EE2703 - Week 2

Hardik Gagrani, EE21B047

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).

- 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

[652]:
```
import numpy as np

def fac(x):
    fact = 1
    for i in range(x):
        fact *= i+1

    return fact

print(fac(8))
%timeit fac(8)
```

```
40320
563 ns ± 2.79 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

Here I used a for loop to calculate factorial of a number.

```
[653]: def fac_recursion(n):
           if n == 1:
               return 1
           else:
               return n*fac_recursion(n-1)

       print(fac_recursion(8))
       %timeit fac_recursion(8)
```

```
40320
752 ns ± 2.08 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

Here I used recursion to solve for factorial of a given number and it can be seen that recursion takes longer time than normal for loop. This is because in recursion the function is called multiple times, always creating a new set of variables and return address and therefore slowing the process of calculating factorial.

3 Linear Equation Solver

```
[654]: def alignment_of_rows(A, B):
           n = len(B)

           for i in range(n):
               pivot = i
               for j in range(i + 1, n):
                   if abs(A[j][i]) > abs(A[pivot][i]):
                       pivot = j
               A[i], A[pivot] = A[pivot], A[i]
               B[i], B[pivot] = B[pivot], B[i]

           return A, B
```

The function “alignment of rows” in this code aligns the rows of two matrices, `A` and `B`, using the absolute values of their constituent elements in the first matrix.

The first step of the function is to identify the row in the first matrix’s column I that has the highest absolute value. Then, in both matrices, this row is switched with row i. For each row in the matrix, this operation is repeated.

[655]:
```python
def upper_triangular(A, B):
    n = len(B)

    for k in range(n-1):
        for i in range(k+1, n):
            norm = A[i][k]/A[k][k]
            for j in range(k, n):
                A[i][j] -= norm*A[k][j]
            B[i] -= norm*B[k]
    return A, B
```

The “upper_triangular” function takes in two matrices, `A` and `B`, and converts them to an upper triangular form.

It does this by looping over all the rows (except the last row), calculating a normalization factor, and updating the elements in the current row and the vector B based on the normalization factor and the corresponding elements in a different row.

[656]:
```python
def back_substitution(A, B):

    n = len(B)
    for i in range(n):
        if(A[i][i]==0):
            if(B[i]==0):
                print("The given system of equations has infinte number of␣
↪solutions")
                return
            else:
                print("The given system of equations has no solutions")
                return

    X = np.zeros(n)
    for i in range(n-1, -1, -1):
        X[i] = B[i]
        for j in range(i+1, n):
            X[i] -= A[i][j]*X[j]
        X[i] /= A[i][i]

    return X
```

The “back substitution” function in this code employs two matrices, `A` and `B`, to solve a system of linear equations.

The first thing the function does is determine whether the diagonal elements of the matrix A are zero. The function writes “The provided system of equations has infinte number of solutions” if they are all zero and the elements of the vector B are likewise zero. The function prints “The provided system of equations has no solutions” if any of the diagonal elements of A are zero and the corresponding element in B is non-zero.

After initialising an array X with zeros, the function solves the system of linear equations represented by A and B using back substitution.

```
[657]: def gauss_elimination(A, B):
           A, B = alignment_of_rows(A, B)
           A, B = upper_triangular(A, B)
           Solution = back_substitution(A, B)
           return Solution
```

Finally creating a function with combination of all the functions defined above to ease the process of getting the solution.

```
[658]: def exceptions(A, B):
           try:
               if len(A) != len(A[0]):
                   raise ValueError("Matrix A must be a square matrix.")

               if len(B) != len(A):
                   raise ValueError("The number of rows in matrix A should be equal to␣
        ↪the number of elements in Matrix b.")

               for i in range(len(A)):
                   for j in range(len(A[0])):
                       if (type(A[i][j]) != float) and (type(A[i][j]) != int):
                           raise ValueError("The data entered has wrong Data Type!␣
        ↪Please check the input data again")

               for i in range(len(B)):
                   if (type(B[i]) != float) and (type(B[i]) != int):
                       raise ValueError("The data entered has wrong Data Type! Please␣
        ↪check the input data again")

               Solution = gauss_elimination(A, B)
               return Solution

           except ValueError as err:
               print(err)
```

The `Exceptions` function is designed to look for any errors that might be present in the provided data. Before passing matrices A and B to the `gauss_elimination` function, the function `exceptions` verifies their validity. It checks that matrix A is square, that the number of rows in A equals the number of elements in B, and that all elements in both matrices are of type float or int.

