EE2703 - Week 2

Aditya Mallick EE21B005

February 8, 2023

[1]:
```
# importing required libraries
import numpy as np
import math
import os
```

[209]:
```
# Simple factorial finder using repeated multiplication in for loop
def simple_factorial(n):
  ans = n
  if (n > 0):
    for i in range(1,n):
      ans *= i
    return ans
  elif (n == 0):
    return 1

print(simple_factorial(10))
%timeit simple_factorial(10)
```

```
3628800
430 ns ± 5.37 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

[210]:
```
# Factorial using python's inbuilt function
print(math.factorial(10))
%timeit math.factorial(10)
```

```
3628800
64.5 ns ± 2.24 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)
```

[211]:
```
# Recursive implementation of factorial
def recursive_factorial(n):
  if (n > 1):
    return n*recursive_factorial(n - 1)
  else:
    return 1

print(recursive_factorial(10))
%timeit recursive_factorial(10)
```

```
3628800
```

```
878 ns ± 11.3 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

First of all, all these tests were run on the **IITM Jupyter Server**, to ensure that it is a fair comparison.

It is clear from the above 3 pieces of code that python's **inbuilt factorial function is much faster** as compared to the other 2 functions that I have written. **Recursion takes the longest time** of the 3.

It is interesting to observe that the **recursive implementation takes about double the amount of time taken by the iterative implementation**. However, this can be explained by the fact that in the recursive function, at each step the copy of the function with all variables is stored in memory (one step). When the base case is reached, the recursive calls are stopped, and the copies in memory return their values one by one (another step). After all values are returned, the function terminates. Thus the loop only has to iterate through every number once whereas the recursive function has to iterate until it reaches the end and then come back as well.

The speed of the inbuilt function comes from the fact that it is written in C. In fact all of the 'math' module functions are written in C. This allows for implementation of time-efficent complex algorithms which are simply called by this function.

[86]:
```
# Random 10x10 matrices for solving
a = np.array([[0, 15, 42, 49, 24, 40, 26, 5, 3, 37],
     [50, 40, -47, 43, -42, -18, -39, -3, -12, -1],
     [12, 18, 33, -45, 31, 48, -18, 8, -7, -38],
     [-28, 27, -49, -35, 5, -2, -37, 26, -22, -41],
     [23, -16, -1, -4, 33, -3, 37, 23, 30, -20],
     [1, 37, -7, 30, 10, 49, 36, 12, 18, -11],
     [41, 49, 19, 38, 43, -2, 36, 16, 93, 40],
     [44, 38, 37, 18, 11, -6, -18, 22, 8, -17],
     [-10, 36, 19, 27, 12, -17, -9, 11, -16, 25],
     [2, 19, 23, 27, -12, 21, 10, -13, 9, 42]])
b = np.array([-4, 33, 29, -14, -1, 19, 42, 41, -3, 5])
```

[87]:
```
def gausselim_partialpivoting(A,B):
    n = len(A)
    ans = [0 for i in range(n)]

    # iterate through the columns
    for counter in range(n - 1):
        # partial pivoting -> Swaps rows based on magnitude of leftmost␣
 ↪element, helps prevent division by zero
        for i in range(counter + 1,n):
            if (abs(A[i][counter]) > abs(A[counter][counter])):
                B[[counter,i]] = B[[i, counter]] # shorthand notation for␣
 ↪swapping numpy rows
                A[[counter,i]] = A[[i, counter]]

        # Forward Elimination -> Converts to an upper triangular matrix
```

