MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.01—Introduction to EECS I
Fall Semester, 2007

Assignment 7, Issued: Tuesday, Oct. 16

To do this week

...in Tuesday software lab

1. Start writing code and test cases for the numbered questions in the software lab. Paste all your code, including your test cases, into the box provided in the "Software Lab" (Part 7.1) problem on the on-line tutor. This will not be graded.

...before the start of lab on Thursday

1. Read the lecture notes.
2. **Complete the Tuesday software lab (Questions 1-14) to hand in Thursday in lab.**
3. Read through the entire description of Thursday's lab.

...in Thursday lab

1. Answer the numbered questions (Questions 15-29) in the circuit lab and demonstrate them to your LA.
2. Do the nanoquiz; it will be based on the material in the lecture notes and the on-line tutor problems due on Thursday.

...before the start of lecture next Tuesday

1. Do the lab writeup, providing written answers (including code and test cases) for Questions 15-29 in this handout.

On Athena machines make sure you do:
`athrun 6.01 update`
so that you can get the `Desktop/6.01/lab7` directory which has the files mentioned in this handout.

- You need the file **resolveConstraints.py, example.py, circuit.py, circuit2.py** for the software lab.
- You need the files **circuitConstraints.py, circuitt.py, circuit2t.py, circuitLine.py, circuitG.py, genGrid.py, genKcl.py** for the circuit lab.

During software lab, if you are using your own laptop, download the files from the course Web site (on the Calendar page). Be sure you have `numpy` installed.

The constraints view of circuits

In this software lab you will learn about resistor networks as constraint systems, then build a tool that makes it easy to solve for the voltages in a circuit layout.

Tuesday Software Lab: Circuit Constraints

You will be downloading and using `resolveConstraints.py`, 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. The instance's method `getConstraintEvaluationFunction` returns a function that evaluates how well a supplied set of numerical values for all the instance's variables satisfies all the instance's constraints. By calling the function `resolveConstraints` with this evaluation function, numerical values are determined for the all the instance variables so that all the constraints are satisfied. These constraint-satisfying values of the variables can be printed by calling `ConstraintSet`'s method `display`.

There are two arguments to the function `addConstraint`. The first argument is a constraint evaluation procedure that takes a list of numerical values for the particular constraint's variables and returns a floating point number that indicates how far the constraint is from being satisfied. The constraint evaluation procedure should return zero if the constraint is exactly satisfied. The second argument to `addConstraint` is a list of strings that are labels for the variables used in the constraint. The order of these variable labels is **important**. The order of the labels should match the order of numerical values in the list used by the constraint evaluation procedure.

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

$$5 * v - 2 * y - 3 = 0$$

and

$$3 * v + 4 * y - 33 = 0.$$

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

To use the constraint class and the function `resolveConstraints` to find the constraint-satisfying values of v and y, first define the two constraint evaluation procedures:

```
def firstEqn():
    # Enforces 5*v - 2*y - 3 = 0, assumes the input is [v,y]
    return lambda x : 5.0*x[0] - 2.0*x[1] - 3.0

def secondEqn():
    # Enforces 3*v + 4*y - 33 = 0, assumes the input is [y,v]
    return lambda x : 3.0*x[1] + 4.0*x[0] - 33.0
```

Second, create an instance of `ConstraintSet` and add the two constraints, being careful to provide variable labels in the same order as used in the constraint procedures:

```
linSys = ConstraintSet()
linSys.addConstraint(firstEqn(),['v', 'y'])
linSys.addConstraint(secondEqn(),['y', 'v'])
```

Finally, call `resolveConstraints` and display the solution:

```
solution = resolveConstraints(linSys.getConstraintEvaluationFunction())
linSys.display(solution)
```

In order to use `resolveConstraints` for circuit problems, one needs an organized approach for generating the variables and constraints for a circuit. In class we discussed the nodal approach for accomplishing this task. The steps in the nodal approach were