If any of these conditions is not met, a `ValueError` with an appropriate error message is raised and printed. If all conditions are met, the function returns the “gauss elimination” solution.

[659]:
```
#Sample Example
A = [[1, 3, 5],
    [4,2, 6],
    [14,41,1]]

B = [6,3, 6]

X = exceptions(A, B)
print(f"The solution for the system of equations using gaussian elimination is␣
 ↪{X}")
%timeit exceptions(A, B)

X = np.linalg.solve(A, B)
print(f"\nThe solution for the system of equations using default numpy function␣
 ↪is {X}")
%timeit np.linalg.solve(A, B)
```

```
The solution for the system of equations using gaussian elimination is
[-1.19822485  0.52810651  1.12278107]
8.5 µs ± 18.9 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)

The solution for the system of equations using default numpy function is
[-1.19822485  0.52810651  1.12278107]
5.22 µs ± 6.96 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

The default function of numpy run's faster than the gaussian elimination code implemented here.

Because it uses highly optimised and makes use of potent linear algebra libraries, the default numpy function to solve a matrix solves it significantly more quickly than a customised Gaussian elimination code.

[660]:
```
a = np.random.rand(10,10)*10 #Creating a 10 by 10 numpy array
b = np.random.rand(10)*100 #Creating a 10 by 1 numpy array

a = a.tolist() #Converting numpy array to list matrix for ease in calculation
b = b.tolist()

X = exceptions(a, b)
print(f"The solution for the system of equations using gaussian elimination is\n␣
 ↪{X}")
%timeit exceptions(a, b)

X = np.linalg.solve(a, b)
print(f"\nThe solution for the system of equations using default numpy function␣
 ↪is\n {X}")
%timeit np.linalg.solve(a, b)
```

```
The solution for the system of equations using gaussian elimination is
 [  8.87906559 -43.46161612   4.54443637  14.01387088  19.32434573
```

```
   1.75713264  -3.42654656  19.56190346 -13.9096041   -3.04002794]
80.6 µs ± 169 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)

The solution for the system of equations using default numpy function is
 [  8.87906559 -43.46161612   4.54443637  14.01387088  19.32434573
   1.75713264  -3.42654656  19.56190346 -13.9096041   -3.04002794]
23.6 µs ± 2.39 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

[661]:
```
import random
import time

a = 10
Random_list = random.sample(range(1, 101), a*a)
B = random.sample(range(101, 1010), a)

Col_Dim = a
Row_Dim = len(Random_list) // Col_Dim

A = [Random_list[i:i + Col_Dim] for i in range(0, len(Random_list), Col_Dim)]

X = exceptions(A, B)
print(f"The solution for the system of equations using gaussian elimination is\n␣
 ↪{X}")
%timeit gauss_elimination(A, B)

X = np.linalg.solve(A, B)
print(f"\nThe solution for the system of equations using default numpy function␣
 ↪is\n {X}")
%timeit np.linalg.solve(A, B)
```

```
The solution for the system of equations using gaussian elimination is
 [-22.57379302   8.7542978   -3.53603603   6.15463231   8.51558615
  -8.42795987  18.51275045   8.02907834   2.72022477  -9.7627865 ]
72.9 µs ± 113 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)

The solution for the system of equations using default numpy function is
 [-22.57379302   8.7542978   -3.53603603   6.15463231   8.51558615
  -8.42795987  18.51275045   8.02907834   2.72022477  -9.7627865 ]
25.8 µs ± 2.95 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

This is an additional list-based matrix creation method. This code and the code above are identical apart from this.

4 Netlist Circuit Solver

DC Circuit Solver

[716]:
```
def read_netlist(filename):

