MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall Semester, 1989

Problem Set 5

Due: In recitation, Wednesday 25 October

Reading: Finish chapter 2, then read through section 3.1.

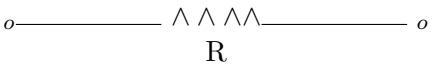
Quiz I Reminder

Quiz I is on Thursday, October 19, in 50-340 (Walker Memorial Gymnasium), from 5-7 PM xor 7-9 PM. You may take the quiz during either of the two sessions, but students taking it in the first session must stay until 7 PM. The quiz is open book.

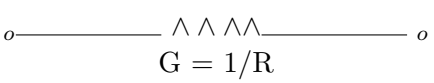
The quiz covers everything up to and including recitation Friday, October 6 (symbols and quotation).

Series-Parallel Resistive Networks

Louis Reasoner, appalled by his tuition bill, is trying to put his computer background to good use by writing an electrical circuit analyzer program that he will rent out to 6.002 students. Louis's partner in this scheme is his roommate, Lem E. Tweakit, a 6-1 major with a flair for hardware. Lem suggests that they begin with a program that computes resistances of simple "linear resistive networks."[1] These are constructed from primitive elements called *resistors*. A resistor is characterized by a number R called the *resistance* of the resistor. It can be depicted as follows:

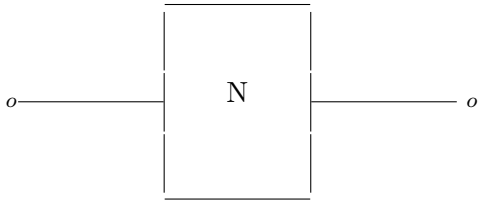


For computational purposes, it is also convenient to consider the *conductance* of a resistor, which is defined to be the reciprocal of the resistance. (Conductance is traditionally denoted by the letter G.)

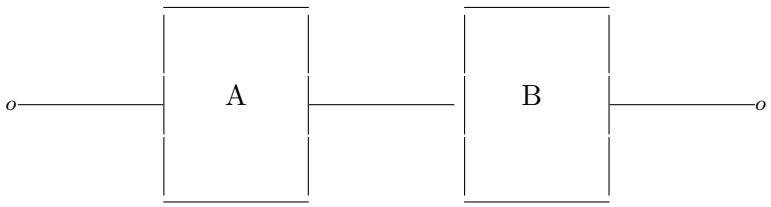


A resistor is the simplest example of a kind of element called a *two-terminal network*, i.e., a network that has exactly two terminals to which other objects can be connected:

[1] In this problem set, we use the words "network" and "circuit" interchangeably. In the context of Louis's program, there is no distinction.



Networks can be combined by attaching their terminals together. A network has a resistance (and conductance) that is determined by the resistances of its parts and the ways in which they are interconnected. Louis's and Lem's initial system will provide two basic methods of combination for constructing two-terminal networks from simpler two-terminal networks. The first method is called *series* combination:



$$R = R_A + R_B$$

The combined resistance of two networks connected in series is the sum of the resistances.

The second method is *parallel* combination:

$$G = G_A + G_B$$

The combined conductance of two networks joined in parallel is the sum of the conductances of the pieces.

After a few hours of work, Louis and Lem have quite an elegant program that handles such series and parallel combinations.[2] The program is provided in the first appendix to this handout, labeled *ps5-res.scm*.

[2]Networks that can be constructed as series and parallel combinations of primitive two-terminal elements are called *series-parallel networks*.

You should read through this code now to get some idea of how it works. There are constructors *make-resistor*, *make-series*, and *make-parallel*, and an operation called *resistance* that computes the resistance of a network by finding the resistances of the parts of a network and combining them appropriately.

Louis and Lem test their program by computing the resistance of the following network:

They type:

```
(define r1 (make-resistor  5))
(define r2 (make-resistor 15))
(define r3 (make-resistor 10))
(define r4 (make-resistor 10))
(define N
  (make-parallel (make-series r1 r2)
                 (make-series r3 r4)))
-->(resistance N)
10.000
```

Indeed, the answer is correct (as Lem verifies).

Exercise 1: Consider the data structure representing the network *N*.

a. Draw the box-and-pointer structure that represents the object *N*.

b. How many times was *resistance* called in computing the resistance of *N*? What were the arguments to *resistance* for each call? (To specify the arguments, you can sketch the part of the network that each argument represents, or you can give a description in English, e.g., "*R3* in parallel with *R4*".)

