EE2703 - Week 2

Amizhthni P.R.K EE21B015 <ee21b015@smail.iitm.ac.in>

February 9, 2023

Assignment questions

1. Write a function to find the factorial of N (N being an input) and find the time taken to compute it. This will obviously depend on where you run the code and which approach you use to implement the factorial. Explain your observations briefly.

[1]:
```
#Recursion code
def factorial(N):
    if N==0 or N==1:
        return 1
    r=N
    N-=1
    a=r*factorial(N)
    return a

N=int(input("Enter positive number"))
print(factorial(N))
%timeit factorial(N)
```

```
Enter positive number 5

120
527 ns ± 24.2 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

We employ recursion to solve this problem of finding the factorial of a number. We first create a **base case** that helps to end the loop. The final number that's called because a decrementer is used is 0. We know that the value of 0 factorial is 1. Initially, the function is called from ___main()___ with N passed as an argument. Then, $N-1$ is passed to the same function through a recursive call. In each such recursive calls, the value of variable r storing the argument N is decreased by 1 and the product stored in a finallly becomes the factorial that is required.

[ ]:
```
# Non recursive code
def nonrecurs_fact(N):

    y=1
    if N!=0 or N!=1:
        for i in range(1,N+1):
            y*=i
    return y

```

```
N=int(input("Enter positive number"))
print(nonrecurs_fact(N))
%timeit nonrecurs_fact(N)
```

This code is pretty straightforward, and does not require the use of recursion. We create a function called nonrecursfact(N) taking the parameter N which is the number whose factorial is to be calculated. Variable y is assigned a value of 1 so that when N takes the values of 1 or 0 the answer is 1. A for loop is then initiated which iterates from 1 to N and multiplies all the integral values until the required number N .The value of that is returned is the required factorial. The recursion function takes much longer than the straightforward solution because the function call is done multiple times, which is is time consuming. This is slightly longer than the straightforward approach of using a single for loop. For some very high inputs, the recursion code might exceed the *recursion* limit and yet our simple code might work.

Question 2

Write a linear equation solver that will take in matrices A and b as inputs, and return the vector x that solves the equation $Ax = b$. Your function should catch errors in the inputs and return suitable error messages for different possible problems.

[12]:
```
import numpy as np
x = 0.000000000000000001*(np.random.rand(10,10)) #e-a very small value #a␣
 ↪2matrix using 2D arrray
y = 0.000000000000000001*(np.random.rand(10))
print('A array',x)
print('B array',y)
A=x.tolist()#to convert numpy array to list, since our 'non-linalg solve' code␣
 ↪uses normal lists instead of numpy arrays
B=y.tolist()#to convert numpy array to list, since our 'non-linalg solve' code␣
 ↪uses normal lists instead of numpy arrays
print('A list',A)
print('B list',B)
```

