Submission1

Siddarth Baruah <ee21b128@smail.iitm.ac.in>

February 8, 2023

0.1 Week2 Assignment

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. **********

[2]:
```
def problem1(N):
    res=1
    for i in range(1,N+1):
        res *= i
    return res
```

1.0.1 Explaination

Starting with the problem 1, I used simple for loop to calculate the factorial. Here I didn't use recurssion since it uses Auxiliary space of N.

[3]:
```
n= 15
print(problem1(n))
```

```
1307674368000
```

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. **********

2.0.1 Solution

[4]:
```
class matrix_solver():
    def __init__(self, Ain, bin):
        self.A= Ain
        self.b= bin
```

```
        self.rowreduce=0
        self.sol=[]
        self.A, self.b= self.rearrange(self.A, self.b)


    def rearrange(self, matrix, b):
        len_row= len(matrix[0])
        len_column= len(matrix)
        if len_column>len_row:
            for row in range(len_row, len_column):
                for column in range(len_row):
                    matrix[0][column]+= matrix[row][column]
                b[0] += b[row]


        for diagonal in range(len_row):
            if matrix[diagonal][diagonal]== 0:
                for row in range(len_column):
                    if matrix[row][diagonal]!=0:
                        for column in range(len_row):
                            matrix[diagonal][column] += matrix[row][column]
                        b[diagonal] += b[row]

        return matrix, b

    def Row_reduce(self):

        lenrow= len(self.A[0])
        lencolumn= len(self.A)


        if lencolumn>=lenrow:
            for iter in range(lenrow):
                #normalizing
                for row_num in range(iter,lencolumn):

                    norm = self.A[row_num][iter]

                    if norm!=0:
                        for column_num in range(iter, lenrow):
                            self.A[row_num][column_num] /= norm
                        self.b[row_num] /= norm

                #row echelon
                for row_num in range(iter+1, lencolumn):

                    if self.A[row_num][iter]:
```

```
                    for column in range(iter, lenrow):
                        self.A[row_num][column] -= self.A[iter][column]
                    self.b[row_num] -= self.b[iter]
            self.rowreduce=1
        #rounding off
        for row in range(lencolumn):
            for column in range(lenrow):
                value_=self.A[row][column]
                self.A[row][column]= round(value_.real,20) + round(value_.
↪imag,20)*1j
        for value in range(lencolumn):
            value_= self.b[value]
            self.b[value]= round(value_.real,10)+ round(value_.imag,10)*1j
        return [self.A,self.b]


    def transpose(self, matrix):
        len_= len(matrix)
        newA=[]
        for row in reversed(matrix):
            new_row=[]
            for num in reversed(row):
                new_row.append(num)
            newA.append(new_row)
        return newA


    def reversed_(self, matrix):
        c=[]
        for i in reversed(range(len(matrix))):
            c.append(matrix[i])
        return c

    def check_solvability(self, matrix, Bin):
        len_row= len(matrix[0])
        len_column= len(matrix)
        prod=1
        if len_column> len_row:
            for row in range(len_row):
                prod *= matrix[row][row]
                if prod ==0:
                    print("No solution")
                    return 0
                for column in range(len_row):
                    if matrix[row][column] != 0:
                        break
                    if  matrix[row][column] != 0 and column== len_row:
```

```
                    if Bin[row]==0:
                        print("Infinite solution")
                        return 0
                    else:
                        print("No solution")
                        return 0
            if Bin[len_row]!= 0:
                print("It is unsolvable!")
                return 0
            return 1
        elif len_column== len_row:
            prod=1
            for diagonal in range(len_row):
                prod *= matrix[diagonal][diagonal]
            if prod==0 and Bin[len_row-1]!=0:
                print("No solution")
                return 0
            elif prod==0 and Bin[len_row-1]==0:
                print("Infinite solution")
                return 0
            else:
                return 1
        else:
            print("Infinite solution")
            return 0

    def return_required_matrix(self, matrix, Bin):
        len_row= len(matrix[0])
        mat= []
        res= []
        for row in range(len_row):
            mat.append(matrix[row])
            res.append(Bin[row])
        return mat, res

    def solve(self):
        self.A, self.b= self.rearrange(self.A, self.b)
        self.A, self.b= self.Row_reduce()

        if self.check_solvability(self.A, self.b):
            self.sol=[]
            self.A, self.b= self.return_required_matrix(self.A, self.b)
            self.A= self.transpose(self.A)
            self.b= self.reversed_(self.b)
            self.sol.append(self.b[0])
            len_= len(self.A[0])
            for row in range(1,len_):
```

```
                sum=0
                for column in range(0, row):
                    sum += (self.A[row][column])*(self.sol[column])
                self.sol.append(self.b[row]- sum)

            return self.reversed_(self.sol)
```

This is a matrix solver which uses Gaussian method to solve a given set of linear equations.

