EE2703 - Week 2

Santhosh S P ee21b119

February 8, 2023

1 Function to find factorial of an integer N

1.1 Implementing factorial using recursion

[ ]:
```
#sample input
x=7

def fact_recursive(n):
    if n>1:
        return n*fact_recursive(n-1)
    else:
        return 1

print(fact_recursive(x))
%timeit fact_recursive(x)
```

```
5040
522 ns ± 6.98 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

1.2 Implementing factorial using a for loop

[ ]:
```
def fact_for(n):
    prod=1
    for i in range(1,n+1):
        prod*=i
    return prod

print(fact_for(x))
%timeit fact_for(x)
```

```
5040
400 ns ± 12.9 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

1.3 Factorial using the built-in factorial from the NumPy library

```
[ ]: import numpy as np
     print(np.math.factorial(x))
     %timeit np.math.factorial(x)
```

```
5040
89.1 ns ± 0.601 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)
```

1.4 Explanation

- The recursive solution takes is slower and takes around 538 ns compared to the for loop solution, which takes only 375 ns beacuse, everytime a function is called a new stack frame is created, it is added to the call stack before evaluating the arguments and running the body of the function, making it slower.
- The built-in factorial function of numpy takes only 85 ns to calculate the result. It is the fastest because numpy functions are written in C to optimize the performance of the code.

2 Solving a system of linear equations by Gauss-Jordan elimination

2.1 Creating matrices A and B filled with random numbers

```
[ ]: import numpy as np
     A=100*np.random.rand(10,10)
     B=100*np.random.rand(10,1)
```

The above code creates a matrix `A` of dimensions 10X10 and matrix `B` of dimensions 10X1 filled with random numbers from 0 to 100.

2.2 Function to solve $Ax = B$ by performing Gauss-Jordan elimination and return the solution

```
[ ]: def GaussJordanSolve(A,B):

         #augmented matrix
         C=np.c_[A, B]


         #we make sure that there are same number of equations as the number of␣
      ↪variables
         #for proper solution
         if np.shape(A)[0]==np.shape(A)[1]==np.shape(B)[0]:
             for i in range(len(A)):
                 lst=list(range(len(C)))
                 lst.remove(i)
```

```
            #checking if the diagonal element is zero
            if C[i][i]!=0:
                C[i]/=C[i][i]
            else:
                #shifting the rows if the diagonal element is zero
                for k in [l for l in lst if l>i]:
                    if C[k][i]!=0:
                        C[[i,k]]=C[[k,i]]
                        C[i]/=C[i][i]
                        break
                break

            #row operations in the augmented matrix
            for j in lst:
                C[j]=C[j]-C[j][i]*C[i]

        #checking for inconsistent system, and infinite solutions
        for row in C:
            if np.all((row == 0)):
                if row[:-1]==0:
                    print("Infinite solutions")
                    return None
                else:
                    print("Inconsistent")
                    return None

        #if the system is consistent the function returns the solution as an␣
 ↪array
        return C[:,-1].reshape(-1,1)
    else:
        print("Make sure that matrix A is a square matrix of dimension n and␣
 ↪matrix B is a matrix of dimensions nx1.")
        return None


#calling the function and printing the result and timing it
solution=GaussJordanSolve(A,B)
print(solution)

%timeit GaussJordanSolve(A,B)
```

```
[[-0.54130254]
 [-0.69933023]
 [ 0.09493242]
 [-1.2276136 ]
 [ 0.59690525]
 [ 0.71401177]]
```

```
 [ 0.184252  ]
 [-0.58370962]
 [ 1.48349274]
 [ 1.32258107]]
373 µs ± 17.6 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

- The above function `GaussJordanSolve()` checks if the number of equations is equal to the number of variables before performing the Gauss-Jordan elimination to return a proper a solution.
- If the diagonal element is non-zero the row is normalized and row operations are carried out.
- If it's zero, the row is swapped with other row in the matrix, it checks if the diagonal element is zero. If yes, it continues with the row operations, if not it tries swapping with a different row and this goes on till the diagonal element is non-zero.
- In the final RREF if a row is full of zeroes, the system has infinite solutions because essentially it has become a n variable system with $(n-1)$ equations. The functions returns `None`.
- In the final RREF if a row is full of zeroes except the final element, the system is inconsistent and the function returns `None`.
- If the system is consistent and reduced to the RREF form, the last column is returned as the solution after converting it into a column vector.

2.3 Using `np.linalg.solve()` to solve $Ax = B$

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

