SPICE Circuit Solver

EE2016 Applied Programming Lab '24

Sanjeev Subrahmaniyan S B

EE23B102

1 Assignment Objective

The objective of this assignment is to write python program to emulate a SPICE circuit solver. The objective is accomplished in the following steps:

1. Read a given SPICE netlist file, which adheres to specific syntax and parse it into the different components and the way they are interconnected to form the circuit.
2. Handle the different nodes and components in the circuit to populate the coefficient and constant matrices which determine the solutions to the system of equations derived from nodal analysis techniques.
3. Solve the system of equations and return the solution.

2 Introduction

SPICE is a circuit simulator, finding widespread use in electrical engineering and industries. In this assignment, such a simulator is realised with specific constraints and assumptions. Although the simulator can be easily extended to the other cases, the assignment does not require those extensions. The specific type of circuits for which the solver is made are ones which are exclusively made up of the following components:

1. Non time-varying resistors of predetermined resistance values
2. Non time-varying independent current sources of predetermined current values
3. Non time-varying independent voltage sources of predetermined voltage values

The assignment's objective is to return a tuple of two dictionaries, which contain the node voltages of all the nodes in the circuit and the currents through the independent voltage sources in the circuit. It is mentioned that all the conventions followed are in strict compliance with SPICE, specifically, *the current from the negative to the positive terminal inside the voltage source is assumed to be positive.*

3 SPICE netlist representation of electric circuits

Electrical circuits are represented textually as a graph of nodes. An example representation and its corresponding circuit follow, directly attached from the problem statement for the assignment.

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

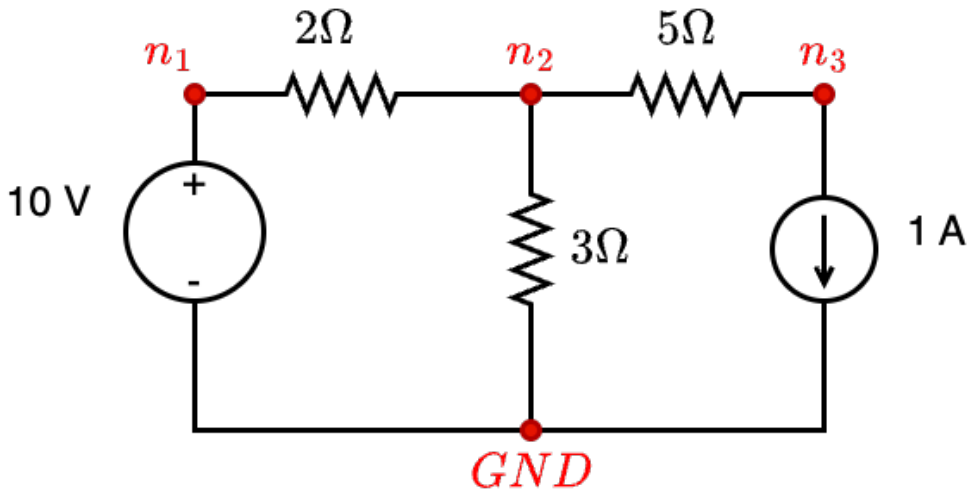

Figure 1: Example electrical circuit and its SPICE netlist

This convention is assumed to be strictly followed for all the test cases and inputs to the program. The specific assumptions are detailed in the next section.

4 Assumptions and input constraints

While writing a program that exhaustively handles all the possible problems is possible, I have assumed the following in my solution. Most commonly encountered circuits should adhere to this without problems, while the program will raise respective errors when failed. The specific parts of the program where these errors will raise are clearly mentioned in the comments with the program.

1. **All circuits have a GND node, whose potential is assumed to be zero and all other potentials are referenced with respect to the GND**. The absence of a GND node will result in abortion of the program. While this can be handled with the feature to use another node as reference, this solution mandates a GND node.

2. **All SPICE netlist files must contain a '.circuit' and '.end' lines to specify the start and end of the circuit**. This solution requires each file to have exactly one of each tags, and will error when not complied with.

