CS 1112 Spring 2020 Project 6, part A **due Tue, May 12, at 11:00 PM EDT**

You must work either on your own or with one partner. If you work with a partner you must first register as a group in CMS and then submit your work as a group. *Adhere to the Code of Academic Integrity.* For a group, "you" below refers to "your group." You may discuss background issues and general strategies with others and seek help from the course staff, but the work that you submit must be your own. In particular, you may discuss general ideas with others but you may not work out the detailed solutions with others. It is not OK for you to see or hear another student's code and it is certainly not OK to copy code from another person or from published/Internet sources. If you feel that you cannot complete the assignment on you own, seek help from the course staff.

Objectives

Completing part A of this project will give you practice with object-oriented programming (including polymorphic behavior) and parameterized graphics, as well as more exposure to cell arrays and file input. You will also get a taste for recursive structures (recursion will be explored more thoroughly in part B).

Ground Rule

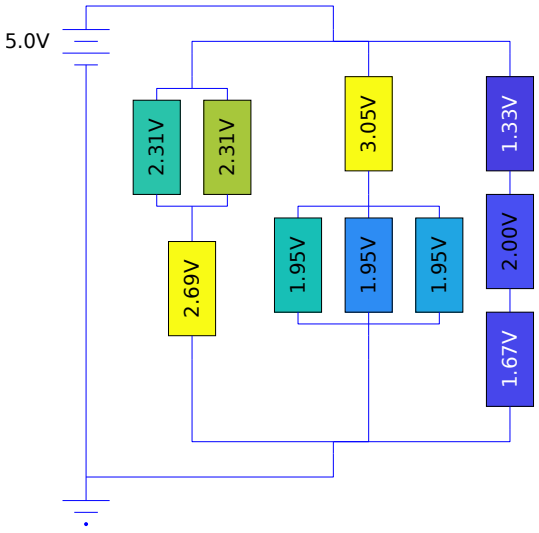
As usual, use only the functions and constructs learned so far in the course (textbook, lectures, exercises, project descriptions).

Object-oriented advice

Most of the power and complexity of object-oriented programs comes from the ways in which different objects are connected, not from writing a lot of complex code for any one class. Therefore, you should expect most of the code you write for this problem to be very simple. When possible, *delegate* as much work as you can to other objects, rather than trying to do it yourself all in one function; making properties private helps encourage this mode of working.

1 Visual circuit simulator

Your task is to build a simulation program for electrical circuits that can visualize how the components in the circuit are connected and where the most heat is being generated. The object modeling (deciding which classes and methods are useful) has already been done; your job is to implement the methods according to their specifications. Along the way you must test each new piece of functionality that you write.



The circuits we will be simulating consist of a battery powering one or more resistive loads connected to each other in series and in parallel. The battery applies a *voltage* (measured in volts, V, and often denoted ΔV) across the top and bottom of the loads, and this voltage causes an electrical *current* (measured in amps, A, and often denoted I) to flow along any wires and through the loads.

Our simulator will represent different kinds of loads as `Components`. A component may be a simple load (like a resistor) or a combination of several other components wired

