Assignment2_EE21B145

Sumeeth C Muchandimath EE21B145 <ee21b145@smail.iitm.ac.in>

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

[11]:
```python
import numpy as np
def fact(n):
    if n == 1:
        return 1
    else:
        return n*fact(n-1)
%timeit fact(100)
%timeit np.math.factorial(100)
```

```
11.5 µs ± 270 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
736 ns ± 19.5 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

2.0.1 Explanation

Here we have used a very simple recursive function to compute factorial.

The time taken is found out by the `%timeit` function. It shows around **11.5**μs for the user defined function, while the in-built function takes **0.736**μs, which is more than 15 times faster.

3 Gaussian Elimination

3.1 Aim

We have been instructed to solve a generic `NxN` system of equations using Gaussian Elimination and compute the time taken for a 10x10 system.

We assume that we recieve an augmented matrix in the variable `mat`. We now code the solver to solve this.

[12]:
```
from pprint import pprint
import numpy as np
N = 10

#start of the gaussian elimination
def gaussElim(mat):
    singular_flag = Elim(mat)  #flag to check if matrix is singular
    if singular_flag!=-1:
        print("It's a Singular Matrix")
        if(mat[singular_flag][N]):  #if diagonal element has non-zero element␣
 ↪and is singular, it has no solution
            print("System is not consistent")
        else:
            print("It has infinitely many solutions")

        return

    return(back_sub(mat))
```

The above block sends the matrix for forward elimination and then checks for errors which may arise, such as having a singular matrix.

If it is a singular matrix, we check if the value at the index returned is `0`, because if so, it may have infinte solutions, else no solution.

[13]:
```
#function to swap two rows of a given matrix
def swap_row(mat,i,j):
    for k in range(N+1):
        temp=mat[i][k]
        mat[i][k]=mat[j][k]
        mat[j][k]=temp
```

This block is a function to swap two rows of a matrix, which will be used in pivoting.

[14]:
```
#function to forward eliminate the matric to a row reduced echelon form
def Elim(mat):
    for k in range(N):
        im=k
        vm=mat[im][k]
        for i in range(k+1,N):
            if(abs(mat[i][k])>vm):  #comparing pivotal values to find the␣
 ↪highest one
                vm=mat[i][k]
                im=i
        if not mat[k][im]:  #checekimg if we get 0
            return k
        if im!=k:
            swap_row(mat,k,im)
        for i in range(k+1,N):
            f=mat[i][k]/mat[k][k]  #getting the factor to divide the below row␣
 ↪with
            for j in range(k+1,N+1):
                mat[i][j]-=mat[k][j]*f
            mat[i][k]=0  #converting lower triangle elements to 0
    return -1
```

This block first does **partial pivoting**, and then swaps rows accordingly, before proceeding with forward elimination to result in a triangular matrix.

[15]:
```
#simple function for back substitution
def back_sub(mat):
    x=[0 for element in range(N)]
    for i in range (N-1,-1,-1):
        x[i]=mat[i][N]
        for j in range(i+1,N):
            x[i]-=mat[i][j]*x[j]
        x[i]=(x[i]/mat[i][i])
    return(x)
```

This is the Back-Substitution function which returns the answer matrix to the `gaussElim` function, which in turn returns it to where it was first called.

[16]:
```
mat = [[1 ,3 ,1 ,2 ,6 ,6 ,0 ,1 ,3 ,5 ,2],
[7 ,0 ,2 ,0 ,5 ,5 ,6 ,3 ,3 ,3 ,3],
[6 ,0 ,6 ,0 ,0 ,8 ,4 ,5 ,3 ,7 ,4],
[8 ,5 ,4 ,9 ,3 ,5 ,3 ,5 ,8 ,7 ,1],
[7 ,6 ,3 ,8 ,9 ,2 ,3 ,8 ,7 ,8 ,2],
[9 ,5 ,7 ,0 ,7 ,7 ,0 ,1 ,8 ,6 ,2],
[3 ,9 ,7 ,9 ,2 ,1 ,7 ,6 ,7 ,1 ,2],
[8 ,5 ,6 ,4 ,4 ,0 ,3 ,7 ,2 ,5 ,9],
[1 ,2 ,7 ,6 ,1 ,5 ,2 ,0 ,8 ,1 ,7],
    [6 ,4 ,4 ,3 ,6 ,2 ,7 ,8 ,5 ,2 ,5]]  #example matrix