```
[[-0.54130254]
 [-0.69933023]
 [ 0.09493242]
 [-1.2276136 ]
 [ 0.59690525]
 [ 0.71401177]
 [ 0.184252  ]
 [-0.58370962]
 [ 1.48349274]
 [ 1.32258107]]
19 µs ± 101 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

2.4 Conclusion

`np.linalg.solve()` is much faster than `GaussJordanSolve()` because numpy is written in C to optimize the performance. Also, they return the same result.

3 Solving circuits by modified nodal analysis

3.1 Reading the file and extracting the required part and mapping node numbers to different node names

[ ]:
```
import numpy as np

#this has all important lines of the netlist (i.e everything b/w .circuit and .
 ↪end excluding them and the .ac line)
file=open("ckt3.netlist","r")
whole_file=file.read()
file.close()

#node_mapping is the dictionary that maps node numbers from 0,1,2.. to node␣
 ↪names
node_mapping={}
node_number=1

#to contain the list of all components (it's basically a nested list)
component_list=[]

#reading only the part between .circuit and .end and the line starting with .ac
net_list=whole_file[whole_file.find('.circuit'):whole_file.find('.end')+4]
for i in whole_file.splitlines():
    if i[:3]=='.ac':
        net_list=net_list+'\n'+i


lines=[i for i in net_list.splitlines() if i not in ['.circuit', '.end']]

for line in lines:      #excluding the lines with .circuit and .end
    if '#' in line:
        components=line[:line.find('#')].split() #excluding the part after #␣
 ↪(comments)
    else:
        components=line.split()
    component_list+=[components]

    #mapping all node names to numbers, starting from 0 for GND and whole␣
 ↪numbers for other nodes
    for i in components[1:-1]:
        if i=='GND':
            node_mapping['GND']=0

    #other nodes are mapped into whole numbers
    for i in components[1:3]:
        if i not in node_mapping and components[0]!='.ac':
```

```
            node_mapping[i]=node_number
            node_number+=1

#printing list of all components and the node mapping
print(component_list)
print(node_mapping)
```

```
[['V1', 'GND', '1', 'dc', '10'], ['R1', '1', '2', '1e3'], ['R2', '2', '3',
'1e3'], ['R3', '3', '4', '1e3'], ['R4', '4', '5', '1e3'], ['R5', '2', 'GND',
'2e3'], ['R6', '3', 'GND', '2e3'], ['R7', '4', 'GND', '2e3'], ['R8', '5', 'GND',
'2e3']]
{'GND': 0, '1': 1, '2': 2, '3': 3, '4': 4, '5': 5}
```

- The above cell, first opens the given file in read mode, reads the contents and closes it.
- We define a dictionary `node_mapping` to map all the different node names to numbers, starting from 0 for GND and natural numbers for other nodes.
- `component_list` is a list of the useful lines in the netlist file including the lines between .circuit and .end defining the components and the frequency of the AC source, excluding the comments.

