May 16, 2017

AC circuit impedence calculator

Object Oriented Programming final project

Dónal Murray
Student ID: 8381250

University of Manchester, Manchester, United Kingdom

Abstract

A program is presented for the manipulation and storage of alternating current circuits. Series and parallel circuits can be constructed using an arbitrary number of resistors, capacitors, inductors or other circuits composed of the listed components. Impedences of all components in the circuit as well as the total impedence of the circuit are calculated and circuits diagrams can be printed. The program allows the user to save the current project to a file and load old projects from a file. The program is designed to fully comply with the current C++ standard and to make use of advanced features of object oriented programming included in C++11.

1 Introduction

This report aims to outline the design and implementation of a program written in C++ to calculate the impedence of alternating current (AC) circuits consisting of any number of ideal resistors, capacitors and inductors in series or parallel. The program attempts to make best use of the advanced idiomatic features of C++.

C++ is an object oriented programming language that was created by Bjarne Stroustrup in 1983, initially with the aim to introduce object oriented programming into C [1] but which has since diverged to become a programming language in its own right and one of the most popular programming languages in the world.

Object oriented programming is centred around classes and their instantiations called objects which contain data and functions which can be either public, allowing the data or function to be accessed from outside of the class, or private, allowing access only within a specified scope. There is a third classification, protected, which will be discussed in section 2.

1.1 AC circuit theory

In contrast with DC circuit theory, where the current through a circuit is opposed only by the resistance, R, of that circuit, in AC circuit theory the current through an AC circuit is opposed by a combination of both the resistance and reactance, X and their combination is called the impedance, Z. Whereas resistance is the opposition to the flow of current, the reactance is the opposition to the *change* in flow of current. This is why the reactance of components is irrelevant with DC circuits, as the current through the circuit is not changing.

Impedance

Mathematically when dealing with AC circuits, complex numbers are used for the impedance with the real part representing the resistance and the imaginary part representing the reactance such that

$$Z = R + jX, \tag{1}$$

where, to agree with convention, $j = \sqrt{-1}$ is used in place of i to avoid confusion with the current which is conventionally denoted I [2]. The program is designed to make all of its calculations with ideal components, which means that a resistor only has resistance and zero reactance and that inductors and capacitors only have reactance and zero resistance. Therefore assuming ideal components, the impedance of a capacitor, inductor and resistor can be found using

$$Z_R = R, \qquad Z_L = j\omega L, \qquad \text{and} \qquad Z_C = \frac{1}{j\omega C} \tag{2}$$

respectively, where ω is the angular frequency of the AC current in $rad\,s^{-1}$ and L and C are the capacitance and inductance respectively [2].

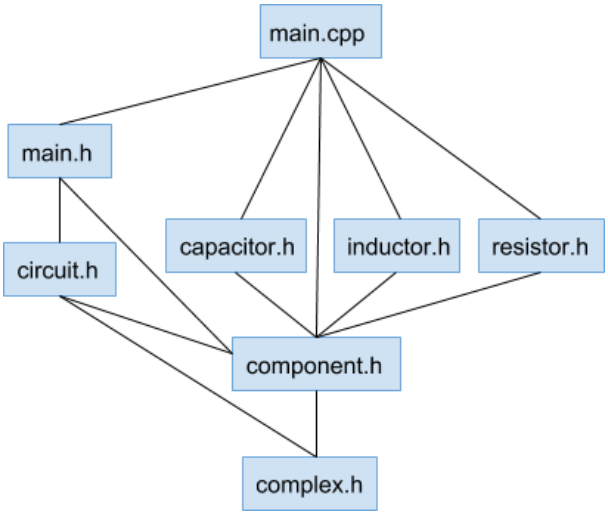

Figure 1: The hierarchy of header files showing the main file at the top and the chain of its dependencies leading down.

With n components connected in series, the total impedance of the circuit can be found by taking the sum of the impedances of the constituent components with

$$Z_{total} = Z_1 + Z_2 + \ldots + Z_n \tag{3}$$

and with n components connected in parallel, the inverse of the total impedance can be found by taking the sum of the inverse of the impedances of the constituent components with

$$\frac{1}{Z_{total}} = \frac{1}{Z_1} + \frac{1}{Z_2} + \ldots + \frac{1}{Z_n}. \tag{4}$$

Both equation 4 and equation 3 also work for nested circuits, as each subcircuit essentially behaves like a component [2].