3. **All current and voltage sources required a type to be specified - either 'ac' or 'dc'**. No default behaviour is assumed in this respect. Note that 'ac' sources are solved without considering the frequency as they do not make a difference for purely resistive circuits.

4. **The circuit must contain atleast one element for the computation**. Absence will error.

5. **The components must be named with the first alphabet being the uppercase version of their identity**. This means resistances must be 'Rxxx', voltage sources must be 'Vxxx' and current sources must be 'Ixxx'. The case sensitivity is emphasised.

6. **The values of each of the components are specified as numbers or as exponential numbers such as '2e3'**. No other input format is expected.

Other less commonly occurring errors like short circuits, hanging elements, invalid connections and floating components are handled in the program. The program has been tested against the corresponding test cases, which are at the end of this report. *Specifically, all circuits which lead to solutions of infinite currents or voltages will raise errors.*

5 The Solution Program

The solution is broadly divided into 3 sections, as outlined earlier in the report. Each of these are implemented as separate functions, all of which contain clear and detailed comments for the reader. The program is explained in steps in this section for completeness.

5.1 Imports

```
1 import numpy as np #Numpy is used to populate matrices and solve the
      equation system
2 from typing import List, Dict, Tuple
3
4 START_OF_CIRCUIT = '.circuit'
5 END_OF_CIRCUIT = '.end'
```

Listing 1: Imports and defines

These lines import the numpy library for handling the equation's matrices and solving them. The typing library is used for concise and clear documentation of the functions. Numpy is notably efficient in processing matrices due to how its background implementation in C and is thus much faster as opposed to conventionaly Gaussian Elimination implemented on python.

5.2 File parsing

The next part of the program reads the file and extracts the useful information(the circuit specifications) as explained in the comments. A number of different error checks are also implemented in this function - for valid existence of the file, compliance to the SPICE netlist format and presence of atleast one component.

```
def parse_file(filename: str) -> List[str]:
    """Parses the circuit file and returns the nodes.

    Parameter:
    filename(str): The name of the SPICE file which needs to be
       solved for.

    Returns:
    List[str]: A list of the nodes in the circuit, which are lines
       containing the component parameters alone.

    This function is used to open the file, read it and discard
       unnecessary data including inline comments starting with '#'.
    The nodes are returned in the form of a list, each of which
       contain the useful part of the SPICE file as a string, which
    can be processed to extract different parameters such as names
       and node names.
    """

    # Check existence of the file and create a filehandle
    try:
        filehandle = open(filename, "r")
    except FileNotFoundError:
        raise FileNotFoundError("Please give the name of a valid
           SPICE file as input")

    # Read the file contents into a list and close the handle
    lines = filehandle.readlines()
    filehandle.close()
```

```
24
25      # Extract useful information from the lines by removing comments
26      # and renameing GND for convenience
27      for i, line in enumerate(lines):
28          lines[i] = line.split("#")[0].strip("\n")
29          lines[i] = lines[i].replace('GND', 'n0')
30
31      # Verify proper structure of the SPICE circuit and find
           boundaries
32      try:
33          st_ind = lines.index(START_OF_CIRCUIT)
34          end_ind = lines.index(END_OF_CIRCUIT)
35          ckt_count = lines.count(START_OF_CIRCUIT)
36          end_count = lines.count(END_OF_CIRCUIT)
37      except ValueError:
38          raise ValueError("Malformed circuit file")
39      if ckt_count != 1 or end_count != 1:
40          raise ValueError("Netlist has too many start/end identifiers
               ")
41      if end_ind - st_ind == 1:
42          raise ValueError("No component found in the netlist")
43      # Sorts the nodes by name for convenience and extract the
           component lines alone
44      lines = lines[st_ind+1:end_ind]
45      lines.sort()
46      return lines
```

Listing 2: File parsing

Note that this function reads the file line-by-line, discards anything that follows the '#' symbol for comments and strips the lines of the newline tag. I have also chosen to replace the GND identifier for convenience and intuitive handling of all the nodes. The compliance checks are implemented by looking for the start and end identifiers and checking their uniqueness and validity.