2.1 Work of each function is as follows:

2.1.1 Rearrange:

This will rearrange the rows of the matrix suxh that it can easily row reduce it. So what it does is that it first check each diagonal and if it finds a diogonal with zero element, it adds another row to it which has a non zero element in the same column.

2.1.2 Row_reduced:

This will row reduce the matrix by first, normalizing the rows and simply do row operations to make a row-echelon matrix.

2.1.3 Transpose:

It transposes a square matrix.

2.1.4 Reveresed:

It flips the elements of a list.

2.1.5 Check_solvability:

It checks if the given matrix is solvable or not. This can only be used after Row_reduced function is used.

2.1.6 return_required_matrix:

After row_reduced and Check_solvability, if all goes right, it removes the extra equations from the linear algebra list.

2.1.7 solve:

so this is the master snippet which does all the work, starting from Reaarange to transpose to row_reduce etc. Finally it finds the solution by starting at the bottom of row echelon matrix and slowy moving up to find all the solutions.

2.2 Now Importing Numpy

[5]:
```
import numpy as np
```

[6]:
```
A= [[7, 6, 6, 6, 7, 5, 5, 4, 2, 2],
 [8, 6, 5, 6, 1, 1, 5, 4, 9, 2],
 [5, 3, 9, 8, 6, 4, 8, 4, 1, 3],
 [3, 5, 8, 8, 6, 5, 6, 4, 8, 1],
 [6, 7, 2, 1, 1, 1, 9, 1, 9, 3],
 [1, 8, 5, 8, 5, 1, 1, 5, 3, 0],
 [9, 9, 0, 3, 9, 3, 4, 2, 7, 5],
 [4, 1, 6, 8, 4, 8, 9, 2, 2, 9],
 [7, 6, 0, 0, 1, 3, 1, 2, 1, 9],
 [7, 0, 7, 7, 5, 5, 4, 1, 1, 0]]
B= [5850, 6088, 6339, 6624, 6847, 7016, 7275, 7586, 7750, 7906]
```

The above is a matrix created for having a solution.

[7]:
```
matrix_solver(A,B).solve()
```

[7]:
```
[(115.79830338963882+0j),
 (1432.499054032567+0j),
 (1467.8080783312425+0j),
 (598.5097274066972+0j),
 (-262.53347173409816+0j),
 (-162.4404953200319+0j),
 (-569.8157050538141+0j),
 (-2873.972531496633+0j),
 (-90.70757667758153+0j),
 (611.4156504996+0j)]
```

[8]:
```
answer= np.linalg.solve(A,B)
answer
```

[8]:
```
array([  115.79830339+0.j,  1432.49905403+0.j,  1467.80807833+0.j,
         598.50972741+0.j,  -262.53347173+0.j,  -162.44049532+0.j,
        -569.81570505+0.j, -2873.9725315 +0.j,   -90.70757668+0.j,
         611.4156505 +0.j])
```

[9]:
```
%timeit np.linalg.solve(A,B)
```

```
15.1 µs ± 503 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

[10]:
```
%timeit matrix_solver(A,B).solve()
```

```
145 µs ± 2.85 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

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 Solution

First declaring some constant that will require afterwards

[11]:
```
#constants

exp= 2.71828182845904
pi= 3.14159265359
```

3.0.2 Function- give_nodes(circuit)

I am starting the solution of this problem by first finding out the nodes of the circuit. For that the following function is used to return a node array.

[12]:
```
def give_nodes(circuit):
    nodes= []
    start=0
    num_vol=0
    for line in circuit:
        line= line.split()
        try:
            if line[0]== ".circuit":
                start=1
                continue
            elif line[0]== '.end':
                start=0
                break
        except:
            pass

        if start:
            try:
                check1= nodes.index(line[1])
            except:
                nodes.append(line[1])
            try:
                check1= nodes.index(line[2])
            except:
                nodes.append(line[2])
    for line in circuit:
        line=line.split()
        try:
            if line[0][0]== 'V':
                nodes.append("I"+line[0])
        except:
```

```
            pass
        try:
            if line[0][0] == "I":
                nodes.append(line[0])
        except:
            pass

    return nodes
```

Lets take an example circuit.

