Calculating the impedance of A.C. LCR circuits using Object Oriented Programming in C++

Meirin Oan Evans
9214122

School of Physics and Astronomy
The University of Manchester

Computing Report

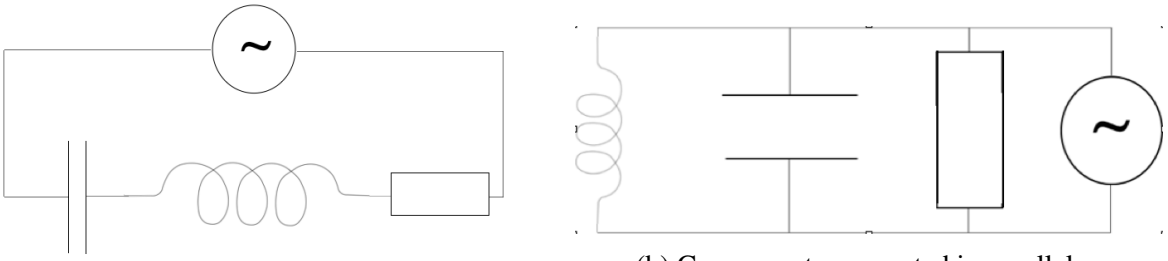
May 2018

Abstract

The objective of this project was to write a code to calculate the impedance of A.C. circuits with an arbitrary number of resistors, capacitors and inductors connected in series or in parallel. This was done using the Object Oriented Programming features available in C++. The code allows the user to create a component library of their own specifications. It subsequently constructs a circuit from this component library whilst connecting components either in series or in parallel. Finally, it outputs the magnitude of impedance and phase shift for each component, as well as the magnitude of impedance and phase shift for the entire circuit.

1 Introduction

This project calculates the impedance of Alternating Current (A.C.) circuits with any number of resistors, capacitors and inductors together in series or in parallel. Examples of simple circuits with 1 of each of these components are shown in Figure 1, part (a) showing the components in series and part (b) showing the components in parallel. When a circuit grows, it becomes tedious to manually calculate the impedance of each component and then the impedance of the whole circuit, therefore a code becomes useful to deal with this problem. The project uses data and function encapsulation, inheritance and polymorphism, and thus displays the 3 pillars of Object Oriented Programming. The code also uses some advanced features of C++ to increase functionality. Some of these features, such as static class members, headers and multiple source files, the list Standard Template Library linear container, exception handling (using try, throw and catch), smart pointers and lambda functions, only became available in C++11 [1].



(a) Components connected in series.

(b) Components connected in parallel.

Figure 1: Circuit diagrams of A.C. circuits with a power supply, capacitor, resistor and inductor, all connected by wires.

2 Code design and implementation

To simplify reading, the code is split into a header file, a .cpp source file that implements the contents of the header file and a .cpp source file that contains the main function. The code was designed to be compiled and run in a Microsoft Visual Studio 2013 Integrated Development Environment as a Win32 Console Application empty project [2].

2.1 Header file

The header file has a .h extension and contains constant definitions such as π (used in the argument of complex numbers) and constants to deal with exceptions using try, throw and catch. It then goes on to define a complex number class to store impedances before defining an abstract base class for components. The derived classes of the component class are resistors, capacitors, inductors, diodes, bulbs, wires and power supplies - most of what is found in an A.C. LCR (inductor, capacitor, resistor) circuit. In addition, a generic circuit class (which is not a derived class) is defined. The circuit class stores the impedance of the whole circuit and contains functions to add impedances for components in series or in parallel, then modifies the total impedance accordingly.

Within the class definitions are declarations for the functions they implement. A header guard is used to prevent constant definitions, class declarations and function declarations from occurring repeatedly when the header file is included in the multiple .cpp source

files. This situation would throw a compile time error, so a header guard is used to only link the header file once, as shown in Figure 2. This is not a problem for .cpp source files as they are only included once by the linker by default.

```
#ifndef CLASS_H // header guard
#define CLASS_H
#include<iostream> // for output to screen
#include<list> // for storing base class pointers
#include<memory> // for clearing cin buffer
const int divideFlag{ -1 }; // for argument exception
const double pi{ 3.141592653589793 }; // for arg
using namespace std;

// complex class
class complex {
  ...
} // end complex class
```

(a) Top of the header file.

```
// Circuit class
class circuit {
  ...
}; // end circuit class

#endif // end header file
```

(b) Bottom of the header file.

