Week2 Assigment

Shah Jainam EE21B122

February 8, 2023

1 Problem 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]:
```python
#Code to Generate N factorial
#Using Reccurssion
def factorial(N):
    if N==1 or N==0:
        return 1
    else:
        return N*factorial(N-1)

#Using a for loop
def fact(N):
    num = 1
    for i in range(N):
        num = num*(i+1)
    return num

#Take appropiate input
try:
    N = int(input('Enter a integer number: '))
    print('This is Reccursion')
    print(factorial(N))
    %timeit factorial(N)

    print('This is a for loop')
    print(fact(N))
    %timeit fact(N)
except:
    print('Invalid Number')
```

```
Enter a integer number: 999
This is Reccursion
402387260077093773543702433923003985719374864210714632543799910429938512398629020
592044208486969404800479988610197196058631666872994808558901323829669944590997
```

```
24504087073759918823627727188732519779505950995276120874975462497043601418278094
64649629105639388743788648733711918104582578364784997701247663288983595573543251
31853239584630755574091142624174743493475534286465766116677973966688202912073791
43853719588249808126867838374559731746136085379534524221586593201928090878297308
43139284440328123155861103697680135730421616874760967587134831202547858932076716
91324484262361314125087802080002616831510273418279777047846358681701643650241536
91398281264810213092761244896359928705114964975419909342221566832572080821333186
11681155361583654698404670897560290095053761647584772842188967964624494516076535
34081989013854424879849599533191017233555566021394503997362807501378376153071277
61926849034352625200015888535147331611702103968175921510907788019393178114194545
25722386554146106289218796022383897147608850627686296714667469756291123408243920
81601537808898939645182632436716167621791689097799119037540312746222899880051954
44414282012187361745992642956581746628302955570299024324153181617210465832036786
90611726015878352075151628422554026517048330422614397428693306169089796848259012
54583271682264580665267699586526822728070757813918581788896522081643483448259932
66043367660176999612831860788386150279465955131156552036093988180612138558600301
43569452722420634463179746059468257310379008402443243846565724501440282188525247
09351906209290231364932734975655139587205596542287497740114133469627154228458623
77387538230483865688976461927383814900140767310446640259899490222221765904339901
88601856652648506179970235619389701786004081188972991831102117122984590164192106
88843871218556461249607987229085192968193723886426148396573822911231250241866493
53143970137428531926649875337218940694281434118520158014123344828015051399694290
15348307764456909907315243327828826986460278986432113908350621709500259738986355
42771967428222487575867657523442202075736305694988250879689281627538488633969099
59826280956121450994871701244516461260379029309120889086942028510640182154399457
15680594187274899809425474217358240106367740459574178516082923013535808184009699
63725242305608559037006242712434169090041536901059339838357779394109700277534720
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000
362 µs ± 4.27 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
This is a for loop
40238726007709377354370243392300398571937486421071463254379991042993851239862902
05920442084869694048004799886101971960586316668729948085589013238296699445909974
24504087073759918823627727188732519779505950995276120874975462497043601418278094
64649629105639388743788648733711918104582578364784997701247663288983595573543251
31853239584630755574091142624174743493475534286465766116677973966688202912073791
43853719588249808126867838374559731746136085379534524221586593201928090878297308
43139284440328123155861103697680135730421616874760967587134831202547858932076716
91324484262361314125087802080002616831510273418279777047846358681701643650241536
91398281264810213092761244896359928705114964975419909342221566832572080821333186
11681155361583654698404670897560290095053761647584772842188967964624494516076535
34081989013854424879849599533191017233555566021394503997362807501378376153071277
61926849034352625200015888535147331611702103968175921510907788019393178114194545
25722386554146106289218796022383897147608850627686296714667469756291123408243920
81601537808898939645182632436716167621791689097799119037540312746222899880051954
44414282012187361745992642956581746628302955570299024324153181617210465832036786
```

```
90611726015878352075151628422554026517048330422614397428693306169089796848259012
54583271682264580665267699586526822728070757813918581788896522081643483448259932
66043367660176999612831860788386150279465955131156552036093988180612138558600301
43569452722420634463179746059468257310379008402443243846565724501440282188525247
09351906209290231364932734975655139587205596542287497740114133469627154228458623
77387538230483865688976461927383814900140767310446640259899490222221765904339901
88601856652648506179970235619389701786004081188972991831102117122984590164192106
88843871218556461249607987229085192968193723886426148396573822911231250241866493
53143970137428531926649875337218940694281434118520158014123344828015051399694290
15348307764456909907315243327828826986460278986432113908350621709500259738986355
42771967428222487575867657523442202075736305694988250879689281627538488633969099
59826280956121450994871701244516461260379029309120889086942028510640182154399457
15680594187274899809425474217358240106367740459574178516082923013535808184009699
63725242305608559037006242712434169090041536901059339838357779394109700277534720
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000
287 µs ± 3.76 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

- For finding factorial of number I have used reccursion method and a for loop method,in reccursion method the fuction is called again and again until N equals 1, this method usese the property N! = N*(N-1)! and solves the problem
- The second method I have used is just used a simple for loop to calculate N! it uses the property N! = 1*2*3…(N-1)*N
- The time of reccusive function is 348μs greater then for loop implementation of 283μs as reccursive fuction is calling the fuction again and again which takes time greater then the for loop implementation; for loop implementation calls the fuction ones and multiplies the values till N and returns the answer and thus is relatively faster then reccursive implementation

2 Problem 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.
- Time your solver to solve a random 10×10 system of equations. Compare the time taken against the `numpy.linalg.solve` function for the same inputs.

2.1 Using Gauss Elemination

[2]:
```python
import numpy as np
def row_swap(A,r1,r2):          #Swapping rows r1 and r2 of matrix A
    A[r1],A[r2] = A[r2],A[r1]