3.2 Extracting different components from the netlist data

[ ]:
```
#extracting different parts from component_list

#contains lines having DC voltage source and DC current source
voltage_source_list=[]
current_source_list=[]

#contains lines having AC voltage source and AC current source
voltage_source_list_AC=[]
current_source_list_AC=[]

#contains lines having resistors, capacitors and inductors
resistor_list=[]
capacitor_list=[]
inductor_list=[]

#based on SPICE syntax we classify and add lines corresponding to the required␣
 ↪element
for i in component_list:
    if (i[0][0]=='V' or i[0][0]=='v') and i[3]=='dc':
        voltage_source_list+=[i]
    elif (i[0][0]=='V' or i[0][0]=='v') and i[3]=='ac':
        voltage_source_list_AC+=[i]
    elif (i[0][0]=='I' or i[0][0]=='i') and i[3]=='dc':
        current_source_list+=[i]
    elif (i[0][0]=='I' or i[0][0]=='i') and i[3]=='ac':
        current_source_list_AC+=[i]
```

```
    elif i[0][0]=='R' or i[0][0]=='r':
        resistor_list+=[i]
    elif i[0][0]=='C' or i[0][0]=='c':
        capacitor_list+=[i]
    elif i[0][0]=='L' or i[0][0]=='l':
        inductor_list+=[i]

#printing the contents of all those lists
print(voltage_source_list)
print(current_source_list)
print(resistor_list)
print(voltage_source_list_AC)
print(current_source_list_AC)
print(capacitor_list)
print(inductor_list)
```

```
[['V1', 'GND', '1', 'dc', '10']]
[]
[['R1', '1', '2', '1e3'], ['R2', '2', '3', '1e3'], ['R3', '3', '4', '1e3'],
['R4', '4', '5', '1e3'], ['R5', '2', 'GND', '2e3'], ['R6', '3', 'GND', '2e3'],
['R7', '4', 'GND', '2e3'], ['R8', '5', 'GND', '2e3']]
[]
[]
[]
[]
```

In the above cell, we define different lists to classify elements of `component_list` into their respective lists, based on the first letter of the lines according to the SPICE syntax.

3.3 Extracting the frequency

[ ]:
```
#to contain the list of frequencies
f_list=[]
for element in component_list:
    if element[0]=='.ac':
        f_list+=[float(element[2])]

#raising an exception if there are multiple frequencies involved, or there are␣
 ↪both
# ac and dc sources
if len(f_list)>1 or (len(f_list)>0 and len(voltage_source_list)>0):
    raise Exception("This program can't solve circuits with more than one␣
 ↪frequency.")


#if there's only a single frequency, calculate the angular frequency
if len(f_list)>0:
```

```
    print(f_list)
    #the angular frequency
    w=2*np.pi*f_list[0]
    print(w)
```

- In the above cell, we extract the frequency of operation of the circuit which is mentioned in the netlist line starting with `.ac`.
- We raise an exception if the circuit has more than one frequency, or it has both AC and DC sources.
- If we deal with only a single frequency, the angular frequency is also calculated.

3.4 Constructing the matrix

[ ]:
```
#the circuit matrix is in the form of Mx=N

#dimension of the matrix
dim=len(node_mapping)-1+len(voltage_source_list)+len(voltage_source_list_AC)


M=np.zeros((dim,dim), dtype=np.complex64)
N=np.zeros((dim,1) , dtype=np.complex64)

print(M)
print(N)
```

```
[[0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]]
[[0.+0.j]
 [0.+0.j]
 [0.+0.j]
 [0.+0.j]
 [0.+0.j]
 [0.+0.j]]
```

- We define `dim` as the number of nodes other than GND and the number of total voltage sources (DC and AC included).
- We define a square matrix `M` of dimension `dim` and a column matrix `N` of dimension `dim`.
- All these matrices contains variables of `np.complex64` datatype to deal with both DC circuit and impedances while solving for the steady state of AC circuits.

