EE2703 Week 2

ANIRUDH B S (EE21B019)

February 6, 2023

1 Assignment

The following are the problems you need to solve for this assignment. You need to submit your code (either as standalone Python script or a Python notebook), a PDF document explaining your solution (either generated from the notebook or a separate LaTeX document), and any supporting files you may have (such as circuit netlists you used for testing your code).

- 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.
- 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.
- 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.

1.1 Bonus assignments

- (Small bonus): after reading in the netlist, allow some or all sources and impedances to be controlled interactively - either using widgets or other mechanisms. On each change you should recompute the currents and voltages and display them.

2 The Solution :-

This assignment consists of codes and documentation for the basic codes on factorial, Gaussian Elimination and Netlist Extractor cum Cirucit Solver. Not to forget, the code for small bonus is included in the Circuit Solver. Another file will be uploaded for the large bonus as instructed in class. !!

2.0.1 Factorial of a Number

Firstly let us use the iterative approach to find the factorial of a number N :

[1]:
```
def factorial(N): #Returns factorial
    if type(N) is not int : #Checks type of input
```

```
        print("Factorial not defined for non integers")
    else :
        if N >= 0 :
            if N == 0 : #Corner case of N being equal to zero.
                return 1
            prod = 1
            for i in range (1, N+1) : #For each number i in 1,...N, we multiply␣
↪i with the running product and store it
                prod = prod * i
            return prod
        else :
            print("Factorial not defined for negative integers")
%timeit factorial(7)
```

```
769 ns ± 17.8 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

[2]:
```
factorial(7)
```

[2]:
```
5040
```

2.0.2 Reason for iterative approach

The iterative approach of factorial computation takes O(n) time on an average. As n becomes larger and larger, it takes more and more time. However, this method of computation of factorial is quite fast for smaller numbers. The relevant checks such as factorials not being defined for fractions or negative numbers have been put in place.

Now, let us see what happens if we use the recursive definition of a factorial.

[3]:
```
def factorialrecurse(N): #Returns factorial
    if type(N) is int : #Integer check
        if N>=0 :
            if N==0 or N == 1: #Basic corner case of N ==0 or N==1
                return 1
            return N * factorialrecurse(N-1) #Use of recursion to return␣
↪factorial of N-1
        else :
            print("Negative factorials do not exist!")
    else :
        print("Factorials not defined for non integers.")
%timeit factorialrecurse(7)
```

```
1.77 µs ± 69.1 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

[4]:
```
factorialrecurse(7)
```

[4]:
```
5040
```

Run	Factorial Iterative (in us)	Factorial Recursive (in us)
1	0.852	1.8
2	0.729	0.716
3	0.795	0.540

2.0.3 Reason for recursive approach

The recursive approach of factorial computation takes O(n) time on an average. As n becomes larger and larger, it takes more and more time. However, this method of computation of factorial is quite fast for smaller numbers. The relevant checks for integers have been put in place. The recursive method is often used by proficient programmers to show their skillset to the world as it is an elegant approach to solve problems.

As is the main aim of recursion, to get the output by repeatedly calling the same function many times, here also the recursive definition of factorial works !

2.0.4 Observations

The computer executes the code in nanoseconds, indicating the processor frequency is in Giga Hertz. My processor runs at 2.30 GHz. (Data taken from system settings)

It appears that the recursive approach takes slightly more time compared to the iterative approach.

As already mentioned in the presentation, the time taken depends entirely on the processor and the time when you are running the code. The output time taken is a function of many external parameters that are beyond our control. When I run the code today, the output may be different from what time output I got yesterday. However, the key take away remains that our processor is able to execute code in nanoseconds.

2.0.5 Gaussian Elimination

[5]:
```
import sys #To exit from terminal incase of inconsistency
import numpy as np
```

[6]:
```
A1 = np.array([[1, 0, 0], [0, 1, 0], [0,0, 1]])
B1 = np.array([[1], [2], [3]])
```