    #Opening the file and reading it line wise
    with open(filename, 'r') as f:
        lines = f.readlines()

    components = [] #Initialising an empty list to store the components.
    nodes = set()
    for line in lines:
        if line.strip() != "": #Processes data only when line is not blank, else␣
↪ignores
            line = line.strip().split() #Splits the line into a list of elements␣
↪and remove whitespaces.

            #If it's a resistor, append a tuple with component type, node1,␣
↪node2, and resistance value to the components list.
            if line[0][0] == 'R':
                components.append(('R', (line[1]), (line[2]), (line[3])))
                nodes.update([(line[1]), (line[2])]) #Adding node1 and node2 to␣
↪the nodes set.

            #If it's a voltage source, append a tuple with component type,␣
↪node1, node2, and voltage value to the components list.
            elif line[0][0] == 'V':
                components.append(('V', (line[1]), (line[2]), (line[4])))
                nodes.update([(line[1]), (line[2])]) #Adding node1 and node2 to␣
↪the nodes set.

            #If it's a current source, append a tuple with component type,␣
↪node1, node2, and current value to the components list.
            elif line[0][0] == 'I':
                components.append(('I', (line[1]), (line[2]), (line[4])))
                nodes.update([(line[1]), (line[2])]) #Adding node1 and node2 to␣
↪the nodes set.

    nodes = list(nodes) #Converting the node set to a list
    nodes.remove('GND') #Removing Ground as a node
    node_dict = {nodes[i]: i for i in range(len(nodes))} #Mapping the nodes to a␣
↪dictionary to avoid different namings of nodes

    print(f"The components in this circuit are: {components}")
    print(f"\nThe nodes in this circuit are: {nodes}")
    print(f"\nThe mapping of nodes is done by: {node_dict}")
    return components, nodes, node_dict
```

The read_netlist function reads a file containing circuit information and returns a list of components, a list of nodes, and a dictionary mapping each node to a unique integer.

The file is read and processed line by line, with each line added to the components list as a tuple if it describes a component (R, V, or I). The nodes used in the component are added to the nodes set and the node_dict dictionary maps each node to a unique integer. The function then prints the information and returns the components, nodes, and node_dict variables.

[717]:
```
def matrix_dim(components, nodes):
    dim = len(nodes)
    for component in components:
        if component[0] == 'V':
            dim += 1

    print(f"\nThe dimension of matrix A is {dim}\n")

    A = np.zeros((dim, dim))
    A = A.tolist()

    B = np.zeros(dim)
    B = B.tolist()

    X = np.zeros(dim)
    X = X.tolist()

    return dim, A, B, X
```

This code defines a function matrix_dim that takes in two arguments: components and nodes.

This function is designed to define matrix dimensions and creating them. Dimension is defined as the number of nodes + the number of independent voltage sources present in the circuit.

After obtaining the dimension, I create numpy arrays A, B, and X with the necessary dimensions and convert them to lists to facilitate matrix calculations.

[718]:
```
def construct_matrices(dim, A, B, X, components, node_dict):
    for component in components:
        if component[0] == 'R':
            if component[1] != 'GND':
                i = node_dict[component[1]]
            if component[2] != 'GND':
                j = node_dict[component[2]]
            value = float(component[3])
            if component[1] != 'GND':
                A[i][i] += 1/value
            if component[1] != 'GND' and component[2] != 'GND':
                A[i][j] -= 1/value
                A[j][i] -= 1/value
            if component[2] != 'GND':
                A[j][j] += 1/value

