AC Circuits Simulator via an Object-Oriented Implementation in C++

Monty Kirner, ID: *10301768*
School of Physics, Department of Natural Sciences
The University of Manchester
(Dated: May 22, 2021)

A program to simulate AC circuits was constructed using the C++ language and its standard template library. By creating an abstract class for components, derived classes consisting of ideal and non-ideal components such as resistors, inductors and capacitors were defined. A circuit class was also created, derived similarly from the abstract component class, which could connect components either in series or parallel and then calculate several properties of the circuit such as the impedance. This was achieved via a user interface, where the user could create separate libraries for components and circuits to then modify these libraries accordingly.

I. INTRODUCTION

In order to apply an alternating-current (AC) to a circuit, a source must periodically reverse the direction of the current such that its magnitude varies sinusoidally with time. This can be compared to a direct current (DC) where the current only flows in one direction. An advantage of using AC for electrical power generation is that the voltage can be changed with a transformer. As a result, energy can be distributed more efficiently over long distances by transmitting through power lines at high voltages, reducing the energy lost due to resistive heating, and then converted to a lower, and thus safer, voltage for general-purpose use [1].

Within this project, an AC circuit simulator is implemented in the C++ programming language which connects resistor, inductor and capacitor components in either series or parallel configurations. The program enabled users to create components and circuit libraries, where the total impedance of a given circuit could then be calculated.

II. AC CIRCUIT THEORY

The impedance of a circuit considers a complex generalization of Ohm's law. By incorporating complex numbers, AC circuits can be quantitatively described through definition of a phase component [2]. The impedance $\mathbf{Z}$ of a two-terminal circuit element is given as

$$\mathbf{V} = \mathbf{ZI}, \tag{1}$$

where $\mathbf{V}$ is the voltage across the component and $\mathbf{I}$ is the current flowing through it [3].

For N elements connected in either series or parallel, the impedance can be calculated using Kirchoff's laws similarly to resistance as

$$Z_s = \sum_{i=1}^{N} Z_i, \qquad \frac{1}{Z_p} = \sum_{i=1}^{N} \frac{1}{Z_i}, \tag{2}$$

where Z_s and Z_p are the series and parallel impedances respectively [4]. Any circuit can therefore be reduced to a single equivalent circuit component with a complex impedance.

In the case of ideal resistors, inductors and capacitors, the impedance is given as

$$\mathbf{Z}_R = R, \qquad \mathbf{Z}_L = i\omega L, \qquad \mathbf{Z}_C = \frac{-i}{\omega C}, \tag{3}$$

where ω is the angular frequency of the circuit and R, L, C correspond to the resistance, inductance and capacitance respectively [1]. Practically however, real-world circuit elements will exhibit non-ideal properties. For instance, the wires connected to the ends of a resistor will generate a magnetic field, thus producing self-inductance. These parasitic effects are common to all three components, and cause the impedance to deviate from the ideal equations when significantly high frequencies are applied.

III. CODE DESIGN AND IMPLEMENTATION

A. File Structure

The code for this project was divided into several different files, allowing for clearer organisation of code and separation of the user interface from the implementation. Individual classes were declared in *header* files (.hpp) that could then be defined in their respective *source* files (.cpp). For instance, ideal and non-ideal resistor components were both declared in *resistors.hpp*, while their function definitions were implemented in *resistors.cpp*. These classes were then utilised within the *main.cpp* file which acted as the user interface. This file was capable of storing user-created libraries for components and circuits, which could then provide details of a given circuit such as its impedance.

B. Hierarchical Inheritance

Hierarchical inheritance is a feature of OOP that allows for multiple derived sub-classes to inherit members

from a base class. For example, a base class can include properties such as data and methods which are common to each sub-class, but the functionality of these methods can be extended differently within each sub-class. This project utilises a three-tier hierarchy. An abstract base class `component` contained protected member data such as the type, impedance (stored using the STL class `complex<double>`) and frequency of the component. This base class also included pure virtual functions which were then overridden by the sub-classes, thus allowing for runtime polymorphism. This can be seen in Listing 1, where the protected member data is shown.

Listing 1: Abstract base component class.

```
class component
{
protected:
  std::string type;
  std::complex<double> impedance;
  double frequency;
  // Series (s) or parallel (p):
  char connection_type;
```

From this base class the ideal resistor, inductor and capacitor were derived by public inheritance. The pure virtual functions were overridden in each of these classes to adapt to the different physical properties of the components. The classes then contained their respective property (resistance, inductance, capacitance) as private data to ensure encapsulation, and could be accessed by `public` get/set methods.

In the third tier, realistic non-ideal components inherited from the respective ideal component, however these included additional private member data for the non-ideal physical values. This member data was then used within the `set_impedance` functions to override the impedance calculation for the non-ideal components.

