Simulation of AC circuits using OOP in C++

A report submitted for the course of
Object-Oriented Programming in C++ (PHYS30762)

May 2021

Author:
Luis Mestre
(9916134)

Abstract

A program was developed in C++ to simulate AC circuits. The program is built solely with the standard library and allows a user to build circuits with an arbitrary number of components in series or parallel. The user can build circuits containing non-ideal resistors, inductors and capacitors. Then, they can obtain the current, voltage, impedance, and phase difference for each circuit. The results for the circuit calculations of these quantities were checked for a simple circuit and found to be correct. Additionally, nested circuits were also implemented to allow the user to build more complex circuit geometries by adding different circuits together.

Contents

1 Introduction

Alternating current (AC) is a type of current flow where the direction of charge flow inverts periodically [1]. This type of current is a cornerstone of modern society and is what makes mainstream access to electricity possible. Other applications of this current flow include radio, audio processing, wifi and many others.

This report will focus on a program developed in C++ to simulate these types of circuits. The program was developed using only the standard library and allows a user to simulate different AC circuits containing resistors, capacitors and inductors. The component non-ideal behavior was modeled using lumped element models containing parasitic effects. These effects are very important for high-frequency circuits as the impedance of the components can change substantially. The models that were used can be seen below in Fig. 1.1 [1] [2; 3].

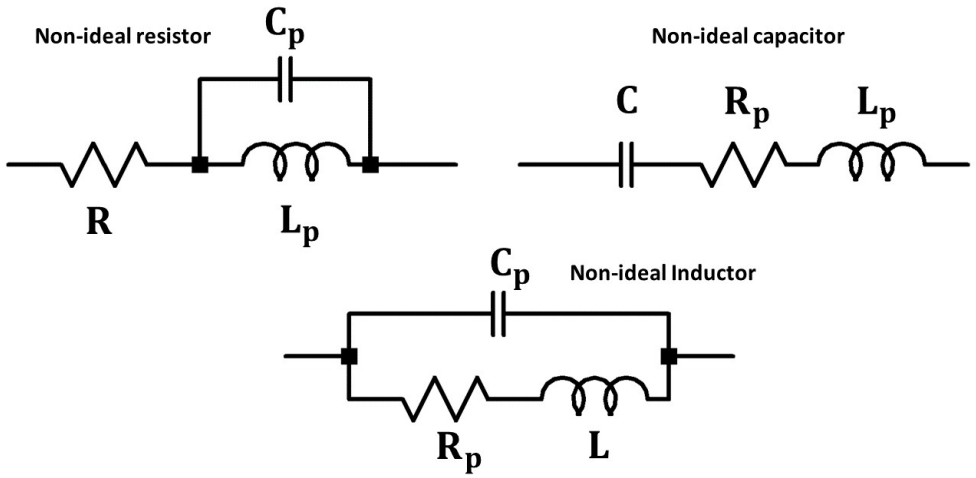

Figure 1.1: Diagram of the non-ideal models of circuit components used. R, C and L denote resistance, capacitance and inductance respectively. The subscript of p denotes parasitic effects.

1.1 Relevant theory

The total impedance of each non-ideal component was calculated by adding the respective contributions of its parasitic effects in parallel or series depending on the lumped element model. Each parasitic effect in the lumped element models is assumed to be a single ideal component. The phase shift between voltage and current and the equations describing the impedance and for each type of ideal component are given in Table 1.1 [1].

[1] Although I have made the diagram the models are from the references listed.

Ideal component	Impedance	Phase shift
Resistor	$\mathrm{Z}_R = R$	0
Capacitor	$\mathrm{Z}_C = \frac{1}{jwC}$	$-\frac{\pi}{2}$
Inductor	$\mathrm{Z}_R = jwL$	$\frac{\pi}{2}$

Table 1.1: Table containing the properties of each type of ideal component. j is $\sqrt{-1}$ and ω is the angular frequency. R, C and L are resistance, capacitance and inductance respectively.

The rules for adding impedance in series and parallel are given by equations [1]:

$$Z_{series} = \sum_{i=1}^{N} Z_i, \tag{1.1}$$

$$Z_{parallel} = \frac{1}{\sum_{i=1}^{N} \frac{1}{Z_i}}. \tag{1.2}$$

Where N is the number of components in series or parallel, Z_i is the impedance of each component and Z_{series} and $Z_{parallel}$ are the total impedances in series and parallel respectively. The total impedance of an AC circuit is given by Ohm's law for AC circuits [1]:

$$Z_{circuit} = \frac{\tilde{V}}{\tilde{I}}. \tag{1.3}$$

Where $\tilde{V}$ and $\tilde{I}$ are the AC voltage and current respectively. The phase difference between them is described by [1]:

$$\Delta\phi = \arg\left(Z_{circuit}\right). \tag{1.4}$$

2 Implementation

The project file structure is orgnised according to the diagram shown in Fig. 2.1. The main program contains the *int main()* function and all the user interface functions. In order to improve the readability of the code, the circuit and component classes were split across different files. Additionally, a libraries class was also made to store all the libraries and relevant library functions.