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 the element's currents to the voltages at the element's terminals.
3. For each circuit node, except the reference node, write a conservation law, also known in circuit analysis as Kirchoff's current law (KCL). 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 resolver to solve circuit problems, it is helpful to have functions which return constraint procedures associated with a circuit's constitutive equations and conservation laws. In the file `circuitConstraints.py`, you will find functions for generating procedures that implement circuit related constraints. Consider two of these procedure generating procedures: `resistor` and `vsrc`. These two functions return procedures that implement the constraints associated with the the constitutive relations of a resistor and a voltage source, respectively. To better understand the `resistor` function, recall that if a current $i_{1,2}$ is flowing from n_1 to n_2 through an R-ohm resistor, then v_{n_1}, v_{n_2} and $i_{1,2}$ must satisfy the constraint

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

Note that the way we have defined the constraint, if $v_{n_1} > v_{n_2}$ then $i_{1,2} > 0$ and current is flowing from n_1 to n_2. If $v_{n_1} < v_{n_2}$ then $i_{1,2} < 0$, and even though $i_{1,2}$ is still defined as the current flowing from n_1 to n_2, the physical current is really flowing from n_2 to n_1.

Similarly, if a current $i_{1,2}$ is flowing from n_1 to n_2 through a V-volt voltage source, then v_{n_1}, v_{n_2} and $i_{1,2}$ must satisfy the constraint

$$v_{n_1} - v_{n_2} - V = 0 \ .$$

Note that for a voltage source, the source current does not appear in the constraint because an ideal voltage source will produce whatever current is necessary to maintain the source voltage. Nevertheless, as we will see later, it is helpful to be consistent about current directions. So try to maintain the practice of defining $i_{1,2}$ so that $i_{1,2} > 0$ implies that the physical current is flowing from n_1 to n_2.

Consider the following example circuit specification, corresponding to the first example in the lecture notes (reproduced here in figure 1). You can find this circuit specification in the file `example.py`.

We start by specifying values for the resistors and the voltage source. Using names here makes it easier to modify the resistor and voltage source values.

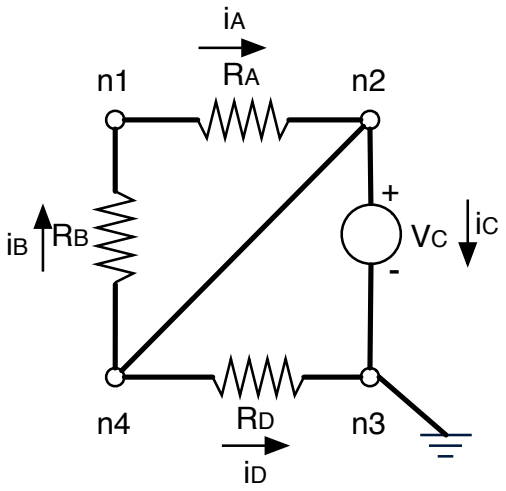

Figure 1: Circuit with three resistors and a voltage source.

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

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

```
ckt = ConstraintSet()
```

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

```
ckt.addConstraint(resistor(ra), ['vn1', 'vn2', 'ia'])
ckt.addConstraint(resistor(rb), ['vn2', 'vn1', 'ib'])
ckt.addConstraint(resistor(rd), ['vn2', 'vn3', 'id'])
ckt.addConstraint(vsrc(vc), ['vn2', 'vn3', 'ic'])
```

In the first line, we are saying that there is an `ra`-ohm resistor between nodes `n1` (with voltage `vn1`) and `n2` (with voltage `vn2`), with an `ia`-amp current flowing through the resistor from `n1` to `n2`. As noted above, it is important to label the directions of the element currents in your diagram and use the variable labels in the order that maintains consistency with the constitutive relations. 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)` mean? If you look in the file `circuitConstraints.py`, you'll see a definition of the function:

```
def resistor(R):
    # assumes a list or tuple x = [vn1, vn2, i12]
    return lambda x : x[0] - x[1] - float(R)*x[2]
```

So, `resistor` is a higher-order function. It consumes a resistance, `R`, and generates a function that expresses a constraint. In this case, the generated function that takes a list of three values `[vn1, vn2, i12]`, and returns `vn1 - vn2 - R * i12`. The returned value will be 0 when the voltages and the current have the correct relationship to one another; that is, when the constraint associated with this resistor constitutive relation is satisfied.