        elif component[0] == 'V':
            if component[1] != 'GND':
                i = node_dict[component[1]]
            if component[2] != 'GND':
                j = node_dict[component[2]]
            value = float(component[3])
            if component[1] != 'GND':
                A[i][dim-1] -= 1
                A[dim-1][i] += 1
            if component[2] != 'GND':
                A[j][dim-1] += 1
                A[dim-1][j] -= 1
            B[dim-1] += value

        elif component[0] == 'I':
            if component[1] != 'GND':
                i = node_dict[component[1]]
            if component[2] != 'GND':
                j = node_dict[component[2]]
            value = float(component[3])
            if component[2] != 'GND':
                B[j] -= value
            if component[1] != 'GND':
                B[i] += value

    return A, B
```

This code defines the construct matrices function, which accepts five arguments: `dim`, `A`, `B`, `X`, `components`, and `node dict`.

The function iterates through the component list, modifying the A and B matrices based on the component type (R for resistor, V for voltage source, I for current source). The `node_dict` dictionary is used to obtain the mapping of nodes to indices in the matrices.

The function updates the elements of the A matrix for a resistor component based on its resistance value and the nodes it connects. If one of the nodes is connected to `GND` (ground), I am ignoring it.

In similar way the Voltage and Current components are treated in order to generate a Modified Nodal Analysis matrix at the end. Keeping this in mind, all of the values in the matrix are updated.

[719]:
```
def final(filename):
    comp, node, node_dict = read_netlist(filename)
    inv_dict = dict(zip(node_dict.values(), node_dict.keys()))
    dim, A, B, X = matrix_dim(comp, node)
    A_final, B_final = construct_matrices(dim, A, B, X, comp, node_dict)
    X = exceptions(A_final, B_final)

    dim_virtual = 0
    for i in inv_dict:
        node_iterate = inv_dict[i]
        dim_virtual += 1
        print(f"The voltage at node {node_iterate} is {X[i]}")

    for i in range(dim_virtual, dim):
        print(f"The current through the voltage source {i-dim_virtual+1} is␣
↪{X[dim_virtual]}")
        dim_virtual += 1

    print("\n")
    return X
```

This code defines a function `final` that accepts one argument: `filename`, which is the name of a file containing a network description in the format specified.

The function first calls `read_netlist` to read the file's `components`, `nodes`, and `node-index mapping`. It also uses the `inv_dict` dictionary to create an inverse mapping from indices to nodes.

The function then calls `matrix_dim` to compute the dimensions of the matrices `A`, `B`, and `X`. The function then invokes `construct_matrices` to update the matrices based on the network components.

The function then calls exceptions on the final matrices `A_final` and `B_final` to solve for the solution matrix `X`.

Finally, I am iterating through the network's nodes and voltage sources, printing the voltage or current at each node or voltage source. The output is the final values of the vector X, which represents the network's voltages at nodes and controlling currents through independent voltage sources. AC Circuit Solver

[720]:
```
def read_netlist_AC(filename):
    with open(filename, 'r') as f:
        lines = f.readlines()
    components = []
    nodes = set()
    frequency = 0

    for line in lines:
        if line.strip() != "":
            line = line.strip().split()
            if line[0] == '.ac':
                frequency += float(line[2])
            if line[0][0] == 'R':
                components.append(('R', (line[1]), (line[2]), (line[3])))
                nodes.update([(line[1]), (line[2])])
            elif line[0][0] == 'V':
                if line[3] == 'ac':
                    components.append(('V', (line[1]), (line[2]), (line[3]),␣
↪(line[4]), (line[5])))
```