The *components.h* file contains the classes relevant to the individual components; the abstract component base class and the resistor, capacitor and inductor classes. The functions of these classes are all declared in the *components.cpp*. The abstract component base class is shown in Fig. 2.2 with all the pure virtual methods used in the resistor, capacitor and inductor classes. The large number of getters present is due to the need for a copy constructor for each component. The *circuits.h* file contains the base circuit and the nested circuit classes. To organise the code better, the nested circuit class inherits the circuit class but also stores a vector of *std::shared_ptr* pointers to circuit objects. This follows the "*is_a*" pattern and has the advantage that functions like *print_circuit_diagram()* can be

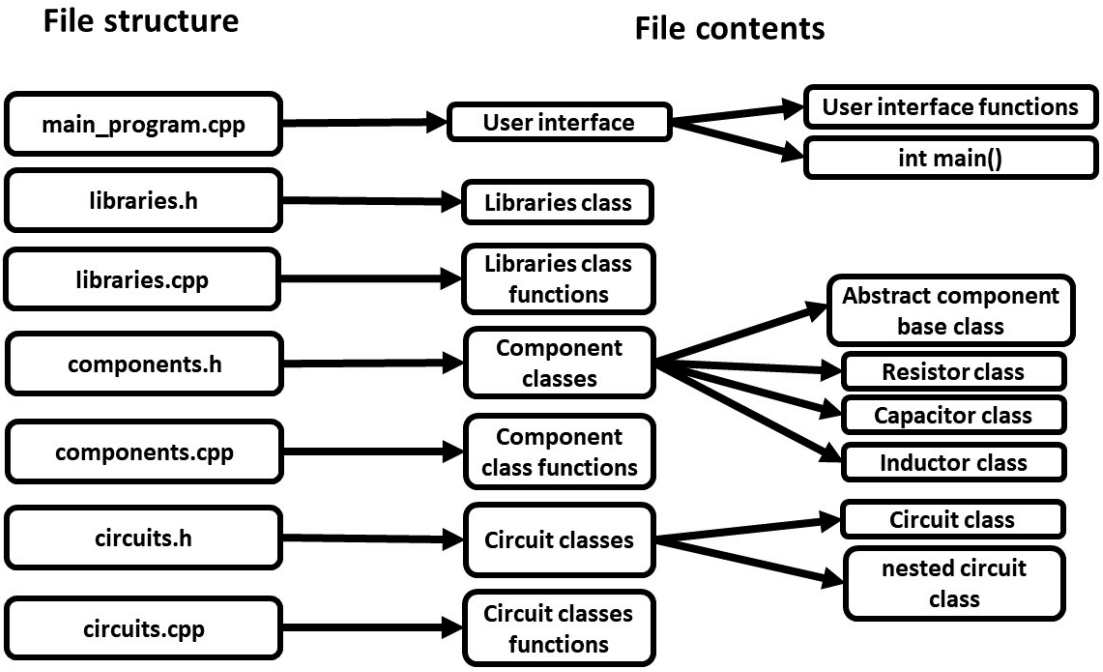

Figure 2.1: Diagram of the file structure and contents for the code of this project.

inherited from the circuit class instead of rewritten. As a result, the nested circuit class is much shorter. The contents of the circuit and nested circuit classes are shown in Fig. 2.3 and Fig. 2.4 respectively. For the circuit class, only the variables are shown due to the large number of class functions inside it.

```
// nested circuit class
class nested_circuit : public circuit
{
private:
    // stores circuits that make up nested cirucit
    std::vector<std::shared_ptr<circuit>> vector_of_circuits;
    // stores number of components of each circuit in the nested circuit
    std::vector<int> vector_of_circuit_sizes;
public:
    nested_circuit() {};
    nested_circuit(double voltage, double frequency, std::string name_parameter)
        : circuit(voltage, frequency, name_parameter) {};

    ~nested_circuit() {};

    // methods specific to this class
    void nested_impedance_series_sum(std::vector<std::shared_ptr<circuit>>);
    void nested_impedance_parallel_sum(std::vector<std::shared_ptr<circuit>>);
    void info();
    void circuit_change_check();

    // getters
    int get_size() const;
}; // end of nested circuit class
```

Figure 2.4: Nested circuit derived class showing all the methods specific to this class.

In the main program, three libraries exist; a component library, a circuit library and a nested circuit library. To avoid a very long main program code, a *Libraries* class was created to implement the three libraries. The data structure to store the libraries in this

```cpp
// base class for all components
class base_components
{
public:
    // virtual destructor
    virtual ~base_components() {};

    //pure virtual methods
    std::complex<double> virtual calculate_impedance(double) = 0;
    void virtual info() = 0;
    void virtual basic_info() = 0;

    // getters
    std::complex<double> virtual get_impedance() const = 0;
    double virtual get_phase_difference () const = 0;
    double virtual get_impedance_magnitude() const = 0;
    double virtual get_frequency() const = 0;
    std::string virtual get_name() const = 0;
    std::string virtual get_type() const = 0;
    double virtual get_component_value() const = 0;
    double virtual get_parasitic_1() const = 0;
    double virtual get_parasitic_2() const = 0;

    // setters
    void virtual set_frequency(const double) = 0;
}; // end of base class for components
```