```
A array [[9.53372772e-19 8.22418945e-19 4.34318220e-19 8.58866841e-19
  1.86635876e-19 5.73345686e-20 6.06050005e-19 7.48324907e-19
  9.70420300e-20 5.37679621e-19]
 [1.44542262e-19 1.99324174e-19 2.96706060e-19 4.02011243e-19
  7.42765556e-19 3.74105093e-20 6.84471039e-19 6.98718459e-19
  5.83587202e-19 1.50270969e-19]
 [9.91524261e-19 7.61106893e-19 6.05870899e-20 2.08325997e-19
  3.71998230e-19 1.63776852e-20 3.41417684e-19 2.06097262e-19
  3.11657493e-19 2.40562473e-19]
 [9.52141943e-19 2.48352029e-19 1.83302557e-20 6.00557742e-19
  8.23669192e-19 1.37272659e-19 7.71976333e-19 6.90735494e-19
  7.02320003e-19 1.46370670e-19]
 [6.06733834e-19 2.80031453e-20 3.21158484e-19 4.95101915e-19
  4.91624818e-22 9.35770093e-19 5.82193489e-19 1.87579033e-19
  2.84454017e-19 1.41069165e-19]
 [5.58846571e-19 7.99350036e-19 9.98709247e-19 9.88895549e-19
  3.02533509e-19 8.74801852e-19 1.61233346e-19 7.99117795e-19
  3.86023807e-20 1.10008960e-19]
 [1.31002669e-19 2.19338062e-19 1.01464002e-19 8.85023924e-19
  7.84834784e-19 9.87030761e-20 2.52880145e-19 4.75370840e-19
  5.82699058e-19 2.17201307e-19]
 [8.00042321e-19 7.62237309e-19 2.65757569e-19 5.48268355e-19
  3.95513960e-20 3.89508686e-19 9.60169253e-19 9.54838453e-19
  3.39758068e-19 4.69375530e-19]
 [1.24588077e-19 5.62336604e-19 1.01651216e-19 9.36160934e-19
  4.64354315e-19 6.85753009e-19 1.64157082e-19 9.83614351e-19
  7.27300389e-19 8.23897998e-19]
 [2.38586495e-19 7.73903491e-19 4.42261056e-19 6.86591062e-19
  5.72455094e-19 6.43534559e-19 6.57041283e-19 7.86644565e-19
  9.06665772e-19 9.67817289e-19]]
B array [3.20956976e-19 3.90806241e-19 8.69135080e-19 1.05675543e-19
 5.33504958e-19 3.13743563e-19 2.43763858e-19 9.63353950e-19
 4.09040348e-19 5.38850083e-19]
A list [[9.533727724786486e-19, 8.224189449380186e-19, 4.343182202363338e-19,
8.588668405839514e-19, 1.8663587564156949e-19, 5.73345686349801e-20,
6.06050005190361e-19, 7.483249067060972e-19, 9.704202998451007e-20,
5.376796211790189e-19], [1.4454226214200895e-19, 1.9932417392136804e-19,
2.967060603789067e-19, 4.0201124344540674e-19, 7.427655563996742e-19,
3.74105093123418e-20, 6.844710386771454e-19, 6.987184587158048e-19,
5.835872020167545e-19, 1.502709686267947e-19], [9.915242611538723e-19,
7.611068926528654e-19, 6.058708990753726e-20, 2.0832599661351804e-19,
3.7199823024005055e-19, 1.6377685211849014e-20, 3.414176842808564e-19,
2.060972622416013e-19, 3.1165749303132053e-19, 2.4056247279493304e-19],
[9.521419434727292e-19, 2.4835202891451815e-19, 1.8330255703795098e-20,
6.005577415170798e-19, 8.236691924640152e-19, 1.3727265887752672e-19,
7.719763334531077e-19, 6.907354937290817e-19, 7.023200032632253e-19,
1.4637067032503581e-19], [6.067338343705117e-19, 2.800314527904846e-20,
3.211584838714202e-19, 4.951019145386878e-19, 4.916248176141114e-22,
9.35770092898732e-19, 5.821934893763761e-19, 1.8757903346055494e-19,
2.8445401683741555e-19, 1.4106916506177403e-19], [5.58846570892377e-19,
7.993500356643931e-19, 9.987092467623327e-19, 9.888955490345356e-19,
3.0253350924962854e-19, 8.748018515647696e-19, 1.6123334574354167e-19,
7.991177950702567e-19, 3.8602380739799163e-20, 1.1000896009462835e-19],
[1.3100266865181055e-19, 2.193380623265262e-19, 1.0146400220331332e-19,
8.850239239267031e-19, 7.848347835367273e-19, 9.870307611475438e-20,
2.528801452470666e-19, 4.75370840328811e-19, 5.826990582223984e-19,
2.1720130660073066e-19], [8.000423206227601e-19, 7.622373089213537e-19,
2.657575686921149e-19, 5.4826835498511465e-19, 3.9551396034586576e-20,
3.8950868637630536e-19, 9.601692532221589e-19, 9.548384529108715e-19,
3.397580684733366e-19, 4.693755301103165e-19], [1.245880769984914e-19,
5.6233660379855555e-19, 1.0165121619779694e-19, 9.36160933838308e-19,
4.643543154229419e-19, 6.85753009405614e-19, 1.641570824581591e-19,
9.836143511295436e-19, 7.273003887505841e-19, 8.238979977182207e-19],
[2.3858649504648567e-19, 7.739034909901012e-19, 4.42261056134946e-19,
6.8659106243141205e-19, 5.724550943274439e-19, 6.435345586124312e-19,
6.570412830258218e-19, 7.866445654679635e-19, 9.066657724384768e-19,
9.678172892603568e-19]]
B list [3.209569755363674e-19, 3.9080624096441955e-19, 8.69135079659928e-19,
1.05675543095765e-19, 5.335049578787222e-19, 3.137435632074721e-19,
2.4376385799964672e-19, 9.63353949629165e-19, 4.090403477204844e-19,
5.388500826650933e-19]
```

[ ]:

```
[19]: #Without pivoting
def gauss_withoutpv(A,B):
    n=len(A)
    norm = A[0][0]
    #Intialzing the first diagonal element as norm
    for i in range(len(A[0])):
        A[0][i]/= norm
        #Dividing the row by that element to make diagonal element 1.
    B[0] = B[0]/norm
    #Doing same row operation on B.
    for i in range(1,n):
        for j in range(0, i):
            norm = A[i][j] / A[j][j]
            for k in range(len(A[i])):
                A[i][k] = A[i][k] - A[j][k] * norm
                #Reducing all the elements in the row.
            B[i] = B[i] - B[j] * norm
            #Performing same operation on B matrix.
        norm = A[i][i]
        #Taking next diagonal element
        for m in range(len(A[i])):
            A[i][m] = A[i][m] / norm
            #Divinding next row by the diagonal element.
        B[i] = B[i] / norm
    #Backward substitution
    x=[0]*n #initialising a zero vector
    for i in range(n-1,-1,-1): #traverses the vector B in reverse
        ele=B[i]
        for j in range(i,n):
            ele-=A[i][j]*x[j]#Each time, a different value of element in vector␣
↪x is obtained, by using the previously stored values in x
            #Doing row operation of reducing upper triangle.
            x[i]=(ele)
    return x
```

```
print(gauss_withoutpv(A,B))

%timeit gauss(A,B)
```

```
[-3.4779743697091776, 3.94771140750451, -8.330393199807418, -1.6139225229770062,
11.56200319172508, 7.103291558025894, 6.295604248710341, -1.8780135031931322,
-15.707210156208141, 3.6142351432412325]
```

```
UsageError: Line magic function `%timeit gauss(A,B)` not found.
```

1st attempt, that uses 3 loops:

[20]:
```
#including pivoting and improvising backsubstitution code
def gauss(A,B):
    n=len(A)
    non=0
    for i in range(n):
        mark=0
        temp=1
        while(mark==0 and temp!=0):
        #Starting while loop to check if diagonal element is zero after we␣
 ↪reduce the rows.
            for j in range(0,i):
                if A[j][j]==0:
                #Skipping if the diagonal element is zero.
                    pass
                else:
                    norm=A[i][j]
                    for k in range(len(A[i])):
                        A[i][k]-=A[j][k]*norm
                        #Reducing all elements in the row.
                    B[i]-=B[j]*norm
                    #Applying same operation on B matrix.
            if abs(A[i][i])==0:
            #Checking if diagonal element is zero.
                if (i+1==n):
                #Flagging if its the last row.
                    temp=0
                for u in range(i+1,n):
                #Checking all rows under that row.
                    if abs(A[u][i])>abs(A[i][i]):
                    #Checking for non zero element in the column.
                        temp=1
                        A[i],A[u]=A[u],A[i]
                        B[i],B[u]=B[u],B[i]
                        #Switching the row with zero diagonal element with a␣
 ↪row with non zero diagonal element.
                        #Doing same row operation on B matrix.
                        break
                    else:
                    #Flagging if we don't find any non zero element in the␣
 ↪column.
                        temp=0
            else:
            #Flagging to break out of the while loop.
                mark=1
        if(temp==1):
        #Checking flag to see if diagonal element is zero.
            norm = A[i][i]
            for m in range(len(A[i])):
            #Dividing all elements in the row by diagonal element.
                A[i][m] = A[i][m] / norm
            #Doing same row operation on the B matrix.
            B[i] = B[i] / norm
        else:
            non=1
    #Backward substitution
    for i in range(n-1,-1,-1):
            for j in range(i):
            #Reducing all elements above the i-th diagonal element to zero and␣
 ↪doing same row operation on B matrix.
                B[j]-=A[j][i]*B[i]
                A[j][i]-=A[i][i]*A[j][i]
    for i in range(n):
        if(A[i][i]==0):
        #Checking if there is zero diagonal element.
            if(B[i]!=0 and len(set(A[i]))==1):
            #If number of unique elements in the row is 1, it is definitely a␣
 ↪zero row.
            #If zero row equals non zero there is no solution.
                return "No Solution"
            else:
                non=1
    if(non==1):
#If there are only zero rows equal to zero, infinite solutions.
        return "Infinite Solutions"
    else:
#If its neither infinite nor no solution,it must have unique solution.
        return B

print(gauss(A,B))
%timeit gauss(A,B)
```

```
[-3.4779743697091776, 3.94771140750451, -8.330393199807418, -1.6139225229770062,
11.56200319172508, 7.103291558025894, 6.295604248710341, -1.8780135031931322,
-15.707210156208141, 3.6142351432412325]
75.1 µs ± 1.51 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

