f'Current_through_Voltage_source {voltage[vol_iter]} :
↪magnitude:  {abs(ele):>10} , phase : <{cmath.phase(ele)*(180/cmath.pi):>10}␣
↪degree>')
                    vol_iter+=1
            # return fun_en_time-fun_st_time


        #From here Below codes is for DC_Analysis.
        # If cnt_dc is 1 and cnt_ac is 0 that is only dc sources are present so␣
↪do the dc analysis.

        elif(cnt_dc==1 and cnt_ac==0):
            storage={} #It's a dictionary storing different elements with their␣
↪given data.
            resistence=[] #for storing resistences
            voltage=[] #For storing voltages
            storage.clear()
            resistence.clear()
            voltage.clear()
            for index in range (start+1,end):
                list=[word for word in content_of_mainfile[index].split('#')[0].
↪split()] #reading the file and storing inside list and with ignoring the␣
↪comments in line
                # print(list)
                if(list[0][0]=='R'):
                    resistence.append(list[0])
                elif(list[0][0]=='V'):
                    voltage.append((list[0]))
                storage[list[0]]=list[1:]

            for ele in storage:
                if(storage[ele][0][0]=='n'):
                    storage[ele][0]=storage[ele][0][1:] #checking for given␣
↪format of nodes if node is 'n12' than ignore 'n' and store '12'
                if(storage[ele][1][0]=='n'):
                    storage[ele][1]=storage[ele][1][1:]

            st_node=set() #store no of nodes uniquly.
            for ele in storage:
                st_node.add(storage[ele][0])
                st_node.add(storage[ele][1])
            size=len(st_node)-1
            n1=0
            for ele in storage:
                if(ele[0]=='V'):
                    size+=1
                    n1+=1
```

```
            cnt=size-n1
            MNAmatrix=[]  # MNA matrix is the mna matrix ans AnsMatrix is the
↪corresponding the B matrix in Ax=B equation.
            AnsMatrix=[]

            # Initialization of MNAmatrix and AnsMatrix
            for row in range(size):
                lt=[]
                for col in range(size):
                    lt.append(0)
                MNAmatrix.append(lt)
                AnsMatrix.append(0)

            # Traversing through each of the element in storage and filling up
↪the MNAmatrix and AnsMatrix according to MNA algorithm.
            for ele in storage:
                if(storage[ele][0]=='GND'):
                    storage[ele][0]='0'

                if(storage[ele][1]=='GND'):
                    storage[ele][1]='0'
                # If ele in storage is 'R' than filling the MNAmatrix
↪accordingly
                if(ele[0]=='R'):
                    a=int(storage[ele][0])
                    b=int(storage[ele][1])
                    if(a!=0 and b!=0):
                        MNAmatrix[a-1][a-1]+=1/float(storage[ele][2])
                        MNAmatrix[a-1][b-1]-=1/float(storage[ele][2])
                        MNAmatrix[b-1][b-1]+=1/float(storage[ele][2])
                        MNAmatrix[b-1][a-1]-=1/float(storage[ele][2])
                    elif(a==0 and b!=0):
                        MNAmatrix[b-1][b-1]+=1/float(storage[ele][2])
                    elif(a!=0 and b==0):
                        MNAmatrix[a-1][a-1]+=1/float(storage[ele][2])
                    else :
                        continue
                elif(ele[0]=='V'):
                    a=int(storage[ele][0])
                    b=int(storage[ele][1])
                    voltageValue=-1*float(storage[ele][3])
                    AnsMatrix[cnt]=float(voltageValue)
                    if(a!=0 and b!=0):

                        MNAmatrix[a-1][cnt]+=1
                        MNAmatrix[cnt][a-1]-=1
```

```
                    MNAmatrix[b-1][cnt]-=1
                    MNAmatrix[cnt][b-1]+=1
                elif(a==0 and b!=0):

                    MNAmatrix[b-1][cnt]-=1
                    MNAmatrix[cnt][b-1]+=1

                elif(a!=0 and b==0):

