EEE21B069_KESHAW_WEEK_2

February 8, 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).

Q1 - 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.

[12]:
```
def factorial(n):
    result = 1
    for i in range(1, n+1):
        result *= i
    return result

n = int(input("Enter a number: "))
print("Factorial:", factorial(n))
%timeit factorial(n)
```

```
Factorial: 120
532 ns ± 90.6 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

The code defines the function "factorial," which accepts a positive integer "n" as input and returns its factorial. The function computes the factorial by iterating through the range "1" to "n+1" and multiplying the "result" by each number in the range using a "for" loop.

The "%timeit" line will run the factorial function several times and give the average execution time. The average execution time will be determined by various factors, including the machine's processing speed, the size of the input "n", and the efficiency of the implementation. When "n" is increased, the execution time might grow exceedingly long.

2 Q2

- 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.

[106]:
```
import numpy as np

def linear_equation_solver(A, b):
    try:
        x = np.linalg.solve(A, b)
    except np.linalg.LinAlgError as e:
        if 'Singular matrix' in str(e):
            return "Error: Singular matrix - system has no unique solution"
        else:
            return "Error: " + str(e)
    return x

A = np.random.rand(10,10)
b = np.random.rand(10)
x = linear_equation_solver(A, b)
print("Solution:", x)
```

```
Solution: [ 1.24666204 -0.43161225 -0.7130019   0.68684441 -0.17305501
-0.31376699
 -0.20565662  0.94017764 -0.66422659  0.6873433 ]
```

Imports the "numpy" library as "np" and writes a function "linear equation solver" that accepts two inputs, a matrix "A" and a vector "b". Using the "np.linalg.solve" function, the function attempts to discover the solution x to the linear equation Ax = b. If an error occurs during the computation (for example, if A is a singular matrix), a custom error message containing details about the mistake is returned. Finally, a random matrix "A" and vector "b" are produced, and the "linear equation solver" function is used with these parameters to determine the answer "x," which is then written to the console.

[14]:
```
%timeit linear_equation_solver(A,b)
```

```
9.81 µs ± 186 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

[107]:
```
def linear_equation_solver1(A, b):
    m = len(A)
    n = len(A[0])
    if m != n:
        return "Error: Matrix A must be square"
    if m != len(b):
        return "Error: Number of rows in A and b must match"

    Ab = []
    for i in range(m):
        row = []
        for j in range(n):
            row.append(A[i][j])
        row.append(b[i])
        Ab.append(row)
```

```
    for i in range(n):
        pivot = i
        for j in range(i+1, n):
            if abs(Ab[j][i]) > abs(Ab[pivot][i]):
                pivot = j
        if pivot != i:
            Ab[i], Ab[pivot] = Ab[pivot], Ab[i]
        if Ab[i][i] == 0:
            return "Error: Singular matrix - system has no unique solution"
        for j in range(i+1, n):
            factor = Ab[j][i] / Ab[i][i]
            for k in range(i, n+1):
                Ab[j][k] = Ab[j][k] - factor * Ab[i][k]

    x = [0] * n
    for i in range(n-1, -1, -1):
        for j in range(i+1, n):
            Ab[i][n] -= x[j] * Ab[i][j]
        x[i] = Ab[i][n] / Ab[i][i]
    return x


N=int(input("Enter a number of Row/Column"))
A = np.random.rand(N,N)
b = np.random.rand(N)
x = linear_equation_solver1(A, b)
print("Solution:", x)
```

```
Solution: [-0.6459007941360819, 0.8877083694942148, 0.5470186300652807]
```