[17]:
```
import numpy as np
def linal(a,b):
     print(np.linalg.solve(a, b))
linal(A,B)
#%timeit linal(A,B)
```

```
[ -3.47797437   3.94771141  -8.3303932   -1.61392252  11.56200319
   7.10329156   6.29560425  -1.8780135  -15.70721016   3.61423514]
```

[18]:
```
import numpy as np
def linal(a,b):
     np.linalg.solve(a, b)
linal(A,B)
%timeit linal(A,B)
```

```
21.2 µs ± 234 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

Time Factor: In numpy array method, all tasks are broken into small segments and are then processed parallelly. An array, in addition, occupies less space than a list(since it stores just one kind of datatype). This also contributes to its speed. This can be technically termed as Homogeneous datatypes with contiguous memory, unlike Lists. The NumPy package also integrates C, C++, and Fortran codes in Python, which have very little execution time compared to Python. Considering the operating frequency of a ssingle threaded CPU to be 2GHz, the ideal time of computation for 10k numbers must be around 0.5 ns. **Precision Factor**: This must stem from the differences in how floating point numerals are handled in C and Python. The internal datatype used in C must be such that it has more precision than its Python counterpart. We know that 64 bit floating points are used in array method.

- We are trying to convert A matrix into an identity matrix, we do this by making all the diagonal elements 1 and making all other elements 0 by consecutive row operations.
- First, we initialize n as the length of the 2d array which is the input matrix.
- Then we initialize a flag 'non' to indicate whether the system of linear equations has a unique solution.
- We start a for loop from 0 to n.
- We initialize two more flags 'temp' and 'mark'.
- We then start a while loop which keeps looping if the diagonal element doesn't become 1.
- We then implement a for loop that runs until i-1 to reduce the i-th row, we run another for loop inside the earlier one for doing subtraction on each element. We do the same row operation on the B matrix.
- We then check if the diagonal element has become 0. If it is zero we check for a non-zero element in a row under it in the same column. If all elements under the given element are 0, it is guaranteed that the matrix doesn't have a unique solution. We then update the 'non' flag and break out of the while loop. We then skip that row and go to the next one. All the row operations are performed on the B matrix too.
- After doing this for all rows we are left with a matrix in which all the lower triangular elements are zero and the diagonal elements are either 1 or 0.
- We then try to make all the upper-triangular elements zero too.

- We start from the last diagonal element and subtract all the elements in the above column by doing row operations. No other row column is affected because all the lower triangular elements are zero.
- We do this by running a loop from n-1 to 0 and once we get out of the loop all the upper triangular elements are zero too. If any diagonal element is zero then the elements above it in that column need not be zero.
- We then check if the matrix has infinite solutions or no solution by checking if a row with all elements as 0 equals a non-zero element in B. If any zero row equals a non-zero element the matrix has no solution, otherwise, it has infinite solutions.

Time order: With pivoting>Gauss elimination without pivoting>Linalg solver

Given a circuit netlist in the form described above, read it in from a file, construct the appropriate matrices, and use the solver you have written above to obtain the voltages and currents in the circuit. If you find AC circuits hard to handle, first do this for pure DC circuits, but you should be able to handle both voltage and current sources.

[41]:
```
import math
import cmath
def create_matrix(file):
    fh=open(file,'r')
     #We use cmath module to handle complex numbers that arise when AC sources␣
 ↪are handled
    s=(fh.read().splitlines()) #The file is opened in read mode and we read the␣
 ↪entire data and split it everytime a newline character is detected. We use␣
 ↪this instead of readlines to avoid \n
    a=s.index('.circuit')#We store the indices of '.circuit' and '.end' to␣
 ↪choose the part of the file that lies in between them and that is required␣
 ↪to generate the MNA matrix.
    b=s.index('.end')
    f={}#We create a dictionary to store the AC sources and their frequencies,␣
 ↪dictionary comes in handy when multiple AC sources are to be used
    acflag=0#We create a flag to know whether or not AC sources are used in the␣
 ↪given circuit
#When the size of the file, in terms of number of lines is greater than the␣
 ↪line count of .end, we realise that ac sources are used and create a␣
 ↪dictionary to hold all source and frequency pairs
    if b!=len(s)-1:
        if s[b+1][0:3]=='.ac':
            acflag=1
    r=v=t=c=l=0
    R=[]
    V=[]
    I=[]
    L=[]
    C=[]
    xx=[]
    for i in (s[a+1:b]):
        xx.append(i.split())
    #From the list that contains the netlist that lies between circuit and end,␣
 ↪we select the ones with just resistors, capacitors, inductors, voltage␣
 ↪sources, current sources and append them to separate lists that store these␣
 ↪values separately
    for i in xx:
        if i[0][0]=='R':
            r+=1
            R.append(i)