[7]:
```
def GaussianElimination(A,b): #Solves a system of linear equations given␣
↪matrices A and b using Gaussian Elimination with parital pivoting (swapping of␣
↪rows included)
    n = A.shape[0]
    m = A.shape[1]
    if m!=n :
        sys.exit("Inconsistent Equation")
    aug = np.zeros((n,n+1), dtype = np.complex128)
    x = np.zeros(n, dtype = np.complex128)   #Solution is initialised to zeros
    aug = np.c_[A,b]  #Function to create the augumented matrix [A|b] by␣
↪concatenating A and b column wise
    try :
```

```
        for i in range(n-1):
            if aug[i][i] == 0: #Checks if pivot is zero
                for l in range(i+1, n):
                    if abs(aug[l][i]) > abs(aug[i][i]):
                        aug[[i,l]] = aug[[l,i]]  #Swaps two rows of the␣
↪augmented matrix
            for j in range (i+1,n):
                if aug[i][i] == 0 :
                    raise ZeroDivisionError("Singular Matrix") #Raises exception
                else :
                    r = aug[j][i]/aug[i][i] #Find the ratio for each row
                for k in range(i,n+1):
                    aug[j][k] = aug[j][k] - r*aug[i][k] #Convert matrix to Row␣
↪Echelon Matrix
        if aug[n-1][n-1] == 0:
            raise ZeroDivisionError("Singular Matrix") #Raises exception
        else :
            x[n-1] = aug[n-1][n]/aug[n-1][n-1] #Start of Back Substitution
        for i in range(n-2,-1,-1):
            x[i] = aug[i][n]
            for j in range(i+1,n):
                x[i] = x[i] - aug[i][j]*x[j] #Back Substitution
            try :
                x[i] = x[i]/aug[i][i]
            except :
                print("Singular Matrix")
        return x
    except ZeroDivisionError:
        print("Singular Matrix")
```

2.0.6 There are two Gaussian Elimination Codes !

- The above Gaussian Elimination code handles complex entries as well which shall be needed for circuit analysis.
- The below Gaussian Elimination works only on real entries, which are used to solve Matrix equations the same way it is done in a Linear Algebra course at school.

The only motivation to keep two codes separately is to highlight the difference that by just changing dtype argumnet in np.zeros() we can convert a matrix from purely real to a matrix that handles complex operations as well !

[8]:
```
def GaussianEliminationReal(A,b): #Solves a system of linear equations given␣
↪matrices A and b using Gaussian Elimination with parital pivoting (swapping of␣
↪rows included)
    n = A.shape[0]
    m = A.shape[1]
    if m!=n :
        sys.exit("Inconsistent Equation")
```

```
    aug = np.zeros((n,n+1))
    x = np.zeros(n)   #Solution is initialised to zeros
    aug = np.c_[A,b]  #Function to create the augumented matrix [A|b] by␣
↪concatenating A and b column wise
    try :
        for i in range(n-1):
            if aug[i][i] == 0: #Checks if pivot is zero
                for l in range(i+1, n):
                    if abs(aug[l][i]) > abs(aug[i][i]):
                        aug[[i,l]] = aug[[l,i]]  #Swaps two rows of the␣
↪augmented matrix
            for j in range (i+1,n):
                if aug[i][i] == 0 :
                    raise ZeroDivisionError("Singular Matrix") #Raises exception
                else :
                    r = aug[j][i]/aug[i][i] #Find the ratio for each row
                for k in range(i,n+1):
                    aug[j][k] = aug[j][k] - r*aug[i][k] #Convert matrix to Row␣
↪Echelon Matrix
        if aug[n-1][n-1] == 0:
            raise ZeroDivisionError("Singular Matrix") #Raises exception
        else :
            x[n-1] = aug[n-1][n]/aug[n-1][n-1] #Start of Back Substitution
        for i in range(n-2,-1,-1):
            x[i] = aug[i][n]
            for j in range(i+1,n):
                x[i] = x[i] - aug[i][j]*x[j] #Back Substitution
            try :
                x[i] = x[i]/aug[i][i]
            except :
                print("Singular Matrix")
        return x
    except ZeroDivisionError:
        print("Singular Matrix")
```

2.0.7 Reason for this implementation of Gaussian Elimination

The above code works on the principle of Gaussian Elimination with partial pivoting. For reference, I have used : https://web.mit.edu/10.001/Web/Course_Notes/GaussElimPivoting.html#:~:text=Gaussian%20Elimination%20

The gist of the code is simple : We iterate over the columns and check if any diagonal element is zero, if it is, then we swap that row with the row having maximum element. Later on we check if even after swapping there is any diagonal element that is zero, if YES, then the matrix is singular with infinitely many or zero solutions, if NO, then we proceed to compute the unique solution to the system of equations. We use forward elimination to convert A|b to an upper triangular matrix and then use backward substitution to extract the answer.