Figure 2.2: Abstract component base class containing all the pure virtual methods that are overridden in the derived component classes.

```cpp
// circuit class
class circuit
{
protected:
    std::vector<std::shared_ptr<COMPONENTS::base_components>> circuit_components_vector;
    std::complex<double> impedance = 0;
    double voltage = 0;
    double frequency_Hz = 0;
    double current = 0;
    std::string circuit_text;
    std::string name = "None";
```

Figure 2.3: Beginning of the circuit base class with all the information about the circuit stored in the class.

class is summarised in Fig. 2.5. These are the variables stored inside the Libraries class. In the main program, these are stored in a static *Libraries* object which is instantiated at the beginning of the code. Within the object, the libraries are stored in different vectors of class pointers. For the component library, the program employs polymorphism to store the different components in a vector of *std::shared_ptr* to component base class objects. The circuit libraries however, are stored in vectors of *std::shared_ptr* which point to circuit and nested circuit objects for the respective libraries. Using pointers to nested circuit objects allows these objects to have access to the functions specific to that class. An important example of this is the *nested_impedance_series_sum* which is used in that class to perform the sum of impedances of a vector of circuits that are connected in series.

```
// variables to count number of members in each library
std::size_t component_library_number = 0;
std::size_t circuit_library_number = 0;
std::size_t nested_circuit_library_number = 0;
// counters for component numbers (used for component names during creation)
std::size_t resistor_count = 1;
std::size_t capacitor_count = 1;
std::size_t inductor_count = 1;
// library vectors
std::vector<std::shared_ptr<COMPONENTS::base_components>> component_library;
std::vector<std::shared_ptr<CIRCUITS::circuit>> circuit_library;
std::vector<std::shared_ptr<CIRCUITS::nested_circuit>> nested_circuit_library;
```

Figure 2.5: Data structure in the libraries class. It contains all the vectors for each library, the library number counters and the component number counters.

2.1 User interface

The *int main()* function in the main program consists of a call to a single function called *user_interface()* which calls all the other functions required based on user input. This function works in a cycle, the function prints out a menu and asks for user input. Then, it allows the user to perform an action based on the input and once the user is finished, it clears the terminal text and takes the user back to the menu to do something else. This occurs indefinitely until the user enters 'e'. This is illustrated in Fig. 2.6. The available menu is shown in Fig. 2.7 with all the possible functions including the demo circuit option which is covered in detail in Appendix A.

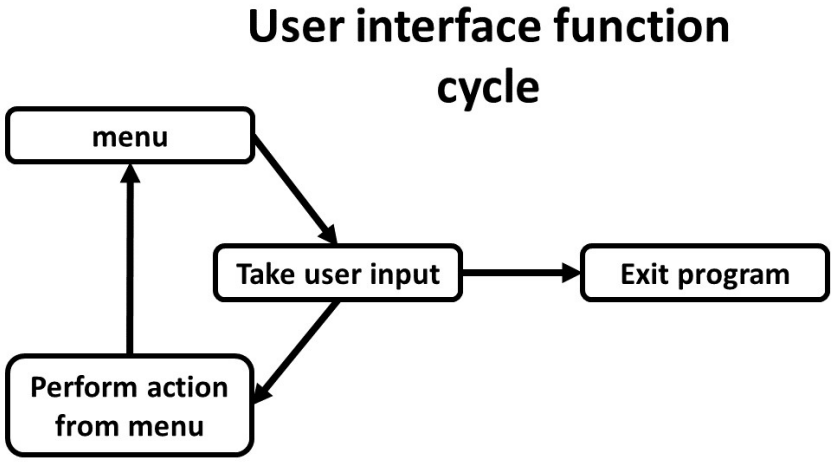

Figure 2.6: Flow chart showing the cycle of the user interface function.

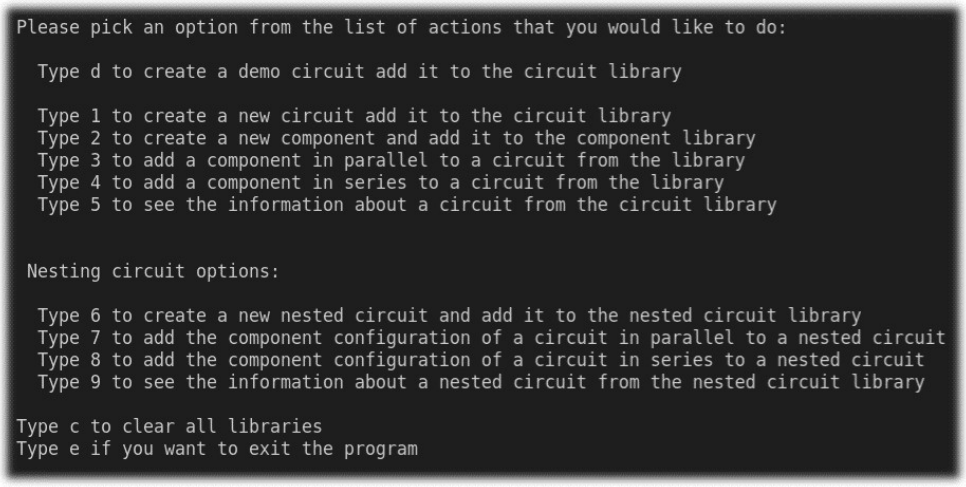

Figure 2.7: Console screenshot showing the user menu with all the options available to the user.