A= np.array([[1 ,3 ,1 ,2 ,6 ,6 ,0 ,1 ,3 ,5 ],
[7 ,0 ,2 ,0 ,5 ,5 ,6 ,3 ,3 ,3 ],
[6 ,0 ,6 ,0 ,0 ,8 ,4 ,5 ,3 ,7 ],
[8 ,5 ,4 ,9 ,3 ,5 ,3 ,5 ,8 ,7 ],
[7 ,6 ,3 ,8 ,9 ,2 ,3 ,8 ,7 ,8 ],
[9 ,5 ,7 ,0 ,7 ,7 ,0 ,1 ,8 ,6 ],
[3 ,9 ,7 ,9 ,2 ,1 ,7 ,6 ,7 ,1 ],
[8 ,5 ,6 ,4 ,4 ,0 ,3 ,7 ,2 ,5 ],
[1 ,2 ,7 ,6 ,1 ,5 ,2 ,0 ,8 ,1 ],
[6 ,4 ,4 ,3 ,6 ,2 ,7 ,8 ,5 ,2 ]] )
b= np.array( [[2],
[3],
[4],
[1],
[2],
[2],
[2],
[9],
[7],
[5]])
ans=gaussElim(mat)
print(np.linalg.solve(A,b))
pprint(ans)
```

```
[[ 0.57592865]
 [-1.1434718 ]
 [ 1.700912  ]
 [ 1.56273285]
 [ 1.1733649 ]
 [ 1.36712171]
 [-1.35775437]
 [ 1.04556496]
 [-1.97475077]
 [-2.06722465]]
[0.575928649457065,
 -1.1434717980768436,
 1.7009120009075778,
 1.5627328459907674,
 1.1733649044776124,
 1.367121709432954,
 -1.3577543680410384,
 1.045564958058256,
 -1.9747507747613344,
 -2.0672246530706593]
```

Here we have initialised a random 10x11 matrix, to feed into the solver.

We have cross verified the result with the `np.linalg.solve()` function, so the solver works properly.

But how long does it take?

[17]:
```
%timeit gaussElim(mat)
%timeit np.linalg.solve(A, b)
```

```
58.4 µs ± 1.94 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
18 µs ± 526 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

After using the `%timeit` function, we see that our solver is more than **3 times** slower than `np.linalg.solve()`.

4 SPICE Simulator

4.1 Aim

To read a `.netlist` file and extract information of the components and their respective nodes as well as their values.

Then use our solver from the previous problem to solve for nodal voltages and auxilliary currents.

We start off by defining our Gaussian Elimination solver from the previous problem but not defining `N`, as it will change based on different `.netlist` files.

[50]:
```
import numpy as np
import sys
from pprint import pprint

#we have used the same function as the previous Program
def gaussElim(mat):
    singular_flag = Elim(mat)
    if singular_flag!=-1:
        print("It's a Singular Matrix")
        if(mat[singular_flag][N]):
            print("System is not consistent")
        else:
            print("It has infinitely many solutions")
        return
    return(back_sub(mat))
```

```
def swap_row(mat,i,j):
    for k in range(N+1):
        temp=mat[i][k]
        mat[i][k]=mat[j][k]
        mat[j][k]=temp

def Elim(mat):
    for k in range(N):
        im=k
        vm=abs(mat[im][k])
        for i in range(k+1,N):
            if(abs(mat[i][k])>vm):
                vm=abs(mat[i][k])
                im=i
        if not mat[k][im]:
            return k
        if im!=k:
            swap_row(mat,k,im)
        for i in range(k+1,N):
            f=mat[i][k]/mat[k][k]
            for j in range(k+1,N+1):
                mat[i][j]-=mat[k][j]*f
            mat[i][k]=0
    return -1

def back_sub(mat):
    x=[0 for element in range(N)]
    for i in range (N-1,-1,-1):
        x[i]=mat[i][N]
        for j in range(i+1,N):
            x[i]-=mat[i][j]*x[j]
        x[i]=(x[i]/mat[i][i])
    return(x)
```

Now we start by defining classes for all passive components and independent sources.

It is much easier to handle attributes related to each components, such as their nodes and value.