[13]:
```
with open("ckt7.txt") as f:
    circuit = f.read().splitlines()
circuitnew=[]
for line in circuit:
    if len(line)==0:
        continue
    else:
        circuitnew.append(line)
circuit= circuitnew

circuitnew=[]
start=0
check=0
end_reach=0
for line in circuit:
    line= line.split()
    if line[0]==".circuit":
        start=1
    if line[0]==".end":
        start=0
        check=1
        end_reach=1
        circuitnew.append(line)
        continue

    if check==1 and end_reach==1 and line[0] != ".ac":
        check=0
        end_reach=0

    if start:
        comment=0
        linenew=[]
        for word in line:
            if word == "#":
                break
            linenew.append(word)
```

```
    if (start==1 or check==1 or end_reach==1):
        linenew=[]
        for word in line:
            if word == "#":
                break
            linenew.append(word)
        circuitnew.append(linenew)

circuit= circuitnew

circuitnew=[]
for line in circuit:
    newline=""
    for word in line:
        newline += word + " "
    circuitnew.append(newline)

circuit= circuitnew
[print(line) for line in circuit]
```

```
.circuit
I1 GND n1 ac 5 0
C1 GND n1 1
R1 GND n1 1000
L1 GND n1 1e-6
.end
.ac I1 1000
```

[13]:
```
[None, None, None, None, None, None, None]
```

We can see that we are importing the circuit correctly.

Now, using the give_node function we get the following output.

[14]:
```
nodes= give_nodes(circuit)
nodes
```

[14]:
```
['GND', 'n1', 'I1']
```

The above array represents all the nodes in a circuit.

3.0.3 Function- check_dc_frequency(circuit)

Then for checking the circuit if it is dc or ac I have used the following funtion. This function will intake a circuit and check. It will go through the circuit and throw error if it has multiple frequencies, or dc and ac in the same circuit.

[15]:
```
def check_dc_ac_frequency(circuit):
    dc_flag=0
    ac_flag=0
```

```
    wc_flag=0
    wc=0
    wcavailable=[]
    for line in circuit:
        line=line.split()

        if line[0][0]== "V" or line[0][0]== "I":
            if line[3]=="dc":
                dc_flag=1
            if line[3]=="ac":
                ac_flag=1
        if line[0]== ".ac":
            wc= float(line[2])
            if float(line[2]) in wcavailable:
                continue
            else:
                wcavailable.append(float(line[2]))
                wc_flag+=1
    if (dc_flag and ac_flag) or (wc_flag>1):
        return 0 ,0
    else:
        if dc_flag:
            return "dc" ,0
        elif ac_flag:
            return "ac" , wc
```

Now, checking if the given circuit is valid.

[16]:
```
check_dc_ac_frequency(circuit)
```

[16]:
```
('ac', 1000.0)
```

The return means that the circuit is dc and has 0 frequency.

3.0.4 Function: make_equation_dc

This function will return coefficient_matrix, const_vector for a given dc circuit.