        if i[0][0]=='V':
            v+=1
            V.append(i)
        if i[0][0]=='I':
            t+=1
            I.append(i)
        if i[0][0]=='L':
            l+=1
            L.append(i)
        if i[0][0]=='C':
            c+=1
            C.append(i)
#Since the AC data consists of nodes 'n' specified in addition to the node␣
 ↪number, we write a separate code that neglects these n strings to give just␣
 ↪the number of the given node
    if acflag==1:
        for i in range(b+1, len(s)):
            yy=s[i].split()

            f[yy[1]]=yy[2]
            w=yy[2]
        for i in range(t):
            for j in range(1,3):
                if I[i][j][0]=="n":
                    I[i][j]=I[i][j][1]
        for i in range(r):
            for j in range(1,3):
                if R[i][j][0]=="n":
                    R[i][j]=R[i][j][1]
        for i in range(c):
            for j in range(1,3):

                if C[i][j][0]=="n":
                    C[i][j]=C[i][j][1]
        for i in range(v):
            for j in range(1,3):
                if V[i][j][0]=="n":
                    V[i][j]=V[i][j][1]
        for i in range(l):
            for j in range(1,3):
                if L[i][j][0]=="n":
                    L[i][j]=L[i][j][1]

#All the matrices may also contain GND instead of 0. In order to facilitate␣
↪numeric handling, we replace all the grounds with zeroes
    for i in range(r):
        for j in range(1,3):
            if R[i][j]=="GND":
                R[i][j]='0'
    for i in range(v):
        for j in range(1,3):
            if V[i][j]=="GND":
                V[i][j]='0'
    for i in range(t):
        for j in range(1,3):
            if I[i][j]=="GND":
                I[i][j]='0'

    for i in range(c):
        for j in range(1,3):
            if C[i][j]=="GND":
                C[i][j]='0'
    for i in range(l):
        for j in range(1,3):
            if L[i][j]=="GND":
                L[i][j]='0'
    ft=0
    if acflag==1:#For circuits like ckt 2 involving both AC and

        for i in V:
            if i[3]=='dc':
                print(f"{file} : Involves both AC and DC, and hence not␣
↪applicable under this question")
                ft=1
                break
        for i in I:
            if i[3]=='dc' and ft==0:
                print(f"{file} : Involves both AC and DC, and hence not␣
↪applicable under this question")
                break
    if ft==1:
        return
```

```
#We then create a list x that contains all the nodes, 0 through the last node.␣
↪This when added with the number of independent voltage sources gives us the␣
↪size of the vector that we need
    x=[]
    for i in range(r):
        if R[i][1] not in x:
            x.append(R[i][1] )
        if R[i][2] not in x:
            x.append(R[i][2] )
    for i in range(c):
        if C[i][1] not in x:
            x.append(C[i][1] )
        if C[i][2] not in x:
            x.append(C[i][2] )
    for i in range(l):
        if L[i][1] not in x:
            x.append(L[i][1] )
        if L[i][2] not in x:
            x.append(L[i][2] )

    for i in range(v):
        if V[i][1] not in x:
            x.append(V[i][1] )
        if V[i][2] not in x:
            x.append(V[i][2] )
#The list is then sorted to make sure that the numbers are in order, i.e. from␣
↪0 to nth node.
    x.sort()
    B=[0]*(len(x)-1)
#Now, we initiate a list B with all zeros, having a length of len(x)-1. This␣
↪minus 1, indicates that the 0 element in x is ignored.