def agumented(A,b):             #Making a Augumented Matrix using matrix A and b
    n = len(A)
    for i in range(len(A)):
        A[i].append(b[i][0])
```

```python
#To calculatee determinant of the matrix in case of singular matrix case
#calculate minor of the matrix A and element ij
def matminor(A,i,j):
    B = []
    for row in A[:i]+A[i+1:]:
        B.append(row[:j] + row[j+1:])
    return B
#Use reccurssion to calculate determinant of the matrix
def det(A):
    determinant = 0
    #-------------------------------------------------------------------
    if len(A) == 1:
        #-------------
        return A[0][0]
    elif len(A) == 2:
        #------------------------------------
        return A[0][0]*A[1][1]-A[0][1]*A[1][0]      #Formula for determinant of the matrix
    else:
        for k in range(len(A)):
            determinant += (pow(-1,k))*A[0][k]*det(matminor(A,0,k))#Use reccursion to solve
                                                                              #for n>2
        #-----------------
        return determinant

def normalization(A,c):   #Normalise the row c of the matrix A
    m = len(A[0])
    n = len(A)
    #n = len(A)
    try:
        for i in range(m):
            if A[c][c] != 1:
                norm = A[c][c]
                for j in range(m):
                    A[c][j] = A[c][j]/norm
    except ZeroDivisionError:
        for i in range(n):
            for j in range(i+1,n):
                if A[j][i] != 0:
                    #swap A[j] with A[i]
                    row_swap(A,j,i)
        normalization(A,c)


def elemination(A,r,c):   #make element rc of the matrix A zero
```

```
    #normalization(A,c)
    m = len(A[0])
    for i in range(m):
        if A[r][c] != 0:
            ele = A[r][c]
            for j in range(m):
                A[r][j] = A[r][j] - ((ele)*A[c][j])

def gauss(A,b):           #Gauss eleminantion Algorithm
    agumented(A,b)
    m = len(A)
    for i in range(m):
        normalization(A,i)
        for j in range(m):
            if i != j:
                elemination(A,j,i)
    return A

def solutions(A,b):      #Extracting the solution after Gauss elemination and␣
 ↪handling edge cases
    n = len(b)
    #print(A)
    d = det(A)
    if d == 0:
        return ('No Solutions')
    else:
        sol = []
        B = gauss(A,b)
        for i in range(len(B)):
            sol.append(B[i][-1])

        return sol
```

[3]:

