MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.01—Introduction to EECS I
Spring Semester, 2008

Assignment 9, Issued: Tuesday, April 8

Overview of this week's work

In software lab

- Start working through the software lab. It is likely that you will have to finish it as homework.

Before the start of your design lab on Apr 10 or 11

- Read the class notes and review the lecture handout.
- Do the on-line tutor problems in section 8.1.
- Read the entire description of the design lab, so that you will be ready to work on it when you get to lab.
- Bring in to design lab, on paper, a solution to question 9. You may do this individually, or with your partner. You should staple this solution to your lab check-off sheet.

In design lab

- Do the nano-quiz.
- Work through the design lab.

At the beginning of your next software lab on Apr 15 or 16

- Submit written solutions to questions 2, 3, 4, 7, and 8 from the software lab and problems to be determined from the design lab. All written work must conform to the homework guidelines on the web page.

Software Lab

You will need to have installed `numpy` for this lab. Instructions are in the 6.01 Software pages. Or use a lab laptop.

In this software lab you will delve into the implementation of the circuit solver that you used last week. In the process, you'll get more insight into a systematic approach to solving circuits as well as into the voltage-controlled voltage-source model of op-amps.

Circuit Constraints

The circuit solver that you used last week took pains to keep all the details of how it operates out of sight. You were able to use it effectively without understanding how it managed to get the solutions for the circuits. This is another instance of the power of abstraction at work. We basically created a new "language" for circuits, just like Python is a language for computation and system functions are a language for signal processing. Now we will look in more detail on how the circuit solver can be implemented.

The solver you used last week is built around the idea of a *constraint* – an arbitrary function of variables that is supposed to evaluate to zero for the "desired" values of the variables. Given a list of constraints, the solver used Newton's method to find values of the variables that "satisfy" (make zero) all the constraints. This is a very general approach but a bit more complicated than we have time to understand this week.

We'll look at a simpler version of the solver that is limited to dealing with constraints that can be formulated as linear equations. Fortunately, this includes resistors, regular voltage sources, and the model of the op-amp as a voltage-controlled voltage-source (without the limitation on the output voltage due to source voltages). This simplified version should help you see what's going on, without the complication of finding numerical solutions for non-linear equations.

In the `lab9` directory, you will find `solveLinearConstraints.pyc`, which contains programs for creating lists of constraints and their associated variables, and then once the list of constraints and variables has been generated, determining values for the variables so that the constraints are satisfied. In particular, a user must first create an instance of the class `ConstraintSet`. Then the user invokes `ConstraintSet`'s method `addConstraint` multiple times, once for each of the constraints. The method `addConstraint` appends a constraint and the constraint's associated variables to the instance. By calling `ConstraintSet`'s method `solve`, values are determined for the variables so that the constraints are satisfied. The values of the variables can be printed by calling `ConstraintSet`'s method `display`.

The function `addConstraint` has one argument, which is a linear equation specified by a set of pairs, each of which consists of a coefficient and a variable name. The constant term in the linear equation is specified in the same way, with the variable name being `None`. We will have a number of standard procedures for generating constraints, one for each type of constraint that we want to have in circuits. But, remember that this approach is simply using linear equations to model the circuits and it can be used for any problem where the solution can be represented as the solution to a set of simultaneous linear equations.

As an example, consider the problem of finding values for x and y that satisfy the two constraints

$$5 * x - 2 * y = 3$$

and

$$3 * x + 4 * y = 33.$$

Of course, $x = 3$ and $y = 6$ satisfy the above two constraints.

To use the `ConstraintSet` class and its method `solve` to find the constraint-satisfying values of `x` and `y`, we first create an instance of `ConstraintSet` and add the two constraints

```
linSys = ConstraintSet()
linSys.addConstraint([(5.0, 'x'), (-2.0, 'y'), (3.0, None)])
linSys.addConstraint([(3.0, 'x'), (4.0, 'y'), (33.0, None)])
```

Note that we're representing the linear equation constraints as lists of tuples of coefficients (which must be numbers) and variable names (which must be strings), except that the constant term has `None` for a variable name. There must be exactly one constant term, and it must be only term on the **right** hand side of the equation.

Finally, we call `solve` and display the solution:

```
solution = linSys.solve()
linSys.display(solution)
```

In order to use this approach for circuit problems, we need an organized approach for generating the variables and constraints for a circuit. The course notes present the nodal approach for accomplishing this task. The steps in the nodal approach are

1. Label all the circuit node voltages and element currents (noting direction), and select a reference (ground) node.
2. For each element, write constitutive equations that relate element currents to the voltages at the element's terminals.
3. For each circuit node, except the reference (ground) node, write a conservation law. That is, insist that the sum of currents entering a node should be equal to the sum of currents leaving a node.

