Analogue Circuits in C++

Ashan Selvaranjan

10500621

The University of Manchester

Object-Oriented Programming in C++

Final Project

May 2022

Abstract

A programme to output circuit impedance and other component information such as individual impedance and phase lag of an analogue circuit is created and tested. The programme allows the user to add 4 different types of components to a circuit in either series and/or parallel. The 4 components include: resistors, conductors, inductors, and light dependent resistors. Calculation of circuit impedance is successful while the circuit design and its limitations are discussed further, along with the importance and application of simulated circuits.

1. Introduction

Creation of an analogue circuit system in the C++ language involves the setup of abstract base classes for components, as well as derived classes for the mandatory components: Resistors, capacitors, and inductors. I have also introduced an additional 4th component, the light dependent resistor, for further unique circuit creation. To calculate the impedances of the circuit and individual components, pure virtual functions are implemented into each component class. These include functions to 'set' and 'get' frequency, as well as to 'get' magnitude, impedance, and phase shift. Where the impedance is being stored as a complex variable using the standard library. The impedances are calculated for all components to give the total circuit impedance. The total circuit impedance is modified by the action of the addition of each component. Impedances are summed differently when components are added in series compared to when components are added in parallel. The main functionality of the code is to allow the user to create a library of different components, stored in vectors,

and use this library to create an AC circuit of arbitrary length to connect components in series and parallel. Moreover, the code allows the user to print out impedance magnitude and phase-shift for each component as well as for the whole circuit.

2. Execution

2.1 Theory behind the code

The code begins by asking the user to input the desired power supply of the circuit in volts. The power supply is the most important piece of the circuit and allows current to flow through all components. The movement of current through components causes an effective resistance which is due to the intrinsic resistance of ohmic resistors, as well as the reactance from capacitors and inductors [1].

Resistors have electrical resistance and are usually used to control the current of a system. Ohmic resistors, which are used in this project, can be described by Ohm's law

$$V = IR. \tag{1}$$

Where V is the power supply in Volts, I is the current travelling through the resistor and R is the resistance. The impedance, Z_R, of a resistor is equal to its resistance. Unlike the capacitor, where impedance can be calculated using

$$Z_C = -j\frac{1}{\omega C}, \tag{2}$$

where Z_C is the impedance of the capacitor, ω is the frequency in Hertz and C is the capacitance in Farads. Similarly, the inductors impedance is calculated using

$$Z_L = 2\pi\omega L, \tag{3}$$

where Z_L is the impedance of the inductor and L is the inductance in Henrys. The impedances of series circuits are simply calculated by the addition of the individual component impedances, however the addition in parallel follows the inverse rule below [1].

$$\frac{1}{Z_T} = \frac{1}{Z_1} + \frac{1}{Z_2} + \cdots + \frac{1}{Z_n} \tag{4}$$

The light dependent resistor can be modelled to show the dependence of light intensity on resistance as

$$R = \frac{k}{Light\ intensity}, \tag{5}$$

where k is a constant and light intensity is in lux. Identical to ohmic resistors, light dependent resistors have impedance equal to their resistance.

The components that are described by equations 2 and 3 are for ideal components. To consider non-ideal components, stray parameters must be included when calculating impedances. Stray effects are unavoidable and are due to the proximity of the parts within the electrical components. Using equations 2 and 3 as well as the impedance and resistance

equality for a resistor, the equations for the impedances of non-ideal (real) components can be described.

$$Z_R = \frac{R+i\omega L_S}{(1-\omega^2 C_S L_S)+i\omega R C_S}, \tag{6}$$

$$Z_C = R_S + i\left(\omega L_S - \frac{1}{\omega C}\right), \tag{7}$$

$$Z_I = \frac{R_S+i\omega L}{(1-\omega^2 C_S L)+i\omega R_S C_S}. \tag{8}$$

Where R_S, C_S and L_S are the stray resistance, capacitance, and inductance respectively.

2.2 File structure

The code includes two source '.cpp' files and one header '.h' file, where the header file includes all declarations of classes. The source files include the member functions of classes as well as the main execution of the code. This is done to improve readability of the project as well as to allow accessibility when editing or altering the code.

2.3 Design and approach

The user should be able to run the programme and input their desired characteristics of the circuit, which are the frequency and power supply. An output of user possibilities of the programme should clearly describe the capabilities of the programme. Creation of components will start the process of creating a circuit, where 4 different components can be chosen from. They are then added to the circuit and finally, the circuit details are outputted. This is the basic framework of how the code was created, Figure 1.

