# *Software Lab 8* *6.01 – Fall 2011*

# Describing Circuits

**Goals:** This software lab seeks to develop a method for describing circuits at a high level of abstraction, and to convert that description into linear equations which can then be solved. This is the method that CMax, and many other systems, use when simulating electrical circuits. You will:

- Understand methods for representing and solving sets of linear equations
- Understand the node-voltages-with-component-currents (NVCC) analysis method for circuits
- Expore a high-level Python representation of circuits, the `Circuit` class
- Implement representations of resistors and opamps
- Implement the `NodeToCurrents` class, to solve for circuit voltages and currents

## 1 Setup

**Using your own laptop**

- Be sure you have the 6.01 software libraries installed.
- Download and unzip `swLab08.zip` into a convenient folder (e.g., `~/Desktop/6.01/swLab08`).

This week's work files are `circSkeleton.py` and `swLab08Work.py`.

## 2 Specifying and solving linear equations

Consider the problem of finding values for $x$ and $y$ that satisfy the two equations:

$$5x - 2y = 3, \quad \text{and}$$
$$3x + 4y = 33.$$

You would probably approach this with the *substitution method*, in which you solve the first equation for $x$, getting $x = \frac{2}{5}y + \frac{3}{5}$, and then substituting that into the second equation, getting

$$\frac{6}{5}y + \frac{9}{5} + 4y = 33.$$

Then, solving for $y$ yields $y = 6$, and substituting $y = 6$ into the $x$ expression yields $x = 3$.

Solving two equations in two unknowns is easy, but humans are not generally good at solving larger systems of the type that we encounter in circuits. Therefore, we will use a computer. We could try to write a computer program to perform the substitution method, but this method is complicated and computationally inefficient. By contrast, ***Gaussian elimination*** is efficient and relatively easy to implement on a computer. We'll use a standard implementation of it from the **Python `numpy` library**.

We will use the **6.01 software module `le`** to represent sets of equations.[1] An **equation** is represented with an instance of **class `le.Equation`**, which takes, at initialization time, three arguments:

- `coeffs`: a list of numerical coefficients for the variables mentioned in the linear equation
- `variableNames`: a list of strings, naming the variables in the equation; the variable names must be listed in the same order as the coefficients
- `constant`: the numerical constant in the equation, with the sign chosen so that the constant is the only term on one side of the equality.

For example, we could represent the equation $-3x + 9.2z = 4$ as

```
le.Equation([-3, 9.2], ['x', 'z'], 4)
```

or by any of several alternative (and equally valid) expressions, including these two:

```
le.Equation([9.2, -3], ['z', 'x'], 4)
le.Equation([3, -9.2], ['x', 'z'], -4)
```

We can represent a **set of equations** using an instance of the **class `le.EquationSet`**. This class takes no parameters at initialization time, and returns an object that represents an empty set of equations. Equations can then be added using the `addEquation` method, whose input is an instance of `le.Equation`, described above. You can find the solution to a set of equations by using the `solve` method of an `le.EquationSet`. The `solve` method returns an instance of `le.Solution`. You can then look up the value of any of the variables by using the `translate` method, which takes, as input, a string that represents the variable of interest, and returns the value of that variable in the solution.

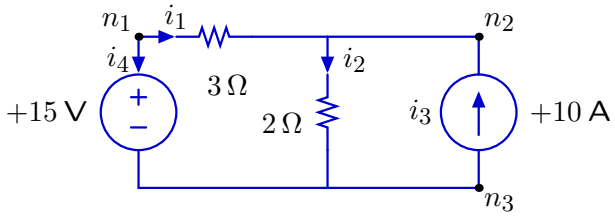
Using these classes, you can describe, then solve, our simple example like this:

```
>>> small = le.EquationSet()
>>> small.addEquation(le.Equation([5, -2], ['x', 'y'], 3))
>>> small.addEquation(le.Equation([3, 4], ['x', 'y'], 33))
>>> sol = small.solve()
>>> sol.translate('x')
3.0
>>> sol.translate('y')
6.0
```

[1] see the 6.01 **Software Documentation** under the **Reference Material** tab of the 6.01 web page.

# 3 NVCC circuit equations

We now customize our equation solving software to solve circuit equations. Our method is based on the node-voltages-with-component-currents (NVCC) method of analysis (see **Section 6.4.1 in the Course Notes**), which we describe by way of the following example, where `'n1'`, `'n2'`, and `'n3'` represent node voltages and `'i1'`, `'i2`, `'i3'`, and `'i4'` represent component currents.




