

Implementation of an AC Circuits Program in C++

Krishan Mistry 8946701

2nd February 2018

School of Physics and Astronomy, University of Manchester
Object Oriented Programming in C++ Report

Abstract

This project calculates the total impedance of an AC circuit. This is done by user defined component library consisting of resistors, capacitors and inductors. The AC circuit can be constructed from this library with the option of nested parallel sub-circuits. The program was implemented through the use of inheritance, polymorphism, recursive functions and making use of advanced C++ features.

Contents

1	Introduction	2
2	Theory	2
2.1	Complex Impedance	2
2.2	Series and Parallel AC Circuits	2
3	Program Design	3
3.1	Complex Class	3
3.2	Input Validation	4
3.3	Component Class	4
3.4	Creating the Component Library	5
4	Circuit Construction	6
4.1	Sub-Circuit Class	6
5	Discussion	8
6	Conclusion	8

1 Introduction

The objective of this project is to create a program in C++ that allows the construction of an AC circuit with an unlimited number of standard components connected in series and parallel. This is done by initially creating a user defined component library consisting of resistors, capacitors and inductors and selecting components from the library to connect in series and parallel. Once the circuit has been created, it returns the total circuit impedance, magnitude and phase shift. The key focus of the program is to demonstrate the power of object oriented programming. The implementation of the program is designed such that it is simple and quick to use.

2 Theory

2.1 Complex Impedance

Circuits that contain an Alternating Current (AC) cannot be described through Ohm's law. The voltage and current of the circuit vary sinusoidally and can be characterised by an angular frequency, ω , and phase, ϕ . By construction, the voltage, \tilde{V} and current, \tilde{I} , (known as phasors) can be expressed as,

$$\begin{aligned}\tilde{V} &= V_0 e^{j(\omega t + \phi_V)}, \\ \tilde{I} &= I_0 e^{j(\omega t + \phi_I)},\end{aligned}\tag{1}$$

where the circuit phase is given by $\Delta\phi = \phi_V - \phi_I$.

Using these redefinitions, Ohm's law can now be modified by including a complex impedance, Z , by the equation,

$$\tilde{V} = \tilde{I}Z,\tag{2}$$

where \tilde{V} is the voltage and \tilde{I} is the current.

Complex impedance values are given by, R , for a resistor, $\frac{1}{j\omega C}$, for a capacitor and $j\omega L$, for an inductor, where R is the resistance, C is the capacitance and L , is the inductance.

2.2 Series and Parallel AC Circuits

The series impedance of an AC circuit is given by,

$$Z_{\text{Ser}} = Z_1 + Z_2 + \dots + Z_N,\tag{3}$$

where $Z_1, Z_2 \dots, Z_N$ are the complex impedances of the components connected in series.

Likewise, the parallel impedance of a circuit is given by,

$$\frac{1}{Z_{\text{Par}}} = \frac{1}{Z_1} + \frac{1}{Z_2} + \dots + \frac{1}{Z_N},\tag{4}$$

which can be rewritten as,

$$Z_{\text{Par}} = \frac{Z_1 Z_2 \dots Z_N}{Z_1 + Z_2 + \dots + Z_N}.\tag{5}$$

The total impedance of the circuit is therefore given by, $Z_{\text{tot}} = Z_{\text{Ser}} + Z_{\text{Par}}$.

3 Program Design

The program design was split into two main sections. The first section allows the user to construct a component library consisting of resistors, capacitors and inductors, all with user defined values. The following section then uses these components to construct the circuit.

3.1 Complex Class

To ease readability of the program, it was split into multiple header files. These files can be 'loaded' into a program in the same way standard library extensions are using the `#include` statement. This can allow the reuse of this code in other programs. The use of header files are an advanced C++ feature, an example of this can be seen in Listing 1. Line 1 demonstrates the use of a header guard which prevents re-definitions of a header file. This stops a possible compiler error. Header files are split into two parts, a `.h` file with an accompanying `.cpp` file. The `.h` file is used as an interface, where all function and class declarations are made. This includes default constructors and destructors. The `.cpp` file is where all the declared function definitions are defined.

Listing 1: Complex Class Header

```
1 #ifndef COMPLEX_H // Header guard
2 #define COMPLEX_H
3 #include <iostream>
4 #include <string>
5
6 using namespace std;
7
8 const double Pi(4 * atan(1)); // Define Pi
9
10 class complex
```

The `complex` class was created as an alternative to the standard library version and allowed the creation and manipulation of complex numbers. The use of the `complex` class is essential in the context of AC circuits which require the use and manipulation of complex numbers. Listing 2 shows the use of some key features of a class. The operator overload defined in line 2 demonstrates an example of the definition of the division (`\`) operation of two complex numbers. This was additionally defined for the addition, subtraction and multiplication of complex numbers. The use of `const` prevents the complex number, which is passed by reference, being modified by the operation. Lines 5 and 6 demonstrate the use of `friend` function to overload the `<<` and `>>` operators for the `complex` class. The use of friendship allows the use of these non-member functions to have access to the member data of the complex class.