Listing 2: Resistor class parameterised constructor.

```
void resistor::set_resistance(const double &res)
{
  if (res < 0.0) {
    throw std::out_of_range{"Cannot have negative
    ↪  resistance."};
  }
  else if (res <= 1e-9) {
    throw std::out_of_range{"Resistance must be
    ↪  above 1 nOhm."};
  }
  else if (res >= 10e9) {
    throw std::out_of_range{"Resistance must be
    ↪  below 10 GOhms."};
  }
  resistance = res;
  // To ensure impedance is adjusted correctly:
  set_impedance();
}
```

All of the derived classes stated above contained parameterised constructors which utilise localised exception handling (discussed in more detail in Section III E). An example of this is shown in Listing 2, where the object is only created if the resistance lies within the range of possible values. Furthermore, every derived class contained move and copy constructors which are crucial for usage with unique pointers.

C. Circuit Class

In addition to the classes given above, a `circuit` class was derived by public inheritance from the `component` class so that users could add circuit elements together. These components could be stored within a `vector` of base class pointers so that runtime polymorphism could be implemented as discussed above. To add elements to this vector, the template function `add_component` was created as shown in Listing 3. This function was both declared and defined in the *circuit.hpp* header file to prevent linkage errors since template classes are generated on demand, and so this was achieved using the namespace `circuits`. A `remove_component` was also included so that the user could remove an element from a circuit of their choice. Three print functions were also added, enabling the user to output the details of a circuit, a list of the component within the circuit or a diagram of the circuit. The results of these print statements can be seen in Figs. 2, 3 and 4.

Listing 3: Circuit class `add_component` function.

```
using namespace circuits;

// Add components to circuit in series or parallel:
template <class T> void circuit::add_component(
  std::shared_ptr<T> &comp,
  const char &conn, const bool &nest)
{
  circuit_comps.push_back(comp->clone() );

  // Set the member data for this component:
  circuit_comps.back()->set_connection_type(conn);
  circuit_comps.back()->set_nested_bool(nest);
  circuit_comps.back()->set_frequency(frequency);

  set_impedance();
}
```

D. User Interface

To access the functionality of the classes described above, a text-based interface was produced in *main.cpp* so that the user could create and modify libraries of

various components and circuits. These libraries were created as `static` data, where its memory was reserved for the entire runtime of the program. Smart pointers are used here to prevent memory leaks and bypass manual cleanup by giving ownership of an allocated resource to an lvalue object.

The interface consisted of a main menu that allowed access to different sub-menus through user input and can be seen in Fig. 1. To select a given option, `switch` statements were used where each `case` corresponded to one of the given options. For the main menu, each `case` called a function defined outside of `main` to improve code readability and usage.

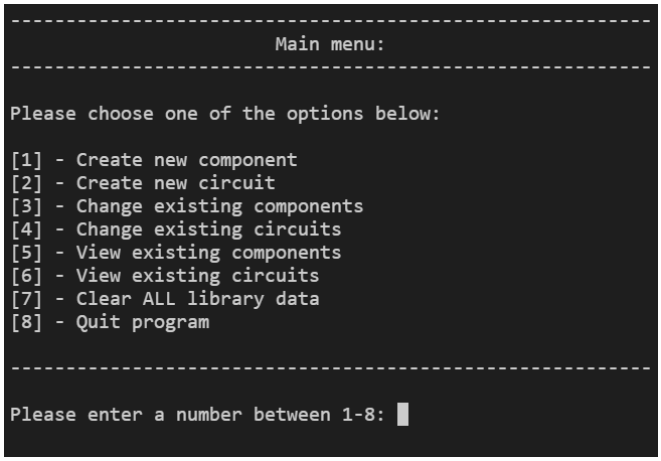

Fig. 1: Main menu of the user interface.

E. Input Validation

To test for valid input, a template function was created as seen in Listing 4. This allowed the code to check that user input was of the correct data type and continue only when the input was sensible. For example, if the interface requested for the resistance of a component which must be stored as a `double`, the code would loop until this condition was met. This template utilises advanced exception handling through the `try`, `throw`, `catch` method. The `try` code block searches for exceptions such as invalid input, which are then 'thrown' via `throw` to the `catch` code block which then decides how to process this error. Throughout the code, multiple `try-catch` code blocks were used to test for valid input from the user, including acceptable input within a given integer range (interface menus) or within the parameterised constructors of components themselves to ensure that the values lie within the appropriate physical limits. In all of these cases, the C++ standard library exceptions header, `<exception>`, was used to throw standard exceptions as objects such as an `invalid_argument` or an `out_of_range` error.