    A=[]
#The elements corresponding to nodes in B are supposed to be 0(Except when␣
↪there are current sources). The elements corresponding to the currents␣
↪through the independent voltage sources are supposed to be entered according␣
↪to the value in the voltage lists.
    for i in V[::-1]:
        B.append(int(i[4]))
    for i in I:
        if int(i[1])!=0:
            B[int(i[1])-1]-=int(i[4])
        if int(i[2])!=0:
            B[int(i[2])-1]+=int(i[4])
#Once list B is completed, with zeros and voltage sources, we now turn to the A␣
↪matrix
```

```
#We iterate through the X list and for all nodes, we carry out the following␣
 ↪operations
    for j in range(1,len(x)):
        #First create a list that can be added to the matrix A. In each␣
 ↪iteration of the loop, we add a list that has all the necessary values of␣
 ↪resistors
        d=[0]*(len(x)-1+v)
        #The length of list d is the size of each row in matrix A
        for i in range(r):
            #We create a loop that iterates over the resistor list and modifies␣
 ↪values in required positions in the list d
            if i<r:
                if x[j] in R[i][1:3]:#The resistor list values between 1 and 3␣
 ↪are taken because they hold the nodes between which the resistor lies
                    index=R[i].index(x[j])
                    #The variable 'index' stores the index of the current loop␣
 ↪value and gets its index from the resistor list

                    if index==1:
                        index2=2
                    elif index==2:
                        index2=1
                    #The index value of the other element is also updated (It␣
 ↪is 2 when the other is 1 and vice versa)
                    d[int(R[i][index])-1]+=float(R[int(i)][3])**-1
                    #The correct position of the list is added with the inverse␣
 ↪of the resistor value
                    if int(R[i][index2])==0:#Care is taken to do no additions␣
 ↪to the list when 0 is one of the nodal voltage values
                        continue
                    else:
                        #The correct position of the list is added with the␣
 ↪inverse of the resistor value of the other node.
                        d[int(R[i][index2])-1]-=float(R[int(i)][3])**-1
#All of these lists are appended to the martrix A. At this stage, A has all the␣
 ↪resistor values updated in it
        A.append(d)
#We next add the current through the voltage sources data in the already␣
 ↪existing rows in the matrix
#This is done by changing the values in the extra columns (after len(x)-1).␣
 ↪That is, we change the value in v number of columns
    kk=0 #We initate a counter to know the position of the voltage␣
 ↪source(within the v number of columns)
```

```
    for j in range(1,len(x)):

        for i in range(v):
                if x[j] in V[i][1:3] and kk<v :
                    kk+=1
#The same logic from resistors is used
                    indexx=V[i].index(x[j])
                    if indexx==1:
                        indexx2=2
                    if indexx==2:
                        indexx2=1
#Care must be taken to make sure that no additions or updations are made when␣
 ↪the ground is encountered
                    if int(V[i][indexx])!=0 :
                        if indexx==1:
                            A[int(V[i][indexx])-1][kk+len(x)-2]+=1
                    #When a given node is first in the netlistpositions (2), we␣
 ↪assign positive polarity to it
                        if indexx==2:
                            A[int(V[i][indexx])-1][kk+len(x)-2]-=1
                    #When a given node is second in the netlistpositions (3),␣
 ↪we assign negative polarity to it
                    if int(V[i][indexx2])!=0 :
                        if indexx2==1:
                             A[int(V[i][indexx2])-1][kk+len(x)-2]+=1
                        if indexx2==2:
                             A[int(V[i][indexx2])-1][kk+len(x)-2]-=1
    #L
#We next update inductor values in the already made A matrix
    for j in range(1,len(x)):
        #Similar logic is applied as in the case of resistors and voltages
        #Variable lf stores the value of the inductor
        #Instead of adding the inverse of resistor value, we add the inverse of␣
 ↪the impedance which is (jwl)
                            #w takes the value of the frequency needed
            #Similarly, the other node is accounted for as well
        for i in range(l):
            if x[j] in L[i][1:3]:
                lf=L[i][3]
                indexx=L[i].index(x[j])
                if indexx==1:
                    indexx2=2
                elif indexx==2:
                    indexx2=1
                A[j-1][int(L[i][indexx])-1]+=(1j*2*math.
 ↪pi*float(w)*float(lf))**-1
```

```
                if int(L[i][indexx2])==0:
                    continue
                else:
                    A[j-1][int(L[i][indexx2])-1]-=(1j*2*math.
↪pi*float(w)*float(lf))**-1
    #C The same logic is used again in the context of capacitors, but with a␣
↪different impedance value of 1/(jwc)
    #The variable lf here stores the value of capacitance
    #We find the index of the current value and also store the index of the␣
↪other value in another variable
           #Instead of adding the inverse of resistor value, we add the inverse␣
↪of the impedance which is 1/jwc
                            #w takes the value of the frequency needed
            #Similarly, the other node is accounted for as well

    for j in range(1,len(x)):
        for i in range(c):
            if x[j] in C[i][1:3]:

                lf=C[i][3]
                indexx=C[i].index(x[j])
                if indexx==1:
                    indexx2=2
                if indexx==2:
                    indexx2=1

