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

Exploration 9, Issued: Thursday, April 10

This exploration is due April 24 or 25 in your design lab.

Option 1: More abstract circuit specifications

This part of the exploration is worth 5 out of 10 points. If you plan to do option 2, you should do it instead of option 1. If you do option 1 in isolation first, you are likely to find that your solution cannot be extended easily to option 2.

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 our job with simpler specifications:

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

c.solve('n3')
```

In this specification, 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 underlying constraint-solving 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, but not specifying currents);
- 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 of software lab 9.
Exploration 2:	Explain what someone would have to do to add a new type of component.

Option 2: Defining and using new primitives

This option is quite difficult. It is worth 10 points. You can do it as an alternative to option 1.

So far, in all of our circuit sepcification methods, we have been able to define primitives and means of combining them. We haven't however, been able to build and use abstractions very effectively during circuit specification. We would like to be able to define the pattern of a voltage divider or an inverting amplifier, and then use it as a component in future circuits we might design.

So, your goal in this exploration is to implement a system that will:

- 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;
- Allow you to take circuits patterns that you have already specified and use them as components in future circuits; and
- Display solutions nicely.

Note that there is an important difference between the generic idea of an inverting amplifier or a buffered voltage divider and any particular instance of one. A particular instance has actual voltages and currents, and you can't put two copies of the same instance in a circuit.

This is a very hard problem that can be approached in several ways. Below, we show you some structures that we built using our solution to this problem. We provide it here to show you the basic idea of what we want, and to give you some idea of one way to go about it. But it is not crucial to approach it this way (for example, you might find it easier or more beautiful to make the program more functional and less object-oriented than we have here).

```
# A voltage divider with resistance r1 on top and r2 on the bottom.
# Needs names of top, middle, and bottom nodes.
class Divider(Circuit):
    def __init__(self, r1, r2, nHi, nOut, nLo):
        # Just make a circuit with two resistors, connected in the
        # right way
        Circuit.__init__(self, [Resistor(r1, nHi, nOut),
                                Resistor(r2, nOut, nLo)])

# Two dividers connected together (in our familiar configuration that
# doesn't divide by 4).  Uses all the same resistance, for simplicity.
class DoubleDivider(Circuit):
    def __init__(self, r, vPlus, vOut, vMinus):
        # Each new instance of this circuit will have a different
        # internal node (the output of the first divider).  So, we
        # have to generate a new name for that node each time
        # through.  (See code for gensym for details).
        middleNode = gensym('divider')
        # Now, make a circuit with two dividers, connected up appropriately.
        Circuit.__init__(self,
                         [Divider(r, r, vPlus, middleNode, vMinus),
                          Divider(r, r, middleNode, vOut, vMinus)])

# A common wiring pattern for an op-amp is a simple follower
```

```
class Follower(Circuit):
    def __init__(self, vIn, vOut):
        Circuit.__init__(self, [OpAmp(vIn, vOut, vOut)])


# A voltage divider with resistance r1 on top and r2 on the bottom.
# Needs names of top, middle, and bottom nodes.  Output of the divider
# is run through a follower.  This is how we made virtual ground.
class BufferedDivider(Circuit):
    def __init__(self, r1, r2, nHi, nOut, nLo):
        # Have to generate a name for the node that is the output of
        # the divider and the input to the follower
        middleNode = gensym('divider')
        Circuit.__init__(self, [Resistor(r1, nHi, middleNode),
                                Resistor(r2, middleNode, nLo),
                                Follower(middleNode, nOut)])

# Connecting two buffered dividers does actually divide by 4
class DoubleBufferedDivider(Circuit):
    def __init__(self, r, vPlus, vOut, vMinus):
        # Have to generate a name for the node that is the output of
        # the first divider and the top of the second divider
        middleNode = gensym('doubleDivider')
        Circuit.__init__(self,
                         [BufferedDivider(r, r, vPlus, middleNode, vMinus),
                          BufferedDivider(r, r, middleNode, vOut, vMinus)])

# Inverting amplifiers are a handy pattern.  We specify input and
# output nodes, and the multiplier we desire
# (so that vOut = multiplier * vIn)
class InvertingAmplifier(Circuit):
    def __init__(self, multiplier, vIn, vOut):
        # Just pick a value for the input resistor
        r = 100.0
        # Need a name for the node that is the minus input to the op amp
        vMinus = gensym('vMinusAmp')
        # Put a resistor on the input and another resistor (r *
        # multiplier) on the feedback path of an op amp
        components = [Resistor(r, vIn, vMinus),
                      Resistor(r*multiplier, vOut, vMinus),
                      OpAmp('gnd', vMinus, vOut)]
        Circuit.__init__(self, components)

# Testing the inverting amplifier
>>> c = Circuit([VSrc(10, 'vIn', 'gnd'),
                 InvertingAmplifier(2, 'vIn', 'vOut')])
>>> c.solve()
[(1, 'vIn'), (-1, 'gnd'), (10, None)]
[(1, 'vIn'), (-1, 'vMinusAmp2'), (-100.0, 'i3'), (0, None)]
[(1, 'vOut'), (-1, 'vMinusAmp2'), (-200.0, 'i4'), (0, None)]
[(1, 'vOut'), (-1, 'gnd'), (-10000, 'gnd'), (10000, 'vMinusAmp2'), (0, None)]
[(1, 'gnd'), (0, None)]
[(1, 'i3'), (1, 'i4'), (0, None)]
[(-1, 'i1'), (-1, 'i3'), (0, None)]
[(-1, 'i4'), (-1, 'iOpAmp5'), (0, None)]
gnd  =  1.46975764892e-15
i1  =  -0.0999800059982
i3  =  0.0999800059982
i4  =  -0.0999800059982
```

```
iOpAmp5  =  0.0999800059982
vIn  =  10.0
vMinusAmp2  =  0.00199940017995
vOut  =  -19.9940017995
```

You might find this strategy for generating brand new names useful:

```
##  Generate new symbols guaranteed to be different from one another
##  Optionally, supply a prefix for mnemonic purposes
class SymbolGenerator:
    def __init__(self):
        self.count = 0
    def gensym(self, prefix = 'i'):
        self.count += 1
        return prefix + str(self.count)

##  Call gensym("foo") to get a symbol like 'foo37'
gensym = SymbolGenerator().gensym
```

Exploration 3: Hand in your code. Show how to define a summer and a differential amplifier. Use a combination of these amplifiers to make a circuit that takes three voltages as inputs and outputs $2 * v_1 + 3 * v_2 - 0.5 * v3$.