L-extensions

Given a 2-terminal network B, we can form a new two-terminal network by attaching an "L-section," which is itself constructed from two two-terminal networks S and P as follows:

This operation is called "L-extension" of a base B by a "series part" S and a "parallel part" P.

Exercise 2: Write a procedure *L-extend* that takes three two-terminal networks – *base*, *series-part*, and *parallel-part* – and combines them using *make-series* and *make-parallel* to produce the extended network as shown above.

Exercise 3: Use your *L-extend* procedure to make a new network that extends the network N from Exercise 1 (as the base) by a 5-ohm resistor (the series part) and a 10-ohm resistor (the parallel part).

Verify that the resulting network has a resistance of 10 ohms. Show the expressions that you typed in order to generate the network and to check its resistance.

By repeatedly L-extending a base by a given series and parallel part, we obtain a circuit called a "ladder." The following diagram shows a 4-stage ladder.

Louis decides that he can easily implement a ladder by making use of his *L-extend* procedure, together with the *repeated* procedure below. Note that *identity* is a procedure that simply returns its argument, and *compose* is a procedure that, given two procedures f and g as arguments, creates a new procedure that applies the composition of f and g to its argument.

```
(define identity (lambda (x) x))
(define compose (lambda (f g)
                  (lambda (x) (f (g x)))))
(define (repeated f n)
  (if (= n 0)
      identity
      (compose f (repeated f (- n 1)))))
```

He defines the following procedure to construct a ladder with a given number of stages:

```
(define (ladder-extension stages base series-part parallel-part)
  ((repeated <EXP-1> stages) <EXP-2>))
```

Exercise 4: Complete the *ladder-extension* procedure. What are the missing expressions <*EXP-1*> and <*EXP-2*>?

Exercise 5: A classical network-theory problem used to plague 6.002 students is to compute the resistance of an infinitely long ladder of 1-ohm resistors:

The procedure `make-tolerance-check` below creates a procedure of two parameters which checks whether those parameters are within `tolerance`:

```
(define make-tolerance-check
  (lambda (tolerance)
    (lambda (number1 number2)
      (< (abs (- number1 number2)) tolerance))))
```

Now, fill in the missing parts of the procedure `solve-infinite-ladder` below, which takes a `tolerance` and calculates the resistance of longer and longer ladders until it finds two answers within `tolerance` of each other. The missing parts are labaled `<exp1>` and `<exp2>`.

```
(define (solve-infinite-ladder tolerance)
  (let ((close-enough? (make-tolerance-check <exp1>))
        (one-ohm-resistor (make-resistor 1)))
    (define (loop circuit circuit-resistance)
```

```
    (let ((new-circuit (l-extend circuit
                                 one-ohm-resistor
                                 one-ohm-resistor)))
      (let ((new-resistance (resistance new-circuit)))
        (if <exp2>
            new-resistance
            (loop new-circuit new-resistance)))))
  (loop one-ohm-resistor (resistance one-ohm-resistor))))
```

Include the expressions you used for `<exp1>` and `<exp2>` in your problem set solutions.

Using your completed `solve-infinite-ladder`, calculate the resistance of an infinite ladder of one-ohm resistors within a tolerance of 0.0001. To what value does the resistance appear to converge? Have you seen this number before?

Exercise 6: Louis and Lem demonstrate their system to Ben Bitdiddle. As an example, they try to compute the resistance of a long ladder. To their surprise, they find that although the program gets correct answers, it seems to run more slowly than before.

Taking a careful look at the code, they find that Louis has changed the definition of *resistance-parallel* so that it now reads:

```
(define (resistance-parallel ckt)
  (/ (* (resistance (left-branch ckt))
        (resistance (right-branch ckt)))
     (+ (resistance (left-branch ckt))
        (resistance (right-branch ckt)))))
```

Explain why this change makes the program run so slowly. As an example, consider a ladder where each of the series, parallel, and base pieces is a simple resistor. If the ladder has N stages:

(a) How many resistors are contained in the ladder?

(b) In computing the resistance of the ladder, how many times will the procedure *resistance-resistor* be run if the system uses the original version of the *resistance-parallel* procedure?

(c) After Louis installs the new version of *resistance-parallel*, how many times will *resistance-resistor* be run in computing the resistance of the N-stage ladder? (You should be able to obtain an exact answer in terms of N. However, partial credit will be given for good partial answers. Show your work.)

(d) How has Louis's change affected the running time of the system? (E.g., slowed it down by a constant factor? slowed it down quadratically? slowed it down exponentially?)

