Assignment 2 - SPICE Circuit Solver

Nishanth Senthil Kumar
Roll No: EE23B049

4th August 2024

Abstract

This report presents an algorithmic approach to solving electrical circuits containing resistors and independent voltage and current sources using nodal analysis. The methodology involves parsing a SPICE netlist, constructing the necessary matrices for nodal analysis, and solving the resulting system of linear equations. Error handling mechanisms are integrated to ensure the solver can gracefully handle malformed inputs or unsolvable circuits.

1 Introduction

This document describes an algorithmic method for generating Modified Nodal Analysis (MNA) equations for circuits containing resistors and independent voltage/current sources. MNA offers a systematic approach to circuit analysis by representing circuit elements and node voltages using matrix equations. Prof. Vinita Vasudevan (my EE2015 professor) had explained modified nodal analysis and this assignment was a attempt to implement it. The algorithm implemented is heavily inspired by a paper written by a member of the Swarthmore College, referenced in the last section.

The core goal is to solve for node voltages and currents through voltage sources, while efficiently handling error conditions. The problem is broken into several key components:

- File input validation and error handling.
- Parsing the circuit description into a graph representation.
- Matrix construction for the MNA method.
- Solving the system of linear equations to find the node voltages and currents.

2 Algorithm Overview

The input to this algorithm is a SPICE-like circuit description, which includes nodes, resistors, independent voltage, and current sources. The algorithm uses the following steps:

1. **Input validation:** Check for valid file name, correct format with `.circuit` and `.end`, and ensure only allowed components (R, V, I) are present.
2. **Parsing:** Convert the circuit description into an adjacency list that represents the graph of nodes and components.
3. **Matrix Construction:** Form the A matrix (comprising the G, B, C, and D matrices) and the Z matrix (comprising current and voltage sources).

4. **Solving:** Use linear algebra to solve the system $Ax = Z$ to find the node voltages and current through voltage sources.

3 Assumptions

These are some assumptions that I have taken while writing this code

- I have assumed that a 'GND' (reference node) will be given in the input, I will raise error otherwise.
- I have assumed duplicate elements are not allowed, I will throw error otherwise.
- I have assumed that if multiple '.circuit' and '.end' are there in the text file, the code will parse from the first '.circuit' to the first '.end', skipping any '.circuit' in the middle, If multiple '.circuit' instances are there, This code will not throw up an error.
- When a Voltage source is given, i have taken the first node to be positive and the second node to be the negative terminal of the voltage source. If a current source is given, I take the current to flow from the first node to the second.
- I have also assumed that the input values wont overflow a float memory type.
- I will also not throw an error if the Voltage or current source is AC, as we could say that the given voltage is the RMS voltage, then the RMS currents and voltages at sources and nodes would be as if a DC voltage of that RMS value is given as input, as there are no reactive elements.
- I have raised ValueErrors with the same messages for multiple different faults, because if i raise custom messages, it might fail the auto-correction while evaluvating.

4 File Input Validation and Error Handling

Before proceeding to circuit analysis, the input file is checked for basic validity. This ensures the presence of the keywords `.circuit` and `.end`, delimiting the circuit description. Furthermore, all components are validated to ensure they belong to the allowed set: resistors, independent current, and voltage sources.

Any malformed file or incorrect components result in immediate error handling. Examples of potential errors:

- Missing `.circuit` or `.end` markers.
- Presence of non-allowed components.
- Incorrect number of arguments for components (e.g., resistors should have 4, sources should have 5).
- Checks if the value of the argument given for the value of the resistance, voltage or current source is of int or float type.
- Also checks if a reference node 'GND' is present as one of the inputs.

The algorithm terminates early if these errors are encountered.

I have taken some assumptions, one of these include that the node 'GND' has to be present as one of the nodes, otherwise we would not have a reference voltage and we would not have a unique answer. The way 'GND' is written is not case sensitive.