```
a = np.random.rand(10,10).tolist()
b = np.random.rand(10,1).tolist()

print('This is solving Ax=b with NumPy')
print(np.linalg.solve(np.array(a), np.array(b)))
%timeit np.linalg.solve(np.array(a), np.array(b))
print()
print('This is solving Ax=b without NumPy')
print(np.array(solutions(a,b)))
%timeit solutions(a,b)
```

```
This is solving Ax=b with NumPy
[[ 83.97198602]
 [-26.14422017]
```

```
 [-28.36046881]
 [ 41.10333139]
 [ 22.89398194]
 [-69.72669974]
 [-61.04808511]
 [  7.24325476]
 [ 44.47335631]
 [-32.14035456]]
18.3 µs ± 711 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)

This is solving Ax=b without NumPy
[ 83.97198602 -26.14422017 -28.36046881  41.10333139  22.89398194
 -69.72669974 -61.04808511   7.24325476  44.47335631 -32.14035456]
2.36 s ± 46.9 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

- The time taken for NumPy is 17μs compares to 2s of pure python code this shows that python is objectively slower then C++ or C as NumPy does calculations in C++ or C and does not call Python whereas doing all this is Pure Python takes considerable amount of time

2.2 Solving using $\mathbf{A}^{-1}\mathbf{b}$

[4]:
```python
import numpy as np
#To calculatee determinant of the matrix in case of singular matrix case
#calculate minor of the matrix A and element ij
def matminor(A,r,c):
    B = []
    for row in A[:r] + A[r+1:]:
        B.append(row[:c] + row[c+1:])
    return B

#Use reccurssion to calculate determinant of the matrix
def det(A):
    determinant = 0
    #-----------------------------------------------------------------------
    if len(A) == 1:
        #-------------
        return A[0][0]
    elif len(A) == 2:
        #-------------------------------------
        return A[0][0]*A[1][1]-A[0][1]*A[1][0]
    else:
        for k in range(len(A)):
            determinant += (pow(-1,k))*A[0][k]*det(matminor(A,0,k))
        #-----------------
        return determinant
#Exchange row with coulmn and column with row
def tran(A):
    n = len(A)
```

```
    B = [[0] * n for i in range(n)]
    #--------------------------------
    for i in range(n):
        for j in range(n):
            B[j][i] = A[i][j]
    #--------------------------------
    return B

#Find Inverse of the matrix A
def inv(A):
    n = len(A)
    #find determinnat
    deter = det(A)
    if deter == 0:
        return 'No Solutions'
    #------------------------------------------------
    if len(A) == 2:
        return [[A[1][1]/deter, -1*A[0][1]/deter],
                [-1*A[1][0]/deter, A[0][0]/deter]]
    else:
        #Find the adjoint matrix co
        co = []
        #---------------------------------------------
        for i in range(n):
            corow = []
            for j in range(n):
                minor = matminor(A,i,j)
                #appending co-factors to the determinant co
                corow.append(pow(-1,i+j)*det(minor))
            co.append(corow)
        #taking transpose to get adjoint of matrix A in co
        co = tran(co)
        #----------------------------------------------
        #divide each element by determinant of A and get the inverse of the␣
 ↪matrix
        for i in range(n):
            for j in range(n):
                co[i][j] = co[i][j]/deter
        #----------------------------------------------
        return co

# Make a function to multiply A and B matricies
def matmult(A,B):
    n = len(A)
    m = len(B[0])
    #to store A*B
    result = [[0] * m for i in range(n)]
```

```
    #------------------------------------------
    for i in range(len(A)):
        for j in range(len(B[0])):
            for k in range(len(B)):
                #multipy row with column
                result[i][j] += A[i][k] * B[k][j]
    #-------------------------------------------
    return result

#find A^-1*b
def  matsolver(A,b):
    if len(A[0]) != len(b):
        return 'Non compatible dimensions of the matricies'
    A_inverse = inv(A)
    result = matmult(A_inverse,b)
    return result
```