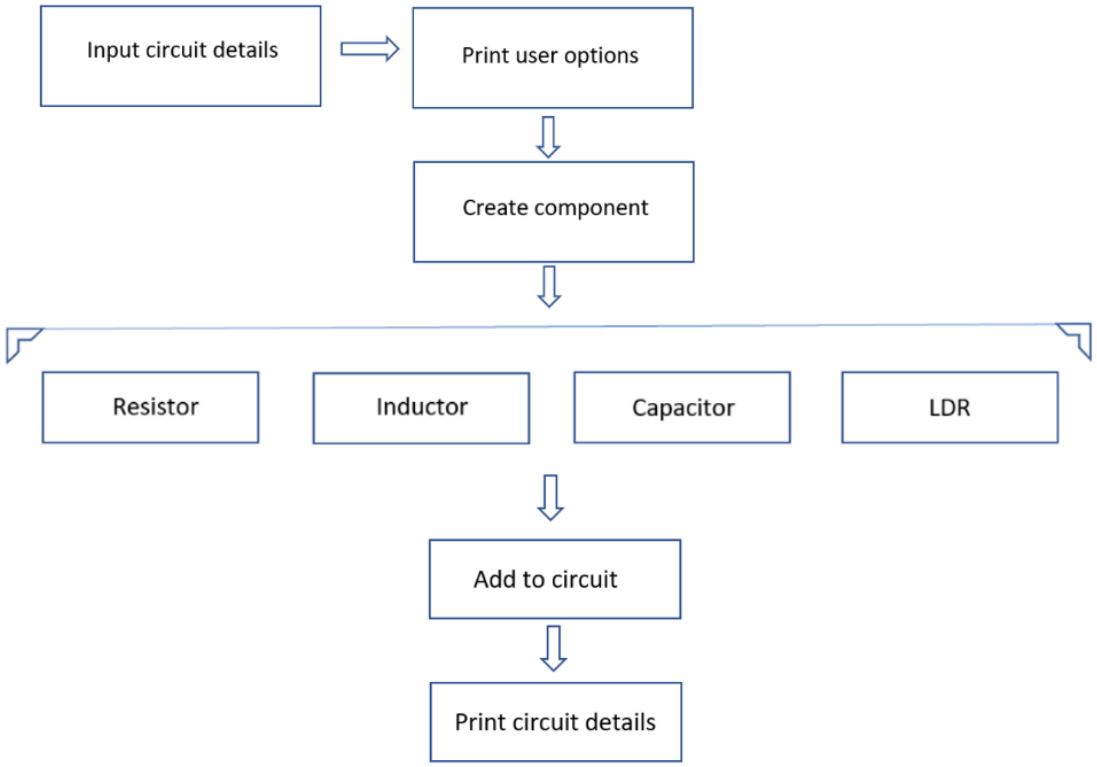

Figure 1: Basic framework of the application from the start of the programme to finish.

The foundation of the code utilizes class hierarchy, where information between classes are shared where appropriate. Like impedance in the *'component'* class, which is shared with all specific component classes. Additionally, in each class there are member functions that utilise polymorphism and are overridden in each case. This can be seen below in Figure 2.

```
32  void capacitor::set_freq(const double f) {
33      frequency_ = 2 * pi * f; //w = 2pi*f
34      //set impedance
35      std::complex<double> Z(0, -1 / (frequency_ * capacitance_)); //Z = -i/wC
36      impedance_ = Z;
37  }
38  void inductor::set_freq(const double f) {
39      frequency_ = 2 * pi * f; //w = 2pi*f
40      //set impedance
41      std::complex<double> Z(0, frequency_ * inductance_); //Z = 2*pi*wL
42      impedance_ = Z;
43  }
```

Figure 2: Two member functions (*'set_freq()'*), that are separately overridden for the capacitor and inductor classes respectively as the calculation for impedance differs.

This is done with many other functions such as *'get_mag()'* and *'details()'* which return the magnitude of impedance and print the details of the component respectively.