Classes also offer modularity as I can easily define another attribute, such as tolerance for resistors, series resistance for voltage sources, and use them for other computations.

We then define lists to hold objects of each components along with counter variables, as I faced some complexity using `len(R)` in some places, so I used `r` instead.

[51]:
```
# Here we start with defining classes for all the components as it will be easy␣
↪to associate nodes and values this way

class resistor:
    def __init__(self,node_ini,node_fin,value):
```

```python
        self.node_ini=node_ini
        self.node_fin=node_fin
        self.value=value
class inductor:
    def __init__(self,node_ini,node_fin,value):
        self.node_ini=node_ini
        self.node_fin=node_fin
        self.value=value
class capacitor:
    def __init__(self,node_ini,node_fin,value):
        self.node_ini=node_ini
        self.node_fin=node_fin
        self.value=value
class voltage_source:
    def __init__(self,node_ini,node_fin,value):
        self.node_ini=node_ini
        self.node_fin=node_fin
        self.value=value
class current_source:
    def __init__(self,node_ini,node_fin,value):
        self.node_ini=node_ini
        self.node_fin=node_fin
        self.value=value
class current_source_ac:
    def __init__(self,node_ini,node_fin,value):
        self.node_ini=node_ini
        self.node_fin=node_fin
        self.value=value
class voltage_source_ac:
    def __init__(self,node_ini,node_fin,value):
        self.node_ini=node_ini
        self.node_fin=node_fin
        self.value=value

# Have initialised lists to hold objects of the components
R=[0 for element in range(20)]
C=[0 for element in range(20)]
L=[0 for element in range(20)]
V=[0 for element in range(20)]
I=[0 for element in range(20)]
V_ac=[0 for element in range(20)]
I_ac=[0 for element in range(20)]

#counter variables to count number of components and nodes
r=c=w=qac=qdc=vac=vdc=dc=ac=num_node=f=0
j=0
```

The below block of code will go through the file line by line and read the information we require.

The `j` counter is used to make sure we only read values in between `.circuit` and `.end`.

While adding an object to their respective list, we also change node `GND` to `0` and replace nodes such as `n3` to `3`.

The `num_node` variable is used to count the number of nodes present in the circuit, as it is required for computation later.

We can check for different `.netlist` files by changing the path in `line 2`.

[52]:
```
#a block to read the file
file1 = open('ckt6.netlist','r') #We can check for other test cases by editing␣
↪the path over here
lines = file1.readlines()
for l in lines:

    #To find the lines between the start and end
    if l.strip() == ".end" or l.strip() == ".circuit":
        j+=1
    if j ==1:
        if l[0] == 'R':
            #this changes the value of the list to an object of the component␣
↪with the attributes and simultanouesly checks for n1 GND and changes them to␣
↪integers
            R[r]=(resistor(l.split()[1].replace('n','').replace('GND','0'),l.
↪split()[2].replace('n','').replace('GND','0'),float(l.split()[3])))
            num_node=max(int(R[r].node_ini),num_node,int(R[r].node_fin))
            r+=1
        elif l[0]=='V':
            if str(l.split()[3])=="ac":  #to check for ac source
                V_ac[vac]=(voltage_source_ac(l.split()[1].replace('n','').
↪replace('GND','0'),l.split()[2].replace('n','').replace('GND','0'),float(l.
↪split()[4])))
                num_node=max(int(V_ac[vac].node_ini),num_node,int(V_ac[vac].
↪node_fin))
                ac+=1
                vac+=1
            elif str(l.split()[3]) == "dc":
                V[vdc]=(voltage_source(l.split()[1].replace('n','').
↪replace('GND','0'),l.split()[2].replace('n','').replace('GND','0'),float(l.
↪split()[4])))
                num_node=max(int(V[vdc].node_ini),num_node,int(V[vdc].node_fin))
                dc+=1
                vdc+=1
        elif l[0]=='I':
            if str(l.split()[3])=="dc":
                I[qdc]=(current_source(l.split()[1].replace('n','').
↪replace('GND','0'),l.split()[2].replace('n','').replace('GND','0'),float(l.
↪split()[4])))
                num_node=max(int(I[qdc].node_ini),num_node,int(I[qdc].node_fin))
                dc+=1
                qdc+=11
            elif str(l.split()[3])=="ac":
                I_ac[qac]=(current_source_ac(l.split()[1].replace('n','').
↪replace('GND','0'),l.split()[2].replace('n','').replace('GND','0'),float(l.
↪split()[4])))
                num_node=max(int(I_ac[qac].node_ini),num_node,int(I_ac[qac].
↪node_fin))
                ac+=1
                qac+=1
        elif l[0]=='L':
            L[w]=(inductor(l.split()[1].replace('n','').replace('GND','0'),l.
↪split()[2].replace('n','').replace('GND','0'),float(l.split()[3])))
            num_node=max(int(L[w].node_ini),num_node,int(L[w].node_fin))
            w+=1
        elif l[0]=='C':
            C[c]=(capacitor(l.split()[1].replace('n','').replace('GND','0'),l.
↪split()[2].replace('n','').replace('GND','0'),float(l.split()[3])))
            num_node=max(int(C[c].node_ini),num_node,int(C[c].node_fin))
            c+=1
    if j==2:  #breaks after ".end" is read
        break
for l in lines:  #loop to find frequency of the AC source
    if l[0:3]=='.ac':
        omega=2*np.pi*float(l.split()[2])
        f+=1
file1.close()
```