```
                    nodes.update([(line[1]), (line[2])])
                else:
                    components.append(('V', (line[1]), (line[2]), (line[3]),␣
↪(line[4])))
                    nodes.update([(line[1]), (line[2])])
            elif line[0][0] == 'I':
                if line[3] == 'ac':
                    components.append(('I', (line[1]), (line[2]), (line[3]),␣
↪(line[4]), (line[5])))
                    nodes.update([(line[1]), (line[2])])
                else:
                    components.append(('I', (line[1]), (line[2]), (line[3]),␣
↪(line[4])))
                    nodes.update([(line[1]), (line[2])])
            elif line[0][0] == 'C':
                components.append(('C', (line[1]), (line[2]), (line[3])))
                nodes.update([(line[1]), (line[2])])
            elif line[0][0] == 'L':
                components.append(('L', (line[1]), (line[2]), (line[3])))
                nodes.update([(line[1]), (line[2])])

    nodes = list(nodes)
    nodes.remove('GND')
    node_dict = {nodes[i]: i for i in range(len(nodes))}

    print(f"The components in this circuit are: {components}")
    print(f"\nThe nodes in this circuit are: {nodes}")
    print(f"\nThe mapping of nodes is done by: {node_dict}")
    print(f"\nThe frequency with which the AC source is operating is␣
↪{frequency}")
    return components, nodes, node_dict, frequency
```

This function is defined in the same way as the DC circuit function in that it reads a file containing circuit information and returns a list of components, a list of nodes, and a dictionary mapping each node to a unique integer.

But the difference here is that I am reading out frequency from the given netlist circuit and, in addition to components R (Resistor), V (Voltage source), I (Current source), I have also defined and updated ‘Capacitor’ ‘(C)’ and ‘Inductor’ ‘(L)’ in the components and nodes list.

[721]:
```
def matrix_dim_AC(components, nodes, node_dict):
    dim = len(nodes)
    for component in components:
        if component[0] == 'V':
            dim += 1

    print(f"\nThe dimension of matrix A is {dim}\n")
    A = np.zeros((dim, dim))
    A = A.tolist()

    B = np.zeros(dim)
    B = B.tolist()

    X = np.zeros(dim)
    X = X.tolist()

    return dim, A, B, X
```

Again, as in DC, this function calculates the required dimension of the matrix and thus defines the matrix with the corresponding dimensions.

Dimension is defined as the number of nodes + the number of independent voltage sources.

[722]:
```
import cmath

def complex_number(mag, phase):
    z = cmath.rect(mag, phase * cmath.pi / 180)
    return z
```

I defined a function `complex number` which uses the cmath module's rect() method to convert magnitude and phase arguments into a complex number. Before passing the phase to the rect() method, which returns the corresponding complex number, it is converted from degrees to radians. This generated complex number is returned by the function.

[723]:
```
def Inductor_impedence(value, frequency):
    mag = value*2*cmath.pi*frequency
    phase = 90
    X_L = complex_number(mag, phase)
    return X_L
```

The `Inductor_impedence` function is used to calculate the Impedance of the respective Inductor. It takes in two arguments: `value` = inductor value, `frequency` = frequency at which the AC circuit is operated, and calculates and returns the inductor's impedance.

[724]:
```
def Capacitor_impedence(value, frequency):
    mag = 1/(value*2*cmath.pi*frequency)
    phase = -90
    X_C = complex_number(mag, phase)
    return X_C
```

The `Capacitor_impedence` function is used to calculate the Impedance of the respective Capacitor. It takes in two arguments: `value` = capacitor's value, `frequency` = frequency at which the AC circuit is operated, and calculates and returns the capacitor's impedance.

[725]:
```
def construct_matrices_AC(dim, A, B, X, components, node_dict, frequency):

    Both_AC = 0
    Both_DC = 0

    for component in components:
        if component[0] == 'R':
            if component[1] != 'GND':
                i = node_dict[component[1]]
            if component[2] != 'GND':
                j = node_dict[component[2]]
            value = float(component[3])
            if component[1] != 'GND':
                A[i][i] += 1/value
            if component[1] != 'GND' and component[2] != 'GND':
                A[i][j] -= 1/value
                A[j][i] -= 1/value
            if component[2] != 'GND':
                A[j][j] += 1/value