[9]:
```
x = GaussianEliminationReal(A1, B1)
print(x)
```

```
[1. 2. 3.]
```

[10]:
```
def npsolve(A,b):
    try :
        x = np.linalg.solve(A,b)
        return x
    except :
        print("Singular Matrix")
```

[11]:
```
x = npsolve(A1,B1)
print(x)
```

```
[[1.]
 [2.]
 [3.]]
```

[13]:
```
A2 = np.random.random((10,10))
B2 = np.random.random((10,1))
```

[14]:
```
%timeit GaussianElimination(A2, B2)
```

```
504 µs ± 19.7 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

[15]:
```
%timeit npsolve(A2, B2)
```

```
9.5 µs ± 101 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

[16]:
```
GaussianEliminationReal(A2, B2)
```

[16]:
```
array([ -5.56872446, -10.33446227,  -3.37351366,   4.40729287,
        12.91345244,  -1.6702017 ,   9.8581886 ,   2.66133085,
        -8.17100885,  -2.29420852])
```

[17]:
```
npsolve(A2, B2)
```

[17]:
```
array([[ -5.56872446],
       [-10.33446227],
       [ -3.37351366],
       [  4.40729287],
       [ 12.91345244],
       [ -1.6702017 ],
       [  9.8581886 ],
       [  2.66133085],
       [ -8.17100885],
       [ -2.29420852]])
```

[18]:
```
print("Note : Number of variables must be equal to number of equations")
m = int(input("Enter the number of equations"))
print("Enter the matrix A")
A3 = np.zeros((m,m))
B3 = np.zeros((m,1))
for i in range(m):
    for j in range(m):
        try :
            print(f"Enter {i+1} {j+1} entry of A")
            A3[i][j] = float(input())
        except TypeError:
            print("Enter a float entry in A")
print("Enter B now")
for i in range(m):
    try :
        print(f"Enter {i+1} entry of B")
        B3[i] = float(input())
    except TypeError:
        print("Enter a float entry in B")
x = GaussianEliminationReal(A3,B3)
y = npsolve(A3,B3)
print (x)
print (y)
```

```
Note : Number of variables must be equal to number of equations
Enter the number of equations3
Enter the matrix A
Enter 1 1 entry of A
2
Enter 1 2 entry of A
2
Enter 1 3 entry of A
0
Enter 2 1 entry of A
4
Enter 2 2 entry of A
4
Enter 2 3 entry of A
0
Enter 3 1 entry of A
1
Enter 3 2 entry of A
1
Enter 3 3 entry of A
1
Enter B now
Enter 1 entry of B
1
```

Run	Gaussian Elimination (in us)	npsolve (in us)
1	489	9.15
2	629	12.5
3	540	8.86

```
Enter 2 entry of B
2
Enter 3 entry of B
3
Singular Matrix
Singular Matrix
None
None
```

2.0.8 Comparision in timing :

The GaussianElimination() function looks to run slower than npsolve() function that works on the standard numpy library based function of solving linear equations. However, as the dimension of the matrix increases, the GaussianElimination() is way slower than npsolve(). However, all problems that might occur have been taken care of in GaussianElimination().

One major drawback of the GaussianElimination() is that it works only for consistent equations, that is, A is a square matrix with non zero determinant. It does not work for a A matrix whose dimension is m x n where m and n are distinct. This is because in such situations, we either have infinite solutions or no solutions

2.0.9 Observation

2.0.10 Spice Netlist Extraction

This code includes the small bonus aspects as well. The idea of small bonus comes from the fact that using a slider widget would enhance the interactability of the circuit solver with the user. This adds a lot of advantage in the domain of circuit analysis as we are able to see the output change upon changing the input or any impedance and thus analyse our circuit further.

[19]:
```
import numpy as np
import ipywidgets as wdg
import sys
import cmath
```

2.0.11 Assumption :

It is assumed that all the above packages are installed. In case it is not installed, you can do a pip install package

[20]:
```
class R: #Created a class for storing R objects
    value = 0
    nodes = ()
    def __init__(self, n1, n2, value) :
```

```
        self.value = value
        self.nodes = (n1, n2)
```

[21]:
```
class C : #Defined a class for capacitor element to store it nicely
    value = 0
    nodes = ()
    def __init__(self,  n1, n2, value) :
        self.value = value
        self.nodes = (n1, n2)
```

[22]:
```
class L : #Defined a class for inductor element to store it nicely
    value = 0
    nodes = ()
    def __init__(self, n1, n2, value) :
        self.value = value
        self.nodes = (n1, n2)
```

[23]:
```
class V : #Created a class for storing V objects
    value = 0
    nodes= ()
    def __init__(self,  n1, n2, value) :
        self.value = value
        self.nodes = (n1, n2)
```

[24]:
```
class I : #Created a class for storing I objects
    value = 0
    nodes = ()
    def __init__(self, n1, n2, value) :
        self.value = value
        self.nodes = (n1, n2)
```

2.0.12 Why classes ?

Classes present a way of storing elements nicely in any OOP. Thus, using classes to store individual elements is interesting. Adding to the above reason, given the expected length of the code, using classes and functions make the working of the code more intuitive and obvious and also provide modularity details to the reader of this code.