In order to use the constraint solver to solve circuit problems, it is helpful to have functions that return constraints associated with a circuit's constitutive equations and conservation laws. In the file `linearCircuitConstraints.py`, there are functions that implement circuit related constraints. The `resistor` and `vsrc` functions implement the constitutive relations associated with a resistor and a voltage source.

In the case that we have an R-ohm resistor between nodes n_1 and n_2, and we let $i_{1,2}$ be the current flowing from n_1 to n_2, then the following constraint has to hold between those variables:

$$v_{n_1} - v_{n_2} - Ri_{1,2} = 0 \ .$$

If we have a V-volt voltage source between nodes n_1 and n_2, and we let $i_{1,2}$ be current flowing from n_1 to n_2, then the following constraint has to hold between those variables:

$$v_{n_1} - v_{n_2} = V \ .$$

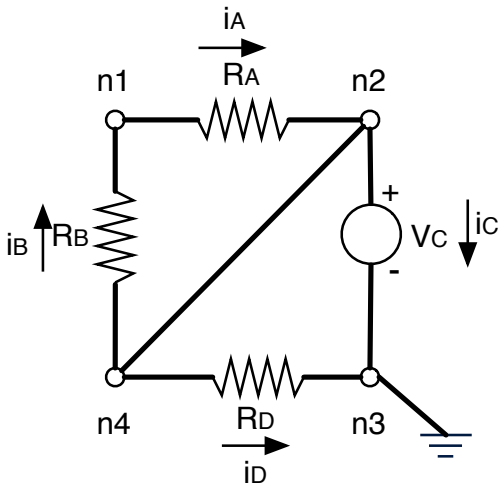

Figure 1: Circuit with three resistors and a voltage source.

Note that V is a constant, not a variable.

Here is an example circuit specification, corresponding to the first example in the lecture notes (reproduced here in figure 1). You can find it in `example.py`.

We start by specifying values for the resistors and the voltage source. Naming them here makes them easy to play with.

```
ra = 100    # 100 ohms
rb = 200    # 200 ohms
rd = 100    # 100 ohms
vc = 10     #  10 volts
```

Next, we make an instance of the `ConstraintSet` class:

```
ckt = ConstraintSet()
```

Now, for each component in the circuit, we specify a constraint relating the voltages on its terminals and the current flowing through it. We have to start, as usual, by thinking up names for each of the node voltages and the currents through the components; those will be the variables in our constraint system.

```
ckt.addConstraint(resistor(ra, ['n1', 'n2', 'ia']))
ckt.addConstraint(resistor(rb, ['n2', 'n1', 'ib']))
ckt.addConstraint(resistor(rd, ['n2', 'n3', 'id']))
ckt.addConstraint(vsrc(vc, ['n2', 'n3', 'ic']))
```

In the first line, we are saying that there is an `ra`-ohm resistor between nodes `n1` and `n2`, with an `ia`-amp current flowing through it, from node `n1` to node `n2`. It is important to label the directions of the currents in your diagram and be sure you use them consistently. Study this example and be sure you understand how it corresponds to the circuit. (Remember that in the figure n_2 and n_4 are really the same node (they're just connected by a wire, and so have the same voltage); so we're just using the name `n2` for both of them here.)

What, exactly, does `resistor(ra, ['n1', 'n2', 'ia'])` mean? If you look in the file `linearCircuitConstraints.py`, you'll see a definition of the function:

```
def resistor(R, x): # R is resistance
    [vn1, vn2, i12] = x
    return [(1.0, vn1), (-1.0, vn2), (-float(R), i12), (0, None)]
```

So, `resistor` is a simple function that returns a linear equation. It consumes a resistance, `R`, and a list of node names (which will be set to the internal variable names `[vn1, vn2, i12]`), and returns `vn1 - vn2 - R * i12 = 0`. The `float` in `-float(R)` is just there to avoid any problems with Python integer arithmetic.