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

    # Back Substitution
    for i in range(n - 1,-1,-1):
        sum = B[i]
        for j in range(n - 1, i - 1, -1):
            sum -= A[i][j]*ans[j]
        ans[i] = sum/A[i][i]

    return(ans)
```

Above is an implementation of gaussian elimination in python with partial pivoting.

Forward Elimination with Partial Pivoting

- The first step involves finding the row with the largest element in the leftmost column (called a **pivot**), and swapping it with the topmost row.
- Then, we eliminate the first element of the subsequent row by normalising the row above it and subtracting.
- Finally, we change our reference frame to the (n - 1) x (n - 1) matrix obtained by hiding the leftmost column and topmost row, and repeat the above steps until we have an upper triangular matrix. Keep in mind that any change made in matrix A will also be reflected in matrix B.
- The use of pivots ensures that we always divide by large numbers. This helps reduce errors caused by rounding or division by zero.

Back Substitution With the upper triangular matrix obtained, we can now sequentially find the values of our unknowns by using the coefficients in A. The method is as follows:

- The last row in A will have one coefficent, thus X_n is found by directly dividing corresponding value in B by this coefficient.
- The row preceding that will have two coefficients. By substituting X_n from before, we can again solve for X_n-1. Thus we can continue like this, finding each variable one after the other by back substituting the known variables.

[93]:
```
# Displaying the solutions
x = np.array([f"X{i + 1}" for i in range (len(b))])
solution1 = gausselim_partialpivoting(a,b)
for i in range(len(solution1)):
    print(f"{x[i]} = {solution1[i]}")

print(np.linalg.solve(a,b))
```

```
%timeit np.linalg.solve(a,b)
%timeit gausselim_paramlist(a,b)
```

```
X1 = 0.30695015227740746
X2 = 0.5753207630766094
X3 = 0.22796832847335496
X4 = 0.20531742281961884
X5 = 0.21414240729551237
X6 = -0.1103265696522512
X7 = -0.06663127690100427
X8 = -1.0830703012912481
X9 = 0.2467718794835007
X10 = -0.6470588235294118
[ 0.30695015  0.57532076  0.22796833  0.20531742  0.21414241 -0.11032657
 -0.06663128 -1.0830703   0.24677188 -0.64705882]
19 µs ± 390 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
325 µs ± 9.61 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

As can be seen above, np.linalg.solve is around 15 times faster than my solver for a 10x10 system of linear equations. It must be using a faster algorithm, perhaps LU decomposition.

0.1 SPICE Netlist Solver

Below is an implementation of a circuit solver that first converts a spice netlist into an understandable circuit. Then, using MNA, we solve for our unkown variables and output them.

[122]:
```
def circuit_simulator(file):
    txt = file.read().splitlines()
    ac_freq = 0

    # checking for ac sources
    for line in txt:
        if line[:3] == '.ac':
            if (line.find('#') != -1):
                line = line[:line.find('#')].strip()
            ac_freq = float(line.split(" ")[-1])

    # cleaning data
    spice_code = txt[txt.index('.circuit') + 1:txt.index('.end')]
    for line in spice_code:
        newline = line
        if (line.find('#') != -1):
            newline = line[:line.find('#')].strip()
        spice_code[spice_code.index(line)] = newline.strip().split(" ")

    # creating component dictionary
    components = {}
```

```
    for line in spice_code:
        if (ac_freq):
            if line[0][0] == 'V' or line[0][0] == 'I':
                if line[3] != 'ac':
                    print("Cannot evaluate both DC and AC sources.")
                    return -1
                else:
                    components[line[0]] = [line[1], line[2], line[4], line[5]]
            else:
                components[line[0]] = [line[1], line[2], line[-1]]
        else:
            components[line[0]] = [line[1], line[2], line[-1]]

    # get A,B matrices
    matrices = mna_builder(components, ac_freq)
    nodes = list(matrices[2].keys())

    # find solution
    solution = gausselim_partialpivoting(matrices[0],matrices[1])

    # print output
    for i in range(len(solution)):
        if (i < len(matrices[2]) - 1):
            print(f"Voltage at node {nodes[i + 1]} = {solution[i]} V")
        else:
            print(f"I{i - len(matrices[2]) + 2} = {solution[i]} A")

    return 0
```

- We have defined a base function which when called prints out the solution matrix. This function takes the file (assumed to be open already) as an input and reads the content. We search the netlist for AC sources, and the '.circuit' and '.end' indicators, all while skipping any comments or junk to ensure that we only get the required data.
- The cleaned data is then iterated through to create a dictionary of components. Each component has its name as the key and the nodes it is connected to, numeric value as values. AC sources have additional parameters that are accounted for.
- In case **multiple AC frequencies and/or DC + AC sources together are found, the function breaks and returns -1, along with an error message.**
- This dictionary serves as the base for our next function, mna_builder. We call mna_builder to retrieve the MNA matrices A and B formed by this circuit, and also the dictionary 'nodes'.
- We then use our matrix solver to solve for the solution and iteratively print the value of each unknown. The 'nodes' dictionary helps us determine the name of each node, and the difference between the number of unknowns and number of nodes (excluding 'GND') is the number of auxiliary currents. For a succesful circuit, the function will return 0.