Listing 2: Complex Class

```
1 // Overload / operator for division, z1/z2
2 complex operator/(const complex &z) const;
3
4 // Make function to overload operators << and >> a friend
5 friend ostream & operator<<(ostream &os, const complex &z);
6 friend istream & operator>>(istream &is, complex &z);
```

3.2 Input Validation

Five functions were created for taking user input, validating and returning the desired value. These functions were implemented in a separate header file (Validation.h) and they were designed to simplify the code readability. An example is if the user is going to enter a component number from the component library they have created. To do this, the EnterNumber() function is called which handles invalid inputs such as a character or a component out of scope and returns the correct entry.

Listing 3: Section of Input Validation

```
1 int EnterNumber(int bound){
2     try{
3         cout << "Please enter the component number you would like to add: "
4             << endl;
5
6         cin >> Num; // Take input
7
8         if (cin.fail() || cin.peek() != '\n'){ // Catch a bad usr input and
9             throw the error
10            cin.clear(); cin.ignore(numeric_limits <streamsize>::max(), '\n');
11            // Clear the stream and ingnore the rest of the line
12            throw 1;
13        } catch (int x){ // Output the error type
14            if (x == 1) cerr << "ERROR: the value entered was not suitable. " <<
15                endl;
16        }
17    }
```

Listing 3 shows the key features of the EnterNumber() function. The basis of this function uses exception handling which is an advanced C++ feature. This starts at line 2 where the try block is implemented. If a condition is met such as a failure in the cin function, an exception is detected and it 'thrown' (line 9) to catch (line 10). Depending on the error that occurred, a unique error message can be displayed (line 11).

3.3 Component Class

The Component class is an abstract base class to which the Resistor, Capacitor and Inductor classes inherit from. In other words, the Resistor class for example is a specialisation of the Component class and inherits its members, such as the frequency. The Component class contains pure virtual functions which can be seen in line 8 in Listing 4. The use of these virtual functions enables the overriding the base class functions in the derived classes. This is useful for functions such as PrintData() and GetImpedence(), which can be defined individually for each class which inherit these functions.

This class makes use of three types of data protection with the level of access specified by either public, private or protected. The frequency in the Component class (line 4) is protected. This means only members of the class and members that inherit the class members have access to this data. private members are only accessible within the class and public members are accessible outside the class. This is the idea of data encapsulation and gives total control over how the data is accessed.

Listing 4: Component Class Key Parts

```

1 class Component
2 {
3 protected:
4     double freq;          // frequency of circuit
5
6 public:
7     Component(){};        // Default constructor
8     virtual ~Component(){}; // Virtual Destructor
9     virtual complex GetImpedance() = 0;

```

3.4 Creating the Component Library

The choice of container for storing the components is the vector. This was chosen because its size does not need to be known when initialised and can be increased/decreased easily using the *push_back()* and *pop_back()* functions. Listing 5 demonstrates the creation of the component library vector. The vector is of a shared pointer type of **Component**. This means that it is a vector of base class pointers to the **Component** class. This is the basic idea of polymorphism. The shared pointer is an advanced feature of C++. This special type of pointer (smart pointer) automatically deletes dynamically allocated memory when it goes out of scope. This removes the need to delete memory limiting the possibility of a memory leak. This specific type of pointer is used because multiple components such as a **Resistor** and **Capacitor** both point to the same **Component** class.

Listing 5: Component Library

```

1 vector <shared_ptr<Component>> CompLibrary; // Vector of Base class
      shared pointers for component library

```

To create the component library, the EnterValue("Freq") function is called. This prompts the user to add the driving frequency of the circuit. Next, the EnterComponent() function is called which allows the user to specify the component type they would like to enter. This choice is fed into the EnterValue() function to prompt the relevant question and input the resistance/capacitance etc. The created component is added to the component library vector. The make_shared<**Resistor**> command is a replacement for **new** when dynamically allocating memory. This allocates memory only once for the pointer and object. In the case of using **new**, memory has to be allocated to the pointer and object separately which is not as ideal.