Throughout the code, the user is asked for input to assign characters, integers, and doubles to variables. The template *'typename'* is utilised to allow multiple input types into the function, see Figure 3. This eliminates the necessity of multiple input functions for different type inputs, which is particularly useful as multiple type inputs are used throughout the programme.

```
15  template<typename t> t return_input(t input) {
16      while (true) {
17          if (std::cin >> input) {
18              break;
19          }
20          else {
21              std::cout << "Enter a valid value!\n";
22              std::cin.clear();
23              std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
24          }
25      }
26      //return any type
27      return input;
28  }
```

Figure 3: Template input function, including a while loop for incorrect type.

2.4 Memory and storage

The data within this code is accessed by pointers of the class *'component'* stored in 4 different vectors for specific use. As well as a vector of class 'circuit' to store the components currently in the circuit being created. The *'component_library'* vector stores access to components that the user has created. *'components_input'* stores access to the specific components that are being added to the circuit. *'series_components'* and *'parallel_components'* stores access to

the components added in series and in parallel respectively, these are both accessed when

```
6     std::vector<component*> component_library;
7     std::vector<component*> components_input;
8     std::vector<component*> series_components;
9     std::vector<component*> parallel_components;
10    std::vector<circuit*> current_circuit;
```

Figure 4: Declaration of vectors to store pointers to component information.

printing out the circuit diagram. The declaration of these vectors is shown in Figure 4.

2.5 Circuit creation

The user is shown a menu of options to start creating an analogue circuit where they will create

```
46      // Instance a resistor is chosen to be created
47      if (type == 'R') {
48          std::cout << "Is the resistor real or ideal? (r/i)" << std::endl;
49          r_i = return_input('a');
50          if (r_i == 'r') {
51              std::cout << "What is the resistance (in Ohms)?" << std::endl;
52              trait = return_input(1.0);
53              std::cout << "What is the stray capacitance (in Farads)?" << std::endl;
54              stray_1 = return_input(1.0);
55              std::cout << "What is the stray inductancce (in Henries)?" << std::endl;
56              stray_2 = return_input(1.0);
57              //construct non ideal capacitor
58              real_resistor* test_r = new real_resistor(trait, stray_1, stray_2);
59              test_r->set_freq(frequency_);
60              component_library.emplace_back(test_r);
61              std::cout << "Component created!" << std::endl;
62          }
```

Figure 5. Creation of a real resistor in the *'create_component()'* function.

components before adding them to a circuit in series or parallel, where a while loop is implemented to allow the user to pick as many components as they would like. The *'create_component()'* function will check if the user input is equal to the component character type. If so, *trait* is assigned by the user and is used as the input for the specific component. A test pointer is created and added to the component library while setting the frequency. Before this, the choice of a real or ideal component is given where a real component requires further input. This can be seen in Figure 5.

The function *'add_series()'* is a function defined in the *circuit* class; it prints the *'component_library'* before asking the user to input which functions to add in series. The user then inputs the specific key of which value they would like added. These chosen components are added to the *'components_input'* vector where they are iterated through and added to the

```
90  void circuit::add_series(std::vector<component*> components_input_) {
91      std::string decision{ "y" };
92      print_library();
93      while (decision == "y") {
94          std::cout << "Which component do you want to add in series? (enter key number)\n" << std::endl;
95          components_input_.emplace_back(component_library[return_input(0)]);
96          std::cout << "Component added!" << std::endl;
97          std::cout << "Would you like to add another component (y/n)?";
98          std::cin >> decision;
99          if (decision == "n") {
100             break;
101         }
102     }
```

Figure 6: While loop in the *'add_series()'* member function for multiple component entries.

'current_circuit' vector which holds information on total circuit impedance. Through this for loop, the impedances of the components being added in series are summated. This function is mirrored for the *'add_parallel()'* function although equation 4 is used to formulate the total impedance of the circuit.

To avoid creation of data that is not being used, both component and circuit vectors are cleared of their pointers before creating a new circuit. This keep the allocation of data to a minimum. A snippet of this code is shown in Figure 7.

```
209         case 'f':
210         {
211             component_library.clear(); //clear the component library vector
212             current_circuit.clear();
213             std::cout << "Library cleared! Start crerating a new circuit." << std::endl;
214             std::cout << "What is the power supply in Volts? (V)" << std::endl;
215             std::cin >> voltage_;
216             std::cout << "What is the frequency in Hertz? (Hz)" << std::endl;
217             std::cin >> frequency_;
218             circuit* created_circuit = new circuit(voltage_, frequency_);
219             current_circuit.emplace_back(created_circuit);
220             break;
221         }
```

Figure 7: The case of the user choosing to clear and create a new circuit, utilising the *'.clear()'* function in the standard library.

