AC CIRCUITS

George McNie

10635213

PHYS30762- Object-Orientated Programming in C++

Project Report

School of Physics and Astronomy

The University of Manchester

May 2023

Abstract

The objective of this C++ programming project is to simulate an arbitrary AC circuit; containing resistors, capacitors and inductors. The total impedance, phase and magnitude is calculated of a circuit made by user input, as well as the impedance, phase and magnitude of each individual component in the circuit. This is achieved using a polymorphic component class and a circuit class. When building a circuit, the user can add components in series and parallel as well as including nested circuits. Once circuits have been defined, the circuit diagrams may be displayed visually. The purpose of this report is to explain the functionality and code features of the program.

Contents

1. Introduction

This program aims to simulate AC circuits in C++ making use of advanced features. The minimum specification design asserts that there must be an abstract class for components from which derived classes can be made: resistor, capacitor and inductor. There must also be an additional class for the circuit, to which components can be added in parallel or series after initialisation, the total circuit impedance is stored in the class. In terms of functionality a user must be able to create a component library from which the circuit can be constructed.

The features in this code that go beyond this main specification include creating nested circuits, having non-ideal components and outputting a visual representation of the circuit.

AC circuit analysis is an essential part to most of modern electronics [1] with the power most electrical devices run on being AC. Therefore, it is essential to be able to model and understand these systems quickly, while having access to all relevant information about the circuit. The first attempt at this was SPICE in 1973, with it's derivatives still being used today [2]. Whilst this simulation doesn't have all of the features of SPICE it attempts to simulate circuits of simple components, using the theory discussed in section 2.

2. Theory

The key concept in AC circuits is that of impedance, Z [3]. Impedance is a complex quantity which is a measure of opposition to current flow. Different components have different impedances which are dependent on the circuit's frequency. For a resistor R, capacitor C, and inductor L the impedances are calculated as,

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

where ω is the angular frequency of the current and j is an imaginary number. This assumes that components are ideal. In reality each component can have properties that effect the rest of the circuit, this is when the component is non-ideal and called a parasitic component. For instance a resistor has a parasitic capacitance and inductance as well as it's resistance, effecting the impedance.

To add obtain the total impedance of a circuit, series components add as,

$$Z_{series} = Z_1 + Z_2 \tag{2}$$

and parallel components as,

$$\frac{1}{Z_{parallel}} = \frac{1}{Z_1} + \frac{1}{Z_2}. \tag{3}$$

3. Code design & Implementation

3.1 Algorithm design

The user interface was designed to be a text-based menu, with menus that gave a range of options for the user to choose from. Using these menus, the intended flow of the program can be seen in Figure 1.

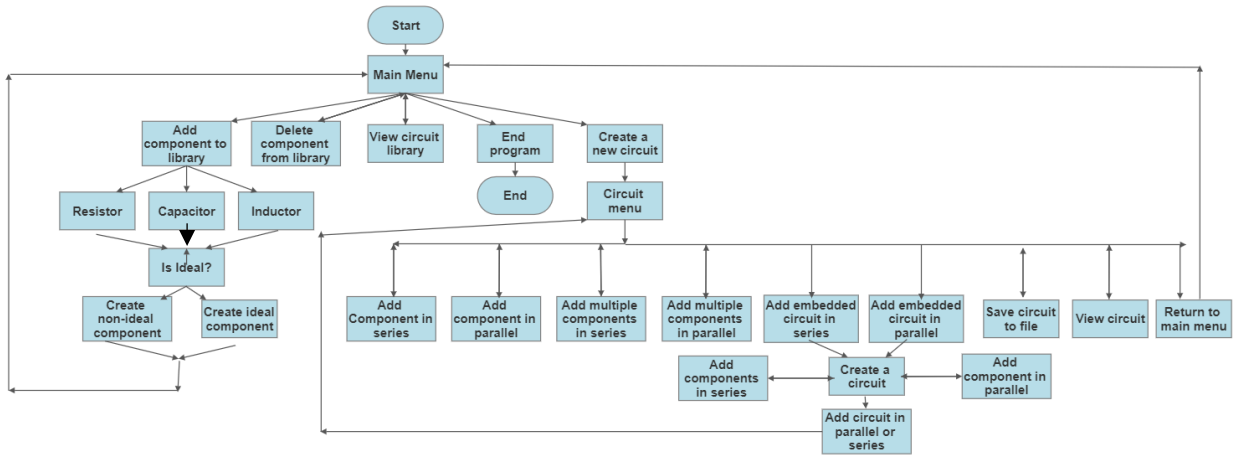

Figure 1 Flow diagram of program. Backward arrows mean once that command is finished it returns to the previous command.