[5]:

```
a = np.random.rand(10,10).tolist()
b = np.random.rand(10,1).tolist()

print('This is solving Ax=b with NumPy')
print(np.linalg.solve(np.array(a), np.array(b)))
%timeit np.linalg.solve(np.array(a), np.array(b))
print()
print('This is solving Ax=b without NumPy')
print(np.array(solutions(a,b)))
%timeit matsolver(a,b)
```

```
This is solving Ax=b with NumPy
[[ 1.29826532]
 [ 0.01017759]
 [-0.71508534]
 [-1.30416039]
 [-0.07868267]
 [ 0.09375643]
 [ 0.66991393]
 [ 0.27031047]
 [ 1.03711746]
 [ 0.14595029]]
20.2 µs ± 1.34 µs per loop (mean ± std. dev. of 7 runs, 100,000 loops each)

This is solving Ax=b without NumPy
[ 1.29826532  0.01017759 -0.71508534 -1.30416039 -0.07868267  0.09375643
  0.66991393  0.27031047  1.03711746  0.14595029]
93 ns ± 0.0634 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)
```

- The time taken for NumPy is 19μs compares to 95ns of pure python code this shows that python is objectively slower then C++ or C as NumPy does calculations in C++ or C and does not call Python whereas doing all this is Pure Python takes considerable amount of time

3 Problem 3

- **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.**

3.0.1 1. Filtering the content of the netlist file

[6]:
```
#import required libraries
import numpy as np
import re

#make a custom error for having more than one frequencies in netlist
class MoreFreqError(Exception):
    pass

#netlist = input('What is the file name? ')

netlist = 'ckt5.netlist'  #<---------uncommet this to hardcode file name

#read the file and load it contents to a list
try:
    with open(netlist,'r') as f:
        para = f.read().splitlines()
except FileNotFoundError:
    print('Make sure the path of the .netlist or .txt file is same as this␣
↪notebook')

#find .circuit and .ac or .end and remove others unneccesaary things
try:
    index1 = para.index('.circuit')
    index2 = para.index('.end')
    string = re.findall('\.ac.*', '\n'.join(para))

    #update the content to have data between .circuit and .end using string␣
↪slicing method
    para = para[index1:index2+1]

    #update content to have data regarding frequencies
    para.extend(string)

    #to check if there is only one frequency
```

```
    w = []
    if len(string)>=1:
        for line in string:
            w_st = line.split()
            w.append(int(w_st[2]))
    try:
        if max(w) != min(w):
            raise MoreFreqError
    except MoreFreqError:
        print('More than one AC frequencys')
    except ValueError:
        pass


    #remove comments from the paragraph or the strings after '#'
    filtered = []
    for line in para:
        #l = re.sub(r'#.*','',line)
        l = line.split('#')
        filtered.append(l[0])
    #the proccesed content is filtered
    print(filtered)
except ValueError:
    print('Invalid Netlist')
#find .ac in the content as .ac comes after .end
```

```
['.circuit', 'R1 GND 1 10', 'V1 GND 1 dc 10', '.end']
```

3.0.2 2. Changing the content to useful data format

[7]:
```
#delete .circuit and .end and return if the circuit is ac or dc and if ac then
# collect operating frequency
def delete_headings(circ):
    data = [' '.join(x.split()) for x in circ]
    mode = circ[-1]
    data = [n for n in data if not n.startswith('.')]
    if mode == '.end':
        return (0,'dc',data)
    else:
        k = mode.split()
        return (k[2],'ac', data)

(w0,mode,y) = delete_headings(filtered)
#frequency of circuit
w0 = int(w0)
flag_ac = 0
flag_dc = 0
```

```
#Making useful data list
data = []
try:
    for line in y:
        line = line.split(' ')
        if line[1] == 'GND':       # Changing the GND node to 0 node for␣
 ↪calculation purposes
            line[1] = '0'
        if line[2] == 'GND':
            line[2] = '0'
        #Handling for non nuumeric nodes
        if line[1][0] == 'n':
            line[1] = line[1][1:]
        if line[2][0] == 'n':
            line[2] = line[2][1:]

        if line[3] == 'dc' or line[3] == 'ac': #Could have use mode
            if line[3] == 'dc':
                flag_dc = 1
            else:
                flag_ac = 1
            del line[3]                       # delete 'ac' or 'dc' strings
        x = ' '.join(line)
        data.append(x)
except:
    print('Invalid Netlist')
data
```

[7]: ['R1 0 1 10', 'V1 0 1 10']