                    MNAmatrix[a-1][cnt]+=1
                    MNAmatrix[cnt][a-1]-=1
                else :
                    continue
                cnt+=1
            elif(ele[0]=='I'):
                #for current source assumption is current is going from␣
↪nodea to node b i.e a-->b  first_node to second_node in the given .netlist␣
↪file .
                a=int(storage[ele][0])
                b=int(storage[ele][1])
                CurrentValue=float(storage[ele][3])
                if(a>0):
                    AnsMatrix[a-1]=AnsMatrix[a-1]-CurrentValue
                if(b>0):
                    AnsMatrix[b-1]=AnsMatrix[b-1]+CurrentValue

        Final_Solution=solve(MNAmatrix,AnsMatrix)  #solution is stored␣
↪inside Final_Solution array.
        # fun_st_time=time.time()
        # Final_Solution=solve(MNAmatrix,AnsMatrix)  #solution is stored␣
↪inside Final_Solution array.
        # fun_en_time=time.time()
        # print('Time taken by solver is : ',fun_en_time-fun_st_time)
        # Now Steps for printing the values of voltages through different␣
↪Nodes and Current through Voltages sources.
        vol_iter=0
        node_size=len(st_node)-1
        i=0
        for ele in Final_Solution:
            if i<node_size:
                print(f'Voltage_Node{i+1} : {ele:>20}')
                i+=1
            else:
                print(f'Current_through_Voltage_source {voltage[vol_iter]} :
↪ {ele:>20}')
                vol_iter+=1
        # return fun_en_time-fun_st_time
```

```
tot_st_time1=time.time()
Ultimate_MNA('/Users/amankumar/Documents/semester4/Applied Programming Lab(APL)/
 ↪assignment2/ac_ex_2.netlist')  #Function calling for reading of file ('Junk␣
 ↪Example.netlist') and solving the circuit based on the data
tot_en_time1=time.time()
t_unoptimised=tot_en_time1-tot_st_time1

# we can call function like Ultimate_MNA('some.netlist') to read and solve the␣
 ↪circuit.
```

```
Voltage_Node1 :magnitude:  4.000000000000004 ,phase : 30.000000000000025
Voltage_Node2 :magnitude:  6.000000000000001 ,phase : 59.99999999999999
Voltage_Node3 :magnitude:  168.43334507813856 ,phase : 39.274768504436835
Voltage_Node4 :magnitude:  91.9622812534155 ,phase : 158.07806718862685
Voltage_Node5 :magnitude:  96.14432133702634 ,phase : 152.54100150606314
Voltage_Node6 :magnitude:  8.92395837347785 ,phase : 38.338066057409314
Voltage_Node7 :magnitude:  533.0346508167578 ,phase : 21.99380938872585
Current_through_Voltage_source V1 :magnitude:  2.0905779540055502 , phase :
<153.40947021894834 degree>
Current_through_Voltage_source V3 :magnitude:  2.0026847750018906 , phase :
<157.2035034661318 degree>
Current_through_Voltage_source V2 :magnitude:  5.12777009468804 , phase :
<-118.8825666749316 degree>
Current_through_Voltage_source V4 :magnitude:  20.25001733246732 , phase :
<21.54500746452838 degree>
```

[11]:
```
# This function will read the given file and calculate the node voltages and␣
 ↪current through the voltages.It will take only the file name as
# input
import cmath
import time
def Ultimate_MNA1(filename):
    import math as mt
    CIRCUIT = '.circuit' # Defining these to easy to handle
    END = '.end'
    AC='.ac'
    start=-1 #start and end index will store the starting and ending index of␣
 ↪line of required data.
    end=-1
    is_ac=False
    with open(filename,'r') as mainfile:

        content_of_mainfile=mainfile.readlines() #storing the content of file␣
 ↪in content_of_mainfile
        for each_line in content_of_mainfile:
```

```
            if(CIRCUIT==each_line[:len(CIRCUIT)]):
                start=content_of_mainfile.index(each_line)

            if(END==each_line[:len(END)]):
                end=content_of_mainfile.index(each_line)