Now, it's time for the KCL constraints. The `kcl` function in `linearCircuitConstraints.py` implements the constraint that the signed sum of currents at a node must equal zero, and the `setGround` implements a constraint forcing a value to zero (typically used to force the reference (ground) node voltage to be zero).

```
ckt.addConstraint(kcl([-1, 1], ['ia', 'ib'])) # n1
ckt.addConstraint(kcl([1,-1,-1, -1], ['ia', 'ib', 'ic', 'id'])) # n2
ckt.addConstraint(setGround('n3'))
```

Remember that a KCL constraint asserts that a bunch of currents sum to zero; so to make a new KCL constraint, you call the function `kcl` with a list of signs, expressed as `1` or `-1`. You should use a `+1` value for currents flowing into the node and a `-1` value for currents flowing out. You also specify a list of variables, in this case, representing the currents flowing into or out of this node, that should satisfy the constraint.

The constraint solver can take a set of n linear equations over n variables and return values for the variables that satisfy all the equations. It does this by calling an equation solver provided in `numpy`.

Finally, we ask the circuit solver to get a solution. A solution is an array (this is a special data type defined by `numpy`, which is different from a list) of values for each of the variables in the constraint system. And then we display it nicely (using the names from the constraint definitions).

```
>>> solution = ckt.solve()
>>> ckt.display(solution)
ia =  -0.0
ib  =  0.0
ic  =  -0.1
id  =  0.1
n1  =  10.0
n2  =  10.0
n3  =  0.0
```

Question 1: Modify `example.py` so to remove the wire connecting nodes n_2 and n_4. Update all the constraints and solve. Which constraints needed to be changed to make this change? Was it easier or harder than removing the wire in the high-level representation from the software lab from week 8?

The abstract model we used for the op-amp was a voltage-controlled voltage source model, and using that model in the inverting amplifier yields the schematic we saw last week (repeated in figure 2)

where the gain of the voltage-controlled voltage source, K, is a very large number. You should add this voltage-controlled voltage source model to the constraint solver, allowing K to be a parameter

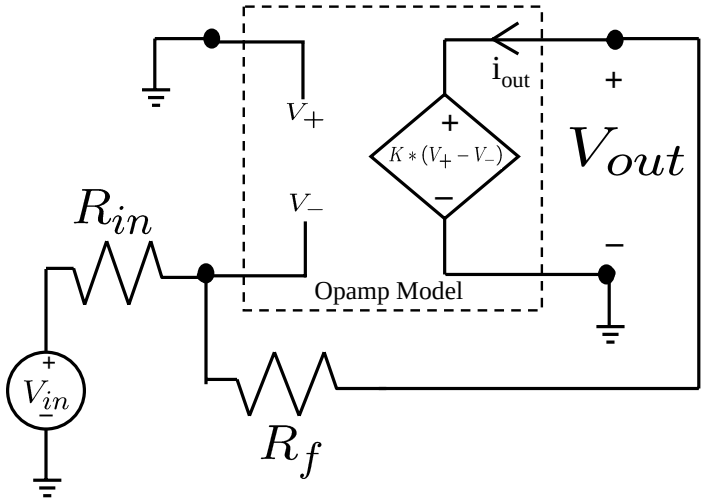

Figure 2: Op Amp model: the voltage-controlled voltage source.

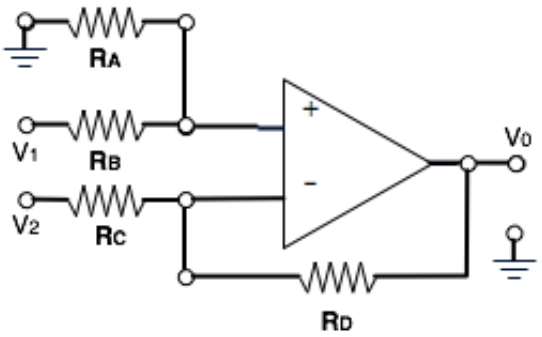

Figure 3: Assume V_1 and V_2 are connected to voltage sources.

like resistance is for resistors. Keep in mind that this model has four terminals, but there is only one nonzero current, i_{out}. Therefore, your constraint should involve four voltages and one current.

Question 2: Implement a constraint function for the voltage-controlled voltage-source model of an op-amp. A skeleton for this function can be found in `linearCircuitConstraints.py`.

Question 3: Note that the output current i_{out} is not used in the op-amp constitutive relations (the same situation occurs with a voltage source). A variable can't exist in our system of equations without any constraints. What constraint will ultimately involve the op-amp current? (Note also that we have said that no current flows **into** the op-amp through the `v+` and `v-` nodes. We don't have to write equations to enforce those constraints; we simply don't introduce those current variables in our model.)