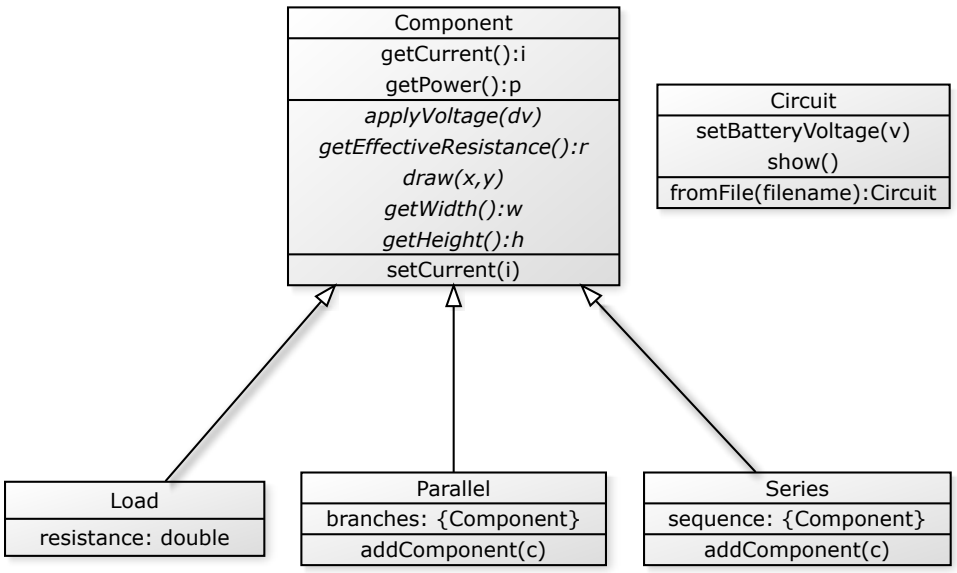
together in a certain way. Our `Component` class defines some features common to all components (like remembering the total current flowing through them), and it also declares several *abstract* methods. These are methods that every subclass of `Component` must define. The `applyVoltage()` method is responsible for performing the electrical calculations and setting the component's current based on the amount of voltage applied. The `getEffectiveResistance()` method provides data needed to perform these calculations when voltage needs to be divided between several components. The `draw()` method allows us to visualize how all of the components in the circuit are connected, assisted by the `getWidth()` and `getHeight()` methods (these allow a component that connects other components to know how big to draw itself, based on how big its constituent components are). Different classes inheriting from `Component` will implement these methods differently, providing what is called *polymorphic behavior*.

Designing our components to be visualizable provides a number of advantages. During development, you will be able to draw simple circuits as you go in order to check your work and see how things should be connected. And once the program is finished, you will be able to see the results of calculations (including currents, voltage drops, and heat dissipated) directly on the circuit schematic, which is much nicer than combing through many lines of text output.

Since you will be testing your work throughout development, start by creating a new script named `testCircuitSim.m`. Each test case you write will be appended to this script so that, by the end of the project, you will have an easy way to test everything that you've written with one click. Re-running old tests even as you focus on later parts of the project is known as *regression testing* and is an important practice in software engineering—it helps make sure you don't break something at the last minute by trying to make "one last, small change." Don't forget to add comments documenting your tests.

1.1 Component types

Here is the class hierarchy for the code that will make up our simulator:



Before writing any code, read the method documentation for all classes very carefully. In order to delegate work effectively, you will need to know what kinds of services other objects can provide. The existing implementations may also give you some hints for how to complete the remaining methods.

1.1.1 Load

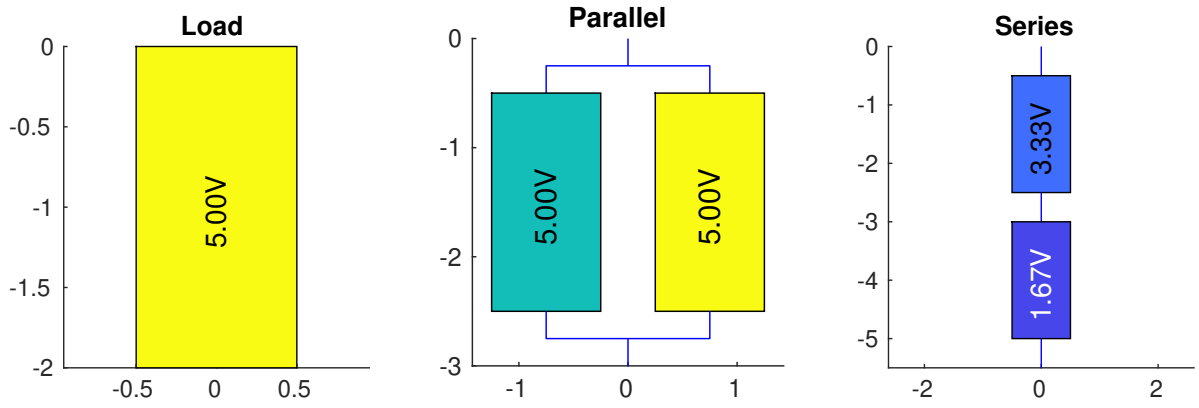
A `Load` component represents a simple resistor. It has one electrical parameter—its resistance R measured in ohms (Ω). It also has two graphical parameters. When a voltage ΔV is applied across a `Load`, the