For the program to work as wanted the user should create several components which are stored in the component library. The component library can be manipulated and viewed. Then the user can create circuits using these components. Components can be added in series or parallel, and embedded circuits can be added. A function to save the circuit to file is also designed to be implemented. At any stage the user will be able to see what they have constructed thus far. Once they are finished with their circuit, they can return to the main menu to add more components to the component library or create a new circuit.

3.2 Class design

All classes, and their children have their own header and implementation file which contain the class declaration and definition respectively. Header guards are implemented to prevent multiple inclusions of the same header file. Two separate classes are used in this code, one makes objects of type component and the other object of type circuit.

3.2.1 Component class

The component class utilises polymorphism. A virtual component base class is made which all subsequent components will inherit from, this class has protected variables impedance, frequency and name which are universal properties of all components, as seen in Figure 2.

```
class component //pure virtual class, has children resistor, capacitor and inductor
{
protected:
    std::complex<double> impedence; //universal to all derived classes
    double frequency;
    std::string name;
public:
    //default constructor
    component(){}
    //destructor
    ~component() {}
    //pure virtual functions
    virtual double get_frequency() = 0;
    virtual void set_frequency(double f) = 0;
    virtual std::complex<double> get_impedance()= 0;
    virtual void set_impedance(std::complex<double> im_in)=0;
    virtual double get_magnitude()  = 0;
    virtual double get_phase() = 0;
    virtual void basic_details() = 0;
    virtual void set_name(int n) = 0;
    virtual void set_name(std::string n) = 0;
    virtual std::string get_name() = 0;
    virtual auto clone() const -> std::shared_ptr<component> = 0;
};
```

Figure 2: Virtual component class. Has functions to get and set variables, print basic details of a component and clone a component.

Next 3 child classes are made, for a resistor, capacitor and an inductor. These inherit the functions seen in Figure 3 and are different depending on the object to line up with the theory seen in Section 2.

```
class para_resistor : public resistor
{
protected:
    double para_c;
    double para_l;
public:
    //default constructor
    para_resistor() : resistor() { para_c = 0; para_l = 0; };
    para_resistor(const double c_in, const double r_in, const double l_in);
    ~para_resistor(){}

    //overriding
    void set_frequency(double f);
    void set_name(int n);
    void basic_details();
    auto clone() const -> std::shared_ptr<component>;

};
```

Figure 3: Parasitic resistor class, grandchild of component class and child of resistor class.

Each of these resistor, capacitor and inductor classes have a child class, used in the instance that these components are non-ideal. Non-ideal components have other properties effecting their impedances (such as a resistor having parasitic resistance and inductance), this is seen implemented in Figure 3.

The component class has a function declared `clone` so that it can clone the component in the component library as a deep copy. The reason for doing this is building for future development of the program. If the current and voltage through each component wanted to be calculated using Kirchhoff's laws and the same component was used twice in the circuit, a standard pointer to the component in the component library wouldn't work as the object would have two different possible values. Therefore, if a deep copy of the component is made and thus stored in the circuit class (see Section 3.2.2) this is possible for the future.

3.2.2 Circuit class

The circuit class is designed to recursively calculate the impedance of the circuit when new components are added in. The circuit diagram, `text_description` is also recursively updated when a component is added into the circuit. As discussed in Section 3.1.1, for future proofing of the code, when a component is made a new copy of the component in the component library is stored in the circuit class.

The circuit's member classes include the crucial `add_parallel` and `add_series` that are used to recursively add the impedance of the circuit when new components are added into the circuit. These functions are overloaded to take arguments to add just one component and to also take the argument of a pointer to an object of type circuit so that embedded circuits can be added.

3.3 Input management

The code relies heavily on the user inputting doubles and integers so a template function is used to reduce the need for multiple functions, as seen in Figure 4. The code is designed to only ever take numerical inputs so this template function is used throughout.

```
template <class c_type> c_type valid_numeric_input(c_type min,c_type max) //template so can validate in integers or doubles depending on what needed
{
  c_type input;
  while (true) {
    std::cin >> input;
    if (std::cin.fail()) { // If nothing is entered it will be true
      std::cin.clear(); // For resetting the bits
      std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n'); //ignore() is used to clear the wrong input entered
      std::cout <<"Please enter a valid input: ";
      continue; //runs loop again
    }
    std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n'); // remove the additional input from the stream
    if (std::cin.gcount() > 1) { // it will return more than 1 if there is something which is not extracted properly
     std::cout <<"Please enter a valid input: ";
     continue;
    }
    if (input > max || input < min){
        std::cout <<"Please enter a input in valid range: ";
        continue;
    }
```

