EE21B126 APL Week 2

Shreya .S. Ramanujam EE21B126

February 8, 2023

The zip folder contains the original jupyter notebook (.ipynb), which can be executed either on local jupyter or on this server. It also contains the exported LaTeX version of the notebook. I have made use of numpy, cmath and sys libraries in this notebook. Firstly, we import the necessary libraries. `cmath` is for complex number calculations.

[1]:
```
import numpy as np
import cmath
import sys
```

0.1 Factorial

[2]:
```
def factorial(n):
    if(n == 0):
        return 1
    return(n*factorial(n-1))
```

The above function recursively calculates the factorial of any integer, using the relation $N! = N \times (N-1)!$

[3]:
```
factorial(6)
```

[3]: 720

[4]:
```
def factorialiterative(n):
    p = 1
    for i in range(1, n+1):
        p *= i
    return(p)
```

This is an alternative iterative implementation of factorial, which uses $N! = 1 \cdot 2 \cdot 3 \ldots N$

[5]:
```
factorialiterative(7)
```

[5]: 5040

1 Gaussian Elimination

[6]:
```
def pivot(A, b, col, n):
    pivotRow = col
    for i in range(col, n):
        if abs(A[i][col]) > abs(A[pivotRow][col]):
            pivotRow = i
    return pivotRow
```

The pivot function finds the largest element in the given column and returns its row index. Later in the code, we switch this large element row to become the "pivot" row (the row that we subtract from all other rows below it, to generate upper triangular matrix). We do this so that we don't divide by small numbers in the normalisation step, since this may result in coefficients that are too big to fit in the 128 bit complex coefficient matrix we use in Gaussian Elimination.

[7]:
```
def gausselimcomplex(A, b):
    n = A.shape
    r = n[0]
    z = np.zeros(n[1], dtype = 'complex') # zero row
    if n[0] < n[1]:
        print("Infinite solutions because no of equations < no of variables")
        return
    if n[0] > n[1]:
        print("No solution!") # Inconsistent equations
        return
    for i in range(r): # current row
        exc = pivot(A, b, i, r) # find largest element in the current column␣
↪and make corresponding row the pivot row
        A[[i, exc]] = A[[exc, i]]
        b[[i, exc]] = b[[exc, i]]
        f1 = A[i][i]
        A[i] = A[i]/f1
        b[i] = b[i]/f1 #normalising the current row
        for j in range(i+1, r): # for all rows below current row
            f2 = A[j][i]
            A[j] = A[j] - f2*A[i] #make corresponding element of subsequent␣
↪rows zero
            b[j] = b[j] - f2*b[i]
            if (z == A[j]).all():
                if b[j] != 0: # if the coefficient matrix has all zero␣
↪coefficients but the b matrix has non zero element, we have no solutions
                    print("Inconsistent equations!")
                    return
                else:
                    print("Infinite solutions!")# if we encounter a zero row,␣
↪implying that one of the equations was a linear combination of the rows␣
↪above it, we have infinite solutions; not enough unique equations
```

```
                return


    x = np.zeros(r, dtype = 'complex') # solution matrix
    x[r-1] = b[r-1]

    for i in range(r-2, -1, -1):
        temp = np.dot(A[i], x)
        x[i] = b[i] - temp

    # print(A)
    # print(b)
    # print(x)
    return x
```

The `gausselimcomplex` function performs Gaussian Elimination. The steps are as follows: - Firstly, we pick a pivot row and take it to the top of the sub matrix we are currently considering - We then normalize the pivot row by dividing by the first element of the row - We then subtract this row from each subsequent row by multiplying with an appropriate factor (first element of each row) to get the first element as zero. This is done to generate the upper triangular matrix for Gaussian Elimination. - Once we have the upper triangular matrix, we back substitute from bottom to top to get the required values of the variables.