[30]:
```
def make_equations_dc(circuit, nodes):


    coefficient_matrix= []
    const_vector= []

    #to relate the GND


    tempequation=[0]*len(nodes)
    tempequation[nodes.index("GND")] += 1
```

```
    coefficient_matrix.append(tempequation)
    const_vector.append(0)

    #for the KCL
    for node in nodes:
        if node=="GND":
            continue
        tempequation= [0]*len(nodes)
        tempsol=[0]
        tempequation_ac= [0]*len(nodes)
        tempsol_ac=[0]
        canappend=0
        for line in circuit:
            line= line.split()
            if node in line[1:3]:
                if line[0][0] == "R":
                    Register= float(line[3])
                    if line.index(node)==1:
                        tempequation[nodes.index(line[1])] -= 1/Register
                        tempequation[nodes.index(line[2])] += 1/Register
                    else:
                        tempequation[nodes.index(line[1])] += 1/Register
                        tempequation[nodes.index(line[2])] -= 1/Register
                    canappend=1
                if line[0][0]== "V" and line[3]=="dc":
                    if line.index(node)==1:
                        tempequation[nodes.index("I"+line[0])] -= 1
                    else:
                        tempequation[nodes.index("I"+line[0])] += 1
                    canappend=1
                if line[0][0]== "I" and line[3]=="dc":
                    if line.index(node)==1:
                        tempequation[nodes.index(line[0])] += 1
                    else:
                        tempequation[nodes.index(line[0])] -= 1
        if canappend:

            coefficient_matrix.append(tempequation)
            const_vector.append(tempsol[0])

    #for the voltages
    for line in circuit:
        tempequation= [0]*len(nodes)
        tempsol=[0]
        line= line.split()
        if line[0][0]=="V" and line[3]=="dc":
            try:
```

```
                tempequation[nodes.index(line[2])] +=-1
            except:
                pass
            try:
                tempequation[nodes.index(line[1])] +=+1
            except:
                pass
            tempsol[0] += float(line[4])
            coefficient_matrix.append(tempequation)
            const_vector.append(tempsol[0])
    #for the current sources
    for line in circuit:
        tempequation= [0]*len(nodes)
        tempsol=[0]
        line= line.split()
        if line[0][0]== "I" and line[3]=="dc":
            tempequation[nodes.index(line[0])] += 1
            tempsol[0] += float(line[4])
            coefficient_matrix.append(tempequation)
            const_vector.append(tempsol[0])

    return coefficient_matrix, const_vector
```

3.0.5 Function- make_equation_ac

This function will return coefficient_matrix, const_vector for the ac circuit. But all of them will be in the frequency domain.

[31]:
```
def make_equations_ac(circuit, nodes, frequency):

    frequency= 2*pi*frequency
    coefficient_matrix= []
    const_vector= []

    #to relate the GND

    tempequation=[0]*len(nodes)
    tempequation[nodes.index("GND")] += 1
    coefficient_matrix.append(tempequation)
    const_vector.append(0)


    #for the KCL
    for node in nodes:
        if node=="GND":
            continue
        pos= 1
```

```
        tempequation= [0]*len(nodes)
        tempsol=[0]

        canappend=0

        for line in circuit:
            line= line.split()

            if node in line[1:3]:
                if line[0][0] == "R":
                    Register= float(line[3])

                    if line.index(node)==pos:
                        tempequation[nodes.index(line[1])] += 1/Register
                        tempequation[nodes.index(line[2])] -= 1/Register
                    else:
                        tempequation[nodes.index(line[1])] -= 1/Register
                        tempequation[nodes.index(line[2])] += 1/Register
                    canappend=1
                if line[0][0] == "L":
                    Impedence= float(line[3])*frequency*1j
                    if line.index(node)==pos:
                        tempequation[nodes.index(line[1])] += 1/Impedence
                        tempequation[nodes.index(line[2])] -= 1/Impedence
                    else:
                        tempequation[nodes.index(line[1])] -= 1/Impedence
                        tempequation[nodes.index(line[2])] += 1/Impedence

                if line[0][0] == "C":
                    Impedence= 1/(float(line[3])*frequency*1j)
                    if line.index(node)==pos:
                        tempequation[nodes.index(line[1])] += 1/Impedence
                        tempequation[nodes.index(line[2])] -= 1/Impedence
                    else:
                        tempequation[nodes.index(line[1])] -= 1/Impedence
                        tempequation[nodes.index(line[2])] += 1/Impedence
                    canappend=1

                if line[0][0]== "V":
                    if line.index(node)==1:
                        tempequation[nodes.index("I"+line[0])] += 1
                    else:
                        tempequation[nodes.index("I"+line[0])] -= 1
                    canappend=1
                if line[0][0]== "I":
                    if line.index(node)==1:
                        tempequation[nodes.index(line[0])] -= 1
```

```
                else:
                    tempequation[nodes.index(line[0])] += 1

                canappend=1

    if canappend:
        coefficient_matrix.append(tempequation)
        const_vector.append(tempsol[0])

#for the voltages
for line in circuit:
    tempequation= [0]*len(nodes)
    tempsol=[0]
    line= line.split()
    if line[0][0]=="V":

        #RHS of the equation
        try:
            tempequation[nodes.index(line[2])] += -1
        except:
            pass
        try:
            tempequation[nodes.index(line[1])] += +1
        except:
            pass

        #LHS of the circuit
        try:
            tempsol[0] += (float(line[4]))*(exp**(float(line[5])*1j))
        except:
            tempsol[0] += float(line[4])
        coefficient_matrix.append(tempequation)
        const_vector.append(tempsol[0])

#for the current sources
for line in circuit:
    tempequation= [0]*len(nodes)
    tempsol=[0]
    line= line.split()
    if line[0][0]== "I":
        tempequation[nodes.index(line[0])] += 1
        try:
            tempsol[0] += (float(line[4]))*(exp**(float(line[5])*1j))
        except:
            tempsol[0] += float(line[4])
```

```
            coefficient_matrix.append(tempequation)
            const_vector.append(tempsol[0])

    return coefficient_matrix, const_vector
```