current flowing through that load is given by an equation known as Ohm's law:

$$I = \Delta V / R$$

A `Load` should be drawn as a simple rectangle whose width and height are given by its graphical parameters (hint: look up the `fill()` function). The arguments to `draw()` specify the coordinates of the top-center point of the rectangle; remember: in normal plot coordinates (unlike for images), the y-coordinate becomes smaller as you move down. The rectangle should be filled with a color indicating the amount of power dissipated by this load (so we can see where the circuit is getting hot); the `powerToColor()` method has been provided for this purpose. Additionally, the voltage across the load should be written inside the rectangle. Use the `text()` function (see Project 1), and look up MATLAB's text properties `Rotation` and `HorizontalAlignment` to help get the label nice and centered. (Bonus: black text on a dark background is hard to read; try changing the text color to white if the average background color value is < 0.5.) Within all component `draw()` methods, you may assume that `hold` is already `on`.

Implement all of the methods marked `% TODO` in `Load.m`. Most of these methods (other than `draw()`) will be extremely simple (just one line). Then **test** your implementations in `testCircuitsSim`. Ensure that, when a voltage of 5 V is applied to a load with a resistance of 50 Ω, the current flowing through the component is 0.1 A. To verify the test, you may simply print the computed current next to the expected value (requiring manual verification), or you may automate the test by throwing an error if the values do not match (recall the `error()` function, or look up how to use `assert()`). Your script should also draw the component in a new figure window (with `axis equal` and `hold on`) to check that your rectangle, background color, and text look good (see figure).



1.1.2 Parallel

A `Parallel` component contains multiple branches connected in parallel. Each branch is represented by a single constituent component. When a voltage is applied across a `Parallel` component, that same voltage gets applied across all of its constituent components. Then the current through the `Parallel` component is equal to the sum of the currents flowing through the constituents. Since a `Parallel` component is just a wrapper around other components, it doesn't have any electrical properties of its own. It does need to maintain a cell array of its branch components, however (a cell array is required instead of a normal array because multiple subclasses of components are allowed). When constructed, a `Parallel` component will be empty (by initializing `branches` in the `properties` list, no explicit constructor is required to do this). Branches must be added after construction by calling the `addComponent()` method.

Even though a `Parallel` component contains multiple other components, it can be treated as a simple resistive load in calculations by computing an *effective resistance*. The effective resistance of N components

connected in parallel is given by the following formula:

$$\frac{1}{R_{\text{par}}} = \sum_{k=1}^{N} \frac{1}{R_k}$$

That is, the reciprocal of its effective resistance is equal to the sum of the reciprocals of the effective resistances of each of its branches.

The `draw()` method has already been implemented. It requires space above and below its constituent components in order to draw branching and converging wires, and it requires horizontal space between each branch. This extra space needs to be accommodated in the `getWidth()` and `getHeight()` methods.

Implement all of the methods marked `% TODO` in `Parallel.m`. Most implementations will require looping over the `branches` property, applying familiar patterns like *accumulation* and *best-in-set*. Then **test** your implementations in `testCircuitsSim`. Ensure that, when two loads with resistances of 100 Ω and 50 Ω are connected in parallel, currents of 0.05 A and 0.1 A respectively flow through the individual loads, a current of 0.15 A flows through the `Parallel` component, and the effective resistance is 33.333 Ω.

1.1.3 Series

A `Series` component connects multiple other components in a sequence so that the same current flows through all constituent components. When a voltage is applied across a `Series` component, only a portion of that voltage will drop across each constituent so that the sum of the voltages across each of them equals the total applied voltage. Like the `Parallel` component, a `Series` component is just a wrapper around other components, so it doesn't have any electrical properties of its own. It does maintain a cell array containing references to its constituents, which is initialized to empty without requiring an explicit constructor. Components can be appended to the sequence by calling the `addComponent()` method.