2.2 Implementation in code

The component classes model the components according to Fig. 1.1. When creating a component using option two in the menu, the user is asked for what type of component they would like to create with options r, c, l for a resistor, capacitor and inductor respectively. The user will be given the option to create an ideal or non-ideal component. If the user decides to create an ideal component, then only a resistance, capacitance, or inductance can be set by the user and the respective parasitic quantities will be set to 0. If the user chooses to create a non-ideal component, they will also be asked to enter the parasitic quantities for the component. A table containing the bounds of all the quantities, for the circuits and each type of component, is shown in Table. 2.1. The bounds of these quantities were estimated based on approximate values found in literature [1; 3]. For example, parasitic capacitances in components can often be in the order of a few pF, to allow for some clearance, nF was used as an upper bound. The large allowed parasitic resistance value for the inductors is due to them often being coils which can have large resistances due to the windings.

Electrical property	Object	Minimum value	Maximum value
Voltage	Circuits	1 mV	100kV
Frequency		1 mHz	10 GHz
Inductance	Inductor	1 fH	10 H
Parasitic resistance		0	1 kΩ
Parasitic capacitance		0	1 nF
Capacitance	Capacitor	1 fF	100 F
Parasitic resistance		0	1 Ω
Parasitic inductance		0	1 μH
Resistance	Resistor	1 mΩ	1 GΩ
Parasitic capacitance		0	1 nF
Parasitic inductance		0	1 μH

Table 2.1: Table containing the bound of all the values for each property of all components.

To keep track of the components, the program automatically assigns a name to it based on the number of already existing components of the same type. If a resistor is created for example, its name is set in the constructor as *"R"+ std::to_string(resistor_count)* and if it is the first resistor to be created its name will be R1. The numbers of components are tracked by the counter variables shown in Fig. 2.5. This makes sure components can later be distinguished in the circuit. When components are created by the user, they are automatically added to the component library vector.

To create circuits, the user must select option one in the menu and then set the frequency and voltage of the power supply as well as a name for the circuit. For components, the frequency is set using the *set_frequency()* function when they are added to a circuit in series or parallel. The frequency of each component is set inside the circuit class' functions for adding components in series and parallel. Inside these functions, a specialised copy constructor is used to copy the base component objects from a vector of *std::shared_ptr* containing the components that are being added to the circuit. This prevents the error of setting each component to a different frequency each time it is added to a different circuit which would cause the component impedances to change across the different circuits in the library. When the components are added to a circuit, the components' name is added to the string *circuit_text* seen in Fig. 2.3. If the components are added in series, then the name is added to the circuit text with a "-|s|", if in parallel, a "-|p|". When going to a new parallel wire, a "$\backslash n$" is added as a delimiter to distinguish between wires. This process allows the *circuit_text* variable in the circuit object to keep track of which components are in series or parallel to produce a circuit diagram. The circuit class effectively stores the circuit as parallel lines and sums the impedance within each line according to whether the components in the line are added in series or parallel.

3 Functionality

3.1 Test run with normal circuits

For a short test, a circuit with two parallel branches and a series branch was generated. The circuit information for this run is shown in Fig. 3.1. To check the results from this circuit ideal resistors were added for simplicity. The manually calculated results are:

$$\text{Total impedance } (\Omega) = \frac{1}{\frac{1}{45+45} + \frac{1}{90+90}} + 30 = 90. \tag{3.1}$$

$$\text{Current (A)} = \frac{90}{90} = 1. \tag{3.2}$$

$$\text{Phase difference (degrees)} = \tan^{-1}\left(\frac{0}{90}\right) = 0^{\circ}. \tag{3.3}$$

The way the circuit diagrams work is by printing the first line with all the components in series. The **connect upline** and **upline** symbolise a vertical connection between the two points. The rule is that each **upline** is connected to the **connect upline** of the previous line. This is depicted in the more elaborate run shown in Fig. 3.2.