Phase difference

The phase difference describes the offset between the voltage and the current. The phase difference is found by taking the argument of the impedance. Capacitors have a phase difference of $-90°$ which means that the voltage lags the current by 90° [3] and inductors have a phase difference of 90° which means that the voltage leads the current by 90° [4].

2 Code design and implementation

In the design of this project, the code was split into seven header files, the hierarchical structure of which is shown in figure 1. These header files contained class definitions, function prototypes and a namespace definition while the implementation of the classes, the full functions and the `main` function were separated into seven source files in line with the practice of abstraction, whereby the user is only exposed to the parts of the program

which they need to actually use the program, and the details like the member data are hidden from them.

Splitting the code into several files makes it easier to reuse code and limits the `#include` preprocessor directives to what is required only for the small piece of code in the file, and not for the entire program, which reduces the risk of name clashes. Separating into different files can also reduce the compile time, as only the parts that have been changed need to be recompiled and the other object files can just be linked by the compiler.

Header guards were put in place to avoid header files being included multiple times. By using the preprocessor directives to define a unique constant for each header file, the compiler could check using a single if statement whether the header had already been included. This avoids compiler errors caused by apparent redefinitions of existing functions and other problems.

A custom complex class, designed for assignment four of Object Oriented Programming was used for dealing with complex numbers instead of using the standard library type definition. The class contained member data of the type double to store the real and imaginary parts and member functions including functions to calculate the modulus and argument of a complex number, and also defined the division of two complex numbers, all of which were used in the calculation of the impedance and the phase difference.

For the components, an abstract base class `Component` was created, from which three derived classes inherited: `Resistor`, `Capacitor` and `Inductor`. The component class defined the functions that the three derived classes had in common – for instance functions to access and modify the label of the components and a general constructor which was called by the derived classes. The component class had a pure virtual function `get_impedance()` which was then defined in the derived classes, using the functions discussed in section 1.1. This meant that polymorphic vectors of base class pointers could be created, which could hold resistors, capacitors and inductors. Due to the virtual function in the component class, the three derived classes share a common interface. In this way the same function makes a different calculation depending on what type of object it was called on.

An abstract base class was created for the circuits which was similar to the component base class in that it virtual functions as a shared interface for the `Series` and `parallel` derived classes. Again, the function to calculate the total impedance was left as a pure virtual function in the base class, to be defined according to the equations discussed in section 1.1 within the series and parallel classes. Another virtual function in the `Circuit` class was a function to print the circuit diagram. This was defined separately in the series and parallel classes so that the circuit diagram could be printed in the correct layout for the circuit type.

A library of components was made from which components could be connected in specified groups to create circuits. A similar functionality was added for all circuits that had been made so that circuits from the circuit library could be nested inside other circuits. This was acheived by making two polymorphic vectors of base class pointers – one for components and the other for circuits. These polymorphic vectors were put inside the namespace `libs` in `main.h` so that the libraries could be accessed from any function using the namespace and the binary scope operator (`libs::`). If the libraries were used multiple times in the same function, `using namespace libs` could be used once to remove the need to type the namespace and binary scope operator repeatedly within the same function.

To ensure that the label of components and circuits is always unique, static member

data was used to keep track of the number of each type of component had been added so far. This is private member data which can be accessed by any object created from the class. By inrementing this integer in the constructor, the number of components could be used to create unique labels.

Smart pointers are template classes defined in the `memory` header in the C++11 standard which can be used like pointers and which take care of memory allocation automatically. By instantiating a pointer using `shared_ptr<T>` a pointer of type `T` is created which can be copied and therefore can be owned by multiple entities, for instance different circuits. The advantage of these pointers is that the data stored in the memory address pointed to by `shared_ptr` is deleted automatically when the data goes out of scope. Smart pointers were not implemented in this project as `dynamic_cast()` cannot compare a smart pointer and a pointer, and this function was key in printing the circuit diagrams. An iterator was used to loop through the polymorphic vector of components their symbols, labels and the magnitude of their impedances were printed. When a base class pointer is dynamically cast to a derived class, if the object stored in the memory pointed to by the base class pointer is not the same as the derived class, the dynamic cast returns a null pointer. In this way, the components of the circuit were classified and the relevant symbol was printed.