        #check for invalid file declearation
        if(start>end):
            print("Invalid declaration of circuit")
            exit()
        if(start==0 and end==0):
            print('No circuit found')
            exit()
        if(start==-1 or end==-1):
            print('Invalid declearation of circuit')
            exit()


        cnt_freq=set()
        womega=0    #frequency of ac source
        for each_line in content_of_mainfile:
            if(len(each_line.split())==0):
                continue
            if(each_line.split()[0]==AC):
                womega=float(each_line.split()[2])  # if there is ac source␣
↪than assigning frequency to womega
                cnt_freq.add(each_line.split()[2])

        #check for if there is more than one frequency source in the file given
        womega=womega*mt.pi*2 #Multiplying through 2*pi.

        # If more than one frequency return because we are not dealing with␣
↪multiple frequency.
        if(len(cnt_freq)>1):
            print("More than one frequency")
            return
        #for counting the no of nodes in the circuit=len(st)
        # print(womega)
        # return

        st=set()  # for counting of different nodes in the file
        cnt_dc=0  #these cnt_dc and cnt_ac are flags that are checking if there␣
↪is ac or dc or both source together.
        cnt_ac=0
        for index in range(start+1,end):
            list=[word for word in content_of_mainfile[index].split('#')[0].
↪split()]
```

```
            if(list[1]!='GND'):
                st.add(list[1])
            if(list[2]!='GND'):
                st.add(list[2])
            if(list[3]=='ac'):
                cnt_ac=1
            if(list[3]=='dc'):
                cnt_dc=1


        # If cnt_ac and cnt_dc both are 1 means It's is also a multiple␣
↪frequency than return from here.
        if(cnt_ac and cnt_dc):
            print('Ac and Dc source together (Multiple Frequency)')
            return
        #If the circuit is ac than execute this if statement it contains the␣
↪ac_analysis.
        if(cnt_ac):

            no_of_node=len(st) #storing the no of unique nodes.

            storage={}  #storage is used to store different types of key value␣
↪pair where key->different element of ckt and value-> is the data related to␣
↪that element
            storage.clear()
            voltage=[] # Storage for different voltage source
            voltage.clear()
            cnt_of_voltage_source=0
            for index in range(start+1,end):
                list=[word for word in content_of_mainfile[index].split('#')[0].
↪split()] #reading the main_file line by line and storing it in list by␣
↪ignoring if any comment is there in that line
                if(list[0][0]=='V'):
                    cnt_of_voltage_source+=1
                    voltage.append(list[0])
                if(list[3]!='ac' and list[3]!='dc'):
                    list[3]=float(list[3])
                storage[list[0]]=list[1:]

            for ele in storage:
                if(storage[ele][0][0]=='n'):
                    storage[ele][0]=storage[ele][0][1:]
                if(storage[ele][1][0]=='n'):
                    storage[ele][1]=storage[ele][1][1:]

            for ele in storage:
                if(ele[0]=='L'):
```

```
                    value=womega*storage[ele][2]#womega is the frequency of ac␣
↪source
                    storage[ele][2]=complex(0,value)
                elif(ele[0] =='C'):
                    value=1/(womega*storage[ele][2])
                    storage[ele][2]=complex(0,-value)

            #making MNAmatrix for solving problem
            size=no_of_node+cnt_of_voltage_source

            cnt=size-cnt_of_voltage_source

            MNAmatrix=[]
            AnsMatrix=[]
            # Initialization of MNAmatrix and AnsMatrix by 0.
            for row in range(size):
                lt=[]
                for col in range(size):
                    lt.append(0)
                MNAmatrix.append(lt)
                AnsMatrix.append(0)
            #  Traversing through each of the element in storage and filling up␣
↪the MNAmatrix and AnsMatrix according to MNA algorithm
            for ele in storage:
                if(storage[ele][0]=='GND'):
                    storage[ele][0]='0'

                if(storage[ele][1]=='GND'):
                    storage[ele][1]='0'

