A.C. Circuits

Erik Germanovic
10496729
Department of Physics and Astronomy
The University of Manchester
May 17, 2022

Abstract

A program to build A.C. circuits and analyse their properties, created using object-oriented programming features of C++ language. The program allows to create an arbitrary number of circuit components and connect them in series or parallel. Moreover, nested circuit creation is available. Implementation of the program utilises advanced features such as polymorphism as well as inheritance. The program was found to be successful in creating complex A.C. circuits and correctly calculating their impedances. Discussion of further program development is provided.

1. Introduction

Alternating current circuits are powered by an alternating source which periodically changes current or voltage about a mean value. The main difference from direct current circuits is the availability of current to pass in both directions. This allows the current flow through capacitors as well as induces electromotive force in inductors [1]. Thus, the complexity of the circuits is enhanced by the availability of different types of components.

Resistors, inductors and capacitors introduce an opposing effect to the alternating current in the circuit. The combined effect of such opposition is called impedance. Impedance extends the concept of resistance to A.C. circuits by including both magnitude and phase components, thus it is represented as a complex number. The magnitude of impedance represents the ratio of the voltage difference amplitude to the current amplitude, whereas the argument indicates the phase difference between the mentioned circuit properties. This is known as Ohm's law and is given by:

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

where V and I are periodic voltage and current respectively and Z is the impedance. The impedances of ideal resistor, ideal capacitor and ideal inductor are represented as the following [1]:

$$Z_R = R \tag{2}$$

$$Z_C = -\frac{i}{\omega C} \tag{3}$$

$$Z_L = i\omega L, \tag{4}$$

where R, C and L indicate resistance, capacitance and inductance respectively and ω is the angular frequency of the circuit. Conveniently, calculation of impedance for components connected in series or parallel follow the same rules as the calculation of combined resistance.

$$Z_S = Z_1 + Z_2 \tag{5}$$

$$\frac{1}{Z_P} = \frac{1}{Z_1} + \frac{1}{Z_2}, \tag{6}$$

where subscript S represents series connection and P is parallel connection.

However, the impedances of different type of components change completely when the assumption of an ideal component is dropped. The changes are caused by parasitic effects which originate from wiring of components as well as different potentials at the ends of components [2]. With parasitic effects, components are modelled as circuits consisting of one resistor, capacitor and inductor. The models of different non-ideal components are shown below [2]:

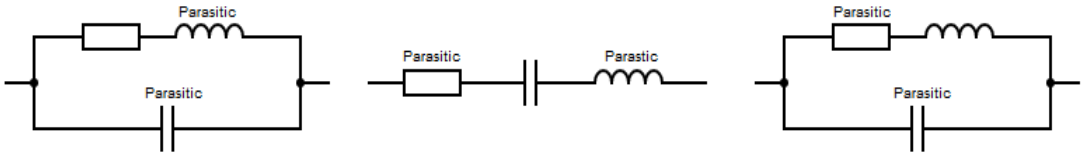

Figure 1. Circuit representations from left to right of real resistor, real capacitor and real inductor.

Using the rules of impedance addition, total impedance of a non-ideal component is expressed as:

$$Z_R = \frac{R+i\omega L_p}{1-\omega^2 L_p C_p + iR\omega C_p} \tag{7}$$

$$Z_C = R_p + i\left(\omega L_p - \frac{1}{\omega C}\right) \tag{8}$$

$$Z_L = \frac{R_p+i\omega L}{1-\omega^2 C_p L + i\omega R_p C_p}, \tag{9}$$

where subscript p indicates parasitic component properties. As can be seen from formulas (7), (8) and (9) the parasitic effects are negligible at low frequency, thus a high frequency is required to test whether the program behaves correctly with non-ideal components.

This program was designed to include both ideal and non-ideal components, allowing the user to specify each of their properties. Once the circuit is built, the program outputs overall information about the circuit, as well as information about individual components.