Note that the constraint resolver can take a set of `n` functions over `n` variables and return values for the variables so that each of the `n` functions will return 0, assuming, of course, that a solution exists. Well, to be perfectly honest, the constraint resolver can fail to find a solution even if one exists. The resolver uses a form of Newton's method, one that applies to multiple equations multiple unknowns, and Newton's method can fail. But not for the circuits you will be using.

Question 1. Describe what the `wire` function in `circuitConstraints.py` does.

Now, it's time for the conservation law, or KCL, constraints. The `kcl` function in `circuitConstraints.py` returns a procedure which implements the constraint that the signed sum of currents must equal zero,

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

Remember that a KCL constraint asserts that at each circuit node, the sum of incoming currents must equal the sum of outgoing currents. So, to make a new KCL constraint, you call the function `kcl` with a list of signs, expressed as `1` or `-1`, that indicate whether a current is leaving or entering a node. The standard sign convention is that `+1` is used for currents flowing out of a node and `-1` is for currents flowing in to a node. So, when you make a KCL constraint, and then you specify a list of variables, in this case, representing the currents flowing into or out of this node, that should satisfy the constraint.

We use the `setGround` function to return a procedure which implements a constraint forcing a value to zero,

```
ckt.addConstraint(setGround, ['vn3'])
```

and is typically used to force the reference node voltage to be zero.

Question 2. What would happen if you changed all the `-1`'s in all your KCL constraints into `+1`'s, and vice versa?

Question 3. What would happen if you flipped the signs on just one KCL constraint?

Question 4. Should you write a KCL constraint for the ground, or reference, node?

Finally, we ask the circuit to export its constraints, and hand these to our constraint resolver 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 = resolveConstraints(ckt.getConstraintEvaluationFunction())
ckt.display(solution)
```

Question 5. Run the above example through Python and be sure you get the same answer as in the notes. If any of the currents are zero, explain why.

Question 6. Modify this example to get rid of the wire between n_2 and n_4. Now what values do you get?

Question 7. If we had the physical circuit rather than a computer model (the circuit without the wire between n_1 and n_2), and flipped resistor `ra` around so its terminals were connected the other way, would that change any voltages in the circuit?

Question 8. If you were to change the line

```
ckt.addConstraint(resistor(ra), ['vn1', 'vn2', 'ia'])
```

to

```
ckt.addConstraint(resistor(ra), ['vn2', 'vn1', 'ia'])
```

before calling the constraint resolver, would the generated solution change? Are there any changes you could make to the KCL constraints so that the solution produced by the constraint resolvers would be exactly the same as before? Be sure to explain any resulting differences in you see in computed currents.

Question 9. If we were to take the voltage source in the same physical circuit, pull it out, and turn it around so that the terminals were connected the other way, would that change the voltages in the actual circuit?

Question 10. If you were to change the line