                if(ele[0]=='R'):
                    a=int(storage[ele][0])
                    b=int(storage[ele][1])
                    if(a!=0 and b!=0):
                        MNAmatrix[a-1][a-1]+=1/float(storage[ele][2])
                        MNAmatrix[a-1][b-1]-=1/float(storage[ele][2])
                        MNAmatrix[b-1][b-1]+=1/float(storage[ele][2])
                        MNAmatrix[b-1][a-1]-=1/float(storage[ele][2])
                    elif(a==0 and b!=0):
                        MNAmatrix[b-1][b-1]+=1/float(storage[ele][2])
                    elif(a!=0 and b==0):
                        MNAmatrix[a-1][a-1]+=1/float(storage[ele][2])
                    else :
                        continue

                elif(ele[0]=='V'):
```

```
                    a=int(storage[ele][0])
                    b=int(storage[ele][1])
                    voltageValue=0

                    voltageValue=-1*complex(float(storage[ele][3])*mt.
↪cos(float(storage[ele][4])*(mt.pi/180)),float(storage[ele][3])*mt.
↪sin(float(storage[ele][4])*(mt.pi/180)))
                    AnsMatrix[cnt]+=voltageValue
                    if(a!=0 and b!=0):

                        MNAmatrix[a-1][cnt]+=1
                        MNAmatrix[cnt][a-1]-=1
                        MNAmatrix[b-1][cnt]-=1
                        MNAmatrix[cnt][b-1]+=1
                    elif(a==0 and b!=0):

                        MNAmatrix[b-1][cnt]-=1
                        MNAmatrix[cnt][b-1]+=1

                    elif(a!=0 and b==0):

                        MNAmatrix[a-1][cnt]+=1
                        MNAmatrix[cnt][a-1]-=1
                    else :
                        continue
                    cnt+=1
                elif(ele[0]=='I'):
                    a=int(storage[ele][0])
                    b=int(storage[ele][1])

                    CurrentValue=0
                    CurrentValue=complex(float(storage[ele][3])*mt.
↪cos(float(storage[ele][4])*(mt.pi/180)),float(storage[ele][3])*mt.
↪sin(float(storage[ele][4])*(mt.pi/180)))
                    if(a>0):
                        AnsMatrix[a-1]=AnsMatrix[a-1]-CurrentValue
                    if(b>0):
                        AnsMatrix[b-1]=AnsMatrix[b-1]+CurrentValue
                elif(ele[0]=='L'):
                    a=int(storage[ele][0])
                    b=int(storage[ele][1])
                    if(a!=0 and b!=0):
                        MNAmatrix[a-1][a-1]+=1/storage[ele][2]
                        MNAmatrix[a-1][b-1]-=1/storage[ele][2]
                        MNAmatrix[b-1][b-1]+=1/storage[ele][2]
                        MNAmatrix[b-1][a-1]-=1/storage[ele][2]
                    elif(a==0 and b!=0):
```

```
                    MNAmatrix[b-1][b-1]+=1/storage[ele][2]
                elif(a!=0 and b==0):
                    MNAmatrix[a-1][a-1]+=1/storage[ele][2]
                else :
                    continue
            elif(ele[0]=='C'):
                a=int(storage[ele][0])
                b=int(storage[ele][1])
                if(a!=0 and b!=0):
                    MNAmatrix[a-1][a-1]+=1/storage[ele][2]
                    MNAmatrix[a-1][b-1]-=1/storage[ele][2]
                    MNAmatrix[b-1][b-1]+=1/storage[ele][2]
                    MNAmatrix[b-1][a-1]-=1/storage[ele][2]
                elif(a==0 and b!=0):
                    MNAmatrix[b-1][b-1]+=1/storage[ele][2]
                elif(a!=0 and b==0):
                    MNAmatrix[a-1][a-1]+=1/storage[ele][2]
                else :
                    continue
        # fun_st_time=time.time()
        Final_Solution=c_solve(MNAmatrix,AnsMatrix)  #solution is stored␣
↪inside Final_Solution array.
        # fun_en_time=time.time()
        # print('Time taken by solver is : ',fun_en_time-fun_st_time)
        # Now Steps for printing the values of voltages through different␣
↪Nodes and Current through Voltages sources.
        vol_iter=0
        i=0
        for ele in Final_Solution:
            if i<no_of_node:
                print(f'Voltage_Node{i+1} :magnitude:  {abs(ele):>10}␣
↪,phase : {cmath.phase(ele)*(180/cmath.pi):>10}')
                i+=1
            else:
                print(f'Current_through_Voltage_source {voltage[vol_iter]} :
↪magnitude:  {abs(ele):>10} , phase : <{cmath.phase(ele)*(180/cmath.pi):>10}␣
↪degree>')
                vol_iter+=1
        # return fun_en_time-fun_st_time