We start by creating an empty equation set:

```
ckt = le.EquationSet()
```

Now, we add one equation for each of the four components:

$$n_1 - n_3 = 15$$
$$n_1 - n_2 = 3i_1$$
$$n_2 - n_3 = 2i_2$$
$$i_3 = 10$$

as follows:

```
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))
```

Next, we need to specify an equation that sets the voltage of the ground node to be zero. We have chosen $n_3$ as ground,

$$n_3 = 0$$

but any other choice would also lead to a valid (but different) solution. We add the ground equation to the equation set with

```
ckt.addEquation(le.Equation([1.0],['n3'], 0.0))
```

Finally, we specify KCL equations for all of the nodes except the ground node:

$$-i_4 - i_1 = 0$$
$$i_1 - i_2 + i_3 = 0$$

which are added to the equation set with

```
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))
```

Now, we can solve the circuit, with the following result:

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

This is convenient, because it saves us from our own algebra errors. Unfortunately, it can be hard to remember to construct exactly the right set of equations (do I have one for each constituent component? do I have all of the current equations for each node? did I use the right names and coefficients for the currents?).

# 4 Describing circuits: the `Circuit` class

A higher-level way to specify a circuit is illustrated as follows:

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

Here we specify each component (e.g., a 15V voltage source, a 3Ω resistor, a 2Ω resistor, and and 10A current source) along with strings that represent the nodes to which that component is connected. The `solve` method then takes the symbolic name for the ground node as its single input and solves the resulting equations using the `le` module discussed previously:

```
>>> c.solve('n3')
Solving equations
*****************
+n1-n3 = 15
+n1-n2-3*i_n1->n2_14 = 0
+n2-n3-2*i_n2->n3_15 = 0
+i_n3->n2_16 = 10
+i_n1->n3_13+i_n1->n2_14 = 0.0
-i_n1->n2_14+i_n2->n3_15-i_n3->n2_16 = 0.0
+n3 = 0
*****************
i_n1->n2_14 = -1.0
i_n1->n3_13 = 1.0
i_n2->n3_15 = 9.0
i_n3->n2_16 = 10.0
n1 = 15.0
n2 = 18.0
n3 = 0.0
```

The first four equations (listed above between the ******* borders) describe the components, the next two are KCL equations, and the last specifies the ground. The solution (listed after the ******* borders) tells us the currents through the components and the voltages at the nodes.

Notice that the `Circuit` class provides two major simplifications. First, the equations for each component were automatically generated from the specified element type (e.g., resistor, voltage source, etc.). Second, we don't have to specify KCL equations at all.

Also notice that names are automatically assigned to the component currents. For example, `i_n1->n2_14` is a current that flows between nodes `n1` and `n2`. We append an additional unique number (in this case `14`) to the name, because there could be multiple components connected in parallel between `n1` and `n2`.

**Wk.8.1.1** Write the `EquationSet` for the circuit shown in the tutor problem. Also, write the more abstract representation (described above) for that circuit. You can debug this in idle using the file `swLab08Work.py`.

# 5 Implementing components: resistors and opamps

The `Circuit` **class** keeps track of all of the components of a circuit as instances of classes that represent each type of component (e.g., the `Resistor` class represents resistors, the `VSrc` represents voltage sources, etc.).

Each component class is a subclass of the `Component` **class**, and must supply two methods:

- `getEquation`, which returns an instance of `le.Equation` that constrains the voltage across the terminals of the component, and
- `getCurrents`, which returns the list of currents that this component introduces to the nodes to which it is connected. Each current is represented as a list `[i, node, sign]`, where `i` is the name of a current variable, `node` is the name of a node, and `sign` is the sign of that current at that node, either `+1` or `-1` (indicating whether the current going in or out of the node).

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. This procedure is useful for implementing names for currents.

**Wk.8.1.2** This problem guides you through implementing the `getEquation` method for the `Resistor` class.

**Wk.8.1.3** This problem guides you through implementing the `OpAmp` **class** as a voltage-controlled voltage source; see **Section 6.6.1 of the Course Notes**.

# 6 Implementing the `NodeToCurrents` class

The heart of the method of NVCC analysis method for solving circuit equations is assembling the correct set of equations, and using a linear equation solver to obtain the node voltages and circuit currents. In this part of the lab, you build the `NodeToCurrents` class which performes these steps to solve for circuit parameters.

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. Calling the `solve` method then does the following:

1. Makes a new empty equation set `es`.
2. Makes a new instance, `n2c`, of the `NodeToCurrents` class. This class keeps track of which currents are flowing into and out of each node.
3. 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`.
4. 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.
5. Solves the equations.

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

**Wk.8.1.4** Implement the `NodeToCurrents` class. Note that the `getKCLEquations` method should return a list of equations (not an EquationSet). There should be one KCL equation in the list for each unique node other than ground. There should also be one equation in the list to specify that the voltage associated with the ground node is zero.
You should debug your code in the `circSkeleton.py` file and then paste it into the Tutor.

6.01SC Introduction to Electrical Engineering and Computer Science

Spring 2011