Applied Programming Lab - Week 2 Assignment
Aayush Patel EE21B003

February 8, 2023

1 Assignment Overview

The following are the problems needed to be solved for this assignment.

- 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 Instructions

- I have used Python Libraries such as **math, cmath, numpy**. These libraries should be already installed in the system.
- Run the Codes in the order they are. (Do not run any cell randomly from the middle) Because the upcoming code requires functions defined earlier.

2 Factorial of a Number

2.0.1 Recursive Method

[1]:
```
def factorial_rec(N):
    if(N<0):
        print("Invalid Input")
    if(N==1):
        return 1
    else:
        return N*factorial_rec(N-1)
```

2.0.2 Iterative Method

[2]:
```
def factorial_iter(N):
    if(N<0):
        print("Invalid Input")
    fac=1
    for num in range(1,N+1):
        fac=fac*num
    return fac
```

2.0.3 Comparision

[3]:
```
x=10
print("By Recursive Method : ")
%timeit factorial_rec(x)
print("By Iterative Method : ")
%timeit factorial_iter(x)
```

```
By Recursive Method :
1.12 µs ± 49.5 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
By Iterative Method :
505 ns ± 5.06 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

In general `Recursions` are slow in implementation, because there are too many function calls and the function calls have to be stored in a stack so that it could return to the caller.

`Iterative` Methods are faster in general and the approach is intuitive, clean and easy to understand.

Here I observed that Recursive Method takes time in orders of **micro seconds**, while the Iterative one took some **nano seconds**. In Recursion, some extra space is also needed to maintain the stack and in some cases there are chances of *Stack Memory Overflow*.

Hence Iterative Method is more efficient.

3 Gauss Elimination

The task is to solve a Matrix Equation **Ax=B** where - A is a (M X N) Matrix - x is a (N X 1) Matrix - B is a (M X 1) Matrix

In Gauss Elimination Method there are basically Two Steps : - **Forward Elimination** : We apply some operations and make the Matrix A in Echelon Form (Upper Triangular Matrix with diagonal elements normalized to 1). - **Backward Substitution** : We substitute the values back one by one starting from the lowest row to the upper row.

3.1 Forward Elimination

[4]:
```
def Forward_Elimination(A,B):
    N=len(A[0])      # Number of Variables
    M=len(A)         # Number of Equations
    flag="Unique Solution"
    for row in range(0,N):
        #Check if Normalization Possible
        if(abs(A[row][row])<=2e-19):
            #Find where its non-0
            swapped=False
            for dummy_row in range(row+1,M):
                if(abs(A[dummy_row][row])>2e-19):
                    #Swap
                    A[dummy_row],A[row]=A[row],A[dummy_row]
                    B[dummy_row],B[row]=B[row],B[dummy_row]
                    swapped=True
                    break
            if(swapped==False):
                #No Unique Solution
                flag=No_Unique_Solution(A,B)
                return A,B,flag
        divisor=A[row][row]
        #Normalization
        for col in range(row,N):
            A[row][col]/=divisor
        B[row]/=divisor
        #Elimination
        for next_rows in range(row+1,M):
            multiplier=A[next_rows][row]
            for col in range(row,N):
                A[next_rows][col]-=multiplier*A[row][col]
            B[next_rows]-=multiplier*B[row]
    flag=No_Unique_Solution(A,B)
    return A,B,flag
```

This function `Forward_Elimination()` takes in arguments A and B which are matrices of the form **(M X N)** and **(M X 1)**. Firstly, it iterates through each row and *normalizes* it. Then it does some *row-operations* on the rows after it to reduce those coefficients to 0.

In the *Normalization Step*, if it find that the normalizing factor i.e the A[row][row] (*divisor* variable) is zero, then it searches other rows below it for non-zero coefficient of A[other_row][row]. If its a hit then it **swaps** that row with the intial row. Note that we have to make changes in both matrices A and B.

If while searching it finds no non-zero coefficient of A[other_row][row] then it means the given Equations do **NOT** have Unique Solution. The *flag* variable which initially stored "Uniqe Solution" is changed to either "No Solution" or "Infinite Solutions" depending on the equations. This is done using the function `No_Unique_Solution()` which has been descibed afterwards.

Note: While comparing the float values we **never check equality** with 0. Rather we check if the value is sufficiently close enough to 0. This is done because of the *Floating Point Approximation* that the Python's Compiler does in the backend. I choose the `Tolerance Level` as *2e-19*, reason begin that we have to later deal with values as small as *1e-12* in case of AC circuits. So the *tolerance level* must be choosen as a number smaller than it. In my opinion values of the order *1e-15* should also work fine.

3.2 Backward Substitution

[5]:
```
def Backward_Substitution(A, B):
    N=len(A[0])      # Number of Variables
    M=len(A)         # Number of Equations
    #Create the list x containing the values of the variables
    x=['none' for i in range(N)]
    for row in range(N-1,-1,-1):
        Sum=B[row]
        for cols in range(N-1,row,-1):
            Sum-=x[cols]*A[row][cols]
        x[row]=Sum
    return x
```

The function `Backward_Substitution()` takes arguments as A and B where A is the RREF (*Row Reduced Echelon Form*). This funtion is beging called only after ensuring that the given set of Linear Equations have a `Unique Solution`. It starts from the Nth row and find the values of the corresponding variables, till the first row. The answers are stored in a list x which is returned in the end.

3.3 Handle cases having NO Solution

[6]:
```
def No_Unique_Solution(A,B):
    N=len(A[0])      # Number of Variables
    M=len(A)         # Number of Equations
    counter=0
    for row in range(M-1,-1,-1):
        check=False
        for col in range(N):
            if(abs(A[row][col])>2e-19):
                check=True
        if(check==False):
            counter+=1
            if(abs(B[row])>2e-19):
                return "No Solution"
    if(counter<M-N):
        return "No Solution"
    elif(counter>M-N):
        return "Infinite Solution"
    else:
        return "Unique Solution"
```

The function `No_Unique_Solution()` is called in the end of Forward Elimination or when during Forward Elimination it doesn't find any row having non-zero value of the Normalizing Factor. `Tolerance Level` is taken as 2e-19. It returns the flags *"Unique Solution"*, *"No Solution"*, *"Infinite Solution"* depeding on the cases.

3.4 Implementation of Gauss Elimination

[7]:
```
import numpy as np
def Gauss_Elimination(A,B):
    if(not isinstance(A,list)):
        A=A.astype(np.float32)
        B=B.astype(np.float32)
    A1,B1,flag=Forward_Elimination(A,B)
    if(flag=="Unique Solution"):
        x=Backward_Substitution(A1,B1)
        return x
    else:
        return flag
```

The function `Gauss_Elimination` firstly calls `Forward_Elimination()` and then `Backward_Substitution()` . Note that if the input has numpy arrays, I have converted the datatype to numpy float. It is done to ensure float division during Normalization Step. If the datatype is int for the numpy arrays it will work with only integer values, because the backend of *Numpy* is in **C language**. We need not worry for lists because Python does implicit type-casting if needed.

3.5 Limitations

[8]:
```
#Case when code fails
A=[
    [7,9,10],
    [4,5,6],
    [11,14,16]
]
B=[1,9,10]
# A is singular Matrix
print(np.linalg.solve(A,B))
print(Gauss_Elimination(A,B))