    #From here Below codes is for DC_Analysis.
    # If cnt_dc is 1 and cnt_ac is 0 that is only dc sources are present so␣
↪do the dc analysis.

    elif(cnt_dc==1 and cnt_ac==0):
```

```
            storage={} #It's a dictionary storing different elements with their␣
↪given data.
            resistence=[] #for storing resistences
            voltage=[] #For storing voltages
            storage.clear()
            resistence.clear()
            voltage.clear()
            for index in range (start+1,end):
                list=[word for word in content_of_mainfile[index].split('#')[0].
↪split()] #reading the file and storing inside list and with ignoring the␣
↪comments in line
                # print(list)
                if(list[0][0]=='R'):
                    resistence.append(list[0])
                elif(list[0][0]=='V'):
                    voltage.append((list[0]))
                storage[list[0]]=list[1:]

            for ele in storage:
                if(storage[ele][0][0]=='n'):
                    storage[ele][0]=storage[ele][0][1:] #checking for given␣
↪format of nodes if node is 'n12' than ignore 'n' and store '12'
                if(storage[ele][1][0]=='n'):
                    storage[ele][1]=storage[ele][1][1:]

            st_node=set() #store no of nodes uniquly.
            for ele in storage:
                st_node.add(storage[ele][0])
                st_node.add(storage[ele][1])
            size=len(st_node)-1
            n1=0
            for ele in storage:
                if(ele[0]=='V'):
                    size+=1
                    n1+=1

            cnt=size-n1
            MNAmatrix=[]  # MNA matrix is the mna matrix ans AnsMatrix is the␣
↪corresponding the B matrix in Ax=B equation.
            AnsMatrix=[]

            # Initialization of MNAmatrix and AnsMatrix
            for row in range(size):
                lt=[]
                for col in range(size):
                    lt.append(0)
                MNAmatrix.append(lt)
```

```
                AnsMatrix.append(0)

            # Traversing through each of the element in storage and filling up␣
↪the MNAmatrix and AnsMatrix according to MNA algorithm.
            for ele in storage:
                if(storage[ele][0]=='GND'):
                    storage[ele][0]='0'

                if(storage[ele][1]=='GND'):
                    storage[ele][1]='0'
                # If ele in storage is 'R' than filling the MNAmatrix␣
↪accordingly
                if(ele[0]=='R'):
                    a=int(storage[ele][0])
                    b=int(storage[ele][1])
                    if(a!=0 and b!=0):
                        MNAmatrix[a-1][a-1]+=1/float(storage[ele][2])
                        MNAmatrix[a-1][b-1]-=1/float(storage[ele][2])
                        MNAmatrix[b-1][b-1]+=1/float(storage[ele][2])
                        MNAmatrix[b-1][a-1]-=1/float(storage[ele][2])
                    elif(a==0 and b!=0):
                        MNAmatrix[b-1][b-1]+=1/float(storage[ele][2])
                    elif(a!=0 and b==0):
                        MNAmatrix[a-1][a-1]+=1/float(storage[ele][2])
                    else :
                        continue
                elif(ele[0]=='V'):
                    a=int(storage[ele][0])
                    b=int(storage[ele][1])
                    voltageValue=-1*float(storage[ele][3])
                    AnsMatrix[cnt]=float(voltageValue)
                    if(a!=0 and b!=0):

                        MNAmatrix[a-1][cnt]+=1
                        MNAmatrix[cnt][a-1]-=1
                        MNAmatrix[b-1][cnt]-=1
                        MNAmatrix[cnt][b-1]+=1
                    elif(a==0 and b!=0):