This block of code is used to check for errors such as using an AC and DC source together or using two AC sources with different frequencies.

[53]:
```
#to check for error states
if (ac*dc!=0):
    print("Not possible to solve with AC and DC sources in one circuit")
    sys.exit()
elif f>1:
    print("Not possible to solve circuit with two different AC frequencies")
    sys.exit()
```

The Modified Nodal Analysis Matrix was generated by using the method specified in this website, Modified Nodal Analysis , which proved to be easy to implement with a few `for` loops and conditional statements.

The following few blocks initialise the `A` Matrix part of the augmented matrix.

The A matrix

The A matrix will be developed as the combination of 4 smaller matrices, G, B, C, and D.

$$A = \begin{bmatrix} G & B \\ C & D \end{bmatrix} \tag{3.3}$$

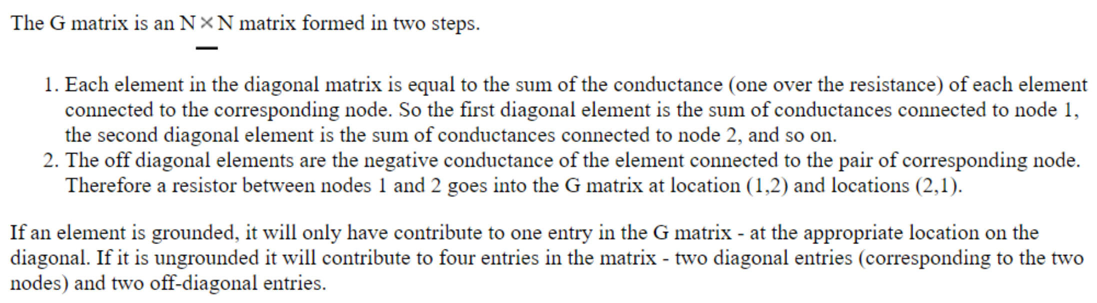
The A matrix is (M+N) × (M+N) (N is the number of nodes, and M is the number of independent voltage sources) and:

- the G matrix is N × N and is determined by the interconnections between the circuit elements
- the B matrix is N × M and is determined by the connection of the voltage sources
- the C matrix is M × N and is determined by the connection of the voltage sources (B and C are closely related, particularly when only independent sources are considered)
- the D matrix is M × M and is zero if only independent sources are considered

