EE21B043_week2[1]

February 10, 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.
- (Large bonus): make a solver that can do real-time transient simulations of a SPICE netlist and update the currents and voltages dynamically. They should also be plotted as a function of time and react to changes. This is something along the lines of https://www.falstad.com/circuit/. Ideally you should be able to do a real-time demo of some experiments you might conduct as part of a basic electronics lab, and simulate the behaviour of an oscilloscope and signal generator.

2 Factorial

2.1 Recursive approach

[17]:
```python
def fac(n):
    if(n<=1):
        return 1
    return n*fac(n-1)

n=int(input("Enter Number "))

print(fac(n))
%timeit fac(n)
```

```
Enter Number  7
```

```
5040
565 ns ± 18.3 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

The factorial function takes integer 'n' as input and returns it factorial
The function starts with a conditional statement 'if (n<=1)', which checks if the input value 'n' is less than or equal to 1. If 'n' is greater than 1, the function returns the result of 'n * fac(n-1)', where 'fac(n-1)' is a **recursive call** to the function 'fac' with an argument of 'n-1'.

The **'%timeit** fac(n)' line is a Jupyter magic command that calculates the execution time of the 'fac' function for the given input 'n'. The command outputs the average time taken to run the function multiple times, which gives a good estimate of the performance of the function.

2.2 For loop approach

[18]:
```python
def fac(n):
    ans=1;
    for i in range(1,n+1):
        ans=ans*i
    return ans
n=int(input("Enter Number "))
print(fac(n));
%timeit fac(n)
```

```
Enter Number  4
```

```
24
271 ns ± 5.8 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

Here also the function returns factorial of a given integer.
The function starts by initializing a variable 'ans' to 1. It then uses a **for loop** to iterate over the range '1 to n+1', and in each iteration, the variable 'ans' is updated as 'ans * i', where 'i' is the current iteration value

2.2.1 Comparision of time between two approaches

There is a difference in time between the recursive and for loop approaches.In the recursive approach, a function is called multiple times with different inputs, which requires **extra time and memory** to keep track of the intermediate results and function call frames.Each time the function is called, a new call frame is created, and the intermediate results are stored in the memory, which can lead to slow performance, especially for large numbers.

On the other hand, in the for loop approach, the calculation of the factorial is done iteratively in a **single function call**. There is no need to keep track of intermediate results or to create new function call frames, which makes the for loop approach more efficient than the recursive approach for calculating the factorial of a large number.

3 Linear Equation Solver

[6]:
```
import random
import numpy as np



def swap(ag_mx,i,j):
    temp=ag_mx[i]
    ag_mx[i]=ag_mx[j]
    ag_mx[j]=temp

def check_swap_row(t,ag_mx,n):
    for l in range(0,n):
        if ag_mx[l][t]!=0:
            swap(ag_mx,t,l)
            break

def lin_eq_solver(A,B):
    n=len(A)
    ag_mx = []
    for i in range(n):
        ag_mx.append(A[i] + [B[i]])
    for t in range(n):
        if ag_mx[t][t] == 0:
            check_swap_row(t,ag_mx,n)
        norm = ag_mx[t][t]
        for i in range(t, n+1):
            ag_mx[t][i]=ag_mx[t][i]/norm
        for i in range(n):
            if i == t:
                continue
            norm = ag_mx[i][t]
            for j in range(t, n+1):
                ag_mx[i][j]=ag_mx[i][j]- norm * ag_mx[t][j]
    for i in range(n):
        if ag_mx[i][i]==0 and ag_mx[i][n]==0:
            return "The given system of linear equations has infinite solutions"
        elif ag_mx[i][i]==0 and ag_mx[i][n]!=0:
            return "The given system of linear equations has no solution"
    ans = [ag_mx[i][n] for i in range(n)]
    return ans

v=int(input("Enter no of variables"))
e=int(input("Enter no of equations"))
A=[]
while v!= e:
    print("Matrix is not square,please enter same no of variables and equations␣
 ↪again.")
    v = int(input("Enter no of variables: "))
    e = int(input("Enter no of equations: "))
try:
    for i in range(v):
        print(f"Enter the coefficients of equation {i+1},where each coefficient␣
 ↪is space separated")
        row=[complex(x) for x in input().split()]
        A.append(row)
    print("Enter the elements of the constant vector, where each element is␣
 ↪space separated")
    B = [complex(x) for x in input().split()]
except ValueError:
        print("please give valid input elements")

print(lin_eq_solver(A,B))
```

```
Enter no of variables 2
Enter no of equations 2

Enter the coefficients of equation 1,where each coefficient is space separated

 1 2

Enter the coefficients of equation 2,where each coefficient is space separated

 2 1

Enter the elements of the constant vector, where each element is space separated

 2 1

[0j, (1-0j)]
```

The code takes two inputs from the user: the number of variables (v) and the number of equations (e).
The code defines several functions to solve the system of linear equations. The 'swap' function swaps two rows of the augmented matrix. The 'check_swap_row' function checks for the non zero

diagonal element, and if so, swaps that row with another row that has a diagonal element as zero.