[130]:
```
def mna_builder(components, ac_freq=0):
```

```
    # nodes dictionary for using later in filling array
    nodes = {}
    x = 0
    for component in components:
        if (components[component][0] not in nodes):
            nodes[components[component][0]] = x
            x += 1
        if (components[component][1] not in nodes):
            nodes[components[component][1]] = x
            x += 1

    #  initialize A and B matrices in AX = B
    nodecount = len(nodes)
    A = np.full((nodecount, nodecount), 0, dtype = complex)
    B = np.full(nodecount, 0, dtype = complex)

    # Filling up the matrices
    for component in components:
        if component[0] == 'R':

            val = float(components[component][2])

            # KCL at node 1 and node 2 creates 4 entries for each resistor, two␣
↪+ve and two -ve
            A[nodes[components[component][0]]][nodes[components[component][0]]␣
↪+= 1/val
            A[nodes[components[component][0]]][nodes[components[component][1]]␣
↪+= -1/val
            A[nodes[components[component][1]]][nodes[components[component][0]]␣
↪+= -1/val
            A[nodes[components[component][1]]][nodes[components[component][1]]␣
↪+= 1/val

        if (ac_freq):
            # AC analysis involves complex representation of V, I, L, C
            if component[0] == 'V':

                phase = float(components[component][3])

                # V value is taken in rectangular form
                val = float(components[component][2])*(math.cos(phase) +␣
↪1j*math.sin(phase))

                # Increasing dimensions to account for additional voltage␣
↪equation
                A = np.column_stack((A, np.full(nodecount, 0)))
```

```
                A = np.row_stack((A, np.full(nodecount + 1, 0)))
                B = np.append(B, val)

                # keeping up with the dimension change
                nodecount += 1

                # auxiliary current
                A[nodes[components[component][0]]][nodecount - 1] += 1
                A[nodes[components[component][1]]][nodecount - 1] += -1

                # voltage additional equation
                A[nodecount - 1][nodes[components[component][1]]] += -1
                A[nodecount - 1][nodes[components[component][0]]] += +1

            elif component[0] == 'L':

                val = float(components[component][2])*1j*ac_freq*2*math.pi

                # Reactance of inductor = j*w*L
                ␣
↪A[nodes[components[component][0]]][nodes[components[component][0]]] += 1/val
                ␣
↪A[nodes[components[component][0]]][nodes[components[component][1]]] += -1/val
                ␣
↪A[nodes[components[component][1]]][nodes[components[component][0]]] += -1/val
                ␣
↪A[nodes[components[component][1]]][nodes[components[component][1]]] += 1/val

            elif component[0][0] == 'C':

                val = 1/(float(components[component][2])*1j*ac_freq*2*math.pi)

                # Reactance of capactitor = 1/j*w*C
                ␣
↪A[nodes[components[component][0]]][nodes[components[component][0]]] += 1/val
                ␣
↪A[nodes[components[component][0]]][nodes[components[component][1]]] += -1/val
                ␣
↪A[nodes[components[component][1]]][nodes[components[component][0]]] += -1/val
                ␣
↪A[nodes[components[component][1]]][nodes[components[component][1]]] += 1/val

            elif component[0] == 'I':

                phase = float(components[component][3])
```

```
                # I value is taken in rectangular form
                val = float(components[component][2])*(math.cos(phase) +␣
↪1j*math.sin(phase))

                # Current source only needs to be filled in B
                B[nodes[components[component][0]]] += val
                B[nodes[components[component][1]]] += -val

        else:
            # DC current and voltage sources
            if component[0] == 'V':

                val = float(components[component][2])

                # Increasing dimensions to account for additional voltage␣
↪equation
                A = np.column_stack((A, np.full(nodecount, 0)))
                A = np.row_stack((A, np.full(nodecount + 1, 0)))
                B = np.append(B, val)

                # keeping up with the dimension change
                nodecount += 1

                # auxiliary current
                A[nodes[components[component][0]]][nodecount - 1] += 1
                A[nodes[components[component][1]]][nodecount - 1] += -1

                # voltage additional equation
                A[nodecount - 1][nodes[components[component][1]]] += -1
                A[nodecount - 1][nodes[components[component][0]]] += +1


            elif component[0] == 'I':

                val = float(components[component][2])

                # Current source only needs to be filled in B
                B[nodes[components[component][0]]] += val
                B[nodes[components[component][1]]] += -val

    # Delete GND row and column in A, and element in B
    A = np.delete(np.delete(A,0,0),0,1)
    B = np.delete(B,0)



    return([A,B,nodes])
```