[25]:
```
def findckt(lines) : #Created a function to find .circuit in netlist
    for line in lines :
        if line == ".circuit" or line == ".circuit\n" :
            return lines.index(line)
    return -1
```

[26]:
```
def findend(lines) : #created a function to find .end in netlist
    for line in lines :
        if line == ".end" or line == ".end\n" :
            return lines.index(line)
```

```
    return -1
```

2.0.13 Use of findckt and findend

The above mentioned functions essentially find .circuit and .end in the circuit. Additionally, a further check has been added later in the main() so that the netlist contains EXACTLY ONE .circuit and .end

[27]:
```
def getMR (M, nodes, value) : #Created a function to change M after considering␣
↪resistances
    n1, n2 = nodes
    M[n1][n1] += 1/value
    M[n1][n2] -= 1/value
    M[n2][n1] -= 1/value
    M[n2][n2] += 1/value
```

[28]:
```
def getMC(nodes, M, frequency, value):#Given a capacitor updates matrix M
    n1, n2 = nodes
    M[n1][n1] += complex(0,2*np.pi*frequency*value)
    M[n1][n2] -= complex(0,2*np.pi*frequency*value)
    M[n2][n1] -= complex(0,2*np.pi*frequency*value)
    M[n2][n2] += complex(0,2*np.pi*frequency*value)
```

[29]:
```
def getML(nodes, M, frequency, value): #Given a inductor, updates the matrix M
    n1, n2 = nodes
    try :
        M[n1][n1] += complex(0,1/(2*np.pi*frequency*value))
        M[n1][n2] -= complex(0,1/(2*np.pi*frequency*value))
        M[n2][n1] -= complex(0,1/(2*np.pi*frequency*value))
        M[n2][n2] += complex(0,1/(2*np.pi*frequency*value))
    except ZeroDivisionError :
        M[n1][n1] += 1
        M[n1][n2] -= 1
        M[n2][n1] -= 1
        M[n2][n2] += 1
```

[30]:
```
def getMV (M, b, nodes, value, n, idx):#Created a function to change M after␣
↪considering voltage sources
    n1, n2 = nodes
    M[n1][n+idx] += 1
    M[n+idx][n1] += 1
    M[n+idx][n2] -= 1
    M[n2][n+idx] -= 1
    b[n+idx] = value
```

[31]:
```

```

```
def getMI (b, nodes, value) : #Created a function to change M after considering␣
↪current sources
    n1, n2 = nodes
    b[n1] += value
    b[n2] -= value
```

2.0.14 Reasons for adopting an OOP based approach

I have adopted a classical OOP form of approaching this problem by creating classes and respective objects and storing them. This is because this would make my code look neater and more presentable. Moreover, it is quite intuitive from the name of the function or class itself the function it is meant to serve. Thus, the OOP form provides an edge over all other possible forms of approaching the given problem.

[32]:
```
n = 0 #Number of nodes in circuit
k = 0 #Number of independent voltage sources in circuit
r = [] #Stores list of resistance objects
v = [] #Stores list of voltage sources objects
cur = [] #Stores list of current source objects
c = [] #Stores list of capacitor objects
ll = [] #Stores list of inductor objects
nodes = {} #Stores dictionary mapping of nodes
```

[33]:
```
w = set() #Stores set of distinct frequencies
frequency  = 0 #Stores last entered frequency of source
phase = 0 #Stores phase of complex quantity
```

[34]:
```
lwr = [] #List that stores widgets to modify resistances
lwc = [] #List that stores widgets to modify capacitances
lwl = [] #List that stores widgets to modify inductances
lwv = [] #List that stores widgets to modify voltage sources
lwi = [] #List that stores widgets to modify current sources
```

2.0.15 Approach to Solve :

This approach slightly defers from the general idea of network nodal analysis.

As taught in EE2015, given n nodes we can have n-1 nodal equations. However here the number of equations is n as I have explicitly added another equation $V_{gnd} = 0$ which is implicitly assumed in the former statement.

Apart from that the age old idea of modified nodal analysis (MNA) is applied considering currents through all the independent voltage sources and voltage at all the nodes and are displayed finally.