```
ckt.addConstraint(vsrc(vc), ['vn2', 'vn3', 'ic'])
```

to

```
ckt.addConstraint(vsrc(vc), ['vn3', 'vn2', 'ic'])
```

before calling the constraint resolver, would the generated solution change? Are there any changes you could make to the KCL constraints so that the solution produced by the constraint resolvers would be exactly the same as before?

Question 11. Run the example described in the file file `circuit.py`, and report the results. Draw a circuit diagram of the circuit described in that file.

Question 12. In the file `circuit2.py`, there's an error. Try running the circuit and explain the resulting error message. Then, fix the error, and draw the circuit diagram.

Add the non-ideal voltage source

The voltage source is an idealized model of a battery. All real batteries have some internal resistance, and this resistance restricts the current that can be provided by the battery. We can model a real battery by creating a more complicated circuit in which we add a resistor in series with an ideal voltage source, but one can also generate a single non-ideal voltage source constraint.

Question 13. Add a function for generating constraints corresponding to non-ideal voltage sources to the file `circuitConstraints.py`.

Question 14. Use your new non-ideal voltage source to reduce the total number of constraints in the `circuit2.py` file. Demonstrate that you still get the same solution.

Go to the on-line Tutor at `http://sicp.csail.mit.edu/6.01/fall07`, choose PS7, and paste the code that you wrote during lab, including your test cases, into the box provided in the "Software Lab" problem. Do this even if you have not finished everything. Your completed answers to these questions are to be handed in **AT THE BEGINNING OF THURSDAY'S LAB.**

If you have extra time, you can start on the Exploration below or start reading about Thursday's lab.

Exploration: Nonlinear resistor

You might try adding a nonlinear resistor constraint to the system (one in which the current is a nonlinear function of the voltage). One example nonlinear resistor has a current-voltage constraint given by

$$R_0 i_R - (v_{n_1} - v_{n_2}) - \beta (v_{n_1} - v_{n_2})^3 = 0.$$

where β is typically less than one, and larger values of β correspond to more nonlinear resistors. Can you find an interesting circuit using such a resistor?

Thursday Lab: Automating the circuit specifications

On Tuesday, we experimented with a constraint resolver that would let us specify the constraints associated with a circuit and then solve for the current and voltage values. Here's the example we did in detail, to refresh your memory.

```
ckt = ConstraintSet()
ckt.addConstraint(resistor(ra), ['vn1', 'vn2', 'ia'])
ckt.addConstraint(resistor(rb), ['vn2', 'vn1', 'ib'])
ckt.addConstraint(resistor(rd), ['vn2', 'vn3', 'id'])
ckt.addConstraint(vsrc(vc), ['vn2', 'vn3', 'ic'])

ckt.addConstraint(kcl([1, -1]), ['ia', 'ib'])  # n1
ckt.addConstraint(kcl([-1,1,1, 1]), ['ia', 'ib', 'ic', 'id'])  # n2
ckt.addConstraint(setGround, ['vn3'])

solution = resolveConstraints(ckt.getConstraintEvaluationFunction())
ckt.display(solution)
```

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. It would be very helpful if we could automatically generate the KCL constraints, but there is a problem. If we wish to allow ourselves to write constitutive equations in any way we'd like, and allow ourselves the flexibility to introduce current and voltage variables in any we please, then we will have to specify the KCL equations. The problem is that, in general, constitutive equations do

not necessary contain enough information for a program to figure out the direction of the currents, or even which terminals the currents enter and leave.

We can, however, decide to be disciplined about how we write constitutive equations so that the KCL equations can be generated automatically. For elements with two terminals, this discipline is quite natural. For example, consider when we use the `resistor` function in `circuitConstraints.py` to add a resistor constitutive constraint, as in

```
ckt.addConstraint(resistor(ra), ['vn1', 'vn2', 'ia'])
```

The two-terminal resistor constraint takes arguments in a specified order with a specified meaning. This two-terminal element is connected between nodes `n1` (with voltage `vn1`) and `n2` (with voltage `vn2`), and there is an `ia`-amp current flowing through the element from `n1` to `n2`. We therefore know that the KCL equation for `n1` will include variable `ia` with a $+1$ sign (since the current is leaving node 1), and the KCL equation for `n2` include the variable `ia` with a -1 sign (since the current is entering node 2).

If we decide that we would like to use almost all of the mechanism in the constraint resolver, but would like to optimize for a special case, a nice mechanism is to use a derived class in Python. Our special case is that we will only consider two-terminal elements, and in particular, two-terminal elements where the constitutive equations have been specified in a disciplined style. That is, the constitutive equations must be written so that the three variables are, in order, the voltage at the node sourcing the current, the voltage at the node sinking the current, and finally the current. If we know this, then every time we add a constitutive constraint, we can store the information needed to eventually generate the KCL constraints after all the constitutive constraints have been specified.

In order to write such a program, consider generating a class derived from `ConstraintSet`,

```
class ConstraintSetKCL(ConstraintSet):
```

and write a couple of methods that will be appended to the already existing methods of `ConstraintSet`. In particular, you will need to write a method

