AC Circuits:
A C++ Project

Thomas John Ross
Student ID: 10138372
School of Physics and Astronomy,
University of Manchester
(Dated: May 4, 2023)

This report presents a program that simulates AC circuits using object orientated programming in C++ and standard libraries. The program allows nested parallel and series circuits with five components: transistor, diode, resistor, inductors and capacitor. The complex impedance and phase difference are calculated for each component and nested circuits. Subsequently, the overall impedance and current are calculated for the entire circuit.

CONTENTS

I. INTRODUCTION AND THEORY

Circuits involve two types of current:, direct current (DC) and alternating current (AC). DC is a unidirectional current that can change in magnitude; when it does, it is called a pulsating DC. In contrast, AC is a current that varies in both magnitude and direction, changing in amplitude and polarity [1]. Circuits can be constructed in parallel and in series with various connecting components, such as resistors, inductors, capacitors, transistors and diodes. The relationship between the voltage and current is quantified by the impedance. It is described by Ohm's law,

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

where V is the periodic voltage, Z is the impedance and I is the periodic current. The impedance for an ideal resistor is given by,

$$Z_{\mathrm{R}} = R, \tag{2}$$

where R is the resistance of the resistor in units of Ohms [1]. The impedance for an ideal capacitor is given by,

$$Z_{\mathrm{C}} = \frac{-i}{\omega C}, \tag{3}$$

where C is the capacitance in units of Farads and ω is the angular frequency of the current flowing through the components [1]. The inductance is given by,

$$Z_{\mathrm{I}} = i\omega L, \tag{4}$$

where L is the inductance in units of Henrys.

The impedance of a transistor:

$$Z_{\mathrm{T}} = R_{\mathrm{eff}}, \tag{5}$$

where R_{eff} is the effective resistance of the transistors small-signal model is given by,

$$R_{\mathrm{eff}} = \frac{V_{\mathrm{BE}}}{h_{\mathrm{FE}} I_{\mathrm{B}}}, \tag{6}$$

where V_{BE} is the base-to-emitter voltage which is assumed to be 0.7 V, h_{FE} is the transistor's current gain, and I_{B} is the base current in units of Amperes [2].

The diode's impedance depends on a bias condition, which states that the forward-biased diodes have a low impedance and the reverse-biased diodes have a very high impedance [2].

The report outlines an object orientated programming (OOP) C++ approach to handling AC circuits. We will show a method for calculating the impedances, phase difference and current for an AC circuit with components connected in parallel or series, capable of handling multiple levels of nested circuits. We will demonstrate the uses of classes, class hierarchy, smart pointers, error handling, input validation, and other advanced C++ features.

II. CODE DESIGN AND IMPLEMENTATION

The program was designed with a simple class structure. An abstract base class was created, called **component_class**, which served all the component classes. A pure virtual function called **get_impedance** was declared, which returns the impedance of a component at a specific frequency. Two other functions called **get_magnitude_impedance** and **get_phase_difference** leverage this function to return the magnitude of the impedance and phase difference,

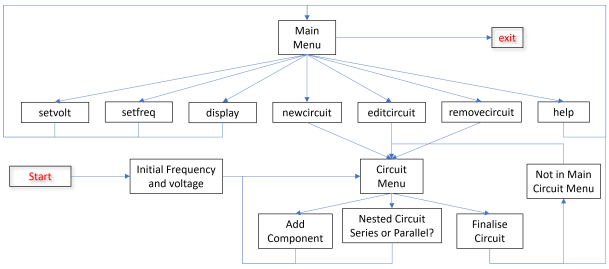

FIG. 1. Flow diagram illustrating user navigation through the program. Notice that the circuit menu allows the user to add components or a nested circuits as well as finalise a nested circuit or main circuit. When finalising if the circuit is not in the main circuit menu and remains in a nested circuit menu, then it will loop back around to the circuit menu.

respectively. The code is broken up into a main function, in which various features of the code are executed, with additional *header* files (.h) containing declarations and *source* files (.cpp), containing the definitions.

A. Class Hierarchy

The resistor, capacitor, inductance, transistor and diode classes are derived classes that publicly inherit from the base component class, which means they can access the public and protected members of the base class. Thus, the code makes use of polymorphism in that the derived classes are derived from the abstract class containing purely virtual functions. There is class called **circuit_class**, which inherits from **component_class** and includes additional data members and member functions. All these classes show the overall class hierarchy.