2.0.16 Regarding use of Global variables

Global variables like n, k, c, lwl are all quite useful in a way. Since, they are used throughout the program, maintaining a single copy of these variables and modifying the values when necessary proves to be a useful thing to do given the humungous nature of this code.

[35]:
```
def reset() : #Function to reset after all the operations with respect to a␣
 ↪netlist are completed.
    global r
    global v
    global cur
    global c
    global ll
    global n
    global k
    global w
    global frequency
    global phase
    global lwr
    global lwc
    global lwl
    global lwv
    global lwi
    global nodes
    #I have imported all the global variables used and reset them all to be zero
    nodes = {}
    r = []
    v = []
    cur = []
    c = []
    ll = []
    n = 0
    k = 0
    w.clear()
    frequency = 0
    phase = 0
    lwr = []
    lwc = []
    lwl = []
    lwv = []
    lwi = []
```

3 IMPORTANT :

The reset() function needs to be run before entering the next netlist. This is done by calling the reset() function after running the netlist and doing the required modifications.

3.0.1 Comments :

Python is an amazingly versatile language. I created a list of objects belonging to one class before, now I just created a list of widgets. That's how amazing python is!!

[36]:
```
def f(x):
    return x #Identity function
```

[37]:
```
def widget(val1): #Used to create widget instance for R, L and C
    w = wdg.interactive(f, x=wdg.FloatSlider(min = 0.0000000001,max =␣
 ↪100000000,step = 0.0000000001, value = val1))
    display(w)
    return w
def widgetvi(val1): #Used to create widget instance for V and I sources
    w = wdg.interactive(f, x=wdg.FloatSlider(min = 0.001,max = 1000,step = 0.
 ↪001, value = val1))
    display(w)
    return w
```

3.0.2 Additional Note :

The widget I shall be using is Float Slider.

I have created two different widget instances because the range of values permitted for current and voltage source and that of resistor, inductor and capacitor are expected to be different.

The ranges of each widget has been explicitly mentioned below (later) for clariy.

[38]:
```
def main():
    name = input("Enter the name of the netlist including extension (.netlist)")
    if name[-8:] != ".netlist" : #Check for .netlist in ending
        print(".netlist not found")
        sys.exit("Netlist should end with .netlist")
    with open(name) as f:
        lines = f.readlines()
    cktcount = 0
    endcount = 0
    global w #Use the set of all W (frequencies)
    for line in lines :
        s = []
        if line == "" or line == "\n":
            continue
        s = line.split()
        if s[0] == ".ac" :
            w.add(float(s[2]))
        if line == ".circuit" or line == ".circuit\n":
            cktcount +=1
        if line == ".end" or line == ".end\n":
            endcount +=1
    if cktcount > 1 : #Check for multiple .circuit
        print("Too many .circuit s")
        sys.exit("Netlist should contain only one .circuit")
    if endcount > 1 : #Check for multiple .end
```

```
        print("Too many .end s")
        sys.exit("Netlist should contain only one .end")
    a = findckt(lines)
    b = findend(lines)
    if a == -1 :
        print (".circuit not found")
    if b == -1 :
        print (".end not found")
    global nodes
    nodes = {} #Stores list of nodes in the circuit and their mapping
    count = 1
    global n
    global k
    global r
    global c
    global ll
    global v
    global cur
    global frequency
    global phase
    #Using global keyword we access the lists and variables that are used␣
↪throughout the code
    if a != -1 and b!=-1 :
        circuit = lines[a+1:b]
        for l in circuit :
            ele = l.split()
            for k_ in [1,2] :
                if ele[k_] == "GND" : #GND is assigned a voltage of zero
                    nodes[ele[k_]] = 0
                if ele[k_] not in nodes.keys():
                    nodes[ele[k_]] = count
                    count += 1
            if l[0] == 'R': #Storing a resistor nicely
                print("Found a resistor")
                ro = R(nodes[ele[1]], nodes[ele[2]],  float(ele[3]))
                r.append(ro)
            if l[0] == 'C': #Storing a capacitor nicely
                print("Found a capacitor")
                co = C(nodes[ele[1]], nodes[ele[2]], float(ele[3]))
                c.append(co)
            if l[0] == 'L': #Storing a inductor nicely
                print("Found a inductor")
                lo = L(nodes[ele[1]], nodes[ele[2]], float(ele[3]))
                ll.append(lo)
            if l[0] == 'V': #Storing a voltage source nicely
                if ele[3] == "ac" : #AC Source
                    phase = float(ele[5])
```

```
                    value = complex(float(ele[4])*np.cos(phase),␣
↪float(ele[4])*np.sin(phase))
                elif ele[3] == "dc" : #DC source
                    value = float(ele[4])
                    w.add(0)
                else : #Checks if AC/ DC entry is left empty
                    sys.exit("Enter AC or DC")
                vo = V(nodes[ele[1]], nodes[ele[2]],value)
                v.append(vo)
                print("Found a voltage source with value: ", vo.value)
                k = k + 1
            if l[0] == 'I': #Storing a current source nicely
                if ele[3] == "ac" : #AC source
                    phase = float(ele[5])
                    value = complex(float(ele[4])*np.cos(phase),␣
↪float(ele[4])*np.sin(phase))
                elif ele[3] == "dc" : #DC source
                    value = float(ele[4])
                    w.add(0)
                else : #Checks if AC/ DC entry is left empty
                    sys.exit("Enter AC or DC")
                io = I(nodes[ele[1]], nodes[ele[2]], value)
                cur.append(io)
                print("Found a current source with value: ", io.value)
        n = len(nodes)
```

