Software Lab 8: Describing Circuits

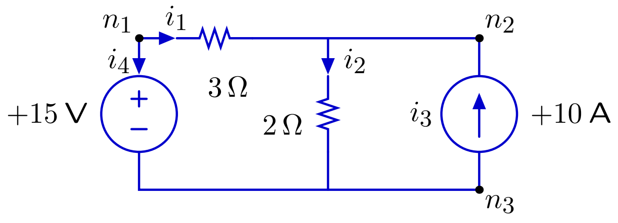
- **Athena machines:** Do `add -f 6.01` and `athrun 6.01 update`.
- **Lab laptop:** Do `athrun 6.01 update`.
- **Personal laptop:** Download software `swlab8.zip` file from course web page.

You can write your code in the file `circSkeleton.py` in `lab8/swLab`

The software lab for this week is to develop a method for describing circuits at a high level of abstraction, and convert that description into linear equations in the representation from software lab 7.

1 Circuit equations

In the last software lab, we wrote and solved sets of linear equations. We can use this software to help us solve circuits. For example, using the NVCC method, we might write the equations for this circuit (see Section 7.6 in the Course Notes). We are using `'n1'`, `'n2'`, etc., as the names of the node voltages and `'i1'`, `'i2'`, etc as the names of the component currents.



as follows:

```
ckt = le.EquationSet()
ckt.addEquation(le.Equation([1.0, -1.0], ['n1', 'n3'], 15.0))
ckt.addEquation(le.Equation([1.0, -1.0, -3], ['n1', 'n2', 'i1'], 0.0))
ckt.addEquation(le.Equation([1.0, -1.0, -2], ['n2', 'n3', 'i2'], 0.0))
ckt.addEquation(le.Equation([1.0], ['i3'], 10.0))
ckt.addEquation(le.Equation([-1.0, -1.0], ['i4', 'i1'], 0.0))
ckt.addEquation(le.Equation([1.0, -1.0, 1.0], ['i1', 'i2', 'i3'], 0.0))
ckt.addEquation(le.Equation([1.0],['n3'], 0.0))
```

And then we could solve it:

```
 ckt.solve()
i1 = -1.0
i2 = 9.0
i3 = 10.0
i4 = 1.0
n1 = 15.0
n2 = 18.0
n3 = 0.0
```

That's convenient, because it saves us from our own algebra errors, but it's hard to keep all those coefficients straight. We will develop software that allows a more compact specification:

```
c = circ.Circuit([circ.VSrc(15, 'n1', 'n3'),
                  circ.Resistor(3, 'n1', 'n2'),
                  circ.Resistor(2, 'n2', 'n3'),
                  circ.ISrc(10, 'n3', 'n2') ])
c.solve('n3')
```

The major simplification is that we don't have to mention the currents when specifying the components and we don't have to specify the KCL equations at all. When we call `c.solve('n3')`, a set of equations is automatically constructed, with node `'n3'` as ground, and then solved:

```
n1 - n3 = 15
n1 - n2 - 3 * i_n1->n2_39 = 0
n2 - n3 - 2 * i_n2->n3_40 = 0
i_n3->n2_41 = 10
i_n1->n3_38 + i_n1->n2_39 = 0.0
-i_n1->n2_39 + i_n2->n3_40 - i_n3->n2_41 = 0.0
n3 = 0
```

The currents are automatically given names. So, `i_n1->n2_39` is a current that flows between nodes `n1` and `n2`.[1]

Wk.8.1.1 For the circuit above write the `EquationSet` and the more abstract representations.

[1] We append an additional unique number (in this case 39) to the name, because, if there are multiple components in parallel between `n1` and `n2`, we need to be able to speak of several different current components between those nodes.

2 Overview of the `Circuit` class

A `Circuit` class instance is created with a list of component instances, as shown above. The key method is the `solve` method, which constructs an equation set from the components and solves it. We will define each component type as a class which can construct the relevant equation for that type of instance (see below).

However, the `solve` method will also need to construct the KCL equations at every node (except the ground). So, we will need to know which components are connected to which nodes. In our implementation, we use the `NodeToCurrents` class to keep track of which component current enters (or leaves) each node. Each component has has a method that provides this information (see below).

Every type of component, for example voltage source, resistor, and op amp, will be a subclass of the `Component` class. Every subclass of the `Component` class has to supply two methods: `getEquation`, which returns an instance of `le.Equation` that contrains the voltage across the terminals of the component, and `getCurrents`, which returns the list of currents that this component adds to the nodes to which it is connected. All two-input components have the same pattern of currents: they make a new current variable when created, and then assert that it flows into their node `n1` and out of their node `n2`. So, we have implemented this pattern as the default `getCurrents` method in the `Component` class.

```
class Component:
    def getCurrents(self):
        return [[self.current, self.n1, +1],
                [self.current, self.n2, -1]]
```

Here is how the `Resistor` component is implemented.

```
class Resistor(Component):
    def __init__(self, r, n1, n2):
        self.current = util.gensym('i_'+n1+'->'+n2)
        self.n1 = n1
        self.n2 = n2
        self.r = r
    def getEquation(self):
        # your code here
```

The `util.gensym` procedure takes a string as an argument and returns a string which is the argument with a unique integer appended to it.

Wk.8.1.2 Implement the `getEquation` method for the `Resistor` class.

Wk.8.1.3 Implement the `OpAmp` class as a voltage-controlled voltage source; see Section 7.8.1 of the Course Notes.

3 Implementing the `Circuit` class

The `Circuit` class has two methods;

```
class Circuit:
    def __init__(self, components):
        self.components = components

    def solve(self, gnd):
        es = le.EquationSet()
        n2c = NodeToCurrents()
        for c in self.components:
            es.addEquation(c.getEquation())
            n2c.addCurrents(c.getCurrents())
        es.addEquations(n2c.getKCLEquations(gnd))
        return es.solve()
```

A circuit is just a list of instances of the `Component` class. When we ask the circuit to solve itself, we provide the name of a node, passed in as parameter `gnd`, which will be the ground node and have voltage 0; then the `solve` method:

0. Makes a new empty equation set `es`.
1. Makes a new instance, `n2c`, of the `NodeToCurrents` class. This class keeps track of which currents are flowing into and out of each node.
2. For each component, adds the equation that describes the relationship between voltage and current that the component induces, and it adds the currents to the appropriate nodes in `NodeToCurrents`.
3. Adds the KCL equations that result from the node-current relationships stored in `n2c`, and one that sets the node named by the `gnd` variable to have voltage 0.
4. Solves the equations.

You can read about the `NodeToCurrents` class and its methods in the documentation.

Wk.8.1.4 Implement the `NodeToCurrents` class. Please debug your code in the `circSkeleton.py` file and then paste it into the Tutor.