                A[j-1][int(C[i][indexx])-1]+=(1j*2*math.pi*float(w)*float(lf))
                if int(C[i][indexx2])==0:
                    continue
                else:
                    A[j-1][int(C[i][indexx2])-1]-=(1j*2*math.
↪pi*float(w)*float(lf))
#  The values now have to be added to the final rows of the A matrix that␣
↪don't come under the nodes in x. These basically assign voltages to nodes␣
↪and so on.
    for j in range(1,len(x)):
        d=[0]*(len(x)-1+v) #since we add an entire row, the new list d that has␣
↪to be appended has to have len(x)-1+v number of elements in it
        for i in range(v):
            if x[j] in V[i][1:3]:
                    indexx=V[i].index(x[j]) #We iterate through the voltage␣
↪lists and assign voltage differences between the required nodes
                    if indexx==1:
                        indexx2=2
                    if indexx==2:
                        indexx2=1
                    if int(V[i][indexx])!=0 :
                        if indexx==1:

                            d[int(V[i][indexx])-1]+=1
                        elif indexx==2:

                            d[int(V[i][indexx])-1]-=1
                    if int(V[i][indexx2])!=0 :

                        if indexx2==1:
                            d[int(V[i][indexx2])-1]+=1
                        elif indexx2==2:
                            d[int(V[i][indexx2])-1]-=1
#We should include a condition that checks the addition of unique lists and␣
↪ignores the condition when 0 voltage i.e GND is encountered
            if list(set(d))!=[0] and (d not in A):
                A.append(d)

    fh.close()
    print(file)
    print('B=',B)
    print('A=',A)
    print('x=',gauss2_withpivot(A,B))
    print(' ')
listss=['ckt1.netlist','ckt2.netlist','ckt3.netlist','ckt4.netlist','ckt5.
↪netlist','ckt6.netlist','ckt7.netlist']
print('The x matrix is decoded as follows: The first n elements correspond to␣
↪the nodes 1 through n. Every element after it till the end of the vector x␣
↪is corresponding to the current through a voltage source')
print(' ')
for i in listss:
    create_matrix(i)
```

```
The x matrix is decoded as follows: The first n elements correspond to the nodes
1 through n. Every element after it till the end of the vector x is
corresponding to the current through a voltage source

ckt1.netlist
B= [0, 0, 0, 0, 5]
A= [[0.00125, -0.00025, 0, 0, 0], [-0.00025, 0.00042500000000000003, -0.000125,
0, 0], [0, -0.000125, 0.000125, 0, 0], [0, 0, 0, 0.0001, -1], [0, 0, 0, -1, 0]]
x= [0.0, 0.0, 0.0, -5.0, -0.0005]

ckt2.netlist : Involves both AC and DC, and hence not applicable under this
question
ckt3.netlist
```

```
B= [0, 0, 0, 0, 0, 10]
A= [[0.001, -0.001, 0, 0, 0, -1], [-0.001, 0.0025, -0.001, 0, 0, 0], [0, -0.001,
0.0025, -0.001, 0, 0], [0, 0, -0.001, 0.0025, -0.001, 0], [0, 0, 0, -0.001,
0.0015, 0], [-1, 0, 0, 0, 0, 0]]
x= [-10.0, -5.029239766081871, -2.573099415204678, -1.4035087719298247,
-0.9356725146198832, -0.004970760233918129]

ckt4.netlist
B= [0, 0, 0, 10]
A= [[0.5, -0.5, 0, -1], [-0.5, 1.0333333333333332, -0.2, 0], [0, -0.2,
0.30000000000000004, 0], [-1, 0, 0, 0]]
x= [-10.0, -5.555555555555556, -3.7037037037037037, -2.2222222222222222]

ckt5.netlist
B= [0, 10]
A= [[0.1, -1], [-1, 0]]
x= [-10.0, -1.0]

ckt6.netlist
B= [0, 0, 0, 5]
A= [[6124.03036408769j, -6283.185307179586j, 0, 0], [-6283.185307179586j,
(0.001+6283.185307179586j), -0.001, 0], [0, -0.001, 0.001, -1], [0, 0, -1, 0]]
x= [(-1.923920880145715e-10-3.1415926534719705e-05j),
(-1.8751873949430718e-10-3.062015181929007e-05j), (-5-6.776263578034403e-21j),
(-0.004999999999812482+3.0620151819290075e-08j)]