3.0.3 Reason for code in the above manner

In the main() function we first check if the net list is valid and then proceed to get the elements and append them to an already existing empty list.

3.0.4 Use of a dictionary to store nodes

I have used a dictionary to store nodes. This is because the node entered by the user could either be a string like n1, n2, GND etc or integers like 1, 2, 0 etc. Either way while reading from the file, it is always a string. So extracting tokens as strings is a better approach as against directly converting them to int. Thus, I have adopted this style of coding.

[39]:
```
def detectchange(x):
    for i__ in range(len(lwr)) : #This detects changes in resistor values
        r[i__].value = lwr[i__].result
    for i__ in range(len(lwc)) : #This detects changes in capacitor values
        c[i__].value = lwc[i__].result
    for i___ in range(len(lwl)): #This detects changes in inductor values
        ll[i___].value = lwl[i___].result
    for j___ in range(len(lwv)): #This detects changes in voltage source values
        mag = lwv[j___].result
        ph = cmath.phase(v[j___].value)
```

```
        v[j___].value = complex(mag*np.cos(ph), mag*np.sin(ph))
    for j__ in range(len(lwi)): #This detects changes in current source values
        mag = lwi[j__].result
        ph = cmath.phase(cur[j__].value)
        cur[j__].value = complex(mag*np.cos(ph), mag*np.sin(ph))
    cktsolver(n,k, r, c, ll, v, cur, 0)
```

3.0.5 Why a detect change function ?

When the detect change function is called the values of R, L, C, V and I are updated according to the need of the user. When there is a change in the value of any one of the parameters, the function detect change is called. This acts like a trigger to this function and thus updates the list and in turn updates the final values. The analogy to this being the clock pulse to a digital circuit. A digital circuit storage element like flip flop is triggered by the rising or falling edge of the clock pulse. Thus, detect change detects the change in state based on x!!

[43]:
```
def cktsolver(n, k, r, c, ll, v, cur, x):#The x taken as input here is a dummy␣
↪variable to make the working of the program dynamic.
    global w
    if len(w) == 1:
        for f in w :
            global frequency
            frequency = f
        M = np.zeros((n+k, n+k), dtype = np.complex128)   #M = zero matrix
        b = np.zeros((n+k,1), dtype = np.complex128)      #b = zero vector
        for res in r:
             getMR(M, res.nodes, res.value)
        for vol in v:
            getMV(M, b, vol.nodes, vol.value, v.index(vol), n)
        for curr in cur :
            getMI(b, curr.nodes, curr.value)
        for cap in c:
            getMC(cap.nodes, M, frequency, cap.value)
        for ind in ll:
            getML(ind.nodes, M, frequency, ind.value)
        M[0] = 0 #Set row corresponding to GND to 0
        M[0][0] = 1 #Set the matrix entry corresponding to GND to 1
        b[0] = 0
        gnd = 0 #To check whether GND Is specified or not.
        for _ in nodes.keys():
            if _ == 'GND' :
                gnd = 1
        if gnd != 1:
            sys.exit("No GND specified")
        x = GaussianElimination(M,b) #Solves the set of equations generated
        if np.all(x) == None : #Checks if solution exists or not
            sys.exit("No solution")
```

```
        nk = list(nodes.keys())
        if frequency == 0 :
            for node_idx in range(len(nodes.values())) :
                print(f"The voltage at node {nk[node_idx]} is {x[node_idx].
↪real}")
            for vol in v:
                print(f"The current through voltage source {vol.value} V between␣
↪nodes {vol.nodes} is {x[n+v.index(vol)].real}")
        else :
            for node_idx in range(len(nodes.values())) :
                print(f"The voltage at node {nk[node_idx]} is {abs(x[node_idx])}␣
↪at an angle {cmath.phase(x[node_idx])} with source.")
            for vol in v:
                print(f"The current through voltage source {vol.value} V between␣
↪nodes {vol.nodes} is {abs(x[n+v.index(vol)])} at an angle {cmath.phase(x[n+v.
↪index(vol)])} with source.")
    else :
        print("Enter a single frequency ONLY.")
```