[8]:
```
p = np.array([[1, 1, 1], [1, -1, 1], [9,8,5]], dtype = float)
q = np.array([[2], [0], [9]], dtype = float)
```

[9]:
```
p = np.random.randn(10, 10)*10
q = np.random.randn(10, 1)*10
a = p
b = q
```

[10]:
```
ans = gausselimcomplex(p, q)
print(ans)
%timeit gausselimcomplex(p, q)
```

```
[11.59210759+0.j -7.66790891+0.j  4.3996422 +0.j -2.58029954+0.j
 -6.20415953+0.j -3.27938511+0.j  4.56930555+0.j  1.7797771 +0.j
  2.50581619+0.j -6.40928509+0.j]
537 µs ± 16.2 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

[11]:
```
print(np.linalg.solve(a, b))
%timeit np.linalg.solve(a, b)
```

```
[[11.59210759]
 [-7.66790891]
 [ 4.3996422 ]
 [-2.58029954]
 [-6.20415953]
 [-3.27938511]
```

```
 [ 4.56930555]
 [ 1.7797771 ]
 [ 2.50581619]
 [-6.40928509]]
20.3 µs ± 902 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

As we can see, `np.linalg.solve` runs approximately 30 times faster than the handwritten Gaussian Elimination solver.

2 Spice Simulator

[12]:
```
def singlefreq(inp):
    currfreq = "nothing" # initializing frequency with invalid value to update␣
 ↪with frequency value
    lineno = -1 # line number of where the frequency information is given in␣
 ↪the input string
    for i in range(len(inp)):
        params = inp[i].split()
        if len(params) == 0:
            continue
        if params[0] == ".ac":
            if currfreq == "nothing":
                currfreq = params[2]
                lineno = i
            if params[2] != currfreq:
                print("Multiple operating frequencies cannot be handled!")
                sys.exit()
    return(lineno)
```

The `singlefreq` function checks if our circuit has single operating frequency for all AC sources, since this solver cannot handle multiple operating frequencies. If multiple frequencies are found, we exit the code execution.

[13]:
```
def matrix_prep(inp, dc, i):
    inp_new = []
    while inp[i] != ".end\n":
        params = inp[i].split() # reading line by line
        component = params[0][0]
        nodeFrom = params[1]
        nodeTo = params[2]
        temp = inp[i]
        if (params[3] == 'ac' and dc) or (params[3] == 'dc' and not dc): # we␣
 ↪remove AC sources when we are solving the purely DC sources and vice versa␣
 ↪to get purely AC or DC netlist
            if component == "V":
                temp = params[0] +' ' + nodeFrom +' ' + nodeTo +' ' + 'dc'+' ' ␣
 ↪+ '0\n' # shorting voltage source by making it 0V
```

```
            if component == "I":
                i += 1
                continue # shorting current source by disconnecting it
        inp_new.append(temp)
        i += 1
    inp_new.append(".end\n")
    freqline = singlefreq(inp)
    if not dc:
        if freqline > -1:
            inp_new.append(inp[freqline]) # adding the line with the frequency␣
↪information
    return inp_new
```

The matrix_prep function scans the matrix and ensures that the netlist has only one particular type of sources; either AC sources or DC sources. This is done to ensure that the netlist being fed into the next segments of the MNA solver has either all AC or all DC sources (assuming all AC sources, if present, have the same frequency).

[14]:
```
def matrix_init(inp, i, dc):
    x = []
    while inp[i] != ".end\n":
        params = inp[i].split()
        component = params[0][0]
        nodeFrom = params[1]
        nodeTo = params[2]
        if component == "V": # adding auxillary current variable for every␣
↪voltage source, flowing from its low to high voltage terminal
            x.append("I" + params[0])
        if (component == "L" or component == "C") and dc:
            i += 1
            continue
        if not(nodeFrom in x): # appending nodes voltage variables (if not␣
↪already added) to the variable matrix
            x.append(nodeFrom)
        if not(nodeTo in x):
            x.append(nodeTo)
        i += 1

    n = len(x)
    M = np.zeros((n,n), dtype = complex) # M is a square matrix with dimensions␣
↪nxn, where n is the number of unique variables in the variable matrix
    b = np.zeros(n, dtype = complex) # b has dimensions equal to the size of␣
↪the variable matrix
    return M, x, b
```

The matrix_init function determines the number of unique node voltage variables and auxillary current variables (one for each voltage source) and creates appropriately sized matrices M (coefficient matrix), b and x (variable names matrix).