        elif component[0] == 'V' and component[3] == 'dc':
            Both_DC += 1
            if component[1] != 'GND':
                i = node_dict[component[1]]
            if component[2] != 'GND':
                j = node_dict[component[2]]
            value = float(component[4])
            if component[1] != 'GND':
                A[i][dim-1] -= 1
                A[dim-1][i] += 1
            if component[2] != 'GND':
                A[j][dim-1] += 1
                A[dim-1][j] -= 1
            B[dim-1] += value

        elif component[0] == 'V' and component[3] == 'ac':
            Both_AC += 1
            if component[1] != 'GND':
                i = node_dict[component[1]]
            if component[2] != 'GND':
                j = node_dict[component[2]]
            magnitude = float(component[4])
            phase = float(component[5])
            value = complex_number(magnitude, phase)
            if component[1] != 'GND':
                A[i][dim-1] -= 1
                A[dim-1][i] += 1
            if component[2] != 'GND':
                A[j][dim-1] += 1
                A[dim-1][j] -= 1
            B[dim-1] += value

        elif component[0] == 'I' and component[3] == 'dc':
            if component[1] != 'GND':
                i = node_dict[component[1]]
            if component[2] != 'GND':
                j = node_dict[component[2]]
            value = float(component[4])
            if component[2] != 'GND':
                B[j] -= value
            if component[1] != 'GND':
                B[i] += value

        elif component[0] == 'I' and component[3] == 'ac':
            if component[1] != 'GND':
                i = node_dict[component[1]]
            if component[2] != 'GND':
                j = node_dict[component[2]]
            magnitude = float(component[4])
            phase = float(component[5])
            value = complex_number(magnitude, phase)
            if component[2] != 'GND':
                B[j] -= value
            if component[1] != 'GND':
                B[i] += value

        elif component[0] == 'C':
            if component[1] != 'GND':
                i = node_dict[component[1]]
            if component[2] != 'GND':
                j = node_dict[component[2]]
            Cap = float(component[3])
            value = Capacitor_impedence(Cap, frequency)
            if component[1] != 'GND':
                A[i][i] += 1/value
            if component[1] != 'GND' and component[2] != 'GND':
                A[i][j] -= 1/value
                A[j][i] -= 1/value
            if component[2] != 'GND':
                A[j][j] += 1/value

        elif component[0] == 'L':
            if component[1] != 'GND':
                i = node_dict[component[1]]
            if component[2] != 'GND':
                j = node_dict[component[2]]
            Ind = float(component[3])
            value = Inductor_impedence(Ind, frequency)
            if component[1] != 'GND':
                A[i][i] += 1/value
            if component[1] != 'GND' and component[2] != 'GND':
                A[i][j] -= 1/value
                A[j][i] -= 1/value
            if component[2] != 'GND':
                A[j][j] += 1/value

    return A, B, Both_DC, Both_AC
```

This function takes the following arguments: `dim`, matrices `A`, `B`, and `X`, `components`, `node dict`, and the `frequency` at which the circuit is stimulated.

The function modifies the values of A, B, and X for each component in components to reflect the component's behaviour in the electrical circuit. A resistor, capacitor, inductor, DC voltage or current source, or AC voltage or current source can be used as the component. The `Capacitor_impedence` and `Inductor_impedence` functions are used to calculate the impedance of a capacitor and an inductor, respectively. The `complex_number` function is used to combine the magnitude and phase of an AC voltage and current source into a complex number.

[748]:
```
def final_AC(filename):
    comp, node, node_dict, freq = read_netlist_AC(filename)
    inv_dict = dict(zip(node_dict.values(), node_dict.keys()))
    dim, A, B, X = matrix_dim_AC(comp, node, node_dict)
    A_final, B_final, DC_Sources, AC_Sources = construct_matrices_AC(dim, A, B,␣
↪X, comp, node_dict, freq)

    if DC_Sources>0 and AC_Sources>0:
        print("The given circuit has more than 1 frequency and therefore it␣
↪can't be solved using this method")
    else:
        X = np.linalg.solve(A_final, B_final)
        dim_virtual = 0
        for i in inv_dict:
            node_iterate = inv_dict[i]
            dim_virtual += 1
            complex_form = cmath.polar(X[i])
            print(f"The voltage at node {node_iterate} is {X[i]} whose magnitude␣
↪is {complex_form[0]} and phase is {complex_form[1]*180/cmath.pi}")

