EE2703 Week 2

Vaddiraju Anshul EE21B151

February 8, 2023

1 Libraries

[28]:
```
import numpy as np
import math
```

2 Factorial and Timeit

2.1 Recursive Algorithm

[2]:
```
x = np.random.randint(0,1000)
def factorial(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n*factorial(n-1)
print(x)
%timeit factorial(x)
```

```
628
210 µs ± 1.28 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

2.2 For Loops

[3]:
```
def factorial(n):
    fac = 1
    for i in range(2,n+1):
        fac = fac*i
    return fac
print(x)
%timeit factorial(x)
```

```
628
167 µs ± 432 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

We can see that the recursive algorith takes more time than the for loop. This is because the recursion algorithm recalls the function n times which takes more time than iteration because it iterates all lines of code again n times where iteration repeats just a single line n times.

3 Linear Equation Solver

3.1 Shuffle:

[29]:
```
def Shuffle(A,B):
    n = len(A)
    f = [k for k in range(n)]
    for i in range(n):
            max_val = abs(A[i][i])
            row = i
            for j in range(i+1,n):
                if abs(A[j][i]) > max_val:
                    max_val = abs(A[j][i])
                    row = j
            A[row],A[i] = A[i],A[row]
            B[row],B[i] = B[i],B[row]
            f[row],f[i] = f[i],f[row]
    #print(f"A = {A}")
    #print(f"B = {B}")
    return A,B
```

I am using shuffle here which takes the maximum value of a column below the current row and pushes it into the current location. This is to avoid any consistent set of eqn being treated inconsistent due to a zero in the diagonal element

3.2 Upper Triangle:

[30]:
```
def UpperTriangle(A,B):
    n = len(A)
    for i in range(n):
        for j in range(i+1,n):
            norm = A[j][i] / A[i][i]
            for k in range(len(A[i])): A[j][k] = A[j][k] - A[i][k] * norm
            B[j] = B[j] - B[i] * norm
            #print(A,B)
    #print("Resultant Upper Triangle:")
    #print(f"A = {A}")
    #print(f"B = {B}")
    return A,B
```

Creates a upper tringular matrix from the previous matrix

3.3 Normalise:

[31]:
```
def Normalise(A,B):
    n = len(A)
    for i in range(n):
        norm = A[i][i]
        for j in range(n):
```

```
            A[i][j] = A[i][j]/norm
        B[i] = B[i]/norm
    #print("Normalised Upper Triangle:")
    #print(f"A = {A}")
    #print(f"B = {B}")
    return A,B
```

It makes all the diagonal elements of the matrix 1 which makes it easier for back substitution in the next step

3.4 Solver:

[32]:
```
def Solver(A,B):
    n = len(A)
    for i in reversed(range(n)):
        for j in range(i):
            B[j] = B[j] - B[i]*A[j][i]
            A[j][i] = A[j][i] - A[i][i]*A[j][i]

    #print("Final Matrix:")
    #print(f"A = {A}")
    #print(f"B = {B}")
    return B
```

It back substitutes the matrix from the last row to the first to find the solution of equations

3.5 Check Solution:

[33]:
```
def CheckSolution(a,b):
    k = []
    n = len(a)
    for i in range(n):
        if a[i][i] == 0: k.append(i)
    for j in k:
        for i in range(n):
            if a[j][i] != 0:
                norm = a[j][i]/a[i][i]
                a[j][i] = a[j][i] - a[i][i]*norm
                b[j] = b[j] - b[i]*norm

        if abs(b[j]) < 1/1000 and b[j] != 0:
            b[j] = 0

    for j in k:
        if b[j] != 0:
            print("System is inconsistent and has no solutions")
            break
        else:
```

```
            if j == k[len(k)-1]: print("System is inconsistent and has infinite␣
↪solutions")
```

Here, if the diagonal element of the upper triangular is 0. It means that the matrix is not singular i.e it is not consistent. - If, the row where the diagonal element is 0 has the value of b[row] as 0 that means there are infinite solutions.
- Else, the matrix has no solutions

3.6 Matrix Solver:

[34]:
```
def GaussianSolver(A,B):
    a,b = Shuffle(A,B)
    n = len(a)
    try:
        a,b = UpperTriangle(a,b)
        a,b = Normalise(a,b)
    except ZeroDivisionError:
        CheckSolution(a,b)
        return None
    else:
        answer = Solver(a,b)
        return answer
```

If the diagonal element is 0, it gives a ZeroDivisionError in the above steps. Hence, the matrix is checked for consistency.

[35]:
```
A = np.random.rand(10,10)
B = np.random.rand(10)
a = A.tolist()
b = B.tolist()
print(GaussianSolver(a,b))
```

```
[-0.7790433269159422, 1.3704247398840599, -0.10634276872391679,
0.7617050482424133, -0.9423996865564889, 0.3225433121471922,
-0.25432069113751293, -0.16572032854599927, 0.7853422000284571,
-0.05894843387472958]
```

4 Netlist Conversion

[36]:
```
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('ckt7.netlist')
print(net)
```

['I1 GND n1 ac 5 0 ', 'C1 GND n1 1', 'R1 GND n1 1000', 'L1 GND n1 1e-6', '.ac I1 1000']

[37]:
```
def MatrixSizeInc(MNA,b):
            k = np.zeros((1,MNA.shape[1]))
            m = np.zeros((MNA.shape[0]+1,1))
            l = [0]
            MNA = np.vstack((MNA,k))
            MNA = np.hstack((MNA,m))
            b = np.vstack((b,l))
            return MNA,b
```

To convert a (n x n) matrix to a (n+1 x n+1) matrix

[38]:
```
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
```

[44]:
```
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()
        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])

            if acdc.startswith("dc"):
                b[i] -= float(value)
                b[j] += float(value)

            if acdc.startswith("ac"):
                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)

            if acdc.startswith("ac"):
                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")
```

- This code implements a function that reads an electrical netlist and creates MNAac and MNAdc and bac and bdc to solve for the voltages and currents in an AC and DC circuit.

- The function splits each line in the netlist and processes components one-by-one, updating the MNA and b arrays based on the type of component. The MNA matrices contain information about the interconnections between the components and the b arrays contain the independent source information.
- The frequency information is taken from the netlist and used in the calculation of components such as capacitors and inductors.

- The function is implemented to handle resistors, current sources, voltage sources, capacitors, and inductors, and returns None if the function encounters a DC circuit with a capacitor,inductor.

[46]:
```
net = NetlistConvert('ckt1.netlist')
try:
        V,b,nodes,check = create_MNA_matrix(net)
        V = V.tolist()
        b = b.tolist()
        a = GaussianSolver(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
```

```
DC :[0.0, 0.0, 0.0, 5.0, 0.0005]
Nodes:{'2': 0, '3': 1, '1': 2, 'GND': 3, '4': 4}
```