Computational exercise: Calculating equivalent resistances
PH 220

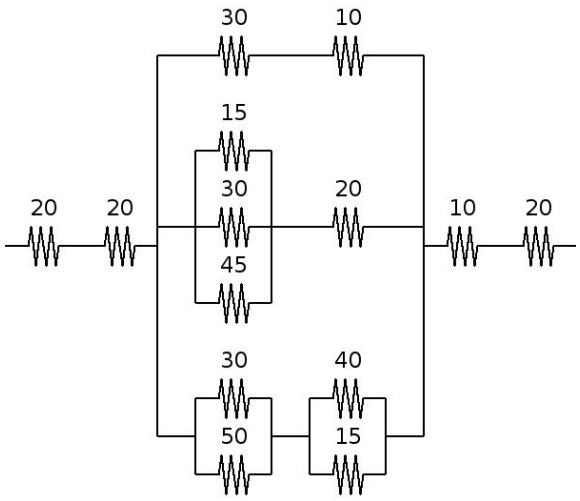
Objective: Write a computer code that reads in a character string representing a complex resistor network, and calculate the equivalent resistance of that network.

Secondary Objectives: Use recursive functions; use command line arguments.

Background: A common task in an introductory circuits course is finding the equivalent resistance of a network of resistors. Doing this generally entails breaking the circuit down into it's most fundamental networks (strictly series or parallel), calculating the effective resistance of those smaller pieces, and then working your way outward to the entire circuit. For circuits involving only a few resistors, this process is fairly straightforward. For circuits involving many resistors with a complicated topography, this can be a rather long process. Hence, the purpose of this exercise is to write a program that will do all of this hard work for us.

How is the computer supposed to know where to start? It turns out that the best way to solve this problem computationally is somewhat different than the way you would attack things by hand. One robust algorithm involves starting at the beginning of the network, assuming that it is just a series circuit, and then plowing through the series elements in sequential order. If one of the series "elements" turns out to be a parallel segment, we store the value of our current series resistance and then switch to a "parallel circuit" function, which will return the effective resistance of the parallel segment. Within this parallel function, we can in turn use our series function to find the effective resistance of the branches of the parallel circuit. Then, if somewhere in that series we encounter another parallel segment, the entire process can simply iterate over again. This keeps going on until we have recursively worked our way through the entire circuit.

In this exercise, we will inform the program of the resistor network's geometry as follows: series resistors are simply numbers separated by spaces. Parallel segments are enclosed in square brackets, with the branches separated by commas. The termination of the network is indicated with an 'X'. For example, this resistor network (resistances listed above the schematic symbols, in Ohms)



would be represented by the following character string

```
20 20 [ 30 10, [15, 30, 45] 20, [30, 50] [40, 15]] 10 20 X
```

We want our program to be robust with respect to white space, i.e. if there are extra spaces between numbers or after commas, or if there is not a space after a comma or bracket. In other words, we want our program to handle cases where the format of the input string is not exact.

To implement this algorithm, we will need to define two functions, which I will at present simply call “series” and “parallel”. These functions will both need to take, as arguments, the character string representing the circuit, and what our present position in this character string is. The functions basically proceed as follows (pseudocode below)

Series function (pass in character string and position as arguments)

- Set initial value of series resistance to zero
- Start cycling through the character string
    - If the next character is a ‘[‘:
        - Increment the current position by one space.
        - Call the parallel function.
    - Otherwise, if the next character is ‘X’, ‘,’, or ‘]’ (each indicates the end of a series segment)
        - Increment the current position by one space.
        - Return the effective resistance of the series.
    - Otherwise, if the next character is a number or decimal (indicates a resistor)
        - Read in the resistance of this resistor
        - Advance the position to the end of the number you just read in
    - Otherwise (this is probably just an extra space)
        - Advance the position by one space.

    - Iterate this process over again.

Parallel function (pass in character string and position as arguments)

- Declare variables for the effective resistance and an array of resistances for each branch. Also declare an integer representing the number of branches.
- Initialize the effective resistance and number of branches to zero.
- As long as the previous character in the string was not a ']'
    - Use the series function to evaluate the resistance of this branch.
    - Increment the variable that is keeping track of the number of branches.
- Now that we have the resistance of each branch, calculate the effective resistance of this parallel segment using $(1/R_1+1/R_2+...)^{-1}$.
- Return the effective resistance.

As an additional requirement for this assignment, I would like your code to pull the name of the input file in as a command line argument. In other words, when you execute your code, you will do so with a command such as:

```
python proj5.py [input file]
```

or

```
./proj5.exe [input file]
```

where "input" is the name of the input file. For those who are working in C, I have included some guidance on how to do this (as well as a few other things you will want to be aware of) at the end of this document. For those working in other languages, you may need to consult google for help.

Exercise requirements: Write a computer code that does the following:

1. Reads a file containing a character string representing a resistor network, as described above, using a command line argument. To check whether your code is working, I would recommend testing with both of the following input files (they represent the same circuit, but have varying amounts of whitespace between elements):

```
20 20 [ 30 10, [15, 30, 45] 20, [30, 50] [40, 15]] 10 20 X
```

```
20 20 [ 30 10, [15, 30, 45  ] 20, [30,50][40, 15]]   10 20 X
```

2. Determines the effective resistance of the circuit, and writes the result to the terminal. There are no specific requirements for the format of the output, though it would be appropriate to include some text (i.e. “The effective resistance of this network is 80.6157 Ohms”). Incidentally, the input files above represent a circuit with an effective resistance of 80.6157 Ohms. Note that you can create your own simpler input files to test your code.

Validation: You can verify whether your code is working correctly or not by feeding it the example input file above, and seeing whether it reproduces the correct result. You could also create other input files for simpler circuits, calculate the effective resistance by hand, and make sure your code is getting the right result.