[15]:
```python
def mnaSolver(inp, dc, i):
    q = i
    M, x, b = matrix_init(inp, q, dc) #initializes the matrix to required size

    if not dc:
        k = inp.index('.end\n')
        k += 1 # index of the frequency
        freq = 0.0 # initialize freq to 0; if dc it stays 0, else we read ac␣
↪frequency from bottom of the netlist
        try:
            freqparams = inp[k].split()
            if len(freqparams) != 0:
                if freqparams[0] == '.ac':
                    freq = float(freqparams[2])
        except:
            pass
        w = 2 * np.pi * freq

    while inp[i] != ".end\n": # reading the netlist components one by one
        params = inp[i].split()
        component = params[0][0]
        nodeFrom = params[1]
        nodeTo = params[2]

        if component == "R": # stamping the matrix for resistors
            value = float(params[3])
            M[x.index(nodeFrom)][x.index(nodeFrom)] += 1/value
            M[x.index(nodeTo)][x.index(nodeTo)] += 1/value
            M[x.index(nodeFrom)][x.index(nodeTo)] += -1/value
            M[x.index(nodeTo)][x.index(nodeFrom)] += -1/value
        if component == "V": # stamping matrix for voltage sources
            dc = (params[3] == 'dc')
            nodeTo = params[1]
            nodeFrom = params[2]
            if dc:
                value = float(params[4])
            else:
                value = float(params[4])*cmath.exp(float(params[5])*1j) # value␣
↪of voltage source becomes magnitude*exp(j*phase angle)
            #iv1 flows from nodeFrom to nodeTo, it comes out of positive␣
↪terminal of voltage source
            M[x.index(nodeTo)][x.index('I'+ params[0])] = -1
            M[x.index(nodeFrom)][x.index('I'+ params[0])] = 1
            M[x.index('I'+ params[0])][x.index(nodeTo)] = 1
            M[x.index('I'+ params[0])][x.index(nodeFrom)] = -1
```

```
            b[x.index('I'+ params[0])] = value


        if component == "I":
            dc = (params[3] == 'dc')
            if dc:
                value = float(params[4])
            else:
                value = float(params[4])*cmath.exp(float(params[5])*1j) # value␣
↪of current source becomes magnitude*exp(j*phase angle)
            b[x.index(nodeFrom)] += -1*value
            b[x.index(nodeTo)] += value

        if component == "C" and not dc: # this mna solver can solve L and C␣
↪only for ac sources
            value = 1/(float(params[3]) * w * 1j) # capacitive impedance for␣
↪capacitor in AC circuit
            M[x.index(nodeFrom)][x.index(nodeFrom)] += 1/value
            M[x.index(nodeTo)][x.index(nodeTo)] += 1/value
            M[x.index(nodeFrom)][x.index(nodeTo)] += -1/value
            M[x.index(nodeTo)][x.index(nodeFrom)] += -1/value

        if component == "L" and not dc:
            value = (float(params[3]) * w * 1j) # inductive impedance for␣
↪inductor in AC circuit
            M[x.index(nodeFrom)][x.index(nodeFrom)] += 1/value
            M[x.index(nodeTo)][x.index(nodeTo)] += 1/value
            M[x.index(nodeFrom)][x.index(nodeTo)] += -1/value
            M[x.index(nodeTo)][x.index(nodeFrom)] += -1/value


        i += 1

    # replacing the redundant GND equation with VGND = 0V
    M[x.index('GND')] = np.zeros(M.shape[1])
    M[x.index('GND')][x.index('GND')] = 1
    b[x.index('GND')] = 0

    # solving for variables
    ans = gausselimcomplex(M, b)

    return ans, x
```

`mnaSolver` takes the netlist as input, goes through the components one by one and appropriately updates the MNA coefficient matrix. For AC circuits, the solver converts L and C values to impedances before updating.