Figure 2: Different parts of the header file to show the use of a header guard. The header guard is the first 2 lines of the header file (apart from the comments) and the last line of the header file.

2.2 Implementation file

The implementation file, which has the same name as the header file but with a different extension, contains the function definitions of the component derived classes. These are functions to set and get the angular frequency in the circuit, to get the impedance of a component, the magnitude and argument of impedance of a component and to return the name of a component. In the component class these are pure virtual functions, which means that an instance of a component class cannot be created. Derived classes have to override every pure virtual function from the base class to create an instance of that class. The technique for defining and overriding pure virtual functions is shown in Figure 3, with the function definitions shown in part (a) and overridden functions shown in part (b).

2.3 Main file

The main file contains the main function which provides the user interface and uses all features that have been declared, defined and implemented in the header as well as implementation files. First, the angular frequency is set, then the details of each component

are read in. Once all components have been entered the code connects the components together either all in series or all in parallel, before displaying the details of the impedance of each component as well as the entire circuit. Finally, the code gives a visual representation of the circuit. A screenshot of the main function is shown in Figure 4.

3 Results

3.1 Code use

The code uses a simple user interface where all interaction is via the keyboard and terminal. A possible use of the code would be to enter the details of each component in a circuit in the initial stage of an electronics lab experiment, and then take the necessary measures to account for circuit impedance when deciding on the size of systematic errors in the final results. Such a situation is shown in Figure 5,

3.2 Input

The code uses minimal input to obtain all the information required. As soon as the circuit angular frequency is entered, the user can select their components by pressing a single letter. The user only needs to enter details of diodes, bulbs, wires and power supplies if they are non-ideal, which means that they have some resistance, capacitance or inductance (called parasitic resistance/capacitance/inductance) [4]. In the same way, the user can enter details of any non-ideal property of a resistor, capacitor or inductor. In the case of a resistor, these non-ideal properties would be capacitance or inductance. The final piece of input is a single letter to say whether the components are in series or in parallel. An example of input is given in Figure 6.

3.3 Output

The real output only begins when all input is finished. The main output is a listing of all components along with their magnitude and phase-shift of impedance. The same is done for each component type, for example all resistors are printed along with their magnitude and phase-shift of impedance. Once this is done, the same information is printed for the whole circuit, as well as how many of each component are present. To simplify the comparison between what is carried out in the code and a real experiment, the circuit is shown in a way that tries to copy a circuit diagram, for example in Figure 7.

4 Discussion

The 3 pillars of Object Oriented Programming are clearly implemented in this code. By defining class functions to decide how the different component classes behave, data and function encapsulation is shown. Inheritance is shown by having the resistor, capacitor, inductor, diode, bulb, wire and power supply classes derive from the component base class. In using the component class as a pure interface, which means one cannot create an instance of a component class, polymorphism is shown.

Some advanced features are also implemented to increase functionality. To display the use of static variables where the access is shared between all objects of a class, a variable for the current number of components is created before the start of the main function in the global part of the code. This variable is incremented in the component constructor

and decremented in the component destructor. As the user enters components the current number of components is printed. A list is used to store and print the components as base class pointers, which is chosen over an array or a vector to easily sort the components. This is done using a lambda function to sort the list in decreasing order of impedance magnitude, as shown in Figure 8.

An example of exception handling with try, throw and catch is used when calculating the argument of a complex number. When both the real and imaginary parts of a complex number are 0, the argument is undefined. This is the statement printed to the error stream by the catch block. Usually, a division (an argument calculation starts with a division) with 0 in the denominator would be undefined, but in this case infinity is returned using numeric_limits<double>::infinity because an inverse tan with 0 in the denominator corresponds to either + or - $\pi/2$, depending on the sign of the numerator. Smart pointers, rather than raw pointers, are used to store components in a list to clarify the ownership of the pointer.