3.5 Adding the individual stamps of each of the elements

[ ]:
```
#adding stamps of resistors
def add_resistor_stamps(resistor_list):
    for res in resistor_list:

        #extracting the node numbers of the component
        n1=node_mapping[res[1]]
        n2=node_mapping[res[2]]

        #extracting the value of the component
        #we convert it to complex
        val=complex(res[3])

        #resistor stamp if both terminals are not GND
        if n1!=0 and n2!=0:
            M[n1-1,n1-1]+=1/val
            M[n1-1,n2-1]+=-1/val
            M[n2-1,n1-1]+=-1/val
            M[n2-1,n2-1]+=1/val

        #resistor stamp if node n2 corresponds to GND
        if n1!=0 and n2==0:
            M[n1-1,n1-1]+=1/val

        #resistor stamp if node n1 corresponds to GND
        if n1==0 and n2!=0:
            M[n2-1,n2-1]+=-1/val

#stamps of dc current sources
def add_current_stamps(current_source_list):
    for cur in current_source_list:

        #extracting the node numbers of the current source
        n1=node_mapping[cur[1]]
        n2=node_mapping[cur[2]]

        #extracting the value of the current source
        val=complex(cur[4])

        #current source stamp when both the nodes are not GND
        if n1!=0 and n2!=0:
            N[n1-1]+=-val
            N[n2-1]+=val

        #current source stamp if node n1 corresponds to GND
        if n1==0 and n2!=0:
```

```
            N[n2-1]+=val

        #current source stamp if node n2 corresponds to GND
        if n1!=0 and n2==0:
            N[n1-1]+=-val


#stamps of dc voltage sources
def add_voltage_stamps(voltage_source_list):
    row_no=len(node_mapping)-1

    for vol in voltage_source_list:

        #extracting the node numbers of the voltage source
        n1=node_mapping[vol[1]]
        n2=node_mapping[vol[2]]

        #extracting the value of the voltage source
        val=complex(vol[4])

        #voltage source stamp when both the nodes are not GND
        if n1!=0 and n2!=0:
            M[row_no,n1-1]+=1
            M[row_no,n2-1]+=-1
            M[n1-1,row_no]+=1
            M[n2-1,row_no]+=-1

            N[row_no]+=val

        #current source stamp if node n2 corresponds to GND
        elif n1!=0 and n2==0:
            M[row_no,n1-1]+=1
            M[n1-1,row_no]+=1

            N[row_no]+=val

        #current source stamp if node n1 corresponds to GND
        elif n1==0 and n2!=0:
            M[row_no,n2-1]+=1
            M[n2-1,row_no]+=1

            N[row_no]+=-val

        row_no+=1

##converting the ac sources and components to phasors
```

```
## as we're dealing with steady state of circuits, we convert a capacitor of␣
 ↪capacitance C
# to a resistor of 1/(j*w*C).
for cap in capacitor_list:


    #we append the required list corresponding to the capacitor to the list of␣
 ↪resistors
    # after changing its value to its impedance
    val=float(cap[3])
    cap[3]=str(complex(0,-1/(w*val)))
    resistor_list.append(cap)

## as we're dealing with steady state of circuits, we convert a inductor of␣
 ↪inductance L
# to a resistor of j*w*L.
for ind in inductor_list:

    #we append the required list corresponding to the inductor to the list of␣
 ↪resistors
    # after changing its value to its impedance
    val=float(ind[3])
    ind[3]=str(complex(0,w*val))
    resistor_list.append(ind)

## we convert AC voltage sources of amplitude V, phase phi to its correspnding␣
 ↪phasor,
# a DC voltage source of voltage Vcos(phi)+j*Vsin(phi)
for ac_vol in voltage_source_list_AC:

    #we append the required list corresponding to the AC voltage source to the␣
 ↪list of DC voltage sources
    # after changing its value to its corresponding phasor
    amp=float(ac_vol[4])
    phi=float(ac_vol[5])

    amp=str(complex(amp*np.cos(phi),amp*np.sin(phi)))
    ac_vol.pop()

    voltage_source_list.append(ac_vol)

## we convert AC current sources of amplitude I, phase phi to its correspnding␣
 ↪phasor
# a DC current source of current Vcos(phi)+j*Vsin(phi)
for ac_cur in current_source_list_AC:
```

```
    #we append the required list corresponding to the AC current source to the␣
 ↪list of DC current sources
    # after changing its value to its corresponding phasor
    amp=float(ac_cur[4])
    phi=float(ac_cur[5])


    amp=str(complex(amp*np.cos(phi),amp*np.sin(phi)))
    ac_cur.pop()

    current_source_list.append(ac_cur)


add_resistor_stamps(resistor_list)
add_current_stamps(current_source_list)
add_voltage_stamps(voltage_source_list)

print(M)
print(N)
```

```
[[ 0.001 +0.j -0.001 +0.j  0.    +0.j  0.    +0.j  0.    +0.j  1.    +0.j]
 [-0.001 +0.j  0.0025+0.j -0.001 +0.j  0.    +0.j  0.    +0.j  0.    +0.j]
 [ 0.    +0.j -0.001 +0.j  0.0025+0.j -0.001 +0.j  0.    +0.j  0.    +0.j]
 [ 0.    +0.j  0.    +0.j -0.001 +0.j  0.0025+0.j -0.001 +0.j  0.    +0.j]
 [ 0.    +0.j  0.    +0.j  0.    +0.j -0.001 +0.j  0.0015+0.j  0.    +0.j]
 [ 1.    +0.j  0.    +0.j  0.    +0.j  0.    +0.j  0.    +0.j  0.    +0.j]]
[[  0.+0.j]
 [  0.+0.j]
 [  0.+0.j]
 [  0.+0.j]
 [  0.+0.j]
 [-10.+0.j]]
```