The 'lin_eq_solver' function is the main function that implements the Gaussian elimination method. It starts by creating an augmented matrix and then performs row operations to transform the matrix to an **identity matrix**. Finally, it checks if the system of linear equations has a unique solution, no solution, or infinite solutions, and returns the solution if there is one.

3.1 Comparing above function and linalg.solve function

[16]:
```
A=[]
for i in range(10):
    A.append([random.randint(1, 20) for i in range(10)])
B=[random.randint(1, 200) for i in range(10)]

print(lin_eq_solver(A,B))
%timeit lin_eq_solver(A,B)

x=np.linalg.solve(A,B)
print(x)
%timeit np.linalg.solve(A,B)
```

```
[-5.147541142786879, 4.209695989078235, -6.450578419592989, -4.977959769741878,
1.0393555313559437, 4.441812590409696, -0.06956016194564185, 5.9379307241151125,
3.5643794192876403, 7.1942722148041325]
87.6 µs ± 2.04 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
[-5.14754114  4.20969599 -6.45057842 -4.97795977  1.03935553  4.44181259
 -0.06956016  5.93793072  3.56437942  7.19427221]
29.2 µs ± 2.78 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

The above code generates a 10x10 coeffecient matrix A and a constant vector B,Where the linear equations are solved using the function defined above. Also we are calculating the execution time of the function.

The time taken to exceute the above functions is different.The **lin_eq_solver funtion takes more time than linalg.solve.**
The lin_eq_solver function defined uses Gaussian elimination method to solve the system of linear equations, while numpy.linalg.solve uses an optimized linear algebra library to solve the system. The optimized linear algebra libraries have been highly optimized for performance and accuracy and use advanced algorithms.

4 circuit simulator

[8]:
```
import numpy as np
def NetlistConvert(file_path):
    with open(file_path, 'r') as file:
            netlist = file.readlines()
    net =[]
    t = 0
    for line in netlist:
        if line[0:2] == ".a": t = 1
        if line[0:2] == ".e": t = 0
        if t == 1:
             linesplit = line.split("#")[0].split('\n')[0].split('  ')
             net.append(linesplit[0])
        if line[0:2] == ".c": t = 1
    return net
net = NetlistConvert('ckt1.netlist')
print(net)
def MatrixSizeInc(MNA, b):
    l = MNA.shape[0]
    MNA_temp = np.zeros((l+1, l+1))
    b_temp = np.zeros((l+1,1))
    MNA_temp[:l, :l] = MNA
    b_temp[:l, :] = b
    return MNA_temp, b_temp
def addRes(MNAdc,value,i,j):
    MNAdc[i][i] += 1/float(value)
    MNAdc[j][j] += 1/float(value)
    MNAdc[i][j] -= 1/float(value)
    MNAdc[j][i] -= 1/float(value)
    return MNAdc
def create_MNA_matrix(netlist):
    nodes = set()
    components = []
    v_type = set()
    k = 0
    t = complex(0,1)
    freq = 0
    if netlist[-1].startswith(".ac"):
      split = netlist[-1].split()
      freq = float(split[2])*2*math.pi
    for line in netlist:
      split_line = line.split()
      print(split_line)
      if len(split_line) == 3:
        continue
      component_type = split_line[0]
      nodes.update([split_line[1], split_line[2]])
      try:
        components.append((component_type, split_line[1],␣
 ↪split_line[2],split_line[3],split_line[4]))
      except IndexError:
        try:
          components.append((component_type, split_line[1],␣
 ↪split_line[2],split_line[3]))
        except IndexError:
            components.append((component_type, split_line[1],␣
↪split_line[2],split_line[3],split_line[4],split_line[5]))
    node_dict = {node: i for i, node in enumerate(nodes)}
    num_nodes = len(nodes)
    num_components = len(components)
    MNA = np.zeros((num_nodes, num_nodes))
    b = np.zeros((num_nodes,1))
    if freq !=0:
        MNA = np.zeros((num_nodes, num_nodes))*t
        b = np.zeros((num_nodes,1))*t
    for component in components:
        try:
         component_type, node1, node2, acdc ,value = component
        except ValueError:
             try:
              component_type, node1, node2,value = component
             except ValueError:
              component_type, node1, node2, acdc ,value, phase = component
        i = node_dict[node1]
        j = node_dict[node2]
        k = node_dict["GND"]
        if component_type[0] == 'R':
            MNAdc = addRes(MNA,value,i,j)
        elif component_type[0] == 'I':
            v_type.update([acdc])
            b[i] -= float(value)
            b[j] += float(value)