3.0.6 Function- make_equations(circuit, nodes)

Now, I am using the functions (check_dc_ac, make dc equation and make ac equation) and returning a linear equations in matrix form.

[32]:
```
def make_equations(circuit,nodes):
    type_of_circuit, frequency= check_dc_ac_frequency(circuit)
    if type_of_circuit=="dc":
        coefficient_matrix, const_vector= make_equations_dc(circuit, nodes)
    elif type_of_circuit=="ac":
        coefficient_matrix, const_vector= make_equations_ac(circuit, nodes,␣
 ↪frequency)
    else:
        print("Invalid Circuit!")
        return 0,0
    return coefficient_matrix, const_vector
```

Now using the example circuit to find its comatric and const vectors.

[33]:
```
coefficient_matrix, const_vector= make_equations(circuit, nodes)
```

3.0.7 Solution

I am calling my matrix solver that was used in question 2 and directly solving it. After that, I am Rounding off the values upto 7 places.

[34]:
```
solution= matrix_solver(coefficient_matrix,const_vector).solve()


for i in range(len(solution)):
    print(f"{nodes[i]} = {solution[i]}")
```

```
GND = 0j
n1 = (-1.333200088e-10+0.0008164557820157671j)
I1 = (5+0j)
```

Here we got the voltages of each node. The term j for zero stays just to indicate its an complex number.

3.0.8 Function- Circuit_analysis(circuit_path)

This is the master function of all. This will just intake the path of the circuit file and then do what ever operation is required to find the solution of the circuit.

[35]:
```
def Circuit_analysis(circuit_path):
    with open(circuit_path) as f:
        circuit = f.read().splitlines()
    circuitnew=[]
    for line in circuit:
        if len(line)==0:
            continue
        else:
            circuitnew.append(line)
    circuit= circuitnew

    nodes= give_nodes(circuit)
    coefficient_matrix, const_vector= make_equations(circuit, nodes)
    try:
        solution= matrix_solver(coefficient_matrix,const_vector).solve()

        for i in range(len(solution)):
            print(f"{nodes[i]} = {solution[i]}")
    except:
        pass
```

3.0.9 Example

Here is an example of the above function

[36]:
```
Circuit_analysis("ckt1.txt")
```

```
GND = 0j
1 = 0j
2 = 0j
3 = 0j
4 = (-5+0j)
IV1 = (-0.0005+0j)
```

[37]:
```
Circuit_analysis("ckt2.txt")
```

```
Invalid Circuit!
```

[38]:
```
Circuit_analysis("ckt3.txt")
```

```
GND = 0j
1 = (-9.999999931764705+0j)
2 = (-5.029239731764705+0j)
3 = (-2.5730993976470584+0j)
4 = (-1.403508762352941+0j)
5 = (-0.935672508235294+0j)
IV1 = (-0.0049707602+0j)
```

[39]:
```
Circuit_analysis("ckt4.txt")
```

```
GND = 0j
1 = (-9.999999999900002+0j)
2 = (-5.555555555500002+0j)
3 = (-3.7037037036666676+0j)
IV1 = (-2.2222222222+0j)
```

[40]:
```
Circuit_analysis("ckt5.txt")
```

```
GND = 0j
1 = (-10+0j)
IV1 = (-1+0j)
```

[41]:
```
Circuit_analysis("ckt6.txt")
```

```
GND = 0j
n3 = (-5.000000000187395-2.015182044054053e-08j)
n1 = (-1.9226547039970782e-10-3.141592653589997e-05j)
n2 = (-1.8739532914109613e-10-3.0620151820440546e-05j)
IV1 = (-0.005+3.06e-08j)
```

[42]:
```
Circuit_analysis("ckt7.txt")
```

```
GND = 0j
n1 = (-1.333200088e-10+0.0008164557820157671j)
I1 = (5+0j)
```