[8]:
```
#intializing variables
rlc_count = 0
v_count = 0
i_count = 0

if flag_ac and flag_dc:    #Cathing case of AC DC mixed circuit
    print('***AC and DC mixed circuit found***')
```

3.0.3 Counting elements and Error handling in netlist

[9]:
```
# number of components and branches
line_count = len(data)
branch_count = 0     #no need for this as line_count = branch_count but done␣
 ↪anyways

#counting number of passive elements rlc and active elemets V ,I
for i in range(line_count):
```

```
    comp = data[i][0]       #the 0th index shows name of component
    value_count = len(data[i].split()) #value count is a nested list containg a␣
↪list
                                       #that contains name of component
                                       #postive node, negative node and value␣
↪of
                                       #the component
    if (comp == 'R') or (comp == 'L') or (comp == 'C'): #add capatlize function
        if value_count != 4:           # handling errors
            print("Error in netlist")
            print('Error in passive elements RLC')
        rlc_count += 1
        branch_count += 1
    elif comp == 'V':
        if value_count != 5 and value_count != 4:
            print("Error in netlist ")
            print('Error in Voltage source/node')
        v_count += 1
        branch_count += 1
    elif comp == 'I':
        if value_count != 4 and value_count != 5:
            print("Error in netlist ")
            print('Error in Current source/branch')
        i_count += 1
        branch_count += 1
    else:
        print('Unknow element found in netlist')
```

3.0.4 Making functions to help in load data from file to a dictionary

[10]:
```
#using data to make a dictionary that can help in extracting data easily
data_dic = {'element':[],'+node':[],'-node':[],'value':[],'phase':[]}

#making a loading function that takes values from data
#and stores to dictionary for passive elements
def load_rlc(line_number):
    line = data[line_number].split()
    data_dic['element'] += [line[0]]
    data_dic['+node'] += [int(line[1])]
    data_dic['-node'] += [int(line[2])]
    data_dic['value'] += [float(line[3])]
    if line[0][0] == 'R':                   #for ac circuit
        data_dic['phase'] += [0.0]
    elif line[0][0] == 'L':
        data_dic['phase'] += [np.pi/2]
```

```
    elif line[0][0] == 'C':
        data_dic['phase'] += [np.pi/2]


#making a loading function that takes values from data
#and stores to dictionary for active elements
def load_sources(line_number):
    line = data[line_number].split()
    if mode == 'dc':
        data_dic['element'] += [line[0]]
        data_dic['+node'] += [int(line[1])]
        data_dic['-node'] += [int(line[2])]
        data_dic['value'] += [float(line[3])]
        data_dic['phase'] += [0.0]
    elif mode == 'ac':
            data_dic['element'] += [line[0]]
            data_dic['+node'] += [int(line[1])]
            data_dic['-node'] += [int(line[2])]
            data_dic['value'] += [float(line[3])]
            try:                                      #for ac mode
                data_dic['phase'] += [float(line[4])]
            except IndexError:
                data_dic['phase'] += [0.0]

    else:
        return('Unknown mode')




# load_rlc(0)
# load_rlc(1)
# load_rlc(2)
```

[11]:
```
def count_nodes():
    n = [([0]*(line_count+1)) for i in range(1)]
    for i in range(line_count - 1):
        n[0][data_dic['+node'][i]] = data_dic['+node'][i]
        n[0][data_dic['-node'][i]] = data_dic['-node'][i]
        #largetst node
    if max(data_dic['-node']) > max(data_dic['+node']):
        largest = max(data_dic['-node'])
    else:
        largest =  max(data_dic['+node'])

    # check for unfilled elements, skip node 0
    for i in range(1,largest):
```

```
        if n[0][i] == 0:
            print('Error in node order')
    return largest
```

[12]:
```
#load data into dictionary
for i in range(line_count):
    comp = data[i][0]
    if (comp == 'R') or (comp == 'L') or (comp == 'C'):
        load_rlc(i)
    elif (comp == 'V') or (comp == 'I'):
        load_sources(i)
    else:
        print('Unknown Elelment Error')

# count number of nodes
num_nodes = count_nodes() #maximum node number in the circuit
```