        elif component_type[0] == 'V':
            v_type.update([acdc])
            if acdc.startswith("dc"):
                MNA,b = MatrixSizeInc(MNA,b)
                l = MNA.shape[0]-1
                MNA[l][i] += 1
                MNA[l][j] -= 1
                MNA[i][l] += 1
                MNA[j][l] -= 1
                b[l] -= float(value)

        elif component_type[0] == 'C':
            if freq == 0:
             print("This function cannot compute DC circuit with a capacitance")
            return None
            MNA[i][i] += t*float(value)*freq
            MNA[j][j] += t*float(value)*freq
            MNA[i][j] -= t*float(value)*freq
            MNA[j][i] -= t*float(value)*freq
        elif component_type[0] == 'L':
            if freq == 0:
             print("This function cannot compute DC circuit with a capacitance")
            return None
            MNA[i][i] += 1/t*float(value)*freq
            MNA[j][j] += 1/t*float(value)*freq
            MNA[i][j] -= 1/t*float(value)*freq
            MNA[j][i] -= 1/t*float(value)*freq
    b = np.squeeze(b)
    MNA = np.delete(MNA, k, axis=0)
    MNA = np.delete(MNA, k, axis=1)
    b = np.delete(b, k)
    v_type = list(v_type)
    if len(v_type) == 1 and v_type[0] == 'dc':
        return MNA, b, node_dict,0
    elif len(v_type) == 1 and v_type[0] == 'ac':
        return MNA, b, node_dict,1
    else:
        return("The code doesn't work for DC+AC supply")
net = NetlistConvert('ckt1.netlist')
try:
    V,b,nodes,check = create_MNA_matrix(net)
    V = V.tolist()
    b = b.tolist()
    a = lin_eq_solver(V,b)
    if check == 0:
      print(f"DC :{a}\nNodes:{nodes}")
    if check == 1:
      print(f"AC :{a}\nNodes:{nodes}")
except ValueError:
    a = create_MNA_matrix(net)
pass
```

```
['R1 GND 1 1e3', 'R2 1 2 4e3', 'R3 2 GND 20e3', 'R4 2 3 8e3', 'R5 GND 4 10e3',
'V1 GND 4 dc 5']
['R1', 'GND', '1', '1e3']
['R2', '1', '2', '4e3']
['R3', '2', 'GND', '20e3']
['R4', '2', '3', '8e3']
['R5', 'GND', '4', '10e3']
['V1', 'GND', '4', 'dc', '5']
DC :[0.0, 5.0, 0.0, 0.0, 0.0005]
Nodes:{'1': 0, 'GND': 1, '4': 2, '2': 3, '3': 4}
```

This code simulates a linear time-invariant electrical circuit. The code reads a netlist file and creates the corresponding modified nodal analysis (MNA) matrix and vector. The MNA matrix and vector can then be used to solve for the voltages and currents in the circuit.

Function Descriptions

NetlistConvert(file_path) This function takes the file path of the netlist file and returns a list of the components in the netlist.

Input
file_path (string): The file path of the netlist file
Output
net (list): A list of the components in the netlist
MatrixSizeInc(MNA, b)
This function increases the size of the MNA matrix and b vector to accommodate additional components.

Input
MNA (2D numpy array): The MNA matrix
b (2D numpy array): The b vector
Output
MNA_temp (2D numpy array): The increased MNA matrix
b_temp (2D numpy array): The increased b vector
addRes(MNAdc, value, i, j)
This function adds the resistance component to the MNA matrix and b vector.

Input
MNAdc (2D numpy array): The MNA matrix
value (float): The resistance value
i (int): The index of node 1 j (int): The index of node 2
Output
MNAdc (2D numpy array): The updated MNA matrix create_MNA_matrix(netlist) This function creates the MNA matrix and b vector from the netlist.

```
[ ]:
```