        for i in range(dim_virtual, dim):
            complex_form = cmath.polar(X[dim_virtual])
            print(f"The current through the voltage source {i-dim_virtual+1} is:␣
↪{X[dim_virtual]} whose magnitude is {complex_form[0]} and phase is␣
↪{complex_form[1]*180/cmath.pi}")
            dim_virtual += 1
        print("\n")
        return X
```

This is the `final_AC` function, the combination of all the earlier defined functions, which performs an AC analysis on an electrical circuit. The function reads the circuit information from the netlist file and takes a filename as input. The function `read_netlist` AC is used to extract information from the circuit such as components, nodes, node mapping, and frequency. The `matrix_dim_AC` function is used to determine the dimension of the matrices A, B, and X required for circuit simulation. The `construct_matrices_AC` function modifies these matrices to reflect the behaviour of electrical circuit components.

The resulting matrices are then used to solve for unknown voltages and currents in the circuit using the NumPy library's np.linalg.solve function.

[741]:
```
X = final("ckt1.netlist")
print(f"The solution vector for the given circuit is {X}")
```

```
The components in this circuit are: [('R', 'GND', '1', '1e3'), ('R', '1', '2',
'4e3'), ('R', '2', 'GND', '20e3'), ('R', '2', '3', '8e3'), ('R', 'GND', '4',
'10e3'), ('V', 'GND', '4', '5')]

The nodes in this circuit are: ['4', '3', '2', '1']

The mapping of nodes is done by: {'4': 0, '3': 1, '2': 2, '1': 3}

The dimension of matrix A is 5

The voltage at node 4 is -5.0
The voltage at node 3 is 0.0
The voltage at node 2 is 0.0
The voltage at node 1 is 0.0
The current through the voltage source 1 is 0.0005


The solution vector for the given circuit is [-5.e+00  0.e+00  0.e+00  0.e+00
5.e-04]
```

[742]:
```
X = final_AC("ckt2.netlist")
print(f"The solution vector for the given circuit is {X}")
```

```
The components in this circuit are: [('R', '1', 'GND', '1e3'), ('R', '1', '2',
'2e3'), ('R', '2', '3', '4e3'), ('V', '3', 'GND', 'ac', '2', '0'), ('V', '2',
'4', 'dc', '5'), ('R', '4', '5', '5e3'), ('I', '5', 'GND', 'dc', '1'), ('R',
'4', '6', '8e3'), ('I', 'GND', '6', 'dc', '10')]

The nodes in this circuit are: ['4', '3', '5', '1', '6', '2']

The mapping of nodes is done by: {'4': 0, '3': 1, '5': 2, '1': 3, '6': 4, '2':
5}
```

```
The frequency with which the AC source is operating is 50.0

The dimension of matrix A is 8

The given circuit has more than 1 frequency and therefore it can't be solved
using this method
The solution vector for the given circuit is None
```

[743]:
```
X = final("ckt3.netlist")
print(f"The solution vector for the given circuit is {X}")
```

```
The components in this circuit are: [('V', 'GND', '1', '10'), ('R', '1', '2',
'1e3'), ('R', '2', '3', '1e3'), ('R', '3', '4', '1e3'), ('R', '4', '5', '1e3'),
('R', '2', 'GND', '2e3'), ('R', '3', 'GND', '2e3'), ('R', '4', 'GND', '2e3'),
('R', '5', 'GND', '2e3')]

The nodes in this circuit are: ['4', '3', '5', '2', '1']

The mapping of nodes is done by: {'4': 0, '3': 1, '5': 2, '2': 3, '1': 4}

The dimension of matrix A is 6

The voltage at node 4 is -1.4035087719298245
The voltage at node 3 is -2.573099415204678
The voltage at node 5 is -0.9356725146198832
The voltage at node 2 is -5.029239766081871
The voltage at node 1 is -10.0
The current through the voltage source 1 is 0.004970760233918128