5 Conclusions

The main point that could be improved in the code would be the ability to print circuits of any number of individually nested circuits, as this would allow visualisation of more situations, such as that shown in Figure 9. This could make the code applicable to some forefront research. Currently, the code allows any number of circuits to be nested and calculates the total circuit impedance, but does not print a visual representation of a circuit with two or more nested circuits within it. It would be a difficult and tedious task to allow the code to do so.

Additionally, the code might be improved if it included more of the C++ advanced features such as function and class templates and namespaces. However, it proved difficult to find a worthwhile use for function templates since not many variable types were used (only the complex number class for storing impedance and double for everything else) and different operations were performed on these 2 types, making a function template unsuitable. Similarly, there was no need for class templates as double and complex were not used in the same way. Throughout the code, there was no case where there was a need to prefix anything with the scope resolution operator (::). Care was taken to avoid name collisions, therefore a namespace would only have complicated the code. This may not have been the case if many external libraries were used, but again, care was taken to minimise the amount of external libraries that were included.

Apart from this, the code successfully calculates the impedance of A.C. LCR circuits, and also extends beyond this, by outputting a circuit in a visually recognisable manner and allowing for non-ideal components. Some advanced features of C++ are implemented to simplify code use as well as to add extra features and functionality.

References

[1] C++, C++11, Bell Labs, Bjarne Stroustrup, Murray Hill, New Jersey, United States , `hyper link to google search for C++11.`

[2] Microsoft Visual Studio, Microsoft Visual Studio 2013, Microsoft Corporation, Redmond, Washington, United States, hyper link to google search for Microsoft Visual Studio 2013.

[3] http://www.oberlin.edu/physics/catalog/demonstrations/em/LRC.html, *Resonance in LRC series circuits*, Oberlin College & Conservatory, Accessed /5/18.

[4] Gao, X., Liou, JJ., Wong, W., Vishwanathan, S., "An improved electrostatic discharge protection structure for reducing triggering voltage and parasitic capacitance", *Solid-State Electronics*, Volume 47, pages 1105-1110, 2003, web page hyper link.

[5] http://audilab.bmed.mcgill.ca/AudiLab/teach/circuit/circuit.html, *Circuit models*, Robert, W., Funnell, J., Accessed /5/18.

Appendix

The number of words in this document is 2327.

This document was last saved on 14/5/2018 at 23:33.

```
// Abstract base class for component
class component {
protected:
  complex impedance;
  double frequency;
  static int numobjects;
public:
  // constructor
  ...
  // pure virtual function to set frequency
  virtual void set_omega(const double angular_freq) = NULL;
  // pure virtual function to get frequency
  virtual double get_omega() const = NULL;
  // pure virtual function to get impedance
  virtual complex get_impedance() const = NULL;
  // pure virtual function to get magnitude of impedance
  virtual double mag_impedance() const = NULL;
  // pure virtual function to get phase difference
  virtual double phase() const = NULL;
  // pure virtual function to get component type
  virtual string type() const = NULL;
  // virtual destructor
  virtual ~component(){ numobjects--; }
}; // end abstract base class for component
```

(a) Definitions of pure virtual functions in the component base class, from the header file.

```
// resistor overriden functions
// override pure virtual function to set frequency
void resistor::set_omega(const double angular_freq) {
  frequency = angular_freq; }
// override pure virtual function to get frequency
double resistor::get_omega() const { return frequency; }
// override pure virtual function to get impedance
complex resistor::get_impedance() const {
  return impedance; }
// override pure virtual function for impedance magnitude
double resistor::mag_impedance() const {
  return impedance.mod(); }
// override pure virtual function to get phase difference
double resistor::phase() const { return impedance.arg(); }
// override pure virtual function to get component type
string resistor::type() const { return "resistor"; }
```

(b) Functions to override the pure virtual functions in the derived classes, from the implementation file.