Listing 4: User input validation using template function.

```
template <class T> T valid_input(
  const std::string& prompt = "")
{
  T valid_input{};

  // Loops until user input is valid:
  bool is_valid = false;
  while (!is_valid) {
    try {
      std::cout << prompt;

      std::string user_input{};
      std::getline(std::cin, user_input);
      std::istringstream stream{user_input};

      // True if it can go into valid_input:
      if (stream >> valid_input && !(stream >>
      ↪  user_input) ) {
        is_valid = true;

      } else {
        throw std::invalid_argument{"Incorrect
        ↪  input type."};
      }
    }
    // Any incorrect input thrown here:
    catch (const std::invalid_argument& ia) {
      std::cout << std::endl;
      std::cerr << "Invalid argument error: ";
      std::cout << std::endl
      << ia.what() << std::endl;
      std::cin.clear();
    }
  }

  return valid_input;
}
```

IV. RESULTS

In order to evaluate whether the code functioned correctly, various RLC series and parallel combinations were tested. The simple case of an ideal RLC series circuit was first considered, constructed with the parameters 100 Ω, 10 mH, 20 μF. From Eqns. 3 and 2, the expected value for the magnitude and phase of the total circuit operating at 60 Hz and 230 V was therefore 163.11 Ω and -0.91 rad respectively. The results for the circuit simulator can be seen in Fig. 2, which agree with this manual calculation. Furthermore, the impedance of each component within the circuit was displayed through one of the user options, where the results are shown in Fig. 3.

The code was also capable of additional functionality such as the output of a diagram for a given circuit. An example of this is shown in Fig. 4 for a more complex circuit consisting of multiple ideal and non-ideal components connected in both series and parallel loops. It was

```
Circuit information:
    Frequency, f = 60.00 Hz,
    Voltage, V = 230.00 V.

Total impedance of circuit:
    Magnitude = 163.11 Ohms,
    and phase = -9.11e-01 radians.

Total current, I = 1.41 A, with phase difference
between the voltage of -9.11e-01 radians.
```

Fig. 2: Output of the calculated values for the simple RLC series circuit.

```
Here is this list of components in this circuit in order:

Component 1 is in series - Resistor:
    Resistance, R = 100.00 Ohms.
    Magnitude = 100.00 Ohms,
    and phase = 0 radians.
Component 2 is in series - Inductor:
    Inductance, L = 1.00e-02 H,
    Magnitude = 3.77 Ohms,
    and phase = 1.57 radians.
Component 3 is in series - Capacitor:
    Capacitance, C = 2.00e-05 F.
    Magnitude = 132.63 Ohms,
    and phase = -1.57e+00 radians.
```

Fig. 3: Output showing the impedance values for each component in the simple RLC series circuit.

```
Note: UPPERCASE = ideal / lowercase = non-ideal.

 O
 |
(~)
 |
 o--[R]--[C]--[l]--[r]
 |
[L]
 |
 o--[r]--[C]--[c]
 |
[R]
 |
 O
```

Fig. 4: Output diagram of a complex circuit consisting of both ideal and non-ideal components.

found that the results for the magnitude and phase were again in agreement with the expected values.

V. DISCUSSION AND CONCLUSIONS

This project successfully demonstrated the object-oriented implementation of an AC circuit simulator using the C++ language. The code possessed the key concepts of OOP including encapsulation, inheritance and polymorphism through appropriate class hierarchy. The ability to create a library of components allowed the user to connect these in series and parallel arrangements to build circuits of arbitrary length, where calculations of the impedance for each comp or circuit was then outputted.

Additional functionalities including a clear user interface, input validation through exception handling, libraries of different circuits and the ability to output a circuit diagram all improved user experience. Furthermore, the inclusion of non-ideal components allowed for more complex AC circuits to be created, while advanced C++ features such as smart pointers, template functions, static data and namespaces improved the underlying code.

Further design choices could have been implemented to improve overall functionality, such as the option to add two circuits together either in series or parallel, which could be achieved by combining the component vectors and calculating the overall impedance. In addition to this, a power source component could be included which must be added to a circuit, also making the joining of two circuits easier to implement. This component could also introduce a time-varying voltage, making for more interesting impedance calculations. The code could also be extended by introducing additional components such as transformers or diodes.

A graphical user interface such as Qt5 could be implemented, allowing the user to add components to an AC circuit schematically, thus displaying the circuit in a more visually appealing manner than through the terminal [5].

An attempt was also made to add nested series and parallel circuits. All the required functionality of this concept, including adding nested components to a circuit through valid user input, was successful. A recursive calculation of the total impedance for a circuit was tested, however the results for the circuits tested were inconsistent with the expected values, and thus this feature was subsequently removed.

[1] I. S. Grant and W. R. Phillips, *Electromagnetism*, Manchester Physics Series, Wiley, 2 ed., 2013.

[2] P. Horowitz and W. Hill, *The Art of Electronics*, Cambridge University Press, New York, 3 ed., 2015.

[3] C. Alexander and M. Sadiku, *Fundamentals of Electric Circuits*, McGraw-Hill, 3 ed., 2007.

[4] S. Kirchhoff, *About the passage of an electric current through a plane, especially a circular one*, Ann. Phys. **140** (1845), no. 4 497.

[5] The Qt Company, *Qt - cross-platform grapical user interface*, https://www.qt.io, Accessed: 17 May 2021.