2. Code design and implementation

2.1 Classes

The program follows a clear class hierarchy which starts with an abstract component base class, used as an interface for all types of components. The base class has frequency and impedance as the only member data. Impedance is stored as a complex number object by utilising a pre-made complex number class. Also, the component class has 6 member functions and a virtual destructor, to ensure that the objects are deleted in a correct order. 4 member functions, returning impedance, impedance magnitude, impedance argument (phase difference) and frequency have definitions in the class, since the implementation is the same for every component. The other 2 member functions are pure virtual functions to set frequency and impedance of a component, as well as print information about it. Three classes: resistor, capacitor and inductor are derived from the abstract base class. Each of them publicly inherits data of the component class and has additional protected member data of resistance, capacitance or inductance based on the component type. Classes for non-ideal resistor, capacitor and inductor are derived from their ideal representation classes. The inheritance is public once again, however each class has additional private member data for parasitic properties. Every class in the program utilises ‘const’ feature on functions or function arguments where appropriate to guarantee that the object is not modified incorrectly. The overall class structure allows to utilise polymorphism throughout the use of the program, hence every created circuit component is stored as an abstract component class pointer.

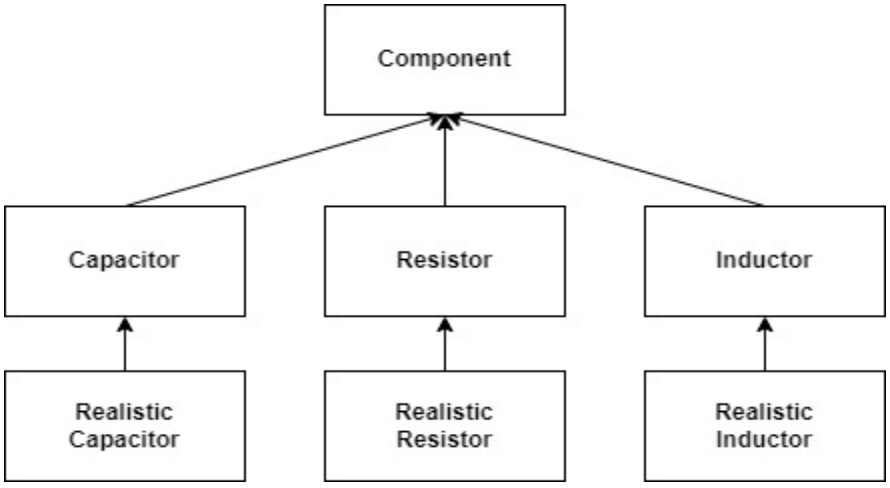

Figure 2. Class hierarchy of circuit components used in the program. The arrows and their directionality represent inheritance.

The program also has a circuit class used to connect components together. Private member data of the class consists of frequency, impedance, vector of integers to specify the current position in the circuit and a vector of pairs which store pointers to the circuit’s components and their positions within the circuit. The use of vectors was motivated by the desire to create arbitrary length circuits. Thus, vector push back feature was found to be most successful in dealing with such a task. Implementation of pair class template allowed to store components and their positions in a circuit as a single unit, which was

beneficial, since most of the tasks involved the use of both properties at the same time.

```
std::vector<std::pair<std::shared_ptr<component>, std::vector<int>>> circuit_components;
```

Figure 3. Utilisation of vector and pair features to store components and their positions within the circuit.

The class has a number of member functions which allow the user to move and create connections within a circuit. For example, stepping into or out of a parallel connection branch and switching branches functions modify the current position in the circuit. These features were introduced, since they allow to create circuits which not only consist of series and parallel connections of components, but also involve complex nesting of both. Moreover, the class provides options to connect both single components and previously created circuits. Other set of member functions were introduced to calculate the number of series connections or parallel connection branches within certain locations of the circuit to ensure that current position movement inside the circuit is correct. The remaining set of member functions were used to calculate the total impedance of the circuit and print its information.

2.2 Design

The component positions within the circuit are encoded using a vector of integers. The vector consists of an odd number of integers specifying series connection positions and parallel connection branch numbers. Both parallel branches and series connections are counted from zero. The information can be interpreted as follows: first number of the vector represents series connection in the main branch. If a parallel connection is present, the second number indicates parallel connection branch. The number after that specifies series connection within the branch. If another parallel connection within a parallel connection branch is present, the rules continue in the same order. Thus, the position vector size is always an odd number. For example, a component position 0 0 0 indicates that it is part of the first full component in series within the main branch, located in first parallel connection branch of the full component and corresponding to the first component in that parallel branch. This allows a simple and effective method to categorise any connection within the circuit. However, the program outputs component connections starting from 1, to make it easier to visualise the circuit.