Figure 3: Different parts of the header and implementation files to show the technique for defining and overriding pure virtual functions. The word virtual is added before the function definition in the component base class to make them virtual functions, and the functions are set to 0 to make them pure virtual functions.

```
// Main code
int main() {
  cout << "Output will be written to screen \& output.txt";
  // Introduce code
  cout << "\nThis code allows you to input resistors ,
  capacitors & inductors in series or parallel";
  // Frequency in circuit
  double freq;
  cout << "What is the circuit angular frequency?" << endl;
  cin >> freq; // read input frequency
  while (cin.fail() || freq <= 0) { // frequency not +ve
    cout << "Input has to be positive" << endl;
    cout << "What is the circuit angular frequency?\n";
    cin.clear();
    cin.ignore(numeric_limits<streamsize>::max(), '\n');
    cin >> freq; // get new frequency
  } // end while frequency not +ve
  cin.clear(); // clear rest of cin buffer
  cin.ignore(numeric_limits<streamsize>::max(), '\n');
  // Create array of base class pointers
  list<shared_ptr<component>> component_library;
  char more_components;
  // Declare resistance, capacitance, inductance
  double R{0},C{numeric_limits<double>::infinity()},L{0};
  // Ask user if they want to enter data from file
  char read_file;
  cout << "Would you like to add the contents of
  components.dat to your component library? (y/n)" << endl;
  cin >> read_file;
  while(cin.fail() || !(read_file=='y' || read_file=='n')){
    cout << "Sorry\nInput has to be y or n.\nWould you like
    to add components.dat to your component library?";
    cin.clear(); // clear rest of cin buffer
    cin.ignore(numeric_limits<streamsize>::max(), '\n');
    cin >> read_file;
  } // end while read_file y/n fail
  ...
```

Figure 4: A part of the main function in the main file, showing some of the code that asks for user input.

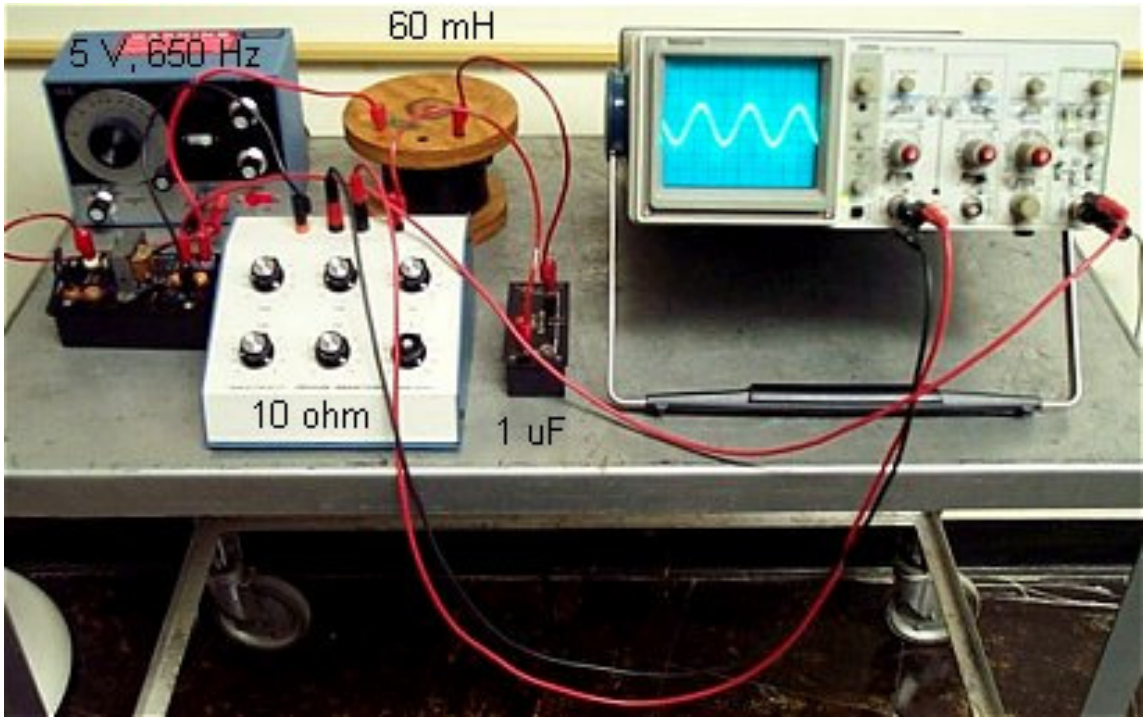
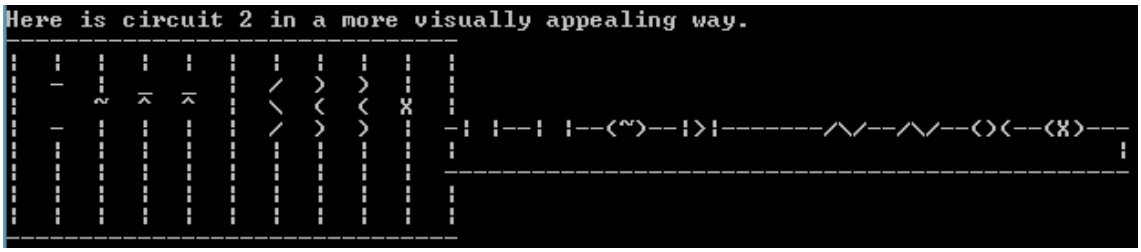