A=[
    [8,9,10],
    [4,5,6],
    [12,14,16]
]
B=[1,9,10]
```

```
[-4.20000000e+01 -4.88498131e-14  2.95000000e+01]
[-1.3333333333333712, -20.3333333333333, 19.333333333333332]
```

- For some equations having Infinite Solutions, my code returns a Unique solution. Although the numpy.linalg.solve() also works here, **but it shouldn't.** The case mentioned above has determinant 0, so it should give Error. This is also a limitation for the np.linalg.solve().
- Both codes give a correct solution from the set of infinite solutions, i.e. it is not a unique solution.
- My impelentation is slower than *np.linalg.solve()* because the backend in numpy is C.C is much faster than Python. Also NumPy is fast because it can do all its calculations without calling back into Python. NumPy package integrates C, C++, and Fortran codes in Python

3.6 Advantage over np.linalg.solve()

[9]:
```
A=[
    [8,9,10],
    [4,5,6],
    [12,14,16]
]
B=[1,9,10]

print(np.linalg.solve(A,B))
print(Gauss_Elimination(A,B))
```

```
[-10.75   0.5    8.25]
Infinite Solution
```

My code gives correct answer that this has infinite solution, but np.linalg.solve() just gives a solution that is correct. This solution is **not unique**.

3.7 10 X 10 Matrix Solving

[10]:
```
A=np.random.randint(low=-100000,high=100000,size=(10,10))
B=np.random.randint(low=-100000,high=100000,size=(10))

print("From linalg.solve() :")
print(np.linalg.solve(A,B))
%timeit np.linalg.solve(A,B)

print("From my Method :")
print(Gauss_Elimination(A,B))
%timeit Gauss_Elimination(A,B)
```