```
addConstraint2T(self, f, variables)
```

which will take constraints written in the disciplined form and save away information so that the KCL equations can be generated later. Keep in mind that you can call the original method in `ConstraintSet` from your new method by using the statement

```
self.addConstraint(f,variables)
```

because this is a derived class.

The method `addConstraint2T()` will have to keep track of contributions to KCL equations for all the nodes in the circuit. Since you do not know all the node names before the first call to `addConstraint2T()`, you will also have to write a method for your derived class that will actually generate all the KCL constraints once all the constitutive constraints have been specified, as in

```
genKcls(self, referenceNode)
```

Note that we have included the ground or reference node in the argument list for generating the KCL equations. No KCL constraint should be generated for the reference node.

In order to get you started, we have generated a file in which to put your derived class `genKcl.py` and started the class definition. We have also converted the two example files, `circuit.py` and

`circuit2.py`, to versions which use the automatic approach to generating the KCL equations. These converted versions are `circuitt.py` and `circuit2t.py`. Your derived class should work with these two example files (after you correct the problem with `circuit2t.py` that you corrected for `circuit2.py`).

To better understand what we expect your derived class to do, consider the following snippet from `circuitt.py`. The KCL equations have been commented out, constitutive equation constraints are generated by calling `addConstraint2T` (which you will add to the derived ConstraintSetKCL class), and `genKcls` is called with the reference node to generate KCL equations for all but the reference node.