The solution vector for the given circuit is [-1.40350877e+00 -2.57309942e+00
-9.35672515e-01 -5.02923977e+00
 -1.00000000e+01  4.97076023e-03]
```

[744]:
```
X = final("ckt4.netlist")
print(f"The solution vector for the given circuit is {X}")
```

```
The components in this circuit are: [('V', 'GND', '1', '10'), ('R', '1', '2',
'2'), ('R', '2', 'GND', '3'), ('R', '2', '3', '5'), ('R', '3', 'GND', '10')]

The nodes in this circuit are: ['3', '1', '2']

The mapping of nodes is done by: {'3': 0, '1': 1, '2': 2}

The dimension of matrix A is 4

The voltage at node 3 is -3.703703703703704
The voltage at node 1 is -10.0
The voltage at node 2 is -5.555555555555556
The current through the voltage source 1 is 2.2222222222222223


The solution vector for the given circuit is [ -3.7037037  -10.
-5.55555556   2.22222222]
```

[745]:
```
X = final("ckt5.netlist")
print(f"The solution vector for the given circuit is {X}")
```

```
The components in this circuit are: [('R', 'GND', '1', '10'), ('V', 'GND', '1',
'10')]

The nodes in this circuit are: ['1']

The mapping of nodes is done by: {'1': 0}

The dimension of matrix A is 2

The voltage at node 1 is -10.0
The current through the voltage source 1 is 1.0


The solution vector for the given circuit is [-10.   1.]
```

[749]:
```
X = final_AC("ckt6.netlist")
print(f"The solution vector for the given circuit is {X}")
```

```
The components in this circuit are: [('V', 'GND', 'n3', 'ac', '5', '0'), ('C',
'n1', 'n2', '1'), ('R', 'n2', 'n3', '1000'), ('L', 'n1', 'GND', '1e-6')]

The nodes in this circuit are: ['n2', 'n3', 'n1']

The mapping of nodes is done by: {'n2': 0, 'n3': 1, 'n1': 2}

The frequency with which the AC source is operating is 1000.0

The dimension of matrix A is 4

The voltage at node n2 is (-1.8751873949628018e-10-3.062015181929011e-05j) whose
magnitude is 3.06201518198643e-05 and phase is -90.00035088109347
The voltage at node n3 is (-5-0j) whose magnitude is 5.0 and phase is -180.0
The voltage at node n1 is (-1.923920880164958e-10-3.1415926534719746e-05j) whose
magnitude is 3.1415926535308854e-05 and phase is -90.00035088109347
The current through the voltage source 1 is:
(0.004999999999812482-3.0620151819290114e-08j) whose magnitude is
0.004999999999906241 and phase is -0.0003508810934678035
```

```
The solution vector for the given circuit is [-1.87518739e-10-3.06201518e-05j
-5.00000000e+00-0.00000000e+00j
 -1.92392088e-10-3.14159265e-05j  5.00000000e-03-3.06201518e-08j]
```

```
[750]: X = final_AC("ckt7.netlist")
       print(f"The solution vector for the given circuit is {X}")
```

```
The components in this circuit are: [('I', 'GND', 'n1', 'ac', '5', '0'), ('C',
'GND', 'n1', '1'), ('R', 'GND', 'n1', '1000'), ('L', 'GND', 'n1', '1e-6')]

The nodes in this circuit are: ['n1']

The mapping of nodes is done by: {'n1': 0}

The frequency with which the AC source is operating is 1000.0

The dimension of matrix A is 1

The voltage at node n1 is (-1.3332000885000973e-10+0.0008164557820158241j) whose
magnitude is 0.000816455782015835 and phase is 90.00000935589411


The solution vector for the given circuit is [-1.33320009e-10+0.00081646j]
```