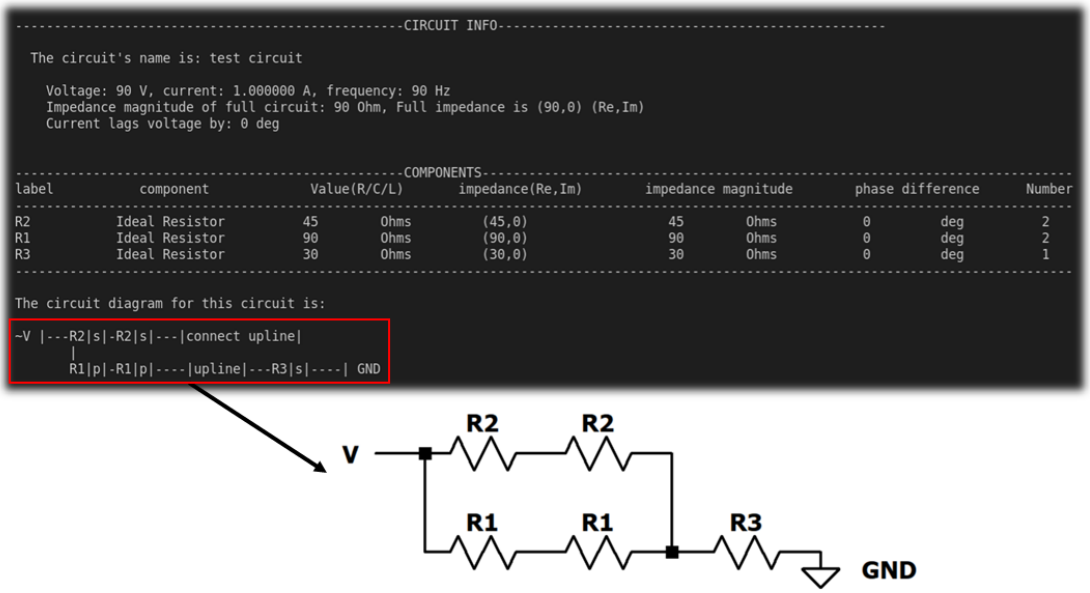

Figure 3.1: Example run of simple circuit with resistors to check results. A table with all the resistors present in the circuit and their information is shown in the top while a circuit diagram is shown below.

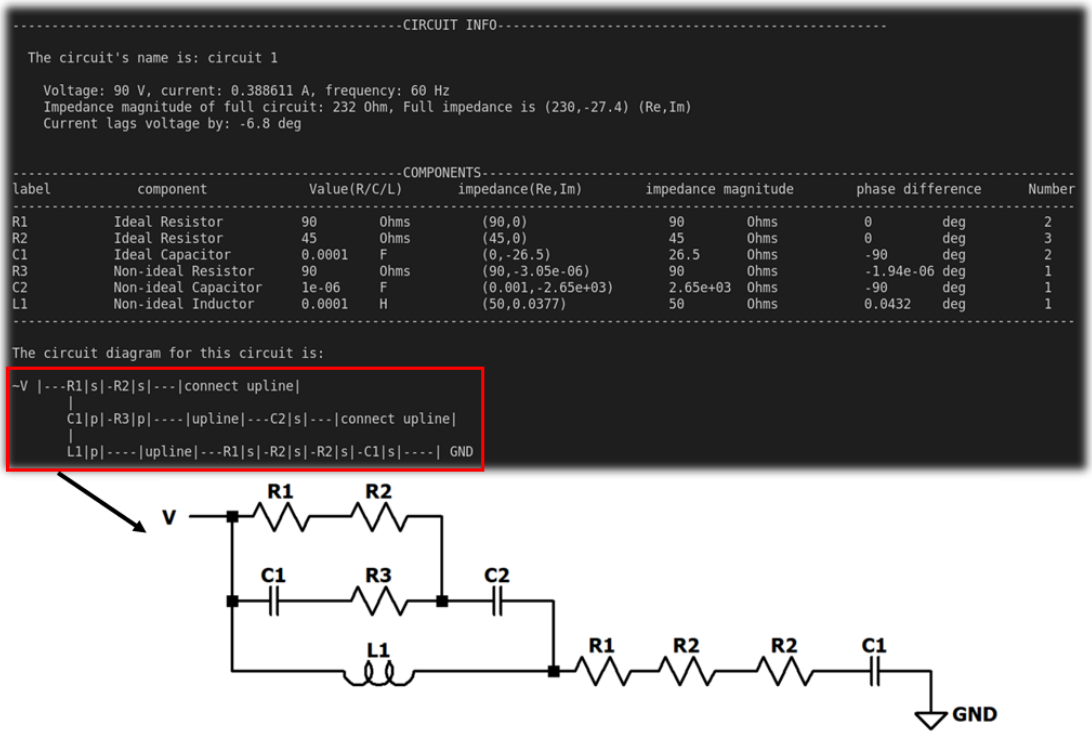

Figure 3.2: Example of a more complex run to illustrate the **upline**, **connect upline** system. A real circuit diagram for the circuit in the terminal output is shown below.

The component table contains all the parameters for each component as well as the number of times each component appears in the circuit. The function that counts the circuit components is a template function called *repeated_name_counter()* and is present inside the circuit class; this allows it to be later used with nested circuits. This was done to prevent the terminal from overcrowding when the circuit contains many repeated components. The

convention used for the phase difference between the voltage and the current is: a positive value means voltage leads current and a negative value means that current leads voltage.

3.2 Test run with nested circuits

To create a nested circuit the user must select option six and enter a voltage, a frequency and a name. Then, the user can go back to the menu and select an option to add circuits from the circuit library to the nested circuit. However, the user can only add circuits that have the same frequency and voltage. The reason for this is that the impedance of the nested circuit is calculated based on the impedance of each circuit added. Therefore, in order to simplify calculations and keep the model simple, the circuits added must have the same parameters so that the inductance of the nested circuit is correctly computed. If the user tries to add circuits that have a different frequency or voltage, then the program will show a warning message to the user indicating the circuits were not added to the nested circuit for this reason. Otherwise, the circuits will be added as the user instructed.