[56]:
```
#the size of matrix would be n+m where n is number of nodes and m is number of␣
↪independent voltage sources
matA=[[0 for i in range(num_node+vac+vdc+1)]for element in␣
↪range(num_node+vac+vdc)]

#here we are first filling in the values of an nxn matrix within the larger␣
↪matrix, with conductance values
for i in range (num_node):
    for j in range(num_node):
        if i==j:
            for n in range(r):
                if int(R[n].node_ini)==i+1 or int(R[n].node_fin)==i+1:
                    matA[i][i]+=1/(float(R[n].value))
            if w!=0:
                for n in range(w):
                    if int(L[n].node_ini)==i+1 or int(R[n].node_fin)==i+1:
                        matA[i][i]+=1/(1j*omega*L[n].value)
            if c!=0:
                for n in range(c):
                    if int(C[n].node_ini)==i+1 or int(C[n].node_fin)==i+1:
                        matA[i][i]+=1j*omega*C[n].value



        else:
            for n in range(r):
                if((int(R[n].node_ini)==i+1 and int(R[n].node_fin)==j+1) or␣
↪((int(R[n].node_ini)==j+1 and int(R[n].node_fin)==i+1))):
                    matA[i][j]-=1/(float(R[n].value))
            if w!=0:
                for n in range(w):
                    if((int(L[n].node_ini)==i+1 and int(L[n].node_fin)==j+1) or␣
↪((int(L[n].node_ini)==j+1 and int(L[n].node_fin)==i+1))):
                        matA[i][j]-=1/(1j*omega*L[n].value)
            if c!=0:
                for n in range(c):
                    if((int(C[n].node_ini)==i+1 and int(C[n].node_fin)==j+1) or␣
↪((int(C[n].node_ini)==j+1 and int(C[n].node_fin)==i+1))):
                        matA[i][j]-=1j*omega*C[n].value
```

The above block of code adds the `G` matrix part of the `A` matrix.

Rules for making the G matrix


The G matrix is an N × N matrix formed in two steps.

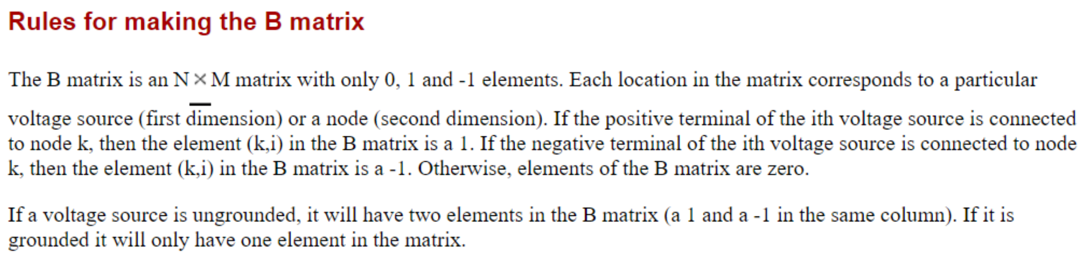
1. Each element in the diagonal matrix is equal to the sum of the conductance (one over the resistance) of each element connected to the corresponding node. So the first diagonal element is the sum of conductances connected to node 1, the second diagonal element is the sum of conductances connected to node 2, and so on.
2. The off diagonal elements are the negative conductance of the element connected to the pair of corresponding node. Therefore a resistor between nodes 1 and 2 goes into the G matrix at location (1,2) and locations (2,1).

If an element is grounded, it will only have contribute to one entry in the G matrix - at the appropriate location on the diagonal. If it is ungrounded it will contribute to four entries in the matrix - two diagonal entries (corresponding to the two nodes) and two off-diagonal entries.

[57]:
```
for i in range(num_node):
    if vac!=0:
        for j in range(vac):
            if int(V_ac[j].node_ini)==i+1:
                matA[i][j+num_node]=1
                matA[j+num_node][i]=1
            elif int(V_ac[j].node_fin)==i+1:
                matA[i][j+num_node]=-1
                matA[j+num_node][i]=-1
    if vdc!=0:
        for j in range(vdc):
            if int(V[j].node_ini)==i+1:
                matA[i][j+num_node]=1
                matA[j+num_node][i]=1

            elif int(V[j].node_fin)==i+1:
                matA[i][j+num_node]=-1
                matA[j+num_node][i]=-1
```

This above block implements the `B` and `C` part of the matrix, it is easy to implement as they both are transposes of each other.