General Series-Parallel Circuits

Louis goes off to try to rent out the system in the 6.002 lab in building 38. His first potential customer is Anna Logg, a sophomore who is busily doing her 6.002 problem set. "Foo," says Anna, "this is useless unless it can handle capacitors and inductors as well as resistors."

Louis goes back home and asks Lem for a short course on circuit theory. Lem says that indeed, interesting circuits have elements called capacitors and inductors, and solving such circuits involves (ugh!) differential equations, and Lem is too tired to give a course in circuit theory (Louis has to take 6.002 next semester anyway). But luckily, Lem explains, it is unnecessary to understand this in order to make a useful system. In particular, explains Lem, capacitors and inductors can be

thought of as "resistors" whose "resistance" (actually called *impedance*) is a complex number that varies with a parameter s, called the "complex frequency."

Lem says that for a capacitor of capacitance C (measured in farads) the "resistance" is $1/sC$. For an inductor of inductance L (measured in henrys) the "resistance" is sL:

For example, a so-called "parallel resonant circuit" can be thought of as a network of 3 "resistors" and analyzed in the same way as before. Only this time, the "resistance" of the network will be a function of s.

Louis meditates on this for a few days and turns for help to the great wizard Alyssa P. Hacker. Alyssa suggests a clever scheme that allows Louis to handle these things without greatly changing his program. The idea is that the "resistance" of a circuit (or a circuit element), instead of being a simple number, will now be a procedure that takes some s as argument. For instance, the "resistance" of a (real) resistor of declared resistance R is represented by a procedure that computes a constant function whose value is R, for any s. For a capacitor of capacitance C, the "resistance" is the procedure that takes s as argument and returns $1/sC$.

The data structure that Alyssa suggests for representing a primitive electrical element is a list of three elements

```
(<type> (<parameter name> <value>) <procedure>)
```

This includes a *type* as in Louis's original program, a "documentation field" that makes printouts of circuits easier to interpret, and the actual "resistance" procedure of the element.

Thus, a (real) resistor is constructed by

```
(define (make-resistor resistance)
  (attach-type 'resistor
               (list (list 'resistance resistance)
                     (lambda (s) resistance)))))
```

A capacitor is constructed by

```
(define (make-capacitor capacitance)
  (attach-type 'resistor
               (list (list 'capacitance capacitance)
                     (lambda (s)
                       (/ 1 (* capacitance s))))))
```

(Note that, as far as the *type* is concerned, a capacitor is a type of resistor, although we can tell the difference by looking at the documentation field.) Inductors are constructed similarly.

Then, for example, the "resistance" of a series combination should be a procedure that, given an argument s, computes the resistances of the branches for that s, and returns their sum:

```
(define (resistance-series ckt)
  (lambda (s)
    (+ ((resistance (left-branch ckt)) s)
       ((resistance (right-branch ckt)) s))))
```

Actually the procedures given above are not quite correct, because in order to use this idea to get information about circuits, one needs to use inputs s that are complex numbers. (Ask Lem if you want to know why, or take 6.002.)

So Louis has to make a further modification: All arithmetic must be changed so that it works with complex numbers. He builds a representation for complex numbers and complex number operations and modifies the code accordingly. The result of his labors (which works, amazingly!) is in the file *ps5-imp.scm*. This is attached as the second appendix to this problem set. You should study the code and use it to do the following exercises.

Exercise 7 Load Louis's new program into Scheme.[3] Build a parallel resonant circuit (as shown above) using a 2-ohm resistor, a 1-farad capacitor, and a 1-henry inductor. Call the circuit *N2*. What is the impedance ("resistance") of *N2* at $s = 2 + 0j$?, at $s = 0 + 2j$? (Note that Louis's code contains a constructor that makes complex numbers of the form $A + Bj$, where j is the square root of -1.) Show the expressions you evaluated in order to generate the circuit and to find the impedance.

Lem has suggested that it is very useful to look at a plot of the magnitude of the impedance of a circuit as a function of imaginary values of the complex frequency s (i.e., values of s for which the real part is 0). Such a plot is called a *frequency response* plot of the circuit.

We have supplied a program that plots functions on the Chipmunk screen. To get an idea of how the program works, type *(line-plot square -2 2 .1)*. This plots values of *square* in the interval from -2 to 2 with an increment of 0.1. Notice that *line-plot* prints the x and y bounds of the plot, in the format *(xmin xmax ymin ymax)*. (The file *ps5-graph.scm*, automatically loaded when you load the problem set, implements the *line-plot* procedure. You need not read or understand the code in the file in order to use *line-plot*.)

[3] A version of this file should have been copied to your disk and into an Edwin buffer when you invoked the "load problem set" command.