```
ckt = ConstraintSetKCL()
ckt.addConstraint2T(resistor(20), ['vn1', 'vn2', 'ir'])
ckt.addConstraint2T(vsrc(5), ['vn1', 'vn2', 'is'])
ckt.addConstraint(setGround, ['vn2'])
#ckt.addConstraint(kcl([1,1]), ['ir', 'is'])
ckt.genKcls('vn2')
```

Question 15. Describe your code for the derived class `ConstraintSetKCL`.

Question 16. Demonstrate that your program works with our test examples above.

Question 17. Generate a `circuitfig1.py` file which uses your program to solve the circuit in figure 1, without the wire.

Question 18. In version of the circuit in figure 1 without the wire, what would happen if we changed `addConstraint2T(resistor(ra), ['vn1', 'vn2', 'ia'])` to `addConstraint2T(resistor(ra), ['vn2', 'vn1', 'ia'])`? Explain why.

Question 19. Also in version of the circuit in figure 1 without the wire, what would happen if we changed `addConstraint2T(vsrc(vc), ['vn2', 'vn3', 'isrc'])` to `addConstraint2T(vsrc(vc), ['vn3', 'vn2', 'isrc'])`

Practice with circuits

Use the software you built to help answer the following questions. For each one, submit the the circuit you constructed and the output it generated when you solved it. Show which values in the output correspond to answers to the questions we ask below.

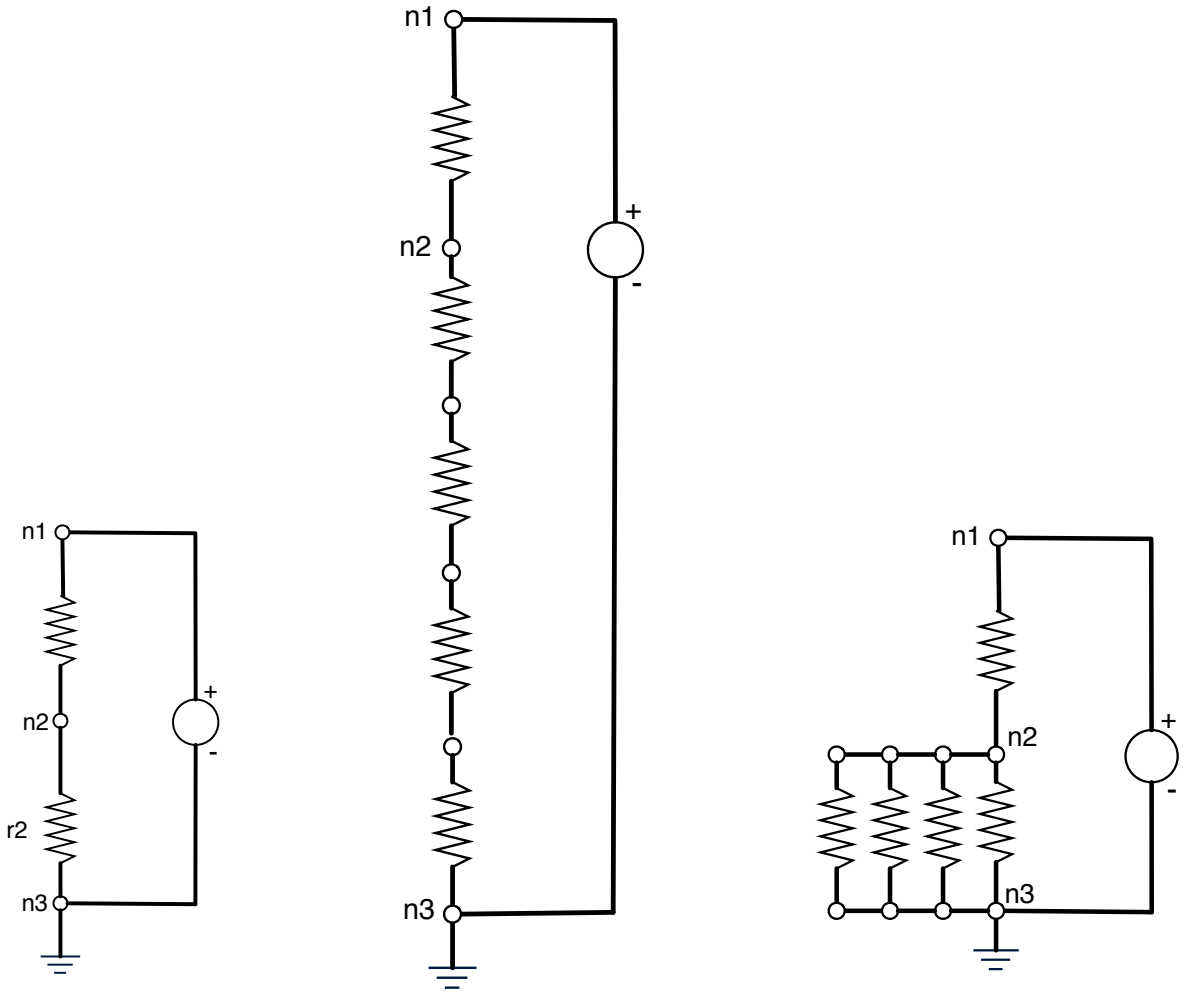

Figure 2: Circuits for Thursday Lab questions

Question 20. Figure 2(a) shows a circuit with two 100Ω resistors and a 10V source. What is the voltage at node n_2?

Question 21. Figure 2(b) shows the circuit in `q1`, but with four 100Ω resistors, wired "in series" in the place of r_2. What is the voltage at node n_2 in this circuit?

Question 22. How could you range the resistance of r_2 in the circuit in figure 2(a) so that the voltage at node n_2 would be the same as it is in the circuit in figure 2(b)?

Question 23. Figure 2(c) shows the circuit in `q1`, but with four 100Ω resistors, wired "in parallel" in the place of r_2. What is the voltage at node n_2 in this circuit?

Question 24. How could you range the resistance of r_2 in the circuit in figure 2(a) so that the voltage at node n_2 would be the same as it is in the circuit in figure 2(c)?

Resistor Lines

There are a variety of applications where it is important to be able to determine if there are conductivity changes in an interior, based on measurements at the periphery. Such problems arise in ground water exploration and medical diagnosis and are referred to generically as inverse conductivity problems. In this problem you will investigate a resistor line model of such problems, but if you prefer, you can work on the more realistic (and more challenging) explorations problem below.

Again, if you wish to work on the explorations problem, you may skip this problem.

We have written a program for generating resistor networks, `genGrid.py`, and along with two circuit files, `circuitLine.py` and `circuitG.py`. The `circuitLine.py` file generates a circuit as shown in figure 3

Use the `genGrid.py` and `circuitLine.py` files to answer the following questions. Note, in answering the questions below, you might find it helpful to think about the resistor line with one column (see below) case.

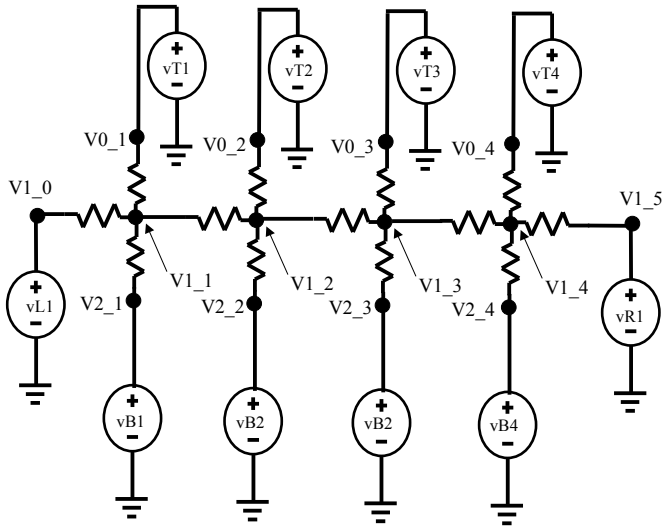

Figure 3: Resistor Line circuit generated by `circuitLine.py`.