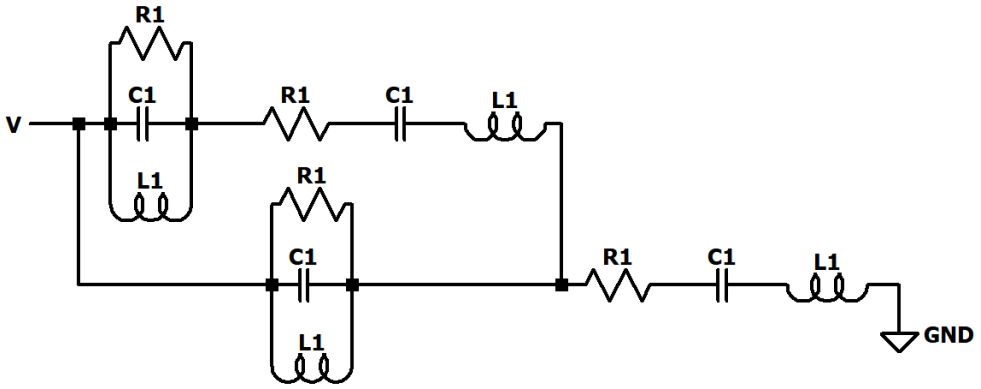

Figure 3.3: Example nested circuit diagram for the example run shown in Fig. 3.4

This type of nesting allows a user to create very large circuit geometries by breaking down individual areas of a circuit into smaller parts. The user can build different circuits of arbitrary length in the circuit library and add them to a larger nested circuit to make much larger circuits. This useful feature could hypothetically allow a user to create more complex non-ideal component models as circuits and build new circuits with these through nesting. Since the user is allowed to name the circuits they create, they could name their non-ideal components R, C or L for example. This would make the presentation clear in the nested circuit diagram.

To show the nesting feature implemented, the circuit in Fig. 3.3 will be simulated. The example run with this nested circuit is shown in Fig. 3.4. The program prints out the information with all the parameters about the nested circuit such as the circuit diagram and a table with the circuits that make it up. Then, it prints the detailed information of the circuits that make it up below. The *repeated_name_counter()* function is now used to, only print out information about each circuit once and also count the number of each circuit in the nested circuit for the table; this avoids terminal overcrowding. A similar template function is also present in the libraries class to make sure the user does not name a circuit with a name already present in the respective circuit library. This ensures that the user does not create circuits with the same name which could make nested circuits confusing.

