Assignment 2 - Spice simulation in Python

Objective

Given a spice circuit, calculate the voltage at each node and calculate the current through the voltage sources.

Current through resistors will be the difference of the voltages of the nodes across the the resistor divided by the resistance. Current through current source will simply be the value of the current source.

The program should raise error in case of wrong file location or malformed spice data.

If the circuit has faulty connections like a loop with only voltage sources or a node with current source in all the branches or connected to such nodes

Approach

- Let the number of nodes be N and the number of voltage sources be B
- Assign index to the nodes and voltage sources
- We operate with N node voltage variables and B current voltage variables
- Write KCL equation of each node in the circuit (N equations)
- Write votlage equation for each voltage source (B equations)
- Thus we have N+B variables and N+B equations which can be solved
- For faulty cases, we get a singular matrix which cannot be solved

Implementation

1. Check input file

Check if the file address is valid using `os.path.exists(filename)` raise `valueError` if the path does not exist

2. Check spice inputs

- The file must have one `.circuit` and one `.end`. I maintained the count of them in a variable and raised error if one or both of them is not present.
- Valid components are ‘V’, ‘I’ and ‘R’. If a line starts with any other character, `valueError` is raised
- Removing the commented part of the lines can be done using ‘

3. Data structure to store the circuit

- Maintain a list of nodes, index of a node is equal to its index in a list
- Create a dictionary to map the nodes to their indices
- Maintain a list and dictionary for voltage sources as well and similary assign indices (for the current variables)
- Create an class `Component` to store each component of the circuit. Voltage sources are also stored in a separate class `Vsource`
- The circuit connections are stored in an *Adjacency list*
- An adjacency list contains the connection to each node. Each element is a list of the nodes connected to that node and the component between them.

4. Writing Equations

- Initialize matrices A, B of size *nxn* and *nx1* where *n=N+B-1*
- The function `Update(i,r)` writes equation for the ith node in the
- It iterates through each node connected to the resistor and updates the matrix using KCL, that is, equation total current outflow equals zero
- Current added for **resistor**:
$$I = \frac{V_1}{R} - \frac{V_2}{R}$$
- Current added for **Current Source**:

$$I = I_{source}$$

Sign depends on the terminal connected to the node

- Current added for **Voltage Source**:
$$I = I_{vsource}$$
The current variable corresponding the voltage source is added. Sign depends on the terminal connected
- Voltage sources have an additional voltage-equation:

$$V_1 - V_2 = V_{source}$$

I am storing these equations in a separate list. They are added to the matrix after the KCL equations are written.

The update function is called for every node in the nodes list

Thus we will have a total of **N+B-1** equations

5. Checking circuit consistency

- Incase of loop with only voltage sources or nodes with current sources in all branches, we get dependent equations, causing the conductance matrix to be singular
- Determinant of the matrix can be calculated using `np.linalg.det()`
- If determinant turns out to be zero, then *ValueError* is raised

6. Solving the equation

- The matrix equation $AX = B$ is solved using `np.linalg.solve(A,B)`
- The solution is a 1D matrix of size N+B-1
- We know that voltage of ground is zero, so we insert a zero at the first position
- Now we have an array of lenght N+B
- For i<=n, ith element represents the voltage of the ith node. for i>n, ith element is the current through the (i-n)th voltage source
- These values are stored in two sepearate dictionaries with keys as the name of nodes and voltage sources respectively
- The two dictionaries are returned as a single tuple

Additional Test Cases

Here some test cases that I verified my code with

1. Shorted resistor: `tc1.ckt`

```
.circuit
V1 n1 GND dc 5
R1 n1 n2 5
R2 GND n2 5
R3 n1 n1 5
.end
```

The resistor is shorted, that is connected between the same nodes. In this case, the equation from both the ends of the resistor cancel each other, giving zero current through the resistor

2. Shorted current source: `tc2.ckt`

```
.circuit
V1 GND n1 dc 5
```

```
R1 n1 GND 5
I1 n1 n1 2
.end
```

When current source is shorted, all the current produced by the current source flows through the shorted wire, thus not affecting the rest of the circuit. My code manages to handle this case effortlessly.

3. Hanging branch: `tc3.ckt`

```
.circuit
V1 GND n1 dc 5
R1 GND n2 dc 5
R2 n2 n1 5
R4 n2 n3 5
V2 n3 n4 dc 10
.end
```

The R4-V2 branch is not a complete loop as n4 is not connected to any node. Thus the current through that branch will be zero. The voltage of n4 is determined by the source V2.

Alternative solution

It possible to solve for the voltages in the circuit without using current variables for voltage sources. This requires writing **supernode** equations across the terminals of the voltage source.

In case multiple voltage sources have common nodes, then they must together be considered as a single super node.

In this approach, we end up with only **N-1** equations(N=number of nodes) rather than **N+B-1** equations.

The `np.linalg.solve` operates at $O(N^3)$ time complexity

Hence the reduction in number of equation can make the algorithm highly efficient, especially for circuits with large number of voltage sources.

The python script `super_nodes.py` implements this method to find the voltage and current solution. I haven't tested this code on many test cases so I am not sure if it will work for all circuits, but it passes the basic tests.

I am also sorry that I didn't get enough time to comment that code properly

Here is the function in `super_nodes.py` which solves for the voltages of the nodes recursively

```
def update(i,r):
   visited[i] = 1
```

```
    updated = 0
    for el in circuit[i]:
        n1 = i
        n2 = el[0]
        print(n1,n2)
        print(A,B,r)
        comp = el[1]
        val = comp.value
        if comp.type == 'R':
            updated+=1
            if n1:
                A[r][n1-1]+=1/val
            if n2:
                A[r][n2-1]+=-1/val
        elif comp.type == 'I':
            updated+=1
            B[r]+=-val
        elif comp.type=='V':
            print(visited)
            if visited_sources[comp.name]>=2:
                break
            visited_sources[comp.name]+=1

            #recursive call
            if visited[n2]==0:
                update(n2,r)
                t = [n1,n2,val]
                stack.append(t)

    if updated:
        r+=1
    return r
```

The difference is in the voltage source case where the update function is called recursively

Here the recursive fuction to find the current after calculating the node voltages. The time complexity for this function is $O(N * B)$

```
def find_current(v):
    visited_sources[v]=1
    n1 = vsource_dict[vlist[v]].n1
    current = 0
    updated = 0
    for el in circuit[n1]:
        n2 = el[0]
        print(n1,n2,current)

        comp = el[1]
        val = comp.value


        if comp.type == 'R':
            # if vpresent:
            #     break
            updated+=1
            current += (output[nodes[n1]]-output[nodes[n2]])/val
            print("//",n1, n2,val,current)

        elif comp.type == 'I':
            # if vpresent:
            #     break
            updated+=1
            print("//",n1, n2,val)
            current+=val
        else:
            if vsource_dict[comp.name].ind == v:
                continue
            if visited_sources[vsource_dict[comp.name].ind]:
                raise ValueError("Circuit error: no solution")
            current += find_current(vsource_dict[comp.name].ind)
    return current
```

THANK YOU