We also check for capacitors and inductors, not requiring other



Rules for making the B matrix

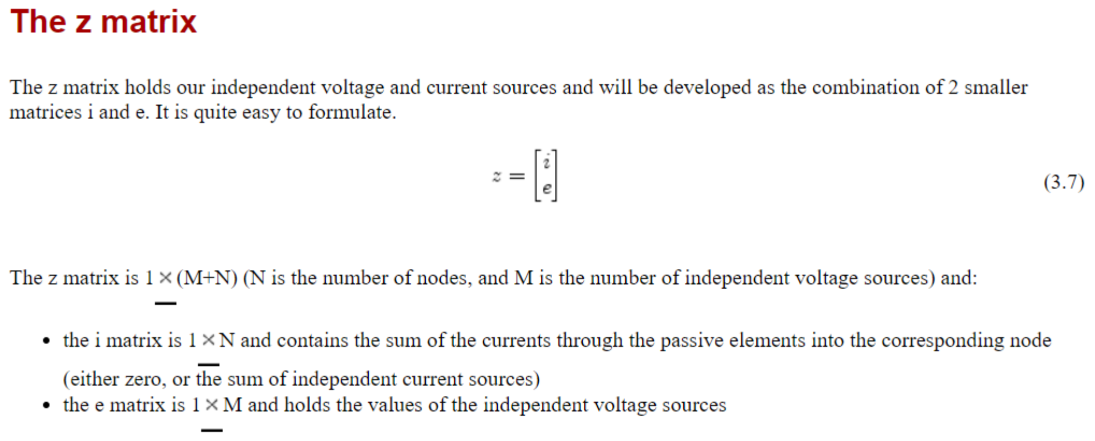
The B matrix is an N × M matrix with only 0, 1 and -1 elements. Each location in the matrix corresponds to a particular voltage source (first dimension) or a node (second dimension). If the positive terminal of the ith voltage source is connected to node k, then the element (k,i) in the B matrix is a 1. If the negative terminal of the ith voltage source is connected to node k, then the element (k,i) in the B matrix is a -1. Otherwise, elements of the B matrix are zero.

If a voltage source is ungrounded, it will have two elements in the B matrix (a 1 and a -1 in the same column). If it is grounded it will only have one element in the matrix.

code.

[58]:
```
#Here we are adding in the values of the last column in the augmented matrix
for i in range(num_node):
    if qac!=0:
        for j in range(qac):
            if i+1 == int(I_ac[j].node_fin):
                matA[i][num_node+vac+vdc]+=float(I_ac[j].value)
    if qdc!=0:
        for j in range(qdc):
            if i+1 == int(I[j].node_fin):
                matA[i][num_node+vac+vdc]+=float(I[j].value)
if ac!=0:
    for i in range(vac):
        matA[i+num_node][num_node+vac+vdc]+=float(V_ac[i].value)
if dc!=0:
    for i in range(vdc):
        matA[num_node+i][num_node+vac+vdc]+=float(V[i].value)
```

This block adds values to the last column of the augmented matrix, which is basically current sources' and voltage sources' values.



The z matrix

The z matrix holds our independent voltage and current sources and will be developed as the combination of 2 smaller matrices i and e. It is quite easy to formulate.

$$z = \begin{bmatrix} i \\ e \end{bmatrix} \tag{3.7}$$

The z matrix is 1 × (M+N) (N is the number of nodes, and M is the number of independent voltage sources) and:

- the i matrix is 1 × N and contains the sum of the currents through the passive elements into the corresponding node (either zero, or the sum of independent current sources)
- the e matrix is 1 × M and holds the values of the independent voltage sources

Now we define `N` for the `gaussElim` function to use.

We then use `for` loops to print the values in presentable form.

[59]:
```
N=num_node+vac+vdc #defining N for the gaussElim function
ans=gaussElim(matA)
for i in range(num_node):
```

```python
    print("V",i+1,"=",ans[i],"V")
for i in range(vac+vdc):
    print("I",i+1,"=",ans[i+num_node],"A")
```

```
V 1 = (-1.9239208801457062e-10-3.141592653471963e-05j) V
V 2 = (-1.8751873949430632e-10-3.062015181929e-05j) V
V 3 = (-5-0j) V
I 1 = (-0.004999999999812482+795.7747154900969j) A
```

After simulating the said `netlist` file on LTSPice, we cross verified the values, and they are correct.

5 Conclusion

We first made a `NxN` solver using Gaussian Elimination. It can be optimised further to make it faster.

We then made a simple `.netlist` solver which can also solve for AC sources. It is able to catch some errors that may have been made in the generation of the `.netlist` file and is able to fix it.

But our solver depends on the speed of our Gaussian Elimination Solver, so we can make it faser by just using the `np.linalg.solve()` function.