5 Parsing

The parsing is done only after a couple of checks, to see if the filename is valid, and if '.circuit' and '.end' are present in the text input file. If either of these are not present, it will throw an appropriate error. These checks are done in separate functions.

If multiple '.circuit' and '.end' are present in the file, i have made it such that the code will parse the data from the first instance of the '.circuit' to the first instance of '.end'.

- The parsing happens line by line, the line is split on the based on white spaces, and all the elements which are empty are deleted, to remove the unnecessary white spaces.
- After the trimming of the white spaces in every line, the first element of the first item is checked, if it does not belong in "RIV", an error is thrown, as we are only working with resistors, voltage and current sources.
- After these checks, the list is trimmed to the first 4 elements of the list if it is a resistor, and first 5 elements otherwise.
- Then the last element of the list is checked if it is of type int or float, and a error is thrown otherwise.
- After this, the data is processed and stored in the form of an adjacency list, a graph.
- I have made the adjacency list as a dictionary with each value corresponding to a key being a list of lists. The keys are the nodes, and the value consists of a list containing three elements, the name of the circuit element, the node it is connected to and the value of the voltage,current or resistance of the circuit element. If the element is a voltage source, and if it is connected with opposite polarity, and extra string 'True' is appended to the list, which will be used later to update the sign of currents.
- Another assumption is that in the case of voltage sources, the first node in the arguments is connected to the positive terminal and the other two to the negative terminal. So in the list that is going to be appended to the second node, the voltage value is multiplied by a factor of -1 to signify that it is connected to the negative terminal.
- Similarly, in the current sources, I have assumed that the current moves from the first node to the second, and the signs have been updated on the basis of that.
- The graph is bi-directed graph, so the list with details is added to both the nodes to which the circuit element is connected between, the only difference in the list is the node to which it is connected to for a particular key.
- The parsing function also counts the number of voltage sources.
- After obtaining the useful information, the 'GND' node is deleted as it is not required and deleting it would simplify some data processing.
- Duplicate circuit elements are not allowed, and I have checked for this by converting a list of circuit elements into a set and comparing the size of each, if they are not equal, there are duplicate elements.

5.1 An example of the node graph

This is an example of how the relations of the nodes are stored in my code, main.py is my code, and I made it print out the node graph for the given input.

This is the given input :

```
.circuit
Vsource n1 GND dc 10
Isource n3 GND dc 1
R1 n1 n2 2
R2 n2 n3 5
R3 n2 GND 3
.end
```

The output when the code 'main.py' is run :

```
> python3 main.py

{'GND': [['Vsource', 'n1', -10.0],
['Isource', 'n3', 1.0], ['R3', 'n2', 3.0]],
'n1': [['Vsource', 'GND', 10.0], ['R1', 'n2', 2.0]],
'n2': [['R1', 'n1', 2.0], ['R2', 'n3', 5.0], ['R3', 'GND', 3.0]],
'n3': [['Isource', 'GND', -1.0], ['R2', 'n2', 5.0]]}
```

A case where the polarity is switched :

```
.circuit
R1 GND 1 10
V1 GND 1 dc 10
.end
```

Node Graph :

```
> python3 main.py

 {'1': [['R1', 'GND', 10.0], ['V1', 'GND', -10.0, 'True']],
 'GND': [['R1', '1', 10.0], ['V1', '1', 10.0]]}
```

Observe how V1 has 'True'(str) as its last element, indicating the reverse polarity.

6 Generating the MNA Matrices

The next step after parsing all the useful data from the input is to generate the nodal equation matrix and solve it, this can be done algorithmically using the Modified nodal analysis. This part of the code is inspired by a paper published by Swarthmore College.