5.3 Identifying and handling components

The next part of the code is targeted to make the different components and nodes in the program easier to handle in the further parts of the code. For this, the solution uses dictionaries.

5.3.1 Why dictionaries?

Dictionaries are used as the data structure due to the functionality offered by them in being able to access elements with an identifier/key. This is particularly useful when the other parameters such as nodes and values of a component are required. *Another option to handle this scenario would be to use instances of defined objects, which can be easily done.*

```
def make_dicts(components: List[List]) -> List[Dict]:
    """
    Makes dictionaries of nodes, resistances, voltage sources and
       current sources in the circuit.

    Parameters:
    components(List[List]): A list of components in the circuit,
       containing names, nodes connected to and the value of the
       component.

    Returns:
    List[Dict]: Returns 4 dictionaries, which map corresponding
       components to their parameter lists.

    This function makes dictionaries of the components in the
       circuit for easy handling during the equation matrix building
       .
    """
    # Initialize dictionary objects to hold values

    nodes = dict() # A dictionary, whose keys are the nodes, and
       values which are the connected components
    Resistances = dict()
    Vsources = dict()
    Isources = dict()

    # Process the components list
    for i in range(len(components)):
        temp = components[i].split() # Split the component into name
           , nodes, and value

        # Check if the node already exists in the dictionary and add
            the component
        if temp[1] in nodes.keys():
            nodes[temp[1]].append(temp[0])
        # If the node does not exist, make a new dictionary key and
           add the component
        else:
            nodes[temp[1]] = [temp[0]]
        if temp[2] in nodes.keys():
```

```
31              nodes[temp[2]].append(temp[0])
32          else:
33              nodes[temp[2]] = [temp[0]]
34
35          # Check if the components already exist in the corresponding
                lists and add them if not
36          if components[i][0] == 'R':
37              if len(temp) != 4: # Check for valid resistance
                    specification
38                  raise ValueError("Invalidly specified resistance
                        element")
39              if temp[0] not in Resistances.keys():
40                  Resistances[temp[0]] = temp
41          elif components[i][0] == 'V':
42              if len(temp) != 5: # Check for valid presence of source
                    type
43                  raise ValueError("Invalidly specified voltage source
                        element")
44              elif temp[0] not in Vsources.keys():
45                  Vsources[temp[0]] = temp
46          elif components[i][0] == 'I':
47              if len(temp) != 5: # Check for valid presence of source
                    type
48                  raise ValueError("Invalidly specified current source
                        element")
49              if temp[0] not in Isources.keys():
50                  Isources[temp[0]] = temp
51          else: # Raise error when components other than resistances
                or voltage/current sources are encountered
52              raise ValueError("Only V, I, R elements are permitted")
53
54      # Check for the valid presence of a GND node, otherwise, deem
            circuit invalid
55      if 'n0' not in nodes.keys():
56          raise ValueError("No GND node found")
57      return [nodes, Resistances, Vsources, Isources]
```

Listing 3: File parsing

For each of the elements, I check if the target dictionary already holds the element and accordingly process it. Another alternative to this would be to use sets, but I chose dictionary for the reasons outlined above.

Note that this segment of the code also checks for the validity of the elements in the circuit and accordingly handles them.