[13]:
```
#The dictionary
print(data_dic)
#The frequency
print(w0)
```

```
{'element': ['R1', 'V1'], '+node': [0, 0], '-node': [1, 1], 'value': [10.0,
10.0], 'phase': [0.0, 0.0]}
0
```

3.0.5 Using dicitionary made above to count the number of nodes in the circuit ignoring the GND or 0 node

3.0.6 Make Matricies to solve the MNA

[14]:
```
#make matricies for calculations
#V matrix to store variable names of voltage nodes
#V = np.zeros((num_nodes,1))
#I matrix to store values of independent current sources
I = np.zeros((num_nodes,1),dtype='complex_')
#G matrix as in conductance matrix for the equations
G = np.zeros((num_nodes,num_nodes),dtype='complex_')


# count the number of element types that affect the size of the B, C, D, E and␣
↪J arrays
k = v_count #number of branches with unknown currents for MNA
if k != 0:
    B = np.zeros((num_nodes,k),dtype='complex_')
    C = np.zeros((k,num_nodes),dtype='complex_')
    D = np.zeros((k,k),dtype='complex_')
    E = np.zeros((k,1),dtype='complex_')
```

```
    #J = np.zeros((k,1),dtype='complex_')
```

3.1 Generating the MNA Matrices

- There are three matrices we need to generate, the **A** matrix, the **X** matrix and the **Z** matrix. Each of these will be created by combining several individual sub-matrices.

3.2 Making of A matrix

- The **A** matrix will be developed as the combination of 4 smaller matrices, G, B, C, and D.

$$\begin{bmatrix} G & B \\ C & D \end{bmatrix}$$

- The A matrix is (m+n)×(m+n) (n is the number of nodes, and m is the number of independent voltage sources) and:
- the G matrix is n×n and is determined by the interconnections between the passive circuit elements (resistors)
- the B matrix is n×m and is determined by the connection of the voltage sources.
- the C matrix is m×n and is determined by the connection of the voltage sources. (B and C are closely related, particularly when only independent sources are considered).
- the D matrix is m×m and is zero if only independent sources are considered.

3.3 G Matrix

- The G matrix is an n×n matrix formed in two steps

1. Each element in the diagonal matrix is equal to the sum of the conductance (one over the resistance) of each element connected to the corresponding node. So the first diagonal element is the sum of conductances connected to node 1, the second diagonal element is the sum of conductances connected to node 2, and so on.
2. The off diagonal elements are the negative conductance of the element connected to the pair of corresponding node. Therefore a resistor between nodes 1 and 2 goes into the G matrix at location (1,2) and locations (2,1).

[15]:
```
for i in range(branch_count):
    n1 = data_dic['+node'][i] #first node is +
    n2 = data_dic['-node'][i] #second node is -
    # iterate through each element and save them in temporary variable g
    # then appy the rules to make G matrix
    comp = data_dic['element'][i][0]   #The first letter shows element type
    if comp == 'R':
        g = 1/data_dic['value'][i]      #1/R
    if comp == 'L':
        if mode == 'ac':
            g = 1/(1j*w0*data_dic['value'][i])  #1/jwL got AC
        else:
            g = np.inf                          #for dc L behaves as close␣
↪circuit in steady state
```

```
    if comp == 'C':
        if mode == 'ac':
            g = 1j*w0*data_dic['value'][i]      #jwC
        else:
            g = 0                               #for dc C behaves as open␣
↪circuit in steady state

    if (comp == 'R') or (comp == 'L') or (comp == 'C'):
        # If neither side of the element is connected to ground
        # then subtract it from appropriate location in matrix.
        if (n1 != 0) and (n2 != 0):
            G[n1-1,n2-1] += -g
            G[n2-1,n1-1] += -g

        # If node 1 is connected to ground, add element to diagonal of matrix
        if n1 != 0:
            G[n1-1,n1-1] += g

        # same for for node 2
        if n2 != 0:
            G[n2-1,n2-1] += g

print(G)
```

```
[[0.1+0.j]]
```

3.4 B matrix

- The B matrix is an n×m matrix with only 0, 1 and -1 elements. Each location in the matrix corresponds to a particular voltage source (first dimension) or a node (second dimension). If the positive terminal of the ith voltage source is connected to node k, then the element (i,k) in the B matrix is a 1. If the negative terminal of the ith voltage source is connected to node k, then the element (i,k) in the B matrix is a -1. Otherwise, elements of the B matrix are zero.