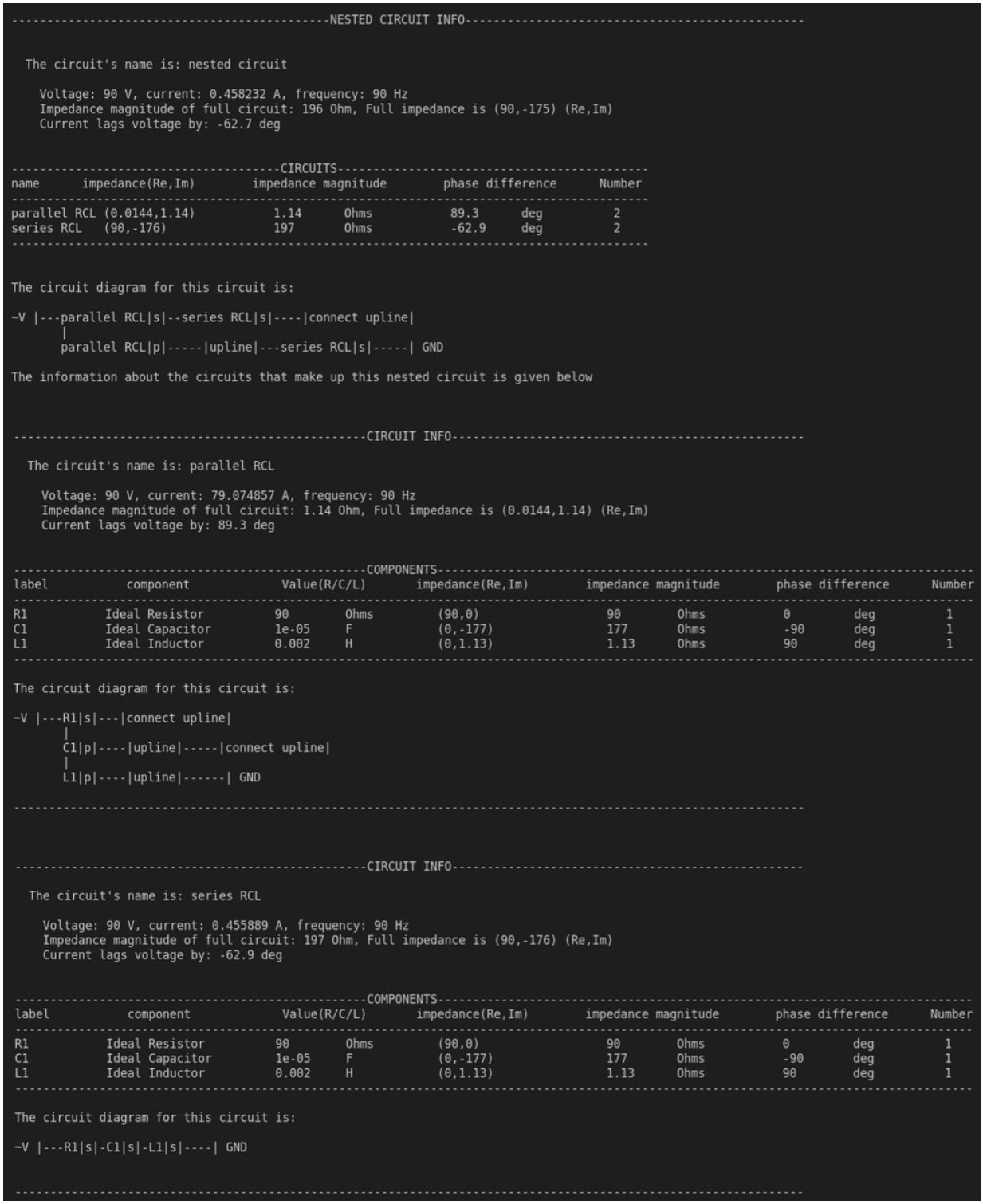

Figure 3.4: Terminal output of example run for nested circuit diagram shown in Fig. 3.3. At the top is the nested circuit diagram and a table summarising the information about the circuits that make it up. On the bottom is the detailed information about each circuit.

3.3 Input validation

An array of input validation measures were implemented so only a few are covered here. The user is only allowed to input values for the components and circuits that are within the bounds shown in Table. 2.1. When dealing with libraries checks are used to make sure the user enters a number that is within the library otherwise, an error message is displayed and the user is asked to enter the numbers again. This prevents crashing from memory leaks due to segmentation errors. Some examples of input validation measures are shown in Figs. 3.5, 3.6 and 3.7.

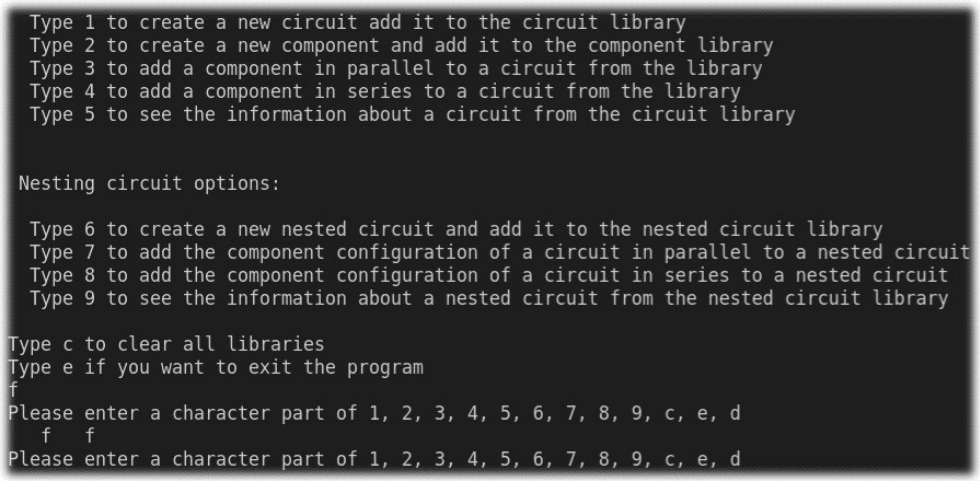

Figure 3.5: User input validation example for different options in the menu.

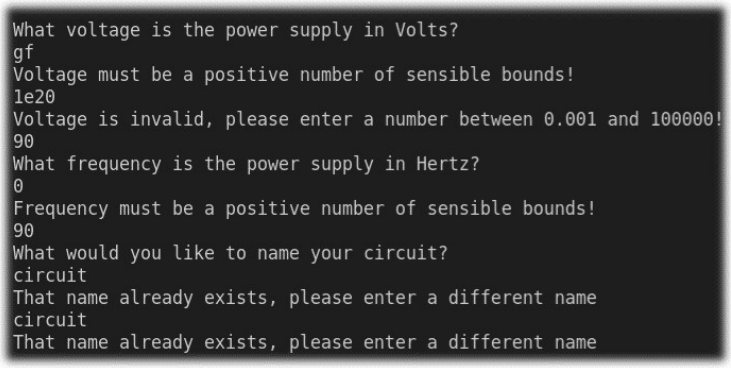

Figure 3.6: User input validation example for checking sensible bounds and non-numeric inputs. Also shown is an attempt to name a circuit with an already existing name in the library.