The printing function to display the component library is carried out with the use of threads. Using threads increases portability and can improve the speed of the code. Parallelism is a technique used within code to carry out tasks in conjunction, to minimise run time. Multithreading is a form of parallelism where several threads are run in parallel. I have used *'auto'* in the *'print_library_()'* function shown in Figure 8. *'auto'* eliminates the use of type specifiers and becomes useful when the type specifier is quite long, this increases the readability of the code. The compiler automatically recognises what type should be assigned and assigns it. Introducing threads into the code has been successful however, the advantages of incorporating threads have not been fully utilised.

```
96  void print_library_() {
97      threads.push_back(std::thread(print_library));
98      for (auto& th : threads) {
99          th.join();
100     }
101     threads.clear();
102 }
```

Figure 8: The use of threads in the print function used to display the component library.

3. Results and application

Calculation of impedance in circuits is necessary when building, testing, and using electronics. For example, in the use of nuclear magnetic resonance spectroscopy the probes used must match an impedance value to maximise power transfer [2]. Likewise, testing certain circuitry can be dangerous without identifying the risks. These risks are managed and calculated with knowledge of the circuit impedance. Outputting the circuit details displays the total impedance due to the circuit components, as well as other circuit information like current, phase shift and frequency. The output in the terminal is shown in Figure 9.

```
Circuit:
        -Frequency: 1Hz
        -Voltage = 1V
        -Current = 0.399848A
        Circuit impedance:
         -Impedance =  (2.50063,-0.0397787)
        -Magnitude = 2.50095 Ohms
        -Phase shift = -0.0159061 rads
```

Figure 9: Terminal displaying circuit details.

The circuit diagram in Figure 10 shows the circuit created after adding a resistor (R) and inductor (I) in series before adding a light dependent resistor (L) and capacitor (C) in parallel. V is the power supply.

```
Circuit diagram:
-------------V--------------
|                           |
------R-------------I-------
|
|-------------L-------------|
|
|-------------C-------------|
```

Figure 10: Example circuit diagram that includes a resistor, inductor, light-dependent resistor, and capacitor.

The code for the circuit diagram reads through the *'series_components'* vector and the *'parallel_components'* vector and outputs the *'circuit_description'* while inputting dashes to represent wires. When adding components in series/parallel they are added to their respective vectors to be printed through a for loop of the size of the vector. Depending on the size of the vector, different sizes of wire are printed to display a correctly connected circuit.

4. Advancing the code

After review, there are areas in the code that could be improved upon. One for example, is the use of smart pointers to access data instead of regular pointers. Smart pointers will clean themselves up after they go out of scope, thus removing fear of most memory leaks. As it can be seen in Figure 9, the printing of the circuit is displayed correctly, however the number of components that are formatted in the correct form is limited. Once the number of components in the circuit exceed 6 components in series the circuit diagram will not be made correctly. This is another area of potential approvement, to allow representation of any circuit independent of the number of circuit components. Furthermore, the *'current_circuit'* vector that holds access to the circuit could hold more than one circuit, so the user is able to view the data of multiple circuits without having to recreate it. These improvements are manageable, and I believe the structure of the code currently allows for these changes to be made.

5. Conclusion

Through testing of the programme, all aims of the project are met. From being able to create a component library, adding components in series/parallel to printing the impedance details of the circuit and its components.

The basic features of an analogue circuit simulator that outputs the circuit impedance based on an arbitrary number of components are successful. As well as this, some extra functionality is included such as an extra choice of component type, choice of real components and the printability of the circuit. The code includes features of C++ that are used in high volume data handling such as multithreading, use of template classes and polymorphism. Management of data accessing, and garbage collection are secure with no memory leaks. However, the security of the data can be improved upon with the use of smart pointers. The usability of the code is clear and easy to follow with output statements to guide the user's choice of inputs.

References

[1] - Callegaro, L. (2012). Electrical Impedance: Principles, Measurement, and Applications. CRC Press, p. 5

[2] - D.D.Traficante, J.A.Simms, and M.Mulcay, "An Approach to Multi-nuclei Capability in Modern NMR Spectrometers", J. Magn. Reason., 1974, 15, 484-497.