```
From linalg.solve() :
[ 0.03061941  2.26369235  0.51224422 -0.77444314  3.39525104  1.30925905
  1.80300032 -0.96814396 -1.89594135 -3.71992834]
8.22 µs ± 997 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
From my Method :
[0.030611038, 2.2636974, 0.51224756, -0.7744447, 3.3952637, 1.3092566,
1.8030082, -0.9681463, -1.8959444, -3.7199311]
489 µs ± 32.2 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

It generates a random 10 X 10 Matrix and solves using Both Methods. Both give the same answers, but the np.linalg.solve() is about 70 times faster than my implementation.

4 Spice

From the given netlist, solve the circuit and give the corresponding outputs.

Note :

- While implementation we need not care about wheter the MNA has solution or not, because those set of equations correspond to a Circuit. Any Circuit would have only one Unique Solution.
- The input frequency is assumed to be in Hz.
- The input values arre assumed to be in SI Units : Volts,Ampere,Ohm,Henry,Farad.
- The Phasor Angle is printed in Degrees.
- The Phasor begin read from the file is assumed to be in Degrees.
- Python Libraries like math and cmath are used here.
- This Circuit Solver solves **only** for
  - Single Frequency Circuits
  - Resistors with DC (Inductor and Capacitor would required Transient Analysis)
  - Resistors, Inductors, Capacitors with AC (Returns the value of Voltages and Currents in Steady State.)

[11]:
```
def circuit_solver(filename):
    with open(filename) as f:
        lines=f.readlines()
        ckt_data=[]
        for line in lines:
            ckt_data.append(line.split())
    check=False
    variables=[]
    dc_flag=False
    ac_flag=False
    #Read the required part in the .netlist file.
    #i.e. the part between .circuit and .end
    for line in ckt_data:
        if(len(line)==0):
            continue
        if(line[0]=='.end'):
            check=False
            break
        if(check==True):
            #Name node i as ni for simplicity.
            if(line[1][0]!='n' and line[1]!='GND'):
               line[1]='n'+line[1]
            if(line[2][0]!='n' and line[2]!='GND'):
                line[2]='n'+line[2]
```

```
            #Add the circuit variables to the variables list.
            variables.append(line[1])
            variables.append(line[2])
            #Check for Voltage Sources
            if(line[0][0] in ['V','v']):
                variables.append("I"+line[0])
                if(line[3]=="dc"):
                    dc_flag=True
                if(line[3]=="ac"):
                    ac_flag=True
            #Check for Current Sources
            elif(line[0][0] in ['I','i']):
                if(line[3]=="dc"):
                    dc_flag=True
                if(line[3]=="ac"):
                    ac_flag=True
        if(line[0]=='.circuit'):
            check=True
    #To deal with the circuit variables, we need
    variables=list(set(variables))
    variables.sort()
    pos={}
    for i in range(len(variables)):
        pos[variables[i]]=i
    if(ac_flag==True and dc_flag==True):
        print("Multiple Frequencies : Involves both AC and DC Sources")
        return
    elif(dc_flag==True):
        dc_solver(pos,ckt_data,variables)
    else:
        ac_solver(pos,ckt_data,variables)
```

The function `circuit_solver()` takes in the input by reading the .netlist file. It then calls the function dc_solver() or ac_solver() depending on the circuit. If we have both AC and DC sources, it means we have Multiple Frequencies which can not be delt here.

The input is taken in the list *ckt_ data* and the corresponding variables are added in the dictionary *pos*. *Pos* contains the position of the circuit variables for referrencing in the MNA Matrix.