[16]:
```
# generate the B Matrix
# loop through all the branches and process independent voltage sources
sources = 0   # count source number
for i in range(branch_count):
    n1 = data_dic['+node'][i]
    n2 = data_dic['-node'][i]
    # process all the independent voltage sources
    #x = df.loc[i,'element'][0]   #get 1st letter of element name
    comp = data_dic['element'][i][0]
    if comp == 'V':
        if v_count > 1:
            if n1 != 0:
                B[n1-1][sources] = 1
```

```
            if n2 != 0:
                B[n2-1][sources] = -1
            sources += 1   #increment source count
        else:
            if n1 != 0:
                B[n1-1] = -1
            if n2 != 0:
                B[n2-1] = +1

B
```

[16]:
```
array([[1.+0.j]])
```

3.5 C Matrix

- The C matrix is an m×n matrix with only 0, 1 and -1 elements. Each location in the matrix corresponds to a particular node (first dimension) or voltage source (second dimension). If the positive terminal of the ith voltage source is connected to node k, then the element (k,i) in the C matrix is a 1. If the negative terminal of the ith voltage source is connected to node k, then the element (k,i) in the C matrix is a -1. Otherwise, elements of the C matrix are zero.
- In other words, the C matrix is the transpose of the B matrix. (This is not the case when dependent sources are present.)

[17]:
```
# generate the C matrix
source = 0   # count source number
for i in range(branch_count):
    #n1 = df.loc[i,'p node']
    n1 = data_dic['+node'][i]
    n2 = data_dic['-node'][i]
    # process all the independent voltage sources
    #x = df.loc[i,'element'][0]   #get 1st letter of element name
    comp = data_dic['element'][i][0]
    if comp == 'V':
        if v_count > 1:
            if n1 != 0:
                C[source][n1-1] = 1
            if n2 != 0:
                C[source][n2-1] = -1
            source += 1   #increment source count
        else:
            if n1 != 0:
                C[0][n1-1] = -1
            if n2 != 0:
                C[0][n2-1] = +1
print(C)
```

```
[[1.+0.j]]
```

3.6 D Matrix

- The D matrix is an m×m matrix that is composed entirely of zeros. (It can be non-zero if dependent sources are considered.)

[18]:
```
print(D)
```

```
[[0.+0.j]]
```

3.7 I Matrix

- The i matrix is an n×1 matrix with each element of the matrix corresponding to a particular node. The value of each element of i is determined by the sum of current sources into the corresponding node. If there are no current sources connected to the node, the value is zero.

[19]:
```
# Current matrix containg current at each node
for i in range(branch_count):
    #n1 = df.loc[i,'p node']
    n1 = data_dic['+node'][i]
    n2 = data_dic['-node'][i]
    # process all the passive elements, save conductance to temp value
    comp = data_dic['element'][i][0]
    if comp == 'I':
        #g = data_dic['element'][i]
        g = data_dic['value'][i]*np.exp(1j*data_dic['phase'][i]) # For AC in,␣
 ↪case dc phase = 0
        #g = data_dic['value'][i]
        # sum the current into each node
        if n1 != 0:
            I[n1-1] = I[n1-1] - g
        if n2 != 0:
            I[n2-1] = I[n2-1] + g

print(I)
```

```
[[0.+0.j]]
```

[20]:
```
#Making matrix to store names of unknown variables
#Using python list instead of numpy array as storing strings is easy in them
rows, cols = num_nodes,1
V = [(''*1) for i in range(num_nodes)]
J = [(''*1) for i in range(v_count)]
```

3.8 V Matrix

- The v matrix is an n×1 matrix formed of the node voltages. Each element in v corresponds to the voltage at the equivalent node in the circuit (there is no entry for ground – node 0).