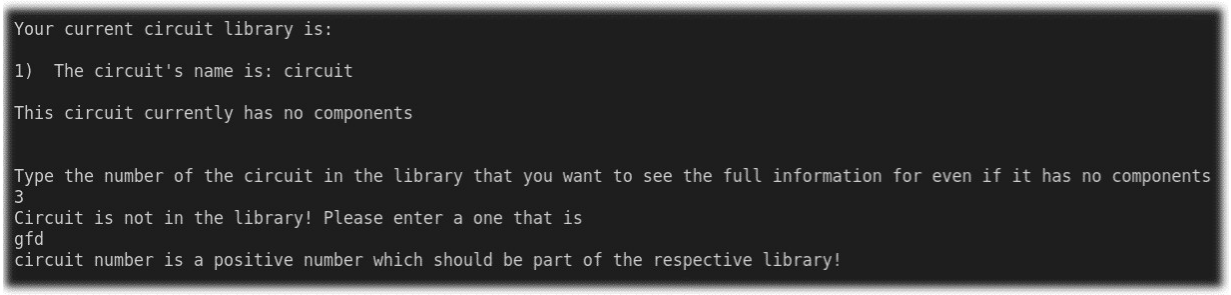

Figure 3.7: User input validation example to check for library number input.

3.4 Advanced C++ features

The component library and the storage of components in the circuit class are based on polymorphism. A component abstract base class is used to interface between the different components. Smart pointers of type *std::shared_ptr* are used to point to different class objects in the different libraries allowing these to be passed to functions. The use of these pointers allows automatic garbage collection as they contain a reference counter and get automatically deleted when their reference count reaches 0.

Inheritance was used with the nested circuit class which inherits the circuit class allowing it to have access to methods from the circuit class, such as *print_circuit_diagram()*, as well as some specific to itself. In addition to this, template functions are used in the circuit and library classes where appropriate to reduce repeated code. A notable example is the *print_library()* function which is used to print all three libraries which contain vectors of *std::shared_ptr* which point to a different type of object depending on the library. Additionally, namespaces are also used to prevent name clashing. Two namespaces exist; *COMPONENTS* and *CIRCUITS*.

Furthermore, exception handling was used as a precaution when allocating new memory to catch any memory leaks through *bad_alloc*. Finally, other features such as iterators, switch statements and ternary expressions and were also used.

4 Final discussion and conclusions

The program that was developed can successfully simulate AC circuits with an arbitrary number of resistors, capacitors or inductors in series and parallel. The program used only the standard C++ library and the advanced C++ features required for the assignment. The program can correctly compute the phase difference between the voltage and current, the current and the impedance of a circuit.

Many improvements could be made to the program. The most relevant improvement would be to develop a more user-friendly graphical user interface with better graphics. This would however require external libraries. Furthermore, more electronic component models such as operational amplifiers could be added by creating more component classes. This would however require significant changes to the code. Finally, a higher-order nesting could be implemented where nested circuits can also be nested with themselves in a recursive fashion.

Word count (Overleaf): 2468 words without references, captions and the appendix.

Bibliography

[1] P. Horowitz. *The art of electronics*. Cambridge University Press, New York, Third edition, 2015.

[2] A. P. Robert. *Analog circuits*. World Class Designs. Newnes/Elsevier, Burlington, MA, 2007.

[3] M. McWhorter et al. *EE 344 High Frequency Laboratory*. Stanford Univ, 1995.

A Appendix: Demo circuit

To aid with testing the program, a demo circuit option was added to the menu. This demo circuit option can only be run once or after all the libraries are cleared. When created, this circuit gets automatically added to the circuit library and its components to the component library so that the user can use them to build other circuits. All the library numbers and component count variables are also updated.

To create the demo circuit, the user must select option d from the menu. Then to view this circuit, the user must go back to the menu and select option 5 to view the circuit library, once in the circuit library the user must then select the respective demo circuit number to see its details. The terminal output for the circuit information looks like what is shown in Fig. A.1

```
--------------------------------------------------CIRCUIT INFO--------------------------------------------------

  The circuit's name is: demo

    Voltage: 90 V, current: 0.483184 A, frequency: 60 Hz
    Impedance magnitude of full circuit: 186 Ohm, Full impedance is (60,-176) (Re,Im)
    Current lags voltage by: -71.2 deg


--------------------------------------------------COMPONENTS--------------------------------------------------
label           component             Value(R/C/L)       impedance(Re,Im)        impedance magnitude        phase difference      Number
--------------------------------------------------------------------------------------------------------------------------------------
R1           Ideal Resistor          90        Ohms          (90,0)                  90        Ohms          0         deg          3
C1           Ideal Capacitor         1e-05     F             (0,-265)                265       Ohms          -90       deg          3
L1           Ideal Inductor          0.002     H             (0,0.754)               0.754     Ohms          90        deg          3
R2           Ideal Resistor          90        Ohms          (90,0)                  90        Ohms          0         deg          3
C2           Ideal Capacitor         1e-05     F             (0,-265)                265       Ohms          -90       deg          3
L2           Ideal Inductor          0.002     H             (0,0.754)               0.754     Ohms          90        deg          3
--------------------------------------------------------------------------------------------------------------------------------------

The circuit diagram for this circuit is:

~V |---R1|s|-C1|s|-L1|s|-R2|s|-C2|s|-L2|s|---|connect upline|
      |
      R1|p|-C1|p|-L1|p|-R2|p|-C2|p|-L2|p|----|upline|-----|connect upline|
      |
      R1|p|-C1|p|-L1|p|-R2|p|-C2|p|-L2|p|----|upline|------| GND
```

Figure A.1: circuit info of example demo circuit