                        MNAmatrix[b-1][cnt]-=1
                        MNAmatrix[cnt][b-1]+=1

                    elif(a!=0 and b==0):

                        MNAmatrix[a-1][cnt]+=1
                        MNAmatrix[cnt][a-1]-=1
                    else :
```

```
                        continue
                    cnt+=1
                elif(ele[0]=='I'):
                    #for current source assumption is current is going from␣
 ↪nodea to node b i.e a-->b  first_node to second_node in the given .netlist␣
 ↪file .
                    a=int(storage[ele][0])
                    b=int(storage[ele][1])
                    CurrentValue=float(storage[ele][3])
                    if(a>0):
                        AnsMatrix[a-1]=AnsMatrix[a-1]-CurrentValue
                    if(b>0):
                        AnsMatrix[b-1]=AnsMatrix[b-1]+CurrentValue

            Final_Solution=c_solve(MNAmatrix,AnsMatrix)  #solution is stored␣
 ↪inside Final_Solution array.
            # fun_st_time=time.time()
            # Final_Solution=solve(MNAmatrix,AnsMatrix)  #solution is stored␣
 ↪inside Final_Solution array.
            # fun_en_time=time.time()
            # print('Time taken by solver is : ',fun_en_time-fun_st_time)
            # Now Steps for printing the values of voltages through different␣
 ↪Nodes and Current through Voltages sources.
            vol_iter=0
            node_size=len(st_node)-1
            i=0
            for ele in Final_Solution:
                if i<node_size:
                    print(f'Voltage_Node{i+1} : {ele:>20}')
                    i+=1
                else:
                    print(f'Current_through_Voltage_source {voltage[vol_iter]} :
 ↪ {ele:>20}')
                    vol_iter+=1
            # return fun_en_time-fun_st_time

tot_st_time=time.time()
Ultimate_MNA1('/Users/amankumar/Documents/semester4/Applied Programming␣
 ↪Lab(APL)/assignment2/ac_ex_2.netlist')  #Function calling for reading of␣
 ↪file ('Junk Example.netlist') and solving the circuit based on the data
tot_en_time=time.time()
t_optimised=tot_en_time-tot_st_time

# we can call function like Ultimate_MNA('some.netlist') to read and solve the␣
 ↪circuit.
```

```
Voltage_Node1 :magnitude:  4.000000000000004 ,phase : 30.000000000000025
```

```
Voltage_Node2 :magnitude:  6.000000000000001 ,phase : 59.99999999999999
Voltage_Node3 :magnitude:  168.43334507813856 ,phase : 39.274768504436835
Voltage_Node4 :magnitude:  91.9622812534155 ,phase : 158.07806718862685
Voltage_Node5 :magnitude:  96.14432133702634 ,phase : 152.54100150606314
Voltage_Node6 :magnitude:  8.92395837347785 ,phase : 38.338066057409314
Voltage_Node7 :magnitude:  533.0346508167578 ,phase : 21.99380938872585
Current_through_Voltage_source V1 :magnitude:  2.0905779540055502 , phase :
<153.40947021894834 degree>
Current_through_Voltage_source V3 :magnitude:  2.0026847750018906 , phase :
<157.2035034661318 degree>
Current_through_Voltage_source V2 :magnitude:  5.12777009468804 , phase :
<-118.8825666749316 degree>
Current_through_Voltage_source V4 :magnitude:  20.25001733246732 , phase :
<21.54500746452838 degree>
```

[12]:
```
print(f'The time taken by Using Optimised code is : ',t_optimised)
print(f'The time taken by using Unoptimised code is : ',t_unoptimised)
```

```
The time taken by Using Optimised code is :  0.0002949237823486328
The time taken by using Unoptimised code is :  0.0006000995635986328
```

3.1.2 Explanation of above output:

- As we can see In solving a circuit the time taken by optimised code is less than unoptimised code .
- the time taken by optimised for the given ckt detail is : 0.000294 sec.
- the time taken by unoptimised for given ckt is : 0.000600 sec. I will attach the used input file in the zip file also to verify the outputs.