Input from the user is taken at multiple points in the file, each time with an input to different data types with different allowed values to be input. To avoid having to repeatedly type out checks for the correct input each time the user enters input, a template function `take_input()` was created, as shown in code snippet reflst:input. The function parameters are defined to be an `initializer_list`, from the header `initializer_list`, which means that the function is called with the argument as a comma separated list of allowed values for the input between two curly braces. The type of the elements of the list of arguments defines the type chosen for the template function, so there is no need to call it in the form `take_input<int>({0,1,2})` unless no allowed inputs are specified. When the program takes input from the user, the input is checked using a while loop whose condition is a lambda which iterates through the vector created by the initializer_list and checks if the input is present in this list of allowed values. The lambda returns a boolean value which determines whether to ask for input again. If an empty list is provided the lambda assumes that there is no range on the input and does not ask for input again as long as the type is correct. Once the user provides valid input, the `take_input()` function returns this value.

3 Results

In this section the output of the program is presented, showing the capabilities of the program and how it is used. Sample inputs are given and the output is shown.

When the program first starts, the title, author and main menu are displayed. The main menu gives the user 9 options, as shown in figure 2. Each of the options can be chosen at any time, with no requirement to progress through the options linearly.

```
AC Circuit Manipulator
  Author: Dónal Murray


Select an option:
1     Add components to library
2     Print component library
3     Create series circuit
4     Create parallel circuit
5     Print circuit library
6     Print a circuit
7     Save project to file
8     Load a project from file
0     Quit

Option:
```

Figure 2: The main menu.

```
Option: 1

Enter q to stop adding components.
Add resistor(r), capacitor(c) or inductor(l)?: r
Enter the resistance in Ω: 100
Add resistor(r), capacitor(c) or inductor(l)?: c
Enter the capacitance in μF: 300
Add resistor(r), capacitor(c) or inductor(l)?: l
Enter the inductance in μH: 170
Add resistor(r), capacitor(c) or inductor(l)?: q
```

Figure 3: The add component menu.

```
Option: 2

-------Component Library-------
| ID  Type       Value         |
-------------------------------
  C1  Capacitor  300μF
  L1  Inductor   170μH
  R1  Resistor   100Ω
```

Figure 4: The print component library option.

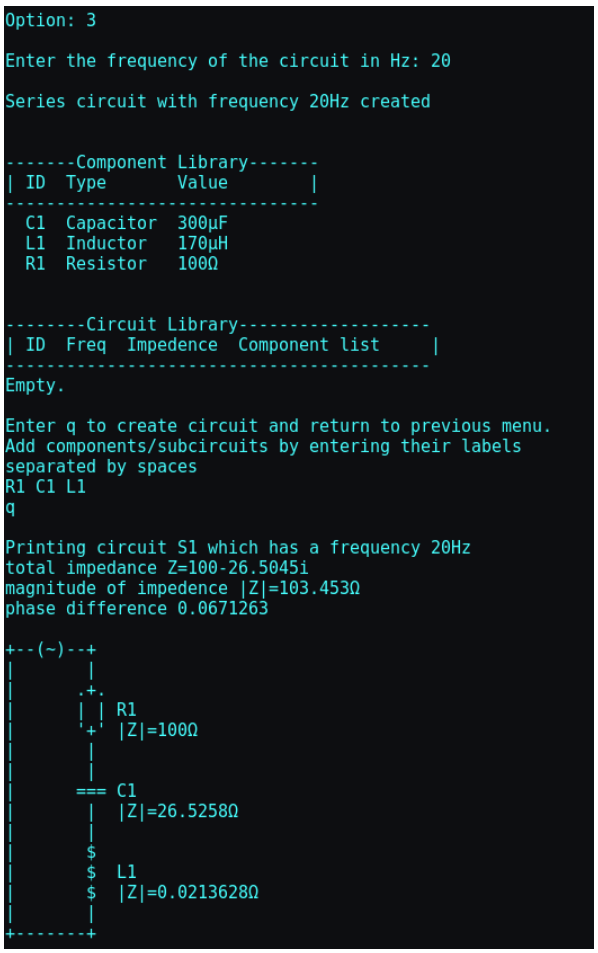

Figure 5: The add series circuit menu.