[108]:
```
%timeit linear_equation_solver1(A,b)
```

```
12.8 µs ± 222 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

This code creates the function “linear equation solver1,” which solves a system of linear equations represented by a square matrix “A” and a vector “b” of length equal to the number of rows in “A”. The function first determines if the matrix “A” is square and whether the number of rows in “A” and “b” match, and then produces an error message if either condition is not fulfilled. The function then generates an augmented matrix “Ab” by attaching the vector “b” to the right of “A”. The matrix “Ab” is then reduced to row echelon form using Gaussian elimination. If throughout the elimination process, any pivot is determined to be zero, the method generates an error message indicating that the system has no unique solution. Finally, the function use back substitution to solve the system of linear equations and provides the result as a vector. The answer is validated by constructing a “NxN” matrix “A” and a vector “b” of length “N” with the “numpy” library and feeding them into the function as inputs. After then, the solution is printed.

3 Q3

- 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.

[118]:
```
import numpy as np
import cmath as cm


pi=3.14159
def circuit_solver(file_name):
    with open(file_name, "r") as file:
        Lines = file.read()

    circ = Lines.splitlines()

    circ.remove('.circuit')
    circ.remove('.end')

    length = len(circ)

    resistor = []
    inductor = []
    capacitor=[]
    voltage = []
    current = []
    nodes = []
    frequencies = []


    for line in circ:
        l = line.split()
        for i in range(1,3):
            if(l[i] == 'GND'):
                l[i] = '0'
        if(l[0][0] == 'R'):
            resistor.append([l[1], l[2], l[3]])
            nodes.extend([l[1], l[2]])

        elif(l[0][0] == 'V'):
            if(l[3] == 'dc'):
                voltage.append([l[1], l[2], l[4], 0])
                nodes.extend([l[1], l[2]])
            elif(l[3] == 'ac'):
                voltage.append([l[1], l[2], l[4], l[5]])
                nodes.extend([l[1], l[2]])
```

```
        elif(l[0][0] == 'I'):
            if(l[3] == 'dc'):
                current.append([l[1], l[2], l[4], 0])
                nodes.extend([l[1], l[2]])
            elif(l[3] == 'ac'):
                current.append([l[1], l[2], l[4], l[5]])
                nodes.extend([l[1], l[2]])

        elif(l[0][0] == 'L'):
            inductor.append([l[1], l[2], l[3]])
            nodes.extend([l[1], l[2]])

        elif(l[0][0] == 'C'):
            capacitor.append([l[1], l[2], l[3]])
            nodes.extend([l[1], l[2]])

        elif(l[0] == '.ac'):
            frequencies.append(l[2])
            length = length-1


    frequency = list(set(frequencies))

    #Check for Multiple frequencies
    if(len(frequency) >1):
        print("Error: Multiple frequencies detected in the circuit, only a␣
↪single frequency is allowed")

    else:
        nodes = list(set(nodes))
        freq = np.array(frequency, dtype = float)
        angu_freq=2*pi*freq

        nodes.sort()

        length1 = len(nodes)
        l_curr = len(current)
        l_vol = len(voltage)

        res_array = np.array(resistor, dtype = float)
        ind_array = np.array(inductor, dtype = float)
        cap_array = np.array(capacitor, dtype = float)

        A = np.zeros((length1+l_curr+l_vol-1 ,length1+l_curr+l_vol-1), dtype =␣
↪complex)
        b = np.zeros(length1+l_curr+l_vol-1, dtype = complex)
```

```
#Now we add a resistive components in the matrix A
for i in range(1, length1):
    for r in res_array:
        if(r[0] == i):
            A[i-1][i-1] += float(1/r[2])
            if(r[1] != 0):
                A[i-1][int(r[1])-1] -= float(1/r[2])

        if(r[1] == i):
            A[i-1][i-1] += float(1/r[2])
            if(r[0] != 0):
                A[i-1][int(r[0])-1] -= float(1/r[2])

#Add a inductive components in matrix A
for j in range(1, length1):
    for l in ind_array:
        Xl = float((angu_freq)*(l[2]))
        if(l[0] == j):
            A[j-1][j-1] += complex(0, Xl)
            if(l[1] != 0):
                A[j-1][int(l[1])-1] -= complex(0, Xl)

        if(l[1] == j):
            A[j-1][j-1] += complex(0, Xl)
            if(l[0] != 0):
                A[j-1][int(l[0])-1] -= complex(0, Xl)