B. Smart Pointers

Smart pointers are an advanced feature within C++ 11, similar to normal pointers but with advanced memory allocation features. **std::shared_ptr** is used extensively throughout the code, and it is a smart pointer that shares ownership of an object through a pointer. It allows **shared_ptr** objects to own the same object, and ensures that **circuit_class** objects are deallocated automatically when they are no longer in use, preventing memory leaks.

C. User Inputs

The user is initially given the option to pick the frequency and voltage for the circuit. Next, the user is taken into a menu where they can choose to add a component in series, a nested series circuit, a nested parallel circuit or finalise the circuit by picking integers 1, 2, 3, and 4, respectively. The circuit cannot be finalised unless the user adds a component or nested circuit. If an attempt is made to finalise the circuit without a component or nested circuit then the user is prompted to do so.

We utilise an interactive command-line interface for user interaction. The interface presents a range of options defined using the **enum** construct. The commands available to the user are **setfreq**, **setvolt**, **display**, **newcircuit**, **editcircuit**, **removecircuit**, **help**, and **exit**. By default, the **enum** construct assigns integer values to the **Command** constants starting from 0 and increasing in increments of 1 for each subsequent constant.

The **get_command()** function reads a string from the user and converts it into one of the **Command** constants, and it is shown in Listing 1. If the input string matches the **Command** constant, then the corresponding **Command** constant is returned; if not then Invalid is returned.

```
1  // Commands user can input
2  enum Command { SetFreq, SetVolt, Display,
       NewCircuit, EditCircuit, RemoveCircuit, Exit
       , Help, Invalid };
3  // Converts user input into command
4  Command get_command() {
5      std::string input;
6      std::cin >> input;
7      if (input == "setfreq") return SetFreq;
8      if (input == "setvolt") return SetVolt;
9      if (input == "display") return Display;
10     if (input == "newcircuit") return NewCircuit
       ;
11     if (input == "editcircuit") return
       EditCircuit;
12     if (input == "removecircuit") return
       RemoveCircuit;
13     if (input == "exit") return Exit;
14     if (input == "help") return Help;
15     return Invalid;
16 }
```

Listing 1. Shows the command enumeration and the user input processing.

The code snippet in Listing 1 is connected to a **switch** statement in the main function, which uses the **Command** constant to determine what action to take next.

The **setfreq** and **setvolt** commands allow the user to change the frequency and voltage of the circuit, respectively, after initially setting them. The **display** command enables the user to view information regarding the circuit, its nested circuits, and their components. The **newcircuit** command lets the user create a new circuit configuration, while the **editcircuit** command allows them to add additional components and nested circuits to existing configurations, offering flexibility after finalising a particular configuration. Additionally, a **removecircuit** feature allowed the user to delete nested circuits by entering the name of the desired circuit when prompted. The **exit** command ends the loop and returns from the main function, terminating the program. The input sequence is shown in Fig. 1.

std::map was used as an associative container that stores elements formed by a combination of a key value and a mapped value. The map is used to store all **circuit_class** objects, and when a nested circuit is created,

it is added to the circuit's map with its name being a key. The map **circuit_parents**, keeps track of the relationship between circuits.

D. Error Handling

There are catch blocks placed at the end of the main function to handle exceptions. Three different catch blocks catch exceptions of type: **std::invalid_argument**, **std::exception** and **{...}** are used. The block that catches exceptions of type **std::invalid_argument** catches arguments of the right type but an invalid value. The block that catches exceptions of type **std::exception** handles all valid standard exceptions not caught by the previous block. The block that catches type **{...}** exception catches all exceptions not previously caught by previous block. It cannot provide any information on the exception it has caught but is used as a fallback to prevent the program from terminating unexpectedly.