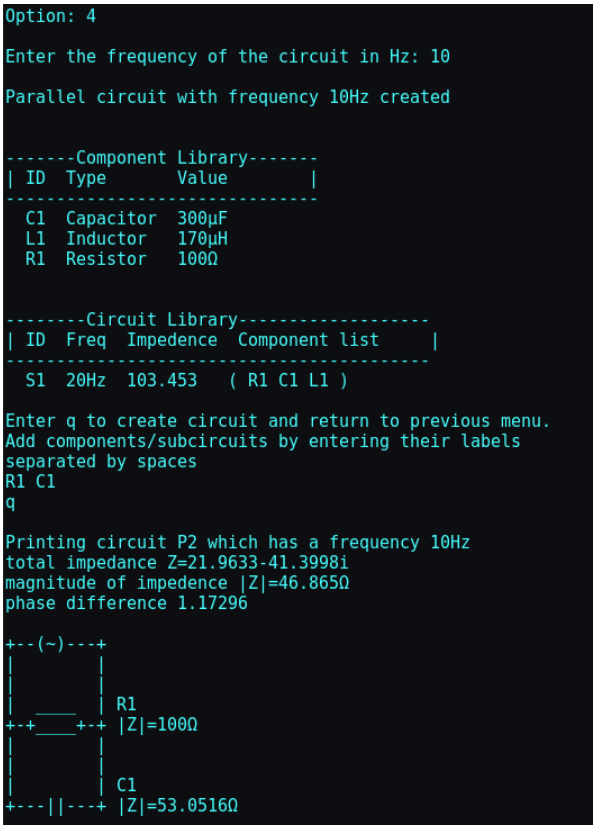

Figure 6: The add parallel circuit menu.

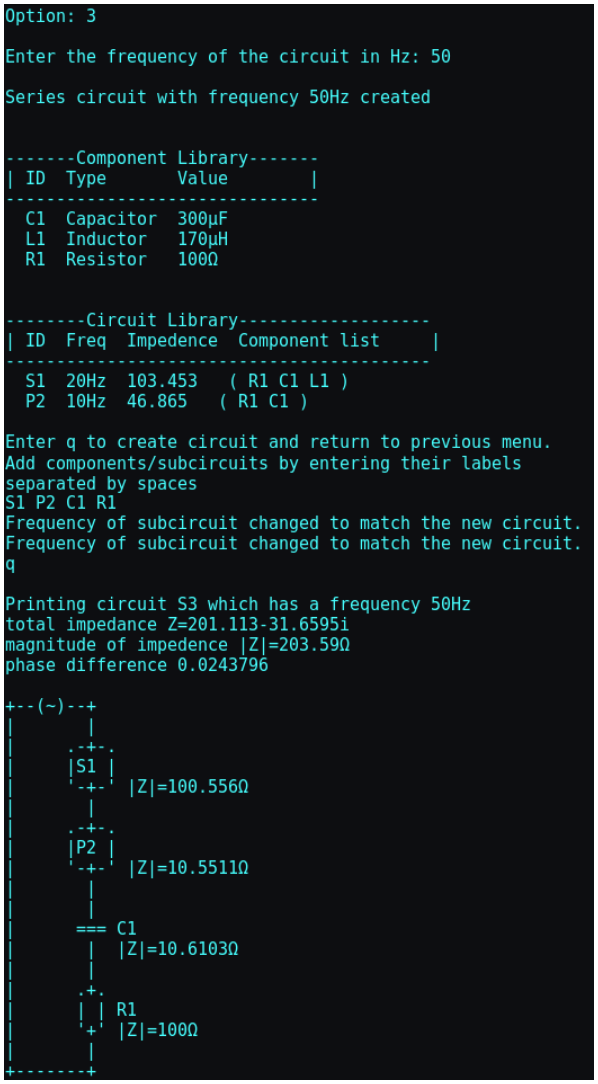

Figure 7: An example of a nested circuit.