Question 25. Modify `genGridResistors` in `genGrid.py` so that the horizontal and vertical resistors can be set to different values

Question 26. Use your modified version of the grid generator, and modify `circuitLine.py` so that it generates the solution to a grid with one row, 4 columns, top and bottom voltages at zero, left voltage at one, right voltage at zero, one ohm horizontal resistors and 10,000 ohm vertical resistors.

Question 27. Describe an accurate way of approximating the right and left voltage source currents generated by the above problem.

Question 28. Suppose only one of the horizontal resistors is set to 10 ohms, the rest of the horizontal resistors are still one ohm and the the vertical resistors are still 10,000 ohms. Is there a best location for the increased horizontal resistor if one wishes to minimize the left source current?

Question 29. Suppose only one of the horizontal resistors is set to 10 ohms, the rest of the horizontal resistors are still one ohm and the the vertical resistors are now 10 ohms. Is there a best location for the increased horizontal resistor if one wishes to minimize the left source current?

Explorations:

As mentioned above there are a variety of applications where it is important to be able to determine if there are conductivity changes in an interior, based on measurements at the periphery. In this exploration, you will investigate a model of such inverse conductivity problems by developing an algorithm for finding which resistor has changed value in 10×10 grid of nominally one ohm resistors, like the one generated by `circuitG.py`.

Suppose one of the resistors in the 10×10 grid has been increased to ten ohms, but you do not know which one. Can you develop an algorithm to determine which resistor was modified by setting the peripheral voltage sources and then measuring `only` the voltage source currents?

To help you think about how to develop an algorithm, consider the following idea.

Suppose the left voltage sources are all one volt, the right voltage sources are all zero volts, and the horizontal resistors are all one. What should the top and bottom voltage sources be so that the left and right voltage source currents are independent of the values of the vertical resistors? What happens when you use the voltage values you just found, but apply them to the case where just one of the horizontal resistors is ten ohms (all the other vertical and horizontal resistors are one ohm). Is there a simple way to tell which row contains the ten ohm resistor from the left and right voltage source currents?

Post-Lab Writeup for Thursday's labs: Due before lecture on Tue. October 23rd.

The Thursday lab solutions should be written in clear English sentences and paragraphs and we expect at least a couple of sentences or a paragraph in answer to **all** of the numbered questions in this lab handout. We also want the code and test examples that you wrote for Thursday's lab. Give us only the code you wrote, don't hand in the code we gave you.

Concepts covered in this assignment

Here are the important points covered in this assignment:

- Learn how to formulate equations for circuits using constitutive relations and conservation laws.
- Understand the issues associated with current directions and KCL equations.
- Gain some practice with circuit equations.