#Adding a inductive components in matrix A
for k in range(1, length1):
    for c in cap_array:
        Xc = -float(1/((angu_freq)*c[2]))
        if(c[0] == k):
            A[k-1][k-1] += complex(0, Xc)
            if(c[1] != 0):
                A[k-1][int(c[1])-1] -= complex(0, Xc)

        if(c[1] == k):
            A[k-1][k-1] += complex(0, Xc)
            if(c[0] != 0):
                A[k-1][int(c[0])-1] -= complex(0, Xc)


diff = length1 - l_vol - l_curr

vol_array = np.array(voltage, dtype = float)
```

```
        for v in vol_array:
            if(v[0] != 0):
                A[diff][int(v[0])-1] += 1
            if(v[1] != 0):
                A[diff][int(v[1])-1] += -1
            b[diff] = cm.rect(v[2], v[3])

            if(int(v[0]) != 0):
                A[int(v[0])-1][length1-1] += 1
            if(v[1] != 0):
                A[int(v[1])-1][length1-1] -= 1

            diff = diff+1

        diff1 = length1 - l_curr
        curr_array = np.array(current, dtype = float)
        for i in curr_array:
            A[diff1][length1-1] += 1
            b[diff1] = cm.rect(i[2], i[3])

            if(int(i[0]) != 0):
                A[int(i[0])-1][length1-1] += 1
            if(i[1] != 0):
                A[int(i[1])-1][length1-1] -= 1

            diff1 = diff1 + 1


        voltage = np.linalg.solve(A, b)
        voltage_phasor = []

        for val in range(length1):
            voltage_phasor.append([cm.polar(x[val])])

        x_matrix = []
        for i in range(1, length):
            x_matrix.append(f"n{i}")

        for i in range(1, l_vol+1):
            x_matrix.append(f"Iv{i}")

        for i in range(1, l_curr+1):
            x_matrix.append(f"Ii{i}")
        print(x_matrix, " = ",  end = '')

        print(voltage)
        print(voltage_phasor)
```

The code reads in an electrical circuit file and performs an analysis of it. The code first opens the file and reads in the lines into the "Lines" variable. It then splits the lines into a list "circ" by line. The code removes the ".circuit" and ".end" strings from the list.

Next, the code initializes various lists to store different components in the circuit, such as resistors, inductors, capacitors, voltages, currents, and frequencies. The code loops through the lines in the "circ" list, splits each line into individual elements, and appends the components to their respective lists. The code also generates a list of nodes present in the circuit.

The code then checks if there are multiple frequencies in the circuit and prints an error if that's the case. If there's only one frequency, the code sorts the nodes list and converts the lists of components into numpy arrays.

The code then initializes an array "A" to store the impedance matrix and an array "b" to store the node voltages. The code adds resistive components to the matrix A by looping through the resistor array and updating the corresponding entries in the matrix. The code also adds inductive components to the matrix A by looping through the inductor array and updating the corresponding entries in the matrix.

Similarly, the code adds capacitive components to the matrix A and the current sources to the matrix "b". Finally, the code returns the impedance matrix "A" and node voltages "b".

[119]:
```
circuit_file = input("Enter Your Netlist File")
# circuit_file="ckt1.netlist"
print(circuit_solver(circuit_file))
```

```
['n1', 'n2', 'n3', 'n4', 'n5', 'Iv1']  = [ 0.e+00+0.j  0.e+00+0.j  0.e+00+0.j
-5.e+00-0.j -5.e-04-0.j]
[[(0.0, 0.0)], [(0.0, 0.0)], [(0.0, 0.0)], [(5.0, -3.141592653589793)],
[(0.0005, -3.141592653589793)]]
None
```