ckt7.netlist
B= [5]
A= [[(0.001+6124.03036408769j)]]
x= [(1.3332000879741771e-10-0.0008164557820158 24j)]
```

- The entire implementation was done using Lists(mainly) and Dictionaries
- We use the node by node method, and write all nodal equation coefficients for the given node in the order of the others nodes across the row
- After doing so for resistors, we repeat the same for capacitors and inductors
- In the already included rows, we might have to include the coefficients for the current through the voltage sources
- After this, we need to add extra rows for denoting voltages of each node connected to an Independent voltage source.
- The explanation given below details the same:
- We use cmath module to handle complex numbers that arise when AC sources are handled
- The file is opened in read mode and we read the entire data and split it everytime a newline character is detected. We use this instead of readlines to avoid \n
- We store the indices of '.circuit' and '.end' to choose the part of the file that lies in between them and that is required to generate the MNA matrix.
- We create a dictionary to store the AC sources and their frequencies, dictionary comes in handy when multiple AC sources are to be used

- We create a flag to know whether or not AC sources are used in the given circuit
- When the size of the file, in terms of number of lines is greater than the line count of .end, we realise that ac sources are used and create a dictionary to hold all source and frequency pairs
- From the list that contains the netlist that lies between circuit and end, we select the ones with just resistors, capacitors, inductors, voltage sources, current sources and append them to separate lists that store these values separately
- Since the AC data consists of nodes 'n' specified in addition to the node number, we write a separate code that neglects these n strings to give just the number of the given node
- All the matrices may also contain GND instead of 0. In order to facilitate numeric handling, we replace all the grounds with zeroes
- We then create a list x that contains all the nodes, 0 through the last node. This when added with the number of independent voltage sources gives us the size of the vector that we need
- The list is then sorted to make sure that the numbers are in order, i.e. from 0 to nth node.
- Now, we initiate a list B with all zeros, having a length of len(x)-1. This minus 1, indicates that the 0 element in x is ignored.
- The elements corresponding to nodes in B are supposed to be 0(Except when there are current sources). The elements corresponding to the currents through the independent voltage sources are supposed to be entered according to the value in the voltage lists.
- Once list B is completed, with zeros and voltage sources, we now turn to the A matrix
- We iterate through the X list and for all nodes, we carry out the following operations
- First create a list that can be added to the matrix A. In each iteration of the loop, we add a list that has all the necessary values of resistors
- The length of list d is the size of each row in matrix A
- We create a loop that iterates over the resistor list and modifies values in required positions in the list d
- The resistor list values between 1 and 3 are taken because they hold the nodes between which the resistor lies

- The variable 'index' stores the index of the current loop value and gets its index from the resistor list
- The index value of the other element is also updated (It is 2 when the other is 1 and vice versa)
- The correct position of the list is added with the inverse of the resistor value
- All of these lists are appended to the martrix A. At this stage, A has all the resistor values updated in it
- We next add the current through the voltage sources data in the already existing rows in the matrix
- This is done by changing the values in the extra columns (after len(x)-1). That is, we change the value in v number of columns
- We initate a counter to know the position of the voltage source(within the v number of columns)
- The same logic from resistors is used
- Care must be taken to make sure that no additions or updations are made when the ground is encountered
- When a given node is first in the netlistpositions (2), we assign positive polarity to it

- When a given node is second in the netlistpositions (3), we assign negative polarity to it

- We next update inductor values in the already made A matrix
- Similar logic is applied as in the case of resistors and voltages
- Variable lf stores the value of the inductor
- Instead of adding the inverse of resistor value, we add the inverse of the impedance which is (jwl). W takes the value of the frequency needed
- The same logic is used again in the context of capacitors, but with a different impedance value of 1/(jwc)
- The variable lf here stores the value of capacitance
- We find the index of the current value and also store the index of the other value in another variable
- Instead of adding the inverse of resistor value, we add the inverse of the impedance which is 1/jwc
- w takes the value of the frequency needed
- The values now have to be added to the final rows of the A matrix that don't come under the nodes in x. These basically assign voltages to nodes and so on.
- Since we add an entire row, the new list d that has to be appended has to have len(x)-1+v number of elements in it
- We iterate through the voltage lists and assign voltage differences between the required nodes
- We should include a condition that checks the addition of unique lists and ignores the condition when 0 voltage i.e GND is encountered

[redacted]