The library was printed out using an iterator over all elements contained in the component library vector. An example of the input and output of the library is shown in Figure 1.

```

What component would you like to enter, a Resistor(R), Inductor(I) or Capacitor
(C) or to Finish(F))?
i
Please enter the Inductance in H:
2e-3
What component would you like to enter, a Resistor(R), Inductor(I) or Capacitor
(C) or to Finish(F))?
f
-----
Component Library
1) 1000 Ohm Resistor
2) 50 Ohm Resistor
3) 200 pF Capacitor
4) 0.002 H Inductor
-----

```

Figure 1: Command window output of the component library.

4 Circuit Construction

Once the component library has been created, the circuit can now be constructed. The circuit is broken down into a set of sub-circuits. This implemented through the `SubCircuit` class. This circuit consists of a `Series` and `Parallel` derived class. Creation of each of these classes are shown in Listing 6. The `FinalCircuit` of type `Circuit`. The `Circuit` class provides the framework where we can store all the sub-circuits. This class contains a vector of base class shared pointers to the `SubCircuit` class. This implementation is similar to the vector of base class pointers of components to build the component library.

Listing 6: Series and Parallel Sub Circuit Creation

```
1 // Add component in series
2 if (Connection == "S" || Connection == "s"){
3     Num = EnterNumber(bound); // Select the component
4     FinalCircuit.AddSubCircuit(shared_ptr<SubCircuit>(make_shared<
5         Series>(CompLibrary[Num])));
6 }
7
8 // Add components in paralell
9 if (Connection == "P" || Connection == "p"){
10     FinalCircuit.AddSubCircuit(shared_ptr<SubCircuit>(make_shared<
11         Parallel>(CreateParalell(CompLibrary, false)))); // Adds a
12         paralell connection to the circuit by calling the
13         CreateParalell function
14 }
```

Each sub-circuit can be considered as an effective component with an impedance. By adding `Series` and `Parallel` sub-circuits to the `FinalCircuit` vector and summing all the impedances, the total impedance of the circuit can be evaluated.

4.1 Sub-Circuit Class

As mentioned in Section 4, the `SubCircuit` class contains two derived classes, `Series` and `Parallel`. The `Series` class consists of a shared pointer to the `Component` class and makes use of the `complex` class function `GetImpedence()` to return the impedance of this connection type. The `Parallel` class uses a recursive definition whereby it contains two shared pointers, `UpperBranch` and `LowerBranch` back to the `SubCircuit` class. This notation can be broken down using Figure 2.

To create a paralell sub-circuit, the `CreateParalell()` function is called. This function is split into two parts, the upper branch section and the lower branch section. The upper branch is considered first where the user is given an option to add a component in series or parallel. By selecting parallel at this stage is equivalent to creating three parallel branches. Recursively, the `CreateParalell()` function is called with the bool set to `true`. This indicates to the user they are now in a sub-branch parallel connection. The use of shared pointers is extremely powerful here handling the memory allocation and deletion.

Selecting series will add components to the branch in series, c.f. the S_3 and S_4 components in Figure 2. By pressing F, the user then moves to the nearest parallel branch. If they are in the Upper Sub-branch, the next component selection will be for the Lower Sub-branch. Once selection for the Upper Branch has been completed, the exact same procedure is followed for the Lower Branch. Finally, the function returns a parallel sub-circuit using