5.4 Matrix population and solution

5.4.1 Mapping nodes

The final function of the program fills the matrices which determine the solution to the system. This is done in steps as follows. The first part of the function maps each of the nodes, which are of string type, to unique numbers, that are intuitive to use with the matrices. The different voltage sources are also mapped to numbers in succession. This is because each of the node except the GND node and each of the voltage sources contribute to one line of the matrices. The GND node is ignored in this mapping.

```
def evalSpice(filename):
    """
    Evaluates the SPICE circuit and returns node voltages and currents through voltage sources.

    Parameters:
    filename(str): The name of the file which contains the circuit to be evaluated.

    Returns:
    Tuple[Dict]: Two dictionaries. The first contains the node voltages of all the nodes in the circuit while
    the second contains the currents through the voltage sources.

    This function runs all routine checks on the file given as input to solve for the circuit variables. It is
    assumed that the circuit consists purely of resistances, and independent current and voltage sources and
    will raise errors when not compliant. It also runs checks on the solvability of the circuit and returns the
    solved variables for evaluation.
    """

    components = parse_file(filename)
    nodes, Resistances, Vsources, Isources = make_dicts(components)

    # Ensure the nodes dictionary contains only unique nodes without redundancies
    for node in nodes.keys():
        nodes[node] = list(dict.fromkeys(nodes[node]))

    # Create mappings of the nodes in the circuit to whole numbers which are used to index the coefficient matrix
    nodemap = {}
    i = 0
    for key in nodes.keys():
```

```
29          if not key == 'n0': #Ignore GND node
30              nodemap[key] = i
31              i += 1
32      # The index i is carefully handled to ensure the matrix is of
            the form required to solve
33      # Create mappings of the voltage sources in the circuit to whole
            numbers, similar to node mapping above
34      vsource_map = {}
35      for v in Vsources.keys():
36          vsource_map[v] = i
37          i += 1
38  # ...Continues in the next section
```

Listing 4: Mapping nodes and sources

5.4.2 Matrix Population

Here, the coefficient and constant matrices are initialized to zero matrices of suitable dimensions. The number of equations required to solve for all the required parameters for a circuit of N nodes and V voltage sources is N + V - 1, which follows from elementary nodal analysis theory.

```
1  # ...Continued
2      # Compute the number of rows and columns the coefficient matrix
           will have
3      num_rows = len(nodes.keys()) + len(Vsources) - 1 # Ignoring the
           GND node
4      num_cols = num_rows
5
6      # Initializing numpy zero matrices for coefficients and
           constants
7      coeffs = np.zeros((num_rows, num_cols))
8      consts = np.zeros((num_rows, 1))
9  # ... Continues
```

Listing 5: matrix initialization

The elements are handled one after another in the for loop. The idea used is that, at every node, the current leaving through a resistance connected to it is (Vi - Vj)/R, where Vi is the potential of the node and Vj is the other node the resistance is connected to. When cast into a matrix, this leads to coefficients of +1 and -1 in the respective positions.

```
1  # ...Continued
2      # Populating the matrices according to nodal analysis equations
3      i = 0
4      for node in nodes.keys():
```

```
        if not node == 'n0':
            for element in nodes[node]:
                if element[0] == 'R':
                    if float(Resistances[element][3]) == 0:
                        raise ValueError("Short circuit leading to
                            infinite current encountered")
                    if Resistances[element][1] == node:
                        coeffs[i][nodemap[Resistances[element][1]]]
                            += 1/float(Resistances[element][3])
                    elif Resistances[element][1] != 'n0':
                        coeffs[i][nodemap[Resistances[element][1]]]
                            -= 1 / float(Resistances[element][3])
                    if Resistances[element][2] == node:
                        coeffs[i][nodemap[Resistances[element][2]]]
                            += 1/float(Resistances[element][3])
                    elif Resistances[element][2] != 'n0':
                        coeffs[i][nodemap[Resistances[element][2]]]
                            -= 1 / float(Resistances[element][3])
# ...Continues
```

Listing 6: Handling resistances

For current sources, the nodal analysis equations take the current entering the node on the right side of the equation. Following SPICE conventions, where the current in a current source is assumed to enter the first node, this leads to a negative contribution to the right hand side matrix. Similarly, it leads to a positive contribution to the other node that it is connected to. Voltage sources are handled similarly, in accordance with the conventions of SPICE.

```
# ...Continued
                elif element[0] == 'I':
                    if Isources[element][1] == node:
                        consts[i][0] -= float(Isources[element][4])
                    else:
                        consts[i][0] += float(Isources[element][4])

                elif element[0] == 'V':
                    if Vsources[element][1] == node:
                        coeffs[i][vsource_map[element]] += 1
                    elif Vsources[element][2] == node:
                        coeffs[i][vsource_map[element]] -= 1
            i += 1
# ...Continues
```