Question 4: Write the constraints for the circuit in figure 3. Pick values of the resistors that give you $V_0 = 2V_1 - 3V_2$. Verify that your solution gives you the expected answer. Be sure you test with values of V_1 and V_2 that demonstrate correctness.

Solving the constraints

Now we need to solve the constraints. That is, we need to find values for the voltages and currents in the circuit that satisfy all of the constraints (linear equations). So, we need to solve a set of simultaneous linear equations.

In general, you will have a set of k linear equations in k unknowns:

$$\begin{aligned} a_{0,0}x_0 + a_{0,1}x_1 + \ldots + a_{0,k}x_k &= c_0 \\ \ldots &= \ldots \\ a_{k,0}x_0 + a_{k,1}x_1 + \ldots + a_{k,k}x_k &= c_k \end{aligned}$$

In `numpy`, there's a handy function to solve sets of linear equations :

```
from numpy import *
x = linalg.solve(A,c)
```

where `A` is a matrix, each of row of which are the coefficients of the linear equations (not including the constant terms) and `c` is a vector of the constant terms.

In numpy, we can build an array and initialize it by giving it a list. To create a vector with 3 elements, do:

```
c = array([0.0, 1.0, 2.0])
```

You can access the elements of this vector as you would a list: `c[1]`, which evaluates to 1.0. To make a matrix with 2 rows and 3 columns, do:

```
A = array([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
```

You can access the elements of the array by: `A[row][column]` or by `A[row,column]`, so `A[1][2]` and `A[1,2]` both evaluate to 6.0. You can change the value of that array element by assignment, such as `A[1,2] = 6.5`.

Question 5: Create an array A and a vector `b` that correspond to the two linear equations in the `linsys` example involving two linear equations. Use `linalg.solve` to find the solution and check it's as expected.

Question 6: Create an array A and a vector `b` that correspond to the constraints in the `circuit.py` file. Start by writing out all of your constraint equations on paper, with every variable in every equation, even if it has a 0 coefficient, and in the same order in each equation. (This is just to see how to set up this type of problem; just type Python expressions along the lines of `A = array([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])`, we'll write a general program later.)

Use `linalg.solve` to find the solution and check it's as expected.

Our strategy for solving circuits will be to scan all the coefficients of the equations from all the circuit constraints and build the matrix `A` and the vector `c` and call `linalg.solve`. Note that we will have to pick an index to correspond to each variable, for example, variable 0 could be i_a, variable 1 could be n_1, variable 2 will be n_2 and so on. A dictionary will serve to keep track of this mapping.

Question 7: We have given you the file `solveLinearConstraintsShell.py` which has a partially written class definition for the `ConstraintSet` class. Read all of the code in that file carefully. Then fill in the indicated sections of the class, in the `addConstraint` and `solve` methods.

Question 8: Test your class definition using at least two of the examples you did earlier, including the subtraction op-amp circuit. You will need to change the `import` statements at the top of the circuit files so that they import `solveLinearConstraintsShell.py`.

Homework due In Design Lab

Question 9: Design and draw the circuit diagram for an op amp circuit to compute $v_1 - v_2$, where v_1 and v_2 are the voltages coming out of the two potentiometers (refer to revised design lab 8 handout for details).

Exploration: More abstract circuit specifications

Solving a circuit this way is a lot easier than using a pencil and paper, but it's still kind of a pain to specify the circuit. We find that we often make mistakes with the signs on the KCL constraints. The fact is that, once we know what the components in the circuit are, and how they're connected together, the entire circuit is specified. In the language we used for last week's lab, we could do the same job as above with a simpler specification:

```
c = Circuit([Resistor(ra, 'n1', 'n2'),
             Resistor(rb, 'n4', 'n1'),
             Resistor(rd, 'n4', 'n3'),
             Wire('n4', 'n2'),
             VSrc(vc, 'n2', 'n3')])

c.solve('n3')
```

We still have to name the nodes, but not the currents. Then, for each component, we just specify its value and its terminals. The order of the terminals still has to agree with the order of the variables in the underlying constraint function.

Use the code from this week's lab to implement a system similar to the one we used last week. Your solution should:

- Provide a convenient way to specify circuits (just saying what the components are and how they are connected together);
- Implement resistors, voltage sources, and op-amps;
- Make it relatively easy to add new components with linear constraints; and
- Display solutions nicely.

Exploration 1:	Hand in your code and demonstrate it on the example in figure 3.
Exploration 2:	Explain what someone would have to do to add a new type of component.