Utilising exception handling, Listing 2 demonstrates this feature in handling the impedance calculation in parallel circuits.

```
std::complex<double> comp_impedance = comp.first
    ->get_impedance(frequency);
if (comp_impedance == std::complex<double>(0.0,
    0.0)) {
    throw std::invalid_argument("Division by
    zero error; Component impededance is zero");
}
total += 1.0 / comp_impedance;
```

Listing 2. Zero division handling

E. Input Validation

Input validation checks were applied to ensure only sensible user inputs were accepted. An example of this is shown in Listing 3, where an integer is set between a specific range using integer parameters **min** and **max**, where the integer values can be adjusted for each use. '`!(std::cin >> value)`' checks for non-integer inputs by detecting if the input stream is in a failed state. '`std::cin.peek() !=`' checks if there are any additional characters in the input stream after the integer value has been entered. If this is the case it will continue to prompt the user for a valid integer.

```
// Function to validate and retrieve an integer
    within a specified range from the user
int get_valid_int(int min = std::numeric_limits<
    int>::min(), int max = std::numeric_limits<
    int>::max()) {
    int value;
    while (!(std::cin >> value) || std::cin.peek
    () != '\n' || value < min || value > max) {
        std::cin.clear();
        std::cin.ignore(std::numeric_limits<std
    ::streamsize>::max(), '\n');
        std::cout << "Invalid input. Please
    enter an integer between " << min << " and "
     << max << ": ";
    }
    return value;
}
```

Listing 3. Input validation checks on integer variable

III. RESULTS

After choosing components and their associated values for each nested circuit and the main circuit as mentioned in Section II C, there is a final output of the type shown in Listing. 4. It shows the impedance and phase difference of each component for the entire circuit set to three significant figures. The output also includes the current and voltage across the entire circuit. The final circuit layout is presented, with components represented by their starting letter (e.g., a diode corresponds to 'D'). Each nested circuit is illustrated by square brackets surrounding it, [...], with parallel components represented by '`||`' and series components shown as '`->`'.

```
==========================================
           Circuit Analysis
==========================================
Entire Circuit:
  Frequency: 100 Hz
  Impedance: 20 + -7.96e-05j ohms
  Magnitude of impedance: 20 ohms
  Phase difference: -3.98e-06 radians
  Voltage: 20 V
  Current: 1 A
==========================================
Main Circuit:
  Frequency: 100 Hz
  Impedance: 20 + -7.96e-05j ohms
  Magnitude of impedance: 20 ohms
  Phase difference: -3.98e-06 radians
Resistor 1:
  Frequency: 100 Hz
  Impedance: 20 + 0j ohms
  Magnitude of impedance: 20 ohms
  Phase difference: 0 radians
Nested Parallel Circuit:
  Frequency: 100 Hz
  Impedance: 0.0035 + -7.96e-05j ohms
  Magnitude of impedance: 0.0035 ohms
  Phase difference: -0.0227 radians
Nested Series Circuit:
  Frequency: 100 Hz
  Impedance: 0.0035 + -7.96e-05j ohms
  Magnitude of impedance: 0.0035 ohms
  Phase difference: -0.0227 radians
Capacitor 1:
  Frequency: 100 Hz
  Impedance: 0 + -7.96e-05j ohms
  Magnitude of impedance: 7.96e-05 ohms
  Phase difference: -1.57 radians
Transistor 1:
  Frequency: 100 Hz
  Impedance: 0.0035 + 0j ohms
  Magnitude of impedance: 0.0035 ohms
  Phase difference: 0 radians
Inductance 1:
```

```
  Frequency: 100 Hz
  Impedance: 0 + 9.42e+03j ohms
  Magnitude of impedance: 9.42e+03 ohms
  Phase difference: 1.57 radians
Diode 1:
  Frequency: 100 Hz
  Impedance: 0 + 0j ohms
  Magnitude of impedance: 0 ohms
  Phase difference: 0 radians
==========================================
Circuit layout: [ R -> [ [ C -> T ] || I ] -> D
    ]
```

Listing 4. Output of circuit configuration with a resistor and inductor in series with a capacitor and resistor in a nested parallel circuit.

IV. DISCUSSION AND CONCLUSION

In summary, an OOP C++ approach was demonstrated for calculating and presenting the impedances, phase difference, and circuit layout for connected parallel and series nested circuits. The code functionalities have been displayed, including polymorphism and inheritance, smart pointers, mapping, exception handling and abstract classes.

While the current implementation provides a robust and flexible foundation for analysing AC circuits, there is still room for improvement and expansion. Future work could implement a graphical user interface (GUI) allowing graphics-based representation of an AC circuit. Additionally, incorporating a feature to save circuit configurations would allow users to retrieve previously designed circuits and build upon them or use them for further analysis.

[1] S. R. Fulton and J. C. Rawlins, *Basic AC circuits* (Sams, 1981).

[2] R. L. Boylestad and L. Nashelsky, *Electronic devices and circuit theory* (Pearson Education India, 2009).