We then solve the MNA equation using the Gaussian Elimination solver coded above.

DC only circuits (circuits 1, 3, 4, 5)

[16]:
```
f = open("ckt1.netlist", "r")
inp = f.readlines()
f.close()
singlefreq(inp) # checking if there is only single operating frequency for all␣
 ↪AC sources in the circuit
ik = int(0)
for ik in range(len(inp)):
    if inp[ik] == ".circuit\n":
        break
ik += 1
j = ik
k = ik
p = ik
while inp[ik] != ".end\n":
    ik += 1

in1 = matrix_prep(inp, True, j) # creating dc netlist
ans_dc, names1 = mnaSolver(in1, True, 0)
print("Values for DC analysis: ")
for x in range(len(names1)):
    print(names1[x] + "\t" + str(ans_dc[x].real))
```

```
Values for DC analysis:
GND     0.0
1       0.0
2       0.0
3       0.0
4       -5.0
IV1     0.0005
```

[17]:
```
f = open("ckt3.netlist", "r")
inp = f.readlines()
f.close()
_ = singlefreq(inp) # checking if there is only single operating frequency for␣
 ↪all AC sources in the circuit
ik = int(0)
for ik in range(len(inp)):
    if inp[ik] == ".circuit\n":
        break
ik += 1
j = ik
k = ik
p = ik
while inp[ik] != ".end\n":
    ik += 1
```

```
in1 = matrix_prep(inp, True, j) # creating dc netlist
ans_dc, names1 = mnaSolver(in1, True, 0)
print("Values for DC analysis: ")
for x in range(len(names1)):
    print(names1[x] + "\t" + str(ans_dc[x].real))
```

```
Values for DC analysis:
IV1     0.004970760233918129
GND     0.0
1       -10.0
2       -5.029239766081871
3       -2.5730994152046787
4       -1.403508771929825
5       -0.9356725146198834
```

[18]:
```
f = open("ckt4.netlist", "r")
inp = f.readlines()
f.close()
_ = singlefreq(inp) # checking if there is only single operating frequency for␣
 ↪all AC sources in the circuit
ik = int(0)
for ik in range(len(inp)):
    if inp[ik] == ".circuit\n":
        break
ik += 1
j = ik
k = ik
p = ik
while inp[ik] != ".end\n":
    ik += 1

in1 = matrix_prep(inp, True, j) # creating dc netlist
ans_dc, names1 = mnaSolver(in1, True, 0)
print("Values for DC analysis: ")
for x in range(len(names1)):
    print(names1[x] + "\t" + str(ans_dc[x].real))
```

```
Values for DC analysis:
IV1     2.2222222222222214
GND     0.0
1       -10.0
2       -5.555555555555557
3       -3.7037037037037037
```

[19]:
```
f = open("ckt5.netlist", "r")
inp = f.readlines()
f.close()
```

```
_ = singlefreq(inp) # checking if there is only single operating frequency for␣
↪all AC sources in the circuit
ik = int(0)
for ik in range(len(inp)):
    if inp[ik] == ".circuit\n":
        break
ik += 1
j = ik
k = ik
p = ik
while inp[ik] != ".end\n":
    ik += 1

in1 = matrix_prep(inp, True, j) # creating dc netlist
ans_dc, names1 = mnaSolver(in1, True, 0)
print("Values for DC analysis: ")
for x in range(len(names1)):
    print(names1[x] + "\t" + str(ans_dc[x].real))
```

```
Values for DC analysis:
GND     0.0
1       -10.0
IV1     1.0
```

AC sources (circuits 6 and 7)