6.1 Mapping

- Since the nodes can have any name it wants, it is important to have some common indexing to refer to each node in the code, hence I have constructed a 2 way mapping of the node name and a index from 0 to n-1, where n is the number of nodes.
- These mappings are stored in a dictionary,
- Example :
  I have made my code to print only the mapping dictionaries (node_mapping_list and node_back_mapping_list)

  For the input:

```
.circuit
Vsource n1 GND dc 10
Isource n3 GND dc 1
R1 n1 n2 2
R2 n2 n3 5
R3 n2 GND 3
.end
```

Output :

```
> python3 main.py

node_mapping_list : {0: 'n1', 1: 'n2', 2: 'n3'}
node_back_mapping_list : {'n1': 0, 'n2': 1, 'n3': 2}
```

These indexing are used throughout the code to reference the node with a common indexing system.

6.2 Generating the Matrices used in MNA

6.2.1 The A Matrix

The A matrix is the key structure in MNA and is constructed from four sub-matrices: G, B, C, and D.

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

The sizes of these matrices depend on the number of nodes n (excluding ground) and the number of independent voltage sources m. The dimensions of A are $(n+m) \times (n+m)$.

6.2.2 The G Matrix

The G matrix is an $n \times n$ conductance matrix. For each node, the diagonal entries represent the sum of conductances connected to the node. Off-diagonal terms represent negative conductances between two connected nodes.

$$G_{ii} = \sum \frac{1}{R_{ij}}, \quad G_{ij} = -\frac{1}{R_{ij}} \quad \text{(if nodes } i \text{ and } j \text{ are connected)}$$

6.2.3 The B Matrix

The B matrix is an $n \times m$ matrix that relates nodes to independent voltage sources. Its entries are either 1, -1, or 0, depending on the polarity of the voltage sources.

$$B_{ij} = \begin{cases} 1, & \text{if positive terminal of } V_j \text{ is at node } i, \\ -1, & \text{if negative terminal of } V_j \text{ is at node } i, \\ 0, & \text{otherwise.} \end{cases}$$

6.2.4 The C Matrix

The C matrix is simply the transpose of the B matrix(only when there are only independent sources:

$$C = B^T$$

6.2.5 The D Matrix

The D matrix is an $m \times m$ zero matrix in this case since there are no dependent sources.

6.2.6 The Z Matrix

The Z matrix consists of known quantities:

$$Z = \begin{bmatrix} I \\ E \end{bmatrix}$$

The I vector contains the sum of currents flowing into each node due to independent current sources, while the E vector contains the values of the independent voltage sources.

6.2.7 The X Matrix

The X matrix is the unknown vector that we solve for:

$$X = \begin{bmatrix} V \\ I_V \end{bmatrix}$$

where V is the vector of node voltages, and I_V is the vector of currents through the independent voltage sources.

These matrices are stored in NumPy arrays, which make it much faster than when we use normal lists.

The reason why NumPy arrays are much faster is because NumPy arrays are densely packed arrays of homogeneous type. Python lists, by contrast, are arrays of pointers to objects, even when all of them are of the same type. So, you get the benefits of locality of reference.

Also, many NumPy operations are implemented in C, avoiding the general cost of loops in Python, pointer indirection and per-element dynamic type checking.

After generating all these matrices, they can be clubbed together to form the A and Z matrices, which can be used to solve for the unknown variables.

7 Solving the System

Once the matrices A and Z are constructed, the solution to the circuit is given by solving the linear equation:

$$X = A^{-1}Z$$

This yields both the node voltages and the currents through the voltage sources.

The solving for the unknown variables is done using LinAlg solve from the NumPy library.

However, when the values of current through unknown sources or nodal voltages are very small, of the order of 10e-7 or less, the floating point errors are big enough for the answer to change quite significantly, and sometimes even the sign might flip (although the magnitude of the error might be very small, less than 10e-5).

Such errors are not acceptable, and one quick way to solve this by the user is to scale all the circuit elements to a range such that the answers come to be a fairly large value, and the scale it back down.

If I had a little more free time, I would have tried to scale the inputs myself in the code when the answers became too small, and scale it back down, or throw an error when the answer is less than a certain threshold where floating point errors become significant, and instruct the user to do the same.

Another alternative is to use some other methods of solving the equations, like some linalg solve functions provided by SciPy, which handle floating points in a much better way.