To perform the electrical calculations, we start by treating the entire sequence as a single effective load. Its effective resistance is simply equal to the sum of the effective resistances of its N constituents:

$$R_{\text{ser}} = \sum_{i=1}^{N} R_k$$

Using the same equation as for the `Load` component, this will tell us the current flowing through the `Series` component, and thus through all components in the sequence. Then, for each constituent component, that equation can be inverted to determine the portion of the total voltage that gets applied across it (using its individual effective resistance R_k):

$$\Delta V_k = I R_k$$

You will need to implement the `draw()` method, which draws each constituent component in a vertical sequence, separating each one by a blue line of height `stemHeight`; a stem should also be drawn above the first component and below the last component. These stems are factored into the given `getHeight()` calculation.

Implement all of the methods marked `% TODO` in `Series.m`. Most implementations will require looping over the `sequence` property, applying familiar patterns like *accumulation* and *best-in-set*. Then **test** your implementations in `testCircuitsSim`. Ensure that, when two loads with resistances of 100 Ω and 50 Ω are connected in series, the current through all components is 0.033 A and the effective resistance of the series is 150 Ω. Draw the component in a new figure (with `axis equal` and `hold on`) to both check your drawing and to verify that the voltages across the loads are 3.33 V and 1.67 V.

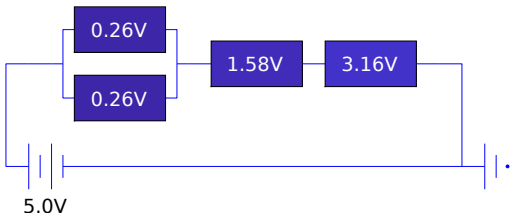
1.2 Assembling a circuit

Now that you have implemented all of the components in our class hierarchy, it's time to arrange them into a circuit. The `Circuit` class encapsulates a load (represented by a root `Component`, which may contain additional components inside of it) and allows the user to attach a battery and visualize the resulting circuit. The `show()` method has already been implemented. The constructor takes one argument specifying the root component. This is convenient for testing, but for larger circuits we will want to load them from a file. Therefore, it also declares a static method named `fromFile()`. A *static* method is like a free function—it's not meant to be called on any one instance of `Circuit`—but it's so closely related to the class that we write it inside the classdef anyway, and we use the name of the class when invoking it. For example, to read a circuit description from the file 'MyCircuit.txt' and store a reference to the resulting `Circuit` object in a variable `c`, you would write:

```
c = Circuit.fromFile('MyCircuit.txt');
```

The text files describing our circuits will look something like this (the numbers in the margin are line numbers and are not present in the file; see also `MyCircuit.txt` bundled with the project files):

```
1 L 50
  L 100
3 P 1 2
  L 200
5 L 400
  S 3 4 5
```



Each line starts with either `L`, `P`, or `S`, indicating whether that line corresponds to a `Load`, `Parallel`, or `Serial` component. Following the letter are one or more numbers separated by spaces. For a `Load`, the number indicates the resistance of that load. For `Parallel` and `Serial`, the numbers indicate which line numbers define the constituent components to be connected in parallel or series. Those line numbers will always be less than the current line number and will never be used more than once. The component on the last line should become the root component of the newly-constructed `Circuit`. In the above example:

- Lines 1 and 2 declare a pair of `Load` components with resistances of 50 Ω and 100 Ω.
- Line 3 declares a `Parallel` component that connects the above two resistors in parallel.
- Lines 4 and 5 declare another pair of `Load` components with resistances of 200 Ω and 400 Ω.
- Line 6 declares a `Series` component that connects the `Parallel` component from line 3, the 200 Ω resistor, and the 400 Ω resistor all in series. This `Series` component will be the root of the circuit.

Implement `Circuit.fromFile()` according to its specifications. You may find `str2cells` from Project 5 useful. Then **test** your implementation in `testCircuitsSim`. You may want to create simple files of your own for testing one component type at a time. However, since those custom files can't be submitted to CMS, be sure to comment out any such lines in your script before submitting so that the graders can run your script. The final test in your script (not commented) should be to load the file `MyCircuit.txt`, install a 5 V battery, and show the resulting circuit.

Submission

Submit your files `Load.m`, `Parallel.m`, `Series.m`, `Circuit.m`, and `testCircuitSim.m` CMS (after registering your group).

Part B will appear in a separate document. Parts A and B have the same due date.