[20]:
```
f = open("ckt6.netlist", "r")
inp = f.readlines()
f.close()
_ = singlefreq(inp) # checking if there is only single operating frequency for␣
↪all AC sources in the circuit
ik = int(0)
for ik in range(len(inp)):
    if inp[ik] == ".circuit\n":
        break
ik += 1
j = ik
k = ik
p = ik
while inp[ik] != ".end\n":
    ik += 1

in2 = matrix_prep(inp, False, k) # creating ac netlist
ans_ac, names2 = mnaSolver(in2, False, 0)
print("Values for AC analysis: ")
for y in range(len(names2)):
    print(names2[y] + "\t Magnitude:" + str('%0.5f' % abs(ans_ac[y])) + "\t␣
↪Phase: " + str(cmath.phase(ans_ac[y])))
```

```
Values for AC analysis:
IV1      Magnitude:0.00500       Phase: -6.124030364011088e-06
GND      Magnitude:0.00000       Phase: 0.0
n3       Magnitude:5.00000       Phase: -3.141592653589793
n1       Magnitude:0.00003       Phase: -1.5708024508252607
n2       Magnitude:0.00003       Phase: -1.5708024508252607
```

[21]:
```
f = open("ckt7.netlist", "r")
inp = f.readlines()
f.close()
_ = singlefreq(inp) # checking if there is only single operating frequency for␣
↪all AC sources in the circuit
ik = int(0)
for ik in range(len(inp)):
    if inp[ik] == ".circuit\n":
        break
ik += 1
j = ik
k = ik
p = ik
while inp[ik] != ".end\n":
    ik += 1

in2 = matrix_prep(inp, False, k) # creating ac netlist
ans_ac, names2 = mnaSolver(in2, False, 0)
print("Values for AC analysis: ")
for y in range(len(names2)):
    print(names2[y] + "\t Magnitude:" + str('%0.5f' % abs(ans_ac[y])) + "\t␣
↪Phase: " + str(cmath.phase(ans_ac[y])))
```

```
Values for AC analysis:
GND      Magnitude:0.00000       Phase: 0.0
n1       Magnitude:0.00082       Phase: -1.5707961635037402
```

AC DC superposition (circuit 2)

[22]:
```
f = open("ckt2.netlist", "r")
inp = f.readlines()
f.close()
_ = singlefreq(inp) # checking if there is only single operating frequency for␣
↪all AC sources in the circuit
ik = int(0)
for ik in range(len(inp)):
    if inp[ik] == ".circuit\n":
        break
```

To apply superposition, solve the circuit twice.

Once considering only DC sources:

[23]:
```
ik += 1
j = ik
k = ik
p = ik
while inp[ik] != ".end\n":
    ik += 1

in1 = matrix_prep(inp, True, j) # creating dc netlist
ans_dc, names1 = mnaSolver(in1, True, 0)
print("Values for DC analysis: ")
for x in range(len(names1)):
    print(names1[x] + "\t" + str(ans_dc[x].real))
```

```
Values for DC analysis:
1       5142.857142857143
GND     0.0
2       15428.57142857143
3       0.0
IV1     -3.8571428571428577
IV2     9.0
4       15423.57142857143
5       10423.57142857143
6       95423.57142857143
```

And once considering only all AC sources:

[24]:
```
in2 = matrix_prep(inp, False, k) # creating ac netlist
ans_ac, names2 = mnaSolver(in2, False, 0)
print("Values for AC analysis: ")
for i in range(len(names2)):
    print(names2[i] + "\t Magnitude:" + str('%0.5f' % abs(ans_ac[i])) + "\t␣
 ↪Phase: " + str(cmath.phase(ans_ac[i])))
```

```
Values for AC analysis:
1        Magnitude:0.28571       Phase: 0.0
GND      Magnitude:0.00000       Phase: 0.0
2        Magnitude:0.85714       Phase: 0.0
3        Magnitude:2.00000       Phase: 0.0
IV1      Magnitude:0.00029       Phase: -0.0
IV2      Magnitude:0.00000       Phase: -0.0
4        Magnitude:0.85714       Phase: 0.0
5        Magnitude:0.85714       Phase: 0.0
6        Magnitude:0.85714       Phase: 0.0
```

For the final answer, we superpose the DC values with the AC values, but multiply all the AC values with $e^{j\omega t}$ before adding them.

This solver can solve for circuits with AC and DC sources together, provided all AC sources have the same operating frequency.