[12]:
```
def dc_solver(pos,ckt_data,variables):
    l=len(variables)
    #Generate the A and B matrix for Modified Nodal Analysis.
    A=[[0 for i in range(l)] for j in range(l)]
    b=[0 for i in range(l)]
    check=False
    #Read the part between .circuit and .end
    for line in ckt_data:
        if(line[0]=='.end'):
            check=False
            break
        if(check==True):
            #Find Resistances
            if(line[0][0] in ['R','r']):
                impedance=float(line[3])
                A[pos[line[1]]][pos[line[1]]]+=1/impedance
                A[pos[line[1]]][pos[line[2]]]-=1/impedance
                A[pos[line[2]]][pos[line[2]]]+=1/impedance
                A[pos[line[2]]][pos[line[1]]]-=1/impedance
            #Find Voltage Sources
            elif(line[0][0] in ['V','v']):
                A[pos["I"+line[0]]][pos[line[1]]]=1
                A[pos["I"+line[0]]][pos[line[2]]]=-1
                b[pos["I"+line[0]]]=float(line[4])
                A[pos[line[1]]][pos["I"+line[0]]]=1
                A[pos[line[2]]][pos["I"+line[0]]]=-1
            #Find Current Sources
            elif(line[0][0] in ['I','i']):
                current=float(line[4])
                b[pos[line[1]]]-=current
                b[pos[line[2]]]+=current
            else:
                #Error if we have Inductance or Capacitance with DC
                #because transient analysis is not possible with Gauss␣
↪Eliimination directly.
                print(".netlist input Error")
                return
        if(line[0]=='.circuit'):
            check=True

    #Removing Redundant Ground Node to get the Official Form of MNA
    A.pop(pos["GND"])
    b.pop(pos["GND"])
    for row in A:
        row.pop(pos["GND"])
    variables.pop(pos["GND"])
    for key in pos:
```

```
        if(pos[key]>pos["GND"]):
            pos[key]-=1
    del pos["GND"]
    #It has been reduced to the MNA form

    x=Gauss_Elimination(A,b)
    print_MNA(x,variables)
```

`dc_solver()` first creates the matrices A and B. Then it generates the MNA Matrices and solves it using Gauss Elimination. Note: Voltage Sources have *positive termial* towards **node1**. Current Sources have *positive terminal* towards **node2**.

[13]:
```
import math
def ac_solver(pos,ckt_data,variables):
    freq_list=[]
    for i in range(len(ckt_data)-1,0,-1):
        if(len(ckt_data[i])==0):
            continue
        if(ckt_data[i][0]=='.ac'):
            freq_list.append(float(ckt_data[i][2]))
    freq_list=list(set(freq_list))
    frequency=freq_list[0]

    #check for Multiple Frequencies.
    if(len(freq_list)!=1):
        print("Multiple frequencies")
        return

    # Solve the circuit for single frequency .
    omega=2*math.pi*frequency
    l=len(variables)
    A=[[0+0j for i in range(l)] for j in range(l)]
    b=[0+0j for i in range(l)]

    check=False
    sources=['V','v','I','i']
    im_j=complex(0+1j)
    for line in ckt_data:
        if(len(line)==0):
            continue
        if(line[0]=='.end'):
            check=False
            break
        if(check==True):
            if(line[0][0] not in sources):
                #Resistor
                impedance=complex(float(line[3]))
```

```
                #Capacitor
                if(line[0][0] in ['C','c']):
                    impedance=1/(im_j*impedance*omega)
                #Inductor
                if(line[0][0] in ['L','l']):
                    impedance=im_j*impedance*omega
                A[pos[line[1]]][pos[line[1]]]+=1/impedance
                A[pos[line[1]]][pos[line[2]]]-=1/impedance
                A[pos[line[2]]][pos[line[2]]]+=1/impedance
                A[pos[line[2]]][pos[line[1]]]-=1/impedance
            #Voltage Source
            elif(line[0][0] in ['V','v']):
                A[pos["I"+line[0]]][pos[line[1]]]=1
                A[pos["I"+line[0]]][pos[line[2]]]=-1
                angle=float(line[5])*math.pi/180
                b[pos["I"+line[0]]]=math.cos(angle)*float(line[4])+math.
↪sin(angle)*float(line[4])*im_j
                A[pos[line[1]]][pos["I"+line[0]]]=1
                A[pos[line[2]]][pos["I"+line[0]]]=-1
            #Current Source
            elif(line[0][0] in ['I','i']):
                angle=float(line[5])*math.pi/180
                current=math.cos(angle)*float(line[4])+math.
↪sin(angle)*float(line[4])*im_j
                b[pos[line[1]]]-=current
                b[pos[line[2]]]+=current
            #Some Error in the .netlist fromat.
            else:
                print(".netlist input Error")
                return
        if(line[0]=='.circuit'):
            check=True
    #Removing Ground Node to get the Official Form of MNA
    A.pop(pos["GND"])
    b.pop(pos["GND"])
    for row in A:
        row.pop(pos["GND"])
    variables.pop(pos["GND"])
    for key in pos:
        if(pos[key]>pos["GND"]):
            pos[key]-=1
    del pos["GND"]
    #It has been reduced to the MNA form

    x=Gauss_Elimination(A,b)
    print_MNA(x,variables)
```