Figure 4: A code snippet of function template for user input

One of the main uses of this template is in getting an integer in a specified range for a menu. The parameters taken in allow for only specific integers to be inputted at this point, seen in Figure 5.

```
int option = valid_numeric_input<int>(1,3);
```

Figure 5: Code snippet showing use of number template when using a menu

The other use of this template is specifying component properties and so it takes a double here, as seen in Figure 6.

```
c = valid_numeric_input<double>(0.0, 1000000000.0);
component_library.emplace_back(new capacitor(c*1e-9));
```

Figure 6: Code snippet showing use of number template when making a component

3.4 Smart pointers

Smart pointers are used throughout the code to provide automatic memory management, avoiding memory leaks and providing a more efficient and safe way to manage memory compared to manual management.

Mainly shared pointers are used in the code, as multiple `std::shared_ptr` objects can own and manage the same object. It also allows for code to be copied between functions, so one can pass the pointer in the function arguments as seen in Figure 7.

```
std::shared_ptr<cir::circuit> created_em = embedded();
user_circuit->add_parallel(created_em);
```

Figure 7: Example of passing a shared pointer to a function `add_parallel` which means the pointer is copied and passed.

The current circuit being used is a unique pointer of type circuit so that it's automatically destroyed when the code goes out of scope.

3.5 Using Statics

Statics are used in this code to store the component library and the current user_circuit defined in interface.cpp meaning they can only be used and accessed in here. This stops them from being changed in all other files. The statics can be used throughout this file and this is done so that the component library can be accessed from all non-member functions. The circuit the user uses is also stored as this for this purpose. Memory is deallocated from these statics when exiting the program by using the `.reset()` function.

3.6 Exception handling

Exception handling to manage memory is achieved in the code by implementing it whenever a new class is made as seen in Figure 8. This means that whenever memory is allocated incorrectly the program will detect a thrown error and exit the code immediately.

```
try{user_circuit.reset(new cir::circuit(voltage, frequency));
}
//catch any exception of bad memory allocation
catch (std::bad_alloc) {
    std::cerr<<"Memory allocation failed!"<<std::endl;
    exit(1); //exit program with error code 3
}
```

Figure 8: Showing some of the error handling done when creating a new circuit

Exception handling is also used when outputting the circuit to a file to avoid run time errors and other errors.

3.7 Using namespaces

Two namespaces have been created in the code to stop any naming conflicts between the circuit's object and the components object as both have member functions `get_impedance`, `get_frequency` and `set_frequency`. The component classes (including all child and grandchild classes) fall under the "cop" namespace and circuit class under the "cir" namespace. Without the namespaces there would be confusion for the compiler and make it difficult to determine which function to call. By placing the classes in different namespaces, this issue is avoided and ensures that each function is called correctly.

3.8 Switch statements

Many menus are used in this code, allowing the user to input a number that gets validated see Section 3.2, which then takes them to a switch statement. The cases in the switch are denoted by integer numbers and execute that case if the corresponding integer is chosen. This made the code more concise than if long if/else statements were used and improves readability.

4. Results

The program accurately follows the algorithmic design specified in section 2.1.

The user interface uses an intuitive method to break up sections and declare that new ones are occurring by using "*" markers as seen in Figure 9. The intuitive menu allows for easy selection to

move between these sections, using numerical inputs as discussed in Section 3.3. The component library viewed in Figure 9 allows the user to always be able to check what they have inputted into it.