Since component impedance is dependent on A.C. circuit frequency, components are firstly initialised without frequency and impedance information. Only after the building of the circuit is finished each component gets assigned a circuit frequency which is then used to calculate the component's impedance. To calculate the total impedance of the circuit recursive definitions are utilised. Circuit components are firstly grouped into full elements within the main branch based on their series connection number. The program then iterates through the grouped elements, identifying whether they consist of multiple branches. If multiple branches are present, the program creates separate circuits out of branches and calls impedance functions on each of them. Effectively, a circuit is broken down into sub-circuits whose impedances can be calculated by simple addition of component impedances.

```
for (int index_1 = 0; index_1 < number_of_series_connections; index_1++) // going over full circuit elements in main branch
{
    if (grouped_components[index_1].size() == 1) { // full circuit element consists of one component case
        impedance = impedance + grouped_components[index_1][0].first->get_impedance();
    } else { // full circuit element consists of more than one component case
        number_of_branches = calculate_branches(grouped_components[index_1]);
        std::vector<complex> reciprocal_branch_impedances(number_of_branches);
        complex total_reciprocal_impedance;
        for (int index_2 = 0; index_2 < number_of_branches; index_2++) // stepping into a branch of full circuit element
        {
            reciprocal_branch_impedances[index_2] = complex(1, 0) / branch_circuit(index_2, grouped_components[index_1]).total_impedance();
            total_reciprocal_impedance = total_reciprocal_impedance + reciprocal_branch_impedances[index_2]; // reciprocal of total impedance of parallel connection
        }
        impedance = impedance + complex(1, 0) / total_reciprocal_impedance;
    }
}
```

Figure 4. Code which demonstrates the recursive calculation of circuit impedance.

Additionally, a static variable is introduced to count the current depth of recursion, since the initial call of the function is intended to reset the impedance of the circuit. Overall, this method allows the calculation of total impedance of circuits with any kind of structure.

```
static int function_call_count{0}; // required to see how many times the function has been called (recursion)
if (function_call_count == 0) {
    impedance = complex{ 0,0 }; // resetting impedance before calculating
}
```

Figure 5. Usage of static variable to count the depth of recursion within the program.

Multiple circuits can have ownership of the same components if a circuit is being created from existing circuits. Thus, shared pointers are used throughout the whole program to store circuit components. Even though component frequency does not match the correct circuit frequency when an existing circuit is connected, the frequency is always reassigned at the end of the built circuit. This overcomes any potential problems due to shared ownership. Also, the use of shared pointers removes the necessity of user defined copy constructor, copy assignment and manual garbage collection.

2.3 User interface

The main program allows the user to test the class implementation through multiple user inputs. Moreover, it effectively constrains the user to call the program functions in correct order, to achieve the desired results. The user interface consists of a menu which allows to select available actions of the circuit through user input. An example of the menu is shown below:

```
List of possible actions:
-----------------------------------------------------------------------------------------

A. Step into a parallel circuit branch (required for nested circuits)
B. Add a component in series
C. Add arbitrary number of components in parallel

-----------------------------------------------------------------------------------------
```

Figure 6. Usage of static variable to count the depth of recursion within the program.

Figure 6 shows that some actions only modify the position within the circuit, whereas others connect circuit components. Based on the current state of the program and the built circuit, main program only allows to perform certain actions from a pre-defined action list. To ensure this, separate action lists and user interface functions were defined for nested and non-nested circuits. In addition to this, a vector of integers was created to specify the nesting status of the circuit. Recursion was used to switch between the mentioned regimes. A vector of circuits was introduced to store the finished circuits and use them for

future circuit building. There was no permanent component container within the main program, since they were directly inputted into circuits. This was utilised through component creator function which returned shared base class pointers to a dynamically allocated component object.

2.4 Validation

To ensure that the user inputs are sensible, exception handling was introduced. This involved different type of validation functions which check whether the user input is of