8 Error Handling

Several checks are incorporated to ensure valid circuit configurations while generating the matrices:

- **Zero resistance:** If a resistor with zero resistance is found, an error is raised.
- **Ill-conditioned matrix:** If the A matrix is singular (i.e., it has a rank deficiency), the circuit is unsolvable, and an error is raised.
  Instead of checking if the determinant is 0, which might not always work due to floating point errors, the rank of the matrix is found, which is more reliable than finding the determinant, which provides a more reliable way of checking if the matrix is singular.

9 Possible Edge Cases

Under the extra folder, I have added some extra test cases, which I will explain here.

- Modelling shorting: Shorting is effectively connecting a resistance less wire between two points, however as you would have realised, my code would throw an error if you tried to put a wire of 0 resistance between two points. Instead, to model shorting, one can effectively connect a voltage source between the two nodes with 0 potential difference. My code would support this and this can be used to short two ends.

```
#shorting.ckt
.circuit
v1 1 GND DC 2
R1 1 2 3
R2 2 GND 4
V3 2 GND dc 0
.end
```

  Expected Output:

```
({'1': 2.0, '2': 0.0, 'GND': 0.0}, {'v1': -0.6666666666666666, 'V3':
↪  0.6666666666666666})
```

- A test case where a non float value is given as value of a circuit element.

```
#invalid_value.ckt
.circuit
R1 GND 1 10
```

```
V1 GND 1 dc J
.end
```

Expected Output : ValueError, with appropriate message

- A test case with duplicate circuit elements

```
#duplicate.ckt
.circuit
v1 1 GND DC 2
R1 1 2 3
R1 2 GND 4
V3 2 GND dc 0
.end
```

Expected Output : ValueError, with appropriate message

- A test case with no GND

```
#no_gnd.ckt
.circuit
v1 1 4 dc 24
v2 3 4 dc 15
r1 1 2 1000
r2 2 3 8100
r3 2 4 4700
.end
```

Expected Output : ValueError, with appropriate message

- A testcase with multiple current and voltage sources (has a supernode)

```
#supernode.ckt
.circuit
Vsource1 2 3 dc 22
I1 1 2 dc 3
I2 1 GND DC 8
I3 GND 3 DC 25
R1 1 3 4
R2 1 2 3
R3 2 GND 1
R4 3 GND 5
.end
```

Expected Output :

```
({'1': 1.0714285714285712, '2': 10.5, '3': 32.5, 'GND': 0.0},
↪  {'Vsource1': 10.642857142857142})
```

- A testcase with a 0 ohm resistor

```
#zero_resistance.ckt
.circuit
R1 1 GND 0
V1 1 GND DC 10
```

```
.end
```

- A test case which fails nodal analysis (while still being a valid circuit). My code will also fail this, as the matrix is singular, I did not have the free time to implement a check for this.

```
#non_nodal.ckt
.circuit
v1 1 gnd DC 8
I1 1 GND DC 8
.end
```

Ideal Output :

```
({'1': 8, 'GND': 0.0}, {'v1': 8})
```

But my code would throw an error becoz the matrix would be singular, a drawback of nodal analysis

10 Possible Extensions to this project

The reason that I have chosen to implement this algorithmic nodal analysis instead of simply forming the equations is that it is quite easy to extend this to dependant sources as well. Some minor modifications to the Z,B and C matrices will do the job.

11 References

- I would like to thank Arjun Krishnaswamy (EE23B009), Deenabandhan N (EE23B021) for going through some test cases and comparing answers to check the correctness of the algorithm, I would also like to thank Medha Girish (EE23B112) for pointing out a minor bug in my code, due to which my current signs were flipped.
- https://lpsa.swarthmore.edu/Systems/Electrical/mna/MNA3.html
- https://NumPy.org/doc/2.0/reference/generated/NumPy.linalg.solve.html
- https://stackoverflow.com/questions/8385602/why-are-NumPy-arrays-so-fast