3.0.6 Cktsolver()

This function essentially develops the matrix M and b required to solve and also solves it. A separate function is definitely needed because on change of input, we need to recall this segment of code to solve for updated values. Thus, I have created it as a separate function.

cktsolver() also checks for a single frequency as this method of solving works only for single frequency sources.

3.0.7 Solution to Problem Scenarios

- A loop consisting purely of voltage sources will cause the system to be inconsistent.
- A node connected purely to current sources will cause the system to be inconsistent.
- A circuit defined with both DC and AC sources has been taken care of.
- Syntax errors of the netlist are penalised.

3.0.8 How are problem scenarios solved in this code ?

- The first two problem scenarios are actually not a problem because a loop full of voltage sources and a node full of current sources are by definition inconsistent, they have either no solution if KCL or KVL is violated or infinite solution if KCL or KVL is satisfied.
- The third scenario of AC and DC is resolved by keeping a set w that keeps tracks of all frequencies, that includes DC which logically is AC with zero frequency.
- The fourth scenario of syntax errors have been appropriately addressed - like .netlist, .cirucit, .end and absence of GND. All these have been shown to work using netlists defined by me.

3.0.9 Small Bonus : Use Widgets

This is the use of widgets to dynamically get the outputs. I have used widgets to dynamically change the values of R, L, C, V and I and use these changes to modify the output.

3.0.10 Additional Note : Regarding convention for V and I sources

- It is assumed that V n1 n2 means Vn1 - Vn2 = V
- It is assumed that I n1 n2 means I enters n1 and leaves n2

3.0.11 Additional Assumption :

It is assumed that magnitude of ac source by default means amplitude and not Vpp (peak to peak voltage).

[41]:
```
main()
```

```
Enter the name of the netlist including extension (.netlist)ckt1.netlist
Found a resistor
Found a resistor
Found a resistor
Found a resistor
Found a resistor
Found a voltage source with value:  5.0
```

[44]:
```
cktsolver(n, k, r, c, ll, v, cur, 0)
```

```
The voltage at node GND is 0.0
The voltage at node 1 is 0.0
The voltage at node 2 is 0.0
The voltage at node 3 is 0.0
The voltage at node 4 is -5.0
The current through voltage source 5.0 V between nodes (0, 4) is -0.0005
```

3.0.12 Note on direction of current through voltage source -

The current through voltage source is the current entering into the battery through the positive terminal of the battery, consistent with passive sign convention.