Grading rubric: Your grade on this exercise, out of 50 points total, will be calculated based on the following criteria:

- Your code correctly calculates the effective resistances of five resistor networks that I pass to it. These networks will include (1) A network consisting only of series elements, (2) A network consisting of only parallel elements, (3) A network consisting of mostly series elements, with one nested parallel segment, (4) A network consisting of one parallel segment, but with series elements in one or more of the branches, and (5) a very complex resistor network, comparable to the one shown in the example. Each correct result will earn five points. (25 points)
- Your code is using command line arguments, as described in the assignment. (10 points)
- Your code is well organized and easily readable by the person grading it. (5 points)
- You have included all of the following comments in your code:
    - A header that describes what the code does and how to use it. (5 points)
    - Frequent descriptions of the variables in your code, and what your code is doing, including any numerical methods that you are employing. (5 points)

Additional notes for those who are programming in C (new programmers): There are a number of things that we will need to do with this code that you may have not seen before. These are described below.

- ***Forward declarations:*** Before I call a function in my program, that function should generally be defined first. In this assignment, we run into a bit of a snag. Specifically, the series function calls the parallel function, and the parallel function recalls the series function. Which one should we put first? Either way, we end up trying to call a function before it has been defined.

  The solution to this conundrum is called a forward declaration. We basically just specify the return type, name, and arguments of a function early in the program, followed with a

semicolon, without actually defining what the function does. For this assignment, those forward declarations will look something like the following:

```
...include statements, defines, etc...

double series(char *c, int *pos);
double parallel(char *c, int *pos);

double series(char *c, int *pos)
{
  ...the actual code for this function, which includes a call to the
  parallel function…
}

double parallel(char *c, int *pos)
{
  ...the actual code for this function, which includes a call to the
  series function…
}

int main(int argc, char *argv[])
{
   ... the main function, which will call "series" at some point...
}
```

Now, when the compiler encounters the call to "parallel" in the "series" function, it will know what that function is.

- ***Pointers and addresses:*** You may have noticed in the function declarations above, that there is a star in front of the variable names. These stars indicate that what I am actually passing to the function is not the value of the variable, but rather the memory address where that variable is found. This does two things for us in this case: (1) I am able to pass an entire array of characters to the function, rather than a single character (we're really just passing the address of the first character in the string), and (2) it allows me to change the value of one of the arguments within the function itself.

  How do we make use of this? In the series and parallel functions, I can refer to the characters within the array in the usual fashion. For example, if I wanted to print out the 7th character in the array, I could put the following line in my series function:

```
double series(char *c, int *pos)
{
  ...
  fprintf(stdout,"7th character is %c.\n",c[6]);
  ...
}
```

(Remember that in C arrays start with element 0, so element 6 is the 7th element). If I need to reference the position variable (the second argument) by value, I simply include the asterisk in front of the variable name. For example, to advance the position by one space, I would write

```
double series(char *c, int *pos)
{
  ...
  *pos=*pos+1;
  ...
}
```

When I return back to the place where I called the function from, the value of *pos will retain any changes I made within my series function.

Last of all, how would I go about calling either of these functions? In other words, how do I pass the addresses of my variables to the function? In the case of an array, we simply pass the array name which, by default, is a pointer to the first element of the array. For my position variable, I simply prepend a '&' character, which indicates I am passing the "address of" the variable:

```
int main(int argc, char *argv[])
{
  ...
  char circuit[10000];
  int position;
  double resistance;
  ...
  resistance=series(circuit,&position);
  ...
}
```

If I was calling the series function within the parallel function (where c and pos were already passed as pointers), I would do so as

```
double parallel(char *c, int *pos)
{
  ...
  rbranch=series(c,pos);
  ...
}
```

- ***Using command line arguments:*** You may have noticed in the previous examples that I am passing arguments (int argc, char *argv[]) to my main function. This is how you use

command line arguments in C. The number of arguments given on the command line is passed into my program via argc, and the character strings holding those arguments are passed in via the array *argv[]. The command that I executed is the first command line argument, and will be recorded in argv[0]. The first additional argument that I pass will be recorded in argv[1]. So, if I wanted to open a file by passing its name as a command line argument, my code might look something like the following:

```
int main(int argc, char *argv[])
{
  ...
  FILE *input;
  ...
  input=fopen(argv[1]);
  ...
}
```

- ***Reading an entire line from a file and storing it in a character array:*** This is a fairly common task, and is pretty straightforward if you use the `fgets` function (in stdio.h). I will simply give an example:

```
int main(int argc, char *argv[])
{
  ...
  char circuit[10000];
  FILE *input;
  ...
  input=fopen(argv[1]);
  fgets(circuit,10000,input);
  ...
}
```

  You can see here that the fgets function takes three arguments: the name of the character string in which I would like to store the information, the length of that character string (so that the function won't try to record information in unallocated memory), and the FILE from which it should read the string.

- ***Reading a decimal number from a string:*** Suppose that I have figured out at what position in my string a certain number begins. For sake of argument, let's say that it begins at the 14th character (i.e. element number 13 in the character array). There's a handy function I can use to read the decimal number from the array, starting at this point. The name of the function is atof. In order to use this function, I need to make sure and have a "`#include <stdlib.h>`" at the beginning of my code. The function is used as follows. In this example, "circuit" is a character array, and I would like to store the number in the variable "resist"

```
double resist;
char circuit[10000];

... read in the character string (array) from the file ...

resist=atof(&circuit[13]);
```

As you can see, the atof function requires only a single argument: the memory address of the character where the number begins, which I can specify using the ‘&’ symbol. (In plain english, the argument to the function would read “the address of circuit[13]”.)