As the name suggests, this function creates the MNA matrix for the circuit from the components present in the input. In case AC components are present, an additional optional argument 'ac_freq' has also been passed. This function only solves circuits having a singular frequency, **multiple AC frequencies and/or DC + AC sources together are not supported**.

The steps are as follows:

- Create a dictionary containing all nodes present in the circuit, including GND by iterating through components and adding all new nodes. Assign the name of the component as key and values are numbers starting from 0, to be used for indexing later.
- Initialize A and B matrices having size equal to the number of nodes found, fill them with zeros.
- Iterate through each component in the circuit. By analysing KCL at each node, we can fill the matrix. For each component, we know that its value will appear 4 times: twice positive at the nodes on the diagonal and twice negative at the non-diagonal elements (symetrically) whose row and column index are determined by the nodes to which it is connected.
- For each voltage source, we must add an extra dimension to the A and B matrices. In A, to account for the new equation corresponding to the source we add a row, and for the new auxiliary current variable passing through it we add a column. In B, we add a new element '0' to match this size change. Without loss of generality we add all these dimensions at the ends of the matrices. The value of the voltage source is filled in B, and we fill '+1's and'-1's in the array A wherever required according to the nodes to which it is connected.
- For each current source, we only need to modify the B matrix at the nodes to which it is connected. **We have considered currents entering to be negative and currents leaving to be postive. Thus by netlist convention, the current is positive in the B matrix at the node it enters, and vice versa.**
- For AC components and sources, the values have been transformed accordingly and the same procedure for filling the matrix is followed.
- After all this is done, we must **remember to delete the row and column in the A matrix corresponding to the GND node, since it should not appear in the final MNA matrix. We included it just to make our code simpler (by not having to check for it whenever we fill the matrix). Similarly delete the corresponding element in the B matrix. By our convention, this has been taken as the first column, row and element.**

We return the values of A and B, (and also the dictionary nodes, which we will use to print output and determine the number of auxiliary currents).

[131]:
```
try:
    filename = input("Enter name of file present in spice_netlists: ")
    if (filename[-8:] != '.netlist'):
        raise ValueError
    if not os.path.isfile(f"spice_netlists/{filename}"):
        raise FileNotFoundError
except ValueError:
    print("Not a valid filename, try again.")
except FileNotFoundError:
```

```
    print('The netlist file you gave is not present in spice_netlists.')
else:
    with open(f"spice_netlists/{filename}", "r") as f:
        circuit_simulator(f)
```

```
Enter name of file present in spice_netlists:  ckt4.netlist
```

```
Voltage at node 1 = (-10-0j) V
Voltage at node 2 = (-5.555555555555556+0j) V
Voltage at node 3 = (-3.7037037037037037+0j) V
I1 = (-2.2222222222222214-0j) A
```

Finally, the user is prompted for a filename input. First we check for validity of filename by ensuring it ends in '.netlist'. Then, if the file exists within the 'spice_netlists' directory, it is opened and passed onto our circuit solver. If either of these conditions aren't passed, then we raise exceptions and exit with an error message.