Figure 5: Photo of a simple example of an electronics lab experiment where using this code could provide a real world application [3].

```
What is the angular frequency in the whole circuit?
1e-6

Would you like to add the contents of components.dat to your component library?
(y/n)
n

Please enter r if you'd like to enter a resistor, c for capacitor, l for inducto
r, d for diode,b for bulb, w for wire or p for power supply.
You only need to enter d, b, w or p if you wish to enter the resistance, capacit
ance or inductance of the these components.
r
What is the resistance of this component?
1
What is the capacitance of this component?
0
What is the inductance of this component?
0
Component number 1
resistor constructor called

Would you like to enter another component to your library? (y/n)
n

Circuit number 1
How many of the 1 + 0i ohm resistor would you like in circuit 1?
1
circuit constructor called

Please enter s to have the components in circuit 1 in series or p for parallel.
s
```

Figure 6: A screenshot of a code fragment that asks for user input. Only one component with minimal details is entered for simplicity, and also to capture as much as possible of this code fragment.

(a) Components connected in series.



(b) Components connected in parallel.

Figure 7: Screenshots of code fragments that display a built circuit in a visually appealing way. As many components as possible are entered to fill the screen.

```
Print magnitude of impedance & phase-shift for each component in circuit 1
power supply, magnitude of impedance = 1 ohm, phase shift = 0 rad
power supply, magnitude of impedance = 4 ohm, phase shift = 0 rad
power supply, magnitude of impedance = 12 ohm, phase shift = 0 rad
capacitor, magnitude of impedance = 4.2735e+007 ohm, phase shift = -1.5708 rad
diode, magnitude of impedance = 8.33333e+012 ohm, phase shift = -1.5708 rad
capacitor, magnitude of impedance = 1e+013 ohm, phase shift = -1.5708 rad
capacitor, magnitude of impedance = 1e+014 ohm, phase shift = -1.5708 rad
wire, magnitude of impedance = 1e+014 ohm, phase shift = -1.5708 rad
capacitor, magnitude of impedance = 1e+014 ohm, phase shift = -1.5708 rad
capacitor, magnitude of impedance = 1e+014 ohm, phase shift = -1.5708 rad
power supply, magnitude of impedance = 1e+015 ohm, phase shift = -1.5708 rad
diode, magnitude of impedance = 1e+016 ohm, phase shift = -1.5708 rad
inductor, magnitude of impedance = 8.33333e+016 ohm, phase shift = -1.5708 rad
wire, magnitude of impedance = 1e+018 ohm, phase shift = -1.5708 rad
resistor, magnitude of impedance = 1e+018 ohm, phase shift = -1.5708 rad
resistor, magnitude of impedance = 1e+018 ohm, phase shift = -1.5708 rad
bulb, magnitude of impedance = 1e+018 ohm, phase shift = -1.5708 rad
inductor, magnitude of impedance = 1e+018 ohm, phase shift = -1.5708 rad
diode, magnitude of impedance = 1e+018 ohm, phase shift = -1.5708 rad
bulb, magnitude of impedance = 1e+018 ohm, phase shift = -1.5708 rad
wire, magnitude of impedance = 1e+018 ohm, phase shift = -1.5708 rad
resistor, magnitude of impedance = 1e+018 ohm, phase shift = -1.5708 rad
inductor, magnitude of impedance = 1e+019 ohm, phase shift = -1.5708 rad
diode, magnitude of impedance = 1e+019 ohm, phase shift = -1.5708 rad
wire, magnitude of impedance = 1e+019 ohm, phase shift = -1.5708 rad
resistor, magnitude of impedance = 1e+019 ohm, phase shift = -1.5708 rad
resistor, magnitude of impedance = 1e+020 ohm, phase shift = -1.5708 rad
inductor, magnitude of impedance = 1e+020 ohm, phase shift = -1.5708 rad
bulb, magnitude of impedance = 1e+020 ohm, phase shift = -1.5708 rad
bulb, magnitude of impedance = 1e+026 ohm, phase shift = -1.5708 rad
```