`ac_solver()` first creates the matrices A and B and also checks if it has Multiple Frequencies. Then it generates the MNA Matrices and solves it using Gauss Elimination. Note: Voltage Sources have *positive termial* towards **node1**. Current Sources have *positive terminal* towards **node2**. The impedance of Capacitor is 1/jwC . The impedance of Inductor is jwL .

The equations are solved in the complex numbers domain.

[14]:
```
import cmath
def print_MNA(x,variables):
    #Convert Radians to Degree
    def radians_to_degree(angle):
        return angle*180/math.pi
    #Use Scientific Notation for printing the values of thec Circuit Variables.
    def conv(value):
        if(isinstance(value,complex)):
            magnitude=abs(value)
            angle=radians_to_degree(cmath.phase(value))
            return "{:e}".format(magnitude)+' <'+str(angle)+' degree>'
        else:
            return "{:e}".format(value)
    #Finally Print the Circuit Variables
    for i in range(len(variables)):
        if(variables[i][0]=='I'):
            print(variables[i]," = ",conv(x[i])," A",sep="")
        else:
            print(variables[i]," = ",conv(x[i])," V",sep="")
```

The function `print_MNA` changes the Rectangular from of Complex Numbers to the Polar Form for AC Circuits. For DC Circuits it just prints the circuit variables as it is. All the values are printed in the Scientific Notation.

[15]:
```
circuits=[
    "ckt1.netlist",
    "ckt2.netlist",
    "ckt3.netlist",
    "ckt4.netlist",
    "ckt5.netlist",
    "ckt6.netlist",
    "ckt7.netlist"]

for filename in circuits:
    print("Circuit Variables for",filename, ":")
    circuit_solver(filename)
    print()
```

```
Circuit Variables for ckt1.netlist :
IV1 = -5.000000e-04 A
n1 = 0.000000e+00 V
n2 = 0.000000e+00 V
n3 = 0.000000e+00 V
n4 = -5.000000e+00 V

Circuit Variables for ckt2.netlist :
Multiple Frequencies : Involves both AC and DC Sources

Circuit Variables for ckt3.netlist :
IV1 = -4.970760e-03 A
n1 = -1.000000e+01 V
n2 = -5.029240e+00 V
n3 = -2.573099e+00 V
n4 = -1.403509e+00 V
n5 = -9.356725e-01 V

Circuit Variables for ckt4.netlist :
IV1 = -2.222222e+00 A
n1 = -1.000000e+01 V
n2 = -5.555556e+00 V
n3 = -3.703704e+00 V

Circuit Variables for ckt5.netlist :
IV1 = -1.000000e+00 A
n1 = -1.000000e+01 V

Circuit Variables for ckt6.netlist :
IV1 = 5.000000e-03 <179.99964911890655 degree> A
n1 = 3.141593e-05 <-90.00035088109347 degree> V
n2 = 3.062015e-05 <-90.00035088109347 degree> V
n3 = 5.000000e+00 <-180.0 degree> V
```

```
Circuit Variables for ckt7.netlist :
n1 = 8.164558e-04 <-89.9999906441059 degree> V
```

5 Conclusion

I have verified these answers by simulating the circuits given here in **LTSpice** All Answers are *correctly matching* with the values obtained from LTSpice when the circuit reaches Steady State.