Listing 7: Handling current sources

Finally, the currents through the voltage sources are added on the constants matrix. If the first node the source is connected to is the node under consideration(this implies the positive

terminal of the source is connected to this node), convention dictates that the current enters the node(current in a voltage source flows from the negative to positive terminal inside a voltage source).

```
# ...Continued
    for v in Vsources.keys():
        if not Vsources[v][1] == 'n0':
            coeffs[i][nodemap[Vsources[v][1]]] += 1

        if not Vsources[v][2] == 'n0':
            coeffs[i][nodemap[Vsources[v][2]]] -= 1
        consts[i] += float(Vsources[v][4])
        i+=1
# ...Continues
```

Listing 8: Handling voltage sources

The use of the variable i is intuitive, and can be understood by looking at how nodal equations are written for nodes in matrix form.

5.5 Solution and remapping

In this part, the system of equation are solved using the Numpy library. An error is raised when the system is unsolvable. Finally, the solution matrix is remapped to the nodes it corresponded to, and the answer is returned in the form of the required dictionaries.

```
# ...Continued
    # Use numpy to solve the system and raise an error if the
       circuit is unsolvable
    try:
        ans = np.linalg.solve(coeffs, consts)
    except np.linalg.LinAlgError:
        raise ValueError("Circuit error: no solution")

    # Create the return dictionaries by mapping the solution values
       to the corresponding nodes
    nodeV = dict()
    for key in nodes.keys():
        if key == 'n0':
            nodeV['GND'] = float(0)
        else:
            nodeV[key] = float(ans[nodemap[key]][0])

    sourceI = dict()
    for key in Vsources.keys():
        sourceI[key] = float(ans[vsource_map[key]][0])
```

```
19
20      # Returns dictionaries for evaluation
21      return (nodeV, sourceI)
```

Listing 9: Solving matrix system

6 Testing and edge case handling

Due to an inherent assumption of ideal sources in SPICE, there are instances of infinite current and similar issues. These edge cases are carefully tackled by raised appropriate errors in the program. All of the extra test cases I tested are in the extra folder as part of the submission folder. I found a few of these in the testbenches given as part of the previous iterations of the course, and wrote many others on my own. A non-exhaustive enumeration of them follows:

1. Circuits without '.end' or '.circuit'
2. Circuits with more than one instances of '.end' or '.circuit'
3. Circuits without a GND node
4. Multiple instances of valid circuits
5. Netlists with junk data and comments
6. Circuits with components other than R, V, I
7. Circuits with very small/very high resistances
8. Circuits with ac sources
9. Pure current sources on one node
10. Hanging components
11. Incompletely specified circuit components
12. Voltage sources in series
13. Short circuits due to direct connections or zero resistance connections
14. Shorted voltage sources
15. Circuits without a closed loop
16. Sources with zero values

It is worth mentioning that using Numpy makes the numerical calculations robust to very high precision and low errors.

7 Discussion and references

7.1 References

Being fairly accustomed to python and Numpy, I did not find the necessity to look up too many references on python. To understand the SPICE notation and convention, I found this page(https://www.seas.upenn.edu/ jan/spice/spice.overview.html) very informative.

7.2 Discussions

I had numerous discussions with friends, many concerning optimizing the solutions and looking for edge cases. A particularly noteworthy discussion was about using supernode analysis and selective node evaluation which could optimize the code to have a smaller complexity than the regular implementation. I chose not to implement it because a decrease from $(n + k)^3$ to $n^3 + k^3$ would not be significant for small circuits. Outside of what is implemented in this program, I also found a discussion on the C implementation of Numpy very informative.

8 Conclusion

With this assignment, I gained an understanding of how professional simulation softwares are implemented at a superficial level. The handling of different components was insightful and highlighted the use of proper conditional statements in python to get the job done. The computational speed differences between Numpy and python was also evident when I attempted solving a system of equations using both the methods.