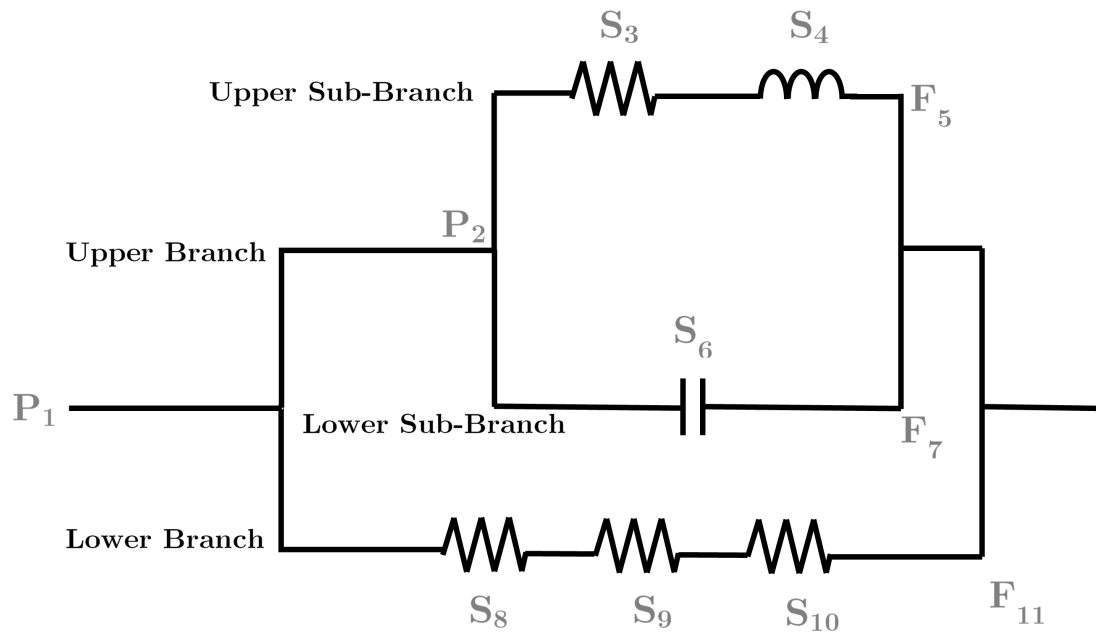


Figure 2: Circuit representation of creating the [Parallel](#) sub-circuit. S represents a component in series being added, P represents a new parallel connection being made and F represents the moving from an upper branch to a lower branch. The numbers indicate the order in which the connection types are made in order to create this specific type of circuit.

the parameterised constructor. This pushes the Upper and Lower branches into their respective [SubCircuit](#) vectors. An example of the typical input for creating the parallel sub-circuit is shown in Figure 3.

```

We will now create the circuit
Is the first component in series (S) or Paralell (P)?
p
Would you like the next component in the upper paralell branch to go in series
(S) or paralell (P)? (Press F to move to a lower paralell branch)
p
Would you like the next component in the Sub branched upper paralell branch to g
o in series (S) or paralell (P)? (Press F to move to a lower paralell branch)
s
Please enter the component number you would like to add:
1
Would you like the next component in the Sub branched upper paralell branch to g
o in series (S) or paralell (P)? (Press F to move to a lower paralell branch)
f
Thanks for entering the components, now moving on the the lower paralell branch
Would you like the next component in the Sub branched lower paralell branch to g
o in series (S) or paralell (P)? (Press F to move to end paralell selection)
s
Please enter the component number you would like to add:
2
Would you like the next component in the Sub branched lower paralell branch to g
o in series (S) or paralell (P)? (Press F to move to end paralell selection)
f
Now moving on the the component after the sub branched paralell selection

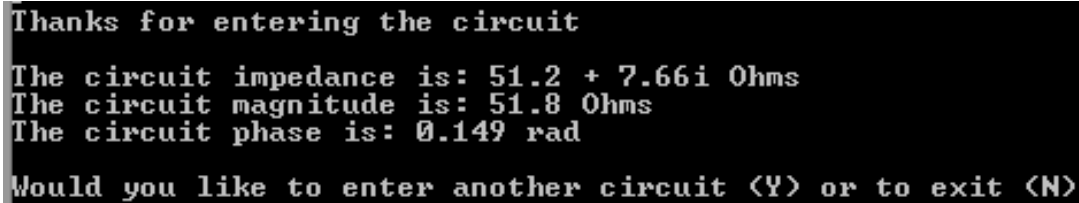
```

Figure 3: Command window example of the input for creating a parallel sub-circuit.

To calculate the impedance of the parallel sub circuit, an iterator is used over the upper and lower branches. This sums all the series components in the branch and also handles the recursive parallel definitions. After these have been calculated, their impedances are summed in parallel using Eqn. 4.

The final output of the program is shown in Figure 4. Once the program has finished, it

prompts the user if they want to create another circuit.



```
Thanks for entering the circuit
The circuit impedance is: 51.2 + 7.66i Ohms
The circuit magnitude is: 51.8 Ohms
The circuit phase is: 0.149 rad
Would you like to enter another circuit (Y) or to exit (N)
```

Figure 4: Command window example of the final output.

5 Discussion

The program successfully computes the total impedance of a circuit with an infinite number of series and parallel connections, however, a possible limitation is that the code does not visualise the circuit. This can make it easy to lose track when creating a deeply nested parallel sub-circuit. This could be achieved using a graphical user interface. In addition, AC circuits can exhibit resonant states similar to a forced oscillator. An extension to this program would be to find possible resonant states in an RLC circuit.

6 Conclusion

This program implemented a wide range of advanced C++ features such as header files, exception handling and smart pointers. The objective was to allow the user to create a component library consisting of resistors, capacitors and inductors. From this library, a circuit with an infinite number of parallel and series connections can be constructed. This was successfully achieved using inheritance, polymorphism and recursive functions. Limitations on the program include a lack of visualisation for the constructed circuit.