- The above cell adds the required stamps in the matrices M and N, for all the components available in the netlist.
- The function `add_resistor_stamps()` takes the `resistor_list` as the input. It loops through it one element at a time. The netlist line say, `R1 n1 n2 V` is converted to its required stamp by:
  - adding $\frac{1}{V}$ at element in n_1^{th} row and n_1^{th} column and n_2^{th} row and to the n_2^{th} column of the matrix `M`.
  - subtracting the value $\frac{1}{V}$ from the element in the n_1^{th} row and n_2^{th} column and from the element in n_2^{th} row and n_1^{th} column of the matrix `M`.
- The function `add_current_stamps()` takes the `current_source_list` as the input. To process the netlist line `I1 n1 n2 I`,
  - the value $-I$ is added to $(n_1 - 1)^{th}$ row of the matrix `N`
  - the value I is added to its $(n_2 - 1)^{th}$ row of `N` matrix.
- The function `add_voltage_stamps()` takes the `voltage_source_list` as the input. To process the netlist line `V1 n1 n2 dc V`:

  - the value `1` is added to the next row after the rows corresponding to the nodal voltages at the $(n_1-1)^{th}$ column and the value `-1` is added to the $(n_2-1)^{th}$ column in the same row of the `M` matrix.
  - the value `1` is added to the next column after the rows corresponding to the nodal voltages at the $(n_1-1)^{th}$ row and the value `-1` is added to the $(n_2-1)^{th}$ row in the same column of the `M` matrix.
  - the value `V` is added to the $(n_1-1)^{th}$ row of the matrix `N`, and the value `-V` is added to the $(n_2-1)^{th}$ row of the matrix `N`.
- Every element in the `capacitor_list` is added as an element of the `resistor_list` with a value of $\frac{-1}{j\omega C}$.
- Every element in the `inductor_list` is added as an element of the `resistor_list` with a value of $j\omega L$.
- Every element in the `voltage_source_list_AC` is added as an element of the `voltage_source_list` with a value of $V\cos\phi + j(V\sin\phi)$, where V is the amplitude of the voltage source and ϕ is its phase.
- Every element in the `current_source_list_AC` is added as an element of the `current_source_list` with a value of $I\cos\phi + j(I\sin\phi)$, where I is the amplitude of the voltage source and ϕ is its phase.

3.6 Solving the matrix equation $Mx = N$

[ ]:
```
circuitsolution=GaussJordanSolve(A=M,B=N)
print(node_mapping)
print(circuitsolution)
```

```
{'GND': 0, '1': 1, '2': 2, '3': 3, '4': 4, '5': 5}
[[-1.0000001e+01+0.j]
 [-5.0292401e+00+0.j]
 [-2.5730996e+00+0.j]
 [-1.4035088e+00+0.j]
 [-9.3567258e-01+0.j]
 [ 4.9707610e-03-0.j]]
```

Matrices `M` and `N` are solved using the `GaussJordanSolve()` function. The solution matrix x is in the form of

$$\begin{bmatrix} V(n_1) \\ V(n_2) \\ \dots \\ I_{voltagesource} \end{bmatrix}$$

.

For circuits with DC components, the solution matrix represents the actual voltages in the nodes and the current through the voltage sources. For circuits with AC components, it represents the voltage in nodes and the current through the voltage source in phasor notation.