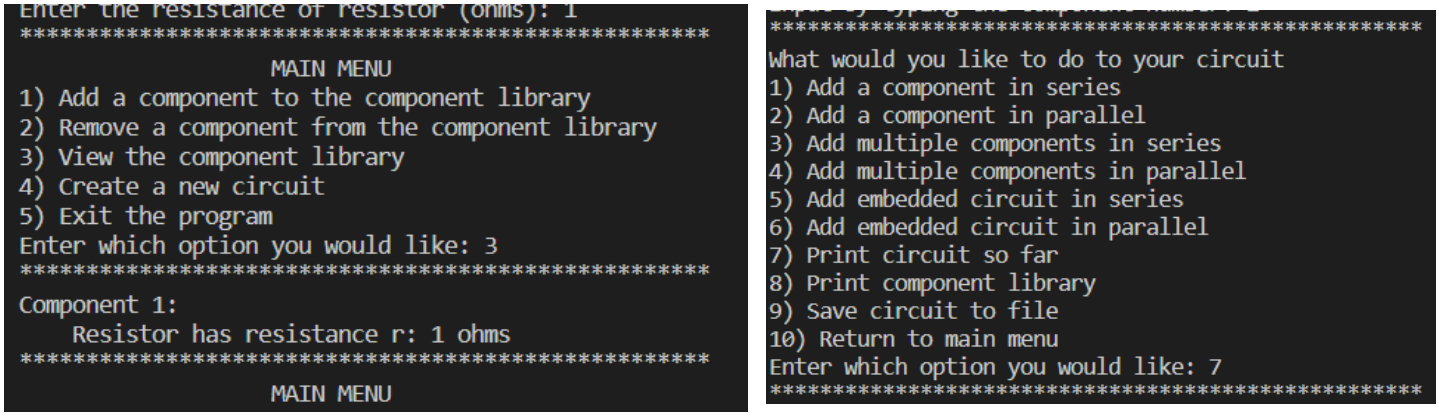

Figure 9: Main menu (left) and circuit menu (right) shown with section breaks giving intuitive feel to program sections

When outputting the circuit, a pause is used (using <chrono> library) to give the user time to read the circuit information that they have requested before going back to the circuit menu. The output of the circuit provides all the information about the circuit, including a circuit diagram where parallel parts are grouped with curly brackets {} and connected by || wires. Components added in series are grouped using square brackets [] and connected by – wires, this can be seen in Figure 10.

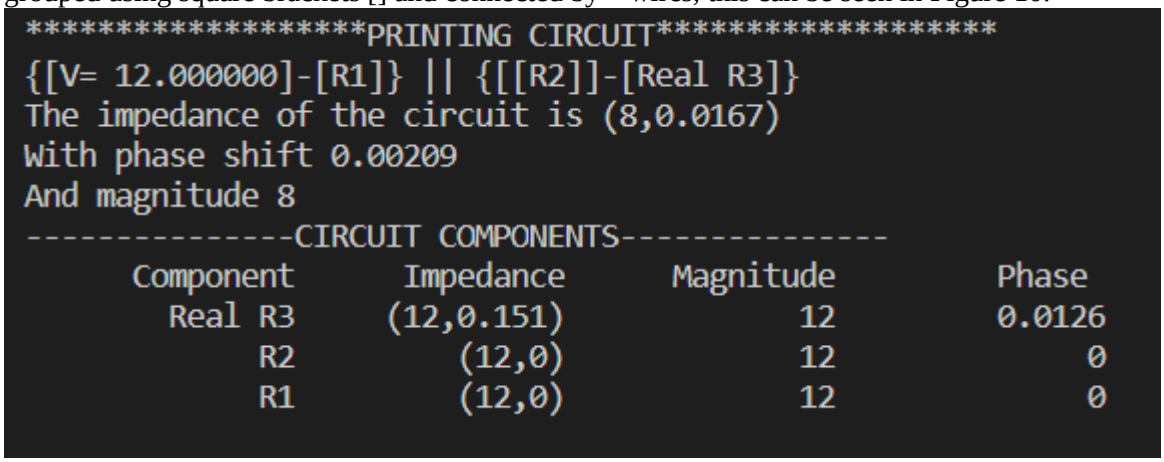

Figure 90: Output when print circuit

The output shows the individual component's impedance and phase as well as the total circuit's, with the output's precision set to 3 significant figures. This output can be saved to a file called anything the user wishes, to check if it's actually saved to file a pause is used again to either print an error message or one saying save was successful.

5. Discussion

Overall, this program meets all the specified minimum requirements and functionality, with additional features such as embedded circuits, non-ideal components, saving to file and a nice output of the circuit created. This was all done while using advanced code features.

To improve this program the currents and voltages through each of the components in the circuit could be calculated, making use of the "future proofing" discussed in Section 3.2.1. Other components could be added such as a transistor or diode making use of the values of current and voltage. In terms of other aspects that would affect the circuit's impedance non-ideal wires could be introduced as a component.

Another thing to improve the functionality of the program would be to let an embedded circuit have as much freedom as the main circuit; so, one could have an embedded circuit in an embedded circuit in perpetuity. This could be done by redesigning the implementation in the program so that the static `user_circuit` was stored in the circuit class and so embedded functions would have constant access to the object when working with the new embedded circuit object.

Finally, a better graphical representation of the circuit could be implemented using the current string of characters used to output a circuit. This would mean that a circuit diagram as standard in industry would be outputted and not just the string seen at the moment.

References

[1] - Trzynadlowski, A.M., 2015. *Introduction to modern power electronics*. John Wiley & Sons.

[2] - Dickerson, S.J. and Clark, R.M., 2021. Use of SPICE Circuit Simulation to Guide Written Reflections and Metacognition. *IEEE Transactions on Education*, *65*(3), pp.471-480.

[3] - Wing, O., 2008. *Classical circuit theory* (Vol. 773). Springer Science & Business Media.