Exercise 8 Make a plot of the frequency response of the circuit *N2* from $s = 0+.001j$ to $s = 0+2j$, stepping s by $.05j$. (Note that the *ps5-imp* file contains a *magnitude* procedure for computing magnitudes of complex numbers.) You should get a curve like the one shown in the figure. The sharp peak is characteristic of a "resonant circuit," and the value of s where the peak occurs is called the *resonant frequency*. Where is the resonant frequency for the circuit *N2*? What expressions(s) did you type to Scheme to generate the plot?

Exercise 9 Use Louis's program to determine how the frequency response changes if you change the resistance of the resistor from 2 to 10 ohms. Sketch the new frequency-response curve, indicating how it differs from the original curve. Does the resonant frequency change? Does the height of the peak change?

NOTE: Everything after this point is OPTIONAL.

Generating Circuits

Louis's program is so successful that it becomes standardly used by almost all students taking 6.002. Eventually, the Course 6 faculty realizes that students are doing circuit-analysis homework simply by using the program, resulting in the shocking state of affairs that most students are able to spend less than 30 hours per week doing 6.002 problem sets! At an emergency meeting of the Course 6 Undergraduate Educational Policy Committee, Prof. Gould suggests that the 6.002 staff can increase the difficulty of the problem sets by asking students to *design* circuits rather than simply to analyze them: Students will be given a list of parts (resistors, capacitors, and inductors)

and asked to use these to build a circuit that has some desired behavior, such as resonance in a given frequency range.

When Louis hears of this plan, he decides to augment his program so that it can do this kind of problem as well. His idea is to write a program that will randomly generate series-parallel circuits using the specified parts, then analyze the circuits to find ones with the desired behavior. (Each of the specified parts will be used exactly once in each circuit.)

We can generate a random circuit as follows: Suppose that S is the set of parts we are supposed to use. We split S (randomly) into two subsets A and B. Then, recursively, we generate a random circuit CA using the parts in A and a random circuit CB using the parts in B. Finally, we return either the series combination of CA and CB or the parallel combination of CA and CB (each with probability 1/2).

Exercise 10: Write a procedure *random-circuit* that uses this strategy to generate random series-parallel circuit. *Random-circuit* should take as argument a set of parts, represented as a list. You should use a separate procedure called *split-list*, which takes as argument a list S and returns a pair of lists A and B, such that each item in S appears either in A or in B (with equal probability). *Random-circuit* is then a recursive procedure that uses *split-list*. (Think about how to stop the recursion: What do you do if one of the lists A or B is empty, or has only one element?) Turn in a listing of your procedure(s).

Exercise 11: Use your program to generate five different circuits whose parts are a 1-farad capacitor, a 2-farad capacitor, a 1-henry inductor, a 1-ohm resistor, and a 2-ohm resistor. Turn in the (pretty-printed) results.

Exercise 12: Using the plotter, determine which (if any) of the circuits you made has a resonant frequency between $s = 0$ and $s = 0 + 2j$. (Resonant behavior is characterized by a sharp peak or a sharp dip at one or more frequencies.)

Program design: Do either one of the next two exercises.

Exercise 13: Describe carefully how you would write a program that automatically tests circuits for resonant behavior. You needn't actually implement the program, but if you do, and have a lot of time, search for series-parallel circuits with four components to find those with resonant behavior. Take the four components to be a 10-ohm resistor, a 2-farad capacitor, a 1-farad capacitor, and a 2-henry inductor. Search the region from $s = 0$ to $s = 0 + 2j$, as before.

Exercise 14: One problem with the random method of generating circuits is that the same circuit might be generated many times, and some circuits might not be tried at all. How would you write a program to generate *all* the circuits that can be constructed from a given list of parts (with each part used exactly once)? Notice that there are really two problems here: one is to make sure that each circuit is generated, and the other is to eliminate duplicates. Eliminating duplicates can be tricky: for instance, given parts *p1*, *p2*, and *p3*, the combination

```
(make-series p1 (make-series p2 p3))
```

represents the same circuit as

```
(make-series p2 (make-series p1 p3))
```

You needn't write any code for this exercise, but be sure to give a good description of how you would go about solving the problem.

Warning/Hint: There is a clever way to eliminate the duplicates, but a straightforward approach can become extremely complicated. Don't be afraid to turn in a partial solution if you don't

discover a complete method. Also, if you actually implement a program and want to test it, you should know that there are 8 distinct circuits that can be constructed with 3 parts, 52 with 4 four parts, and 472 with 5 parts.