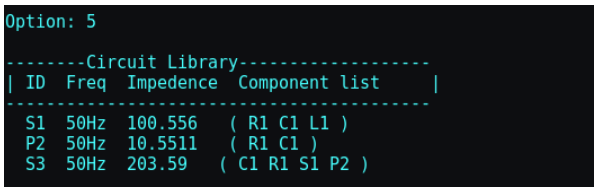

Figure 8: The print circuit library option.

Figure 9: The menu for printing a circuit diagram.

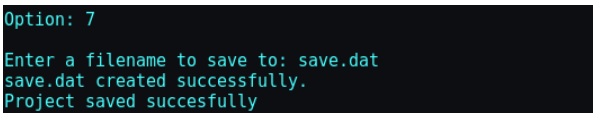

Figure 10: The menu for saving a project to file.

Figure 11: The exit message.

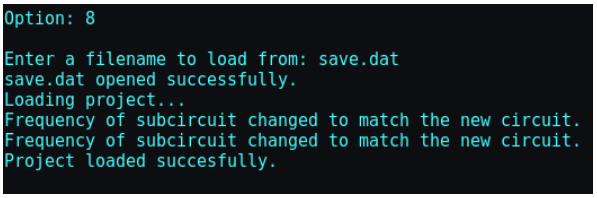

Figure 12: The menu for loading a project from file.

```
#SaveFile
[Components]
  C1  Capacitor  300µF
  L1  Inductor   170µH
  R1  Resistor   100Ω
[Circuits]
  S1  50Hz  100.556   ( R1 C1 L1 )
  P2  50Hz  10.5511   ( R1 C1 )
  S3  50Hz  203.59    ( C1 R1 S1 P2 )
[End]
```

Figure 13: An example of a save file.

4 Discussion

A possible extension of this project would be the ability to create circuits of non-ideal components, taking into consideration that real components have a combination of resistance and reactance and can have an arbitrary phase difference. Changing the circuit class to a class template would mean that circuits could be built using either ideal component objects or non-ideal component objects.

Another possible extension would be a bill of materials calculator for the components that have been used to build the circuit. Using databases for a company like Maplin [5], a list of components with their product codes and an estimate of the total price could be generated to make it easy for the user to translate their plans to a real circuit.

5 Conclusion

The aim of the project was to produce a program which could calculate the impedence and phase difference for a parallel or series circuit consisting of any arbitrary number of capacitors, inductors and resistors. The program can not only perform these tasks but also has additional features including:

- the ability to nest an arbitrary number of series or parallel subcircuits and then recursively calculate the impedence and phase difference of each of the constituent parts to give the total impedence and phase difference for the entire circuit;
- the ability to print circuit diagrams of any complexity through the use of placeholders for the subcircuits;
- the ability to save a project and load it at a later date, allowing for the same component library to be used repeatedly for different projects.

The program demonstrated the use of the C++11 standard library, using many advanced features of object oriented programming in their proper context and using `throw`/`catch` exception handling to deal with errors that would occur at runtime.

References

[1] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd ed., 2000.

[2] J. O'Malley, *Schaum's outline of theory and problems of basic circuit analysis*, McGraw-Hill, New York, NY, USA, 2nd ed., 1992.

[3] T. R. Kuphaldt, *Ac capacitor circuits*, `https://www.allaboutcircuits.com/textbook/alternating-current/chpt-4/ac-capacitor-circuits/`. Accessed: 13/05/2017.

[4] T. R. Kuphaldt, *Ac inductor circuits*, `https://www.allaboutcircuits.com/textbook/alternating-current/chpt-3/ac-inductor-circuits/`. Accessed: 13/05/2017.

[5] Maplin, *Maplin electronics is a company registered in england no. 1264385*, http://www.maplin.co.uk/.