- For example if a circuit has three nodes, the v matrix is

$$\begin{bmatrix} V_1 \\ V_2 \\ V_3 \end{bmatrix}$$

[21]:
```
# Voltage at each node
for i in range(num_nodes):
    V[i] = V[i] + 'v' +f'{i+1}'

V
```

[21]: ['v1']

3.9 J Matrix

- The j matrix is an m×1 matrix, with one entry for the current through each voltage source. So if there are two voltage sources V1 and V2, the j matrix will be:

$$\begin{bmatrix} I_{V1} \\ I_{V2} \end{bmatrix}$$

[22]:
```
# matrix J for current through voltage sources
sources = 0    # count source number
for i in range(branch_count):
    # process all the passive elements
    comp = data_dic['element'][i][0]
    if comp == 'V':
        #J[sorces] = sympify('I_{:s}'.format(df.loc[i,'element']))
        J[sources] = 'I_{:s}'.format(data_dic['element'][i])
        sources += 1

J
```

[22]: ['I_V1']

3.10 E matrix

- The **E** matrix is an m×1 matrix with each element of the matrix equal in value to the corresponding independent voltage source.

[23]:
```
# generate the E matrix
source = 0    # count source number
for i in range(branch_count):
    # process all the passive elements
    #get 1st letter of element name
    comp = data_dic['element'][i][0]
    if comp == 'V':
```

```
        #E[source] = data_dic['element'][i]
        E[source] = data_dic['value'][i]*np.exp(1j*data_dic['phase'][i])
        source += 1

print(E)
```

```
[[10.+0.j]]
```

3.11 Z Matrix

- The z matrix holds our independent voltage and current sources and will be developed as the combination of 2 smaller matrices i and e. It is quite easy to formulate

$$\begin{bmatrix} I \\ E \end{bmatrix}$$

- The z matrix is (m+n)×1 (n is the number of nodes, and m is the number of independent voltage sources) and:
  - The **I** matrix is n×1 and contains the sum of the currents through the passive elements into the corresponding node (either zero, or the sum of independent current sources).
  - The **E** matrix is m×1 and holds the values of the independent voltage sources.

[24]:
```
#matrix containg independent voltage source and current sources
# Z = I + E
Z = np.concatenate((I,E))
Z
```

[24]:
```
array([[ 0.+0.j],
       [10.+0.j]])
```

3.12 X Matrix

- The x matrix holds our unknown quantities and will be developed as the combination of 2 smaller matrices v and j. It is considerably easier to define than the A matrix.

$$\begin{bmatrix} V \\ J \end{bmatrix}$$

- The **X** matrix is (m+n)×1 (n is the number of nodes, and m is the number of independent voltage sources) and:
- the **V** matrix is n×1 and hold the unknown voltages
- the **J** matrix is m×1 and holds the unknown currents through the voltage sources

[25]:
```
#matrix containg node voltage and current through independent voltage sources
# unknown variables matrix
# X = V + J
X = V[:] + J[:]
print(X)
```

```
['v1', 'I_V1']
```

3.13 Making Matrix A

$$\begin{bmatrix} G & B \\ C & D \end{bmatrix}$$

[26]:
```
#The A matrix is (m+n) by (m+n) and will be developed as the combination of 4␣
↪smaller matrices, G, B, C, and D.
n = num_nodes
m = v_count
A = np.zeros((m+n,m+n),dtype='complex_')
for i in range(n):
    for j in range(n):
        A[i,j] = G[i,j]

if v_count > 1:
    for i in range(n):
        for j in range(m):
            A[i,n+j] = B[i,j]
            A[n+j,i] = C[j,i]
else:
    for i in range(n):
        A[i,n] = B[i]
        A[n,i] = C[0][i]

A  # display the A matrix
```

[26]:
```
array([[0.1+0.j, 1. +0.j],
       [1. +0.j, 0. +0.j]])
```

3.13.1 Solving the MNA

[27]:
```
# Solve AX = Z using either gauss or inverse method
if mode =='dc':
    print(np.array(solutions(A.tolist(),Z.tolist())))
    print(X)
else:
    print(np.array(matsolver(A.tolist(),Z.tolist())))
    print(X)
```

```
[10.+0.j -1.-0.j]
['v1', 'I_V1']
```

refrences: https://lpsa.swarthmore.edu/Systems/Electrical/mna/MNAAll.html