Figure 8: A screenshot of a code fragment that shows the component list sorted in decreasing order of magnitude of impedance; displaying the use of the list Standard Template Library linear container and a lambda function.

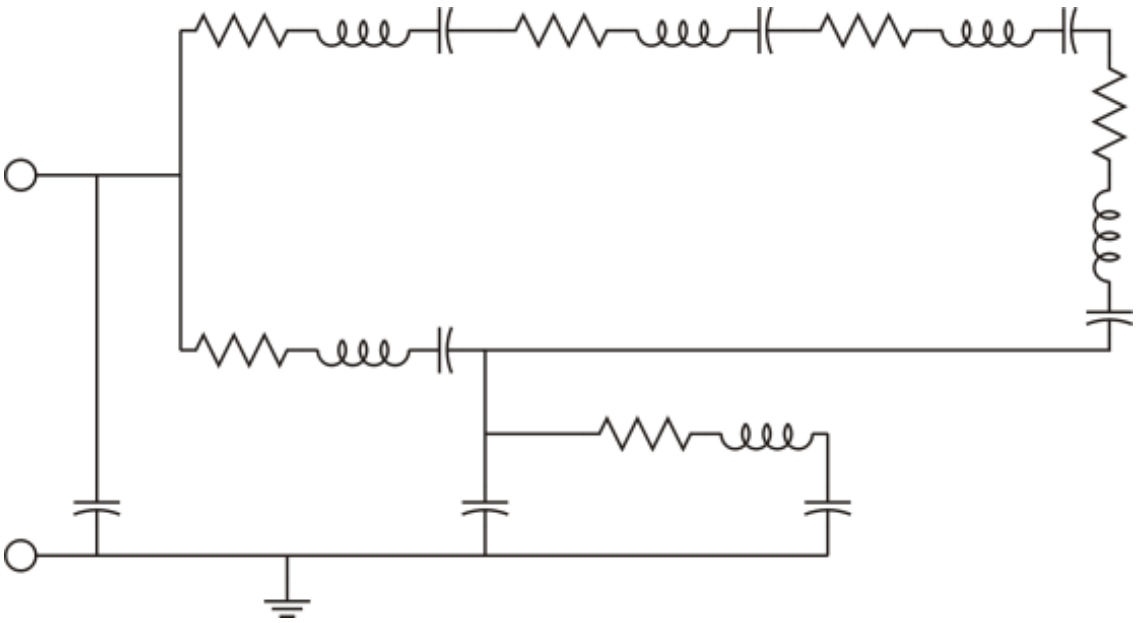


Figure 9: Circuit diagram for an example of a complicated LCR circuit where it might be useful to have a code feature whereby different components can be connected in series or in parallel, rather than all components being connected in the same way [5]. This particular example models different parts of the middle-ear as electrical components with resistance, capacitance and inductance. The setup looks complicated as different components are connected in series or in parallel, but it still only contains resistors, capacitors, inductors, wires and a power supply.