3.0.13 DO NOT FORGET TO RESET !

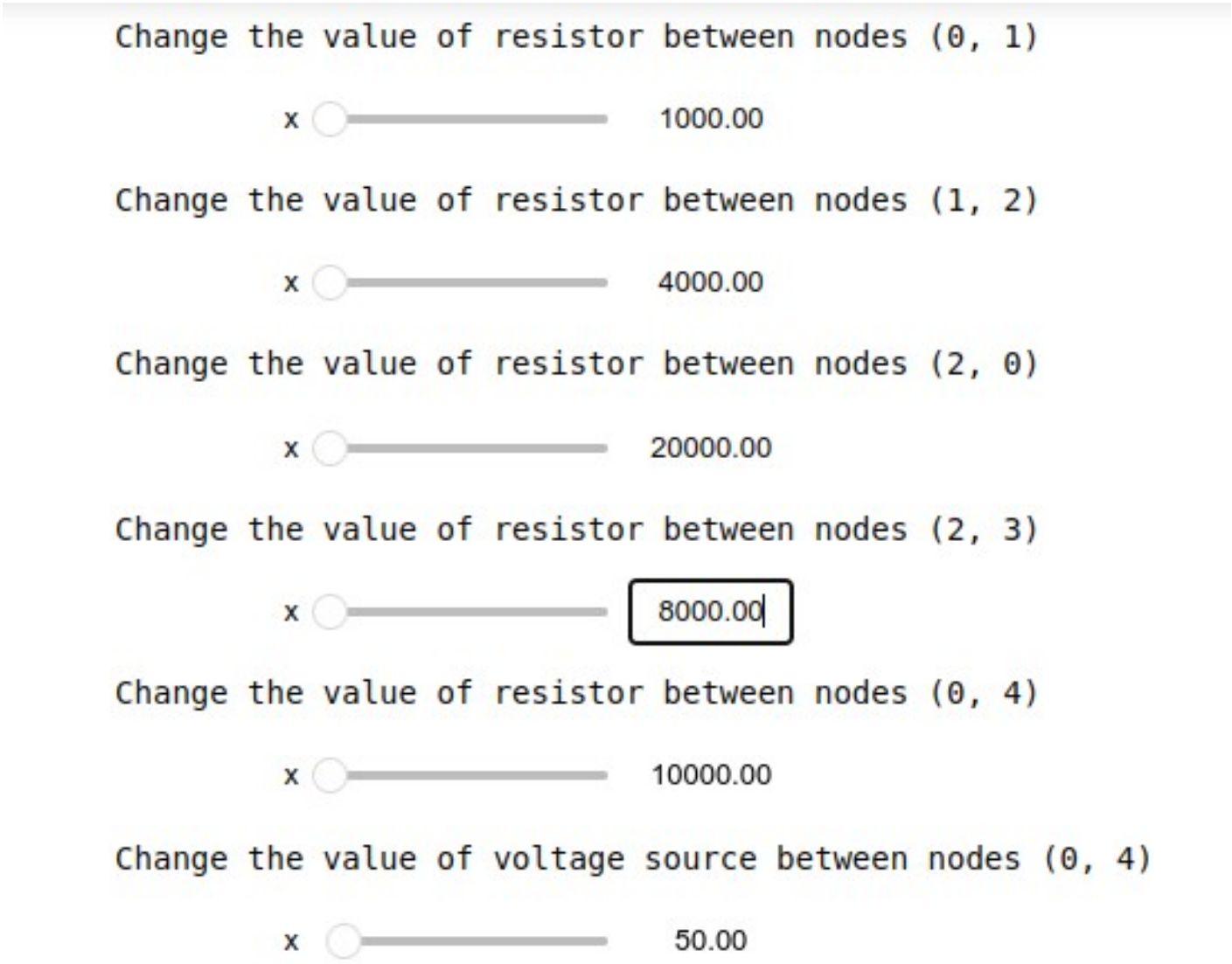
The reset () function is called below. Do not forget to run it before checking the next netlist.

3.0.14 Range of Widgets :

- R values can range from 10^{-10} Ω to 10^{8} Ω
- C values can range from 10^{-10} F to 10^{8} F
- L values can range from 10^{-10} H to 10^{8} H
- V values can range from 10^{-3} V to 10^{3} V
- A values can range from 10^{-3} A to 10^{3} A

[45]:
```
for res in r : #Used to change value of resistor dynamically
    print("Change the value of resistor between nodes "+str(res.nodes))
    lwr.append(widget(res.value))
for cap in c : #Used to change value of capacitor dynamically
    print("Change the value of capacitor between nodes "+str(cap.nodes))
```

```
    lwc.append(widget(cap.value))
for ind in ll : #Used to change value of inductor dynamically
    print("Change the value of inductor between nodes "+str(ind.nodes))
    lwl.append(widget(ind.value))
for vol in v: #Used to change value of voltage source dynamically
    print("Change the value of voltage source between nodes "+str(vol.nodes))
    lwv.append(widgetvi(abs(vol.value)))
for curr in cur: #Used to change value of current source dynamically
    print("Change the value of current source between nodes "+str(curr.nodes))
    lwi.append(widgetvi(abs(curr.value)))
wdg.interact_manual(detectchange, x=wdg.fixed(0))
```



Run Interact

```
The voltage at node GND is 0.0
The voltage at node 1 is 0.0
The voltage at node 2 is 0.0
The voltage at node 3 is 0.0
The voltage at node 4 is -49.99999999999999
The current through voltage source (49.99999999999999+0j) V between nodes (0, 4) is -0.004999999999999999
```

Out[54]: <function __main__.detectchange(x)>

3.0.15 When to call reset() ?

Call the above function only when you are done with the current netlist and want to load the next netlist.

[46]:
```
reset()
```

3.0.16 Note :

I have used additional netlists which have been uploaded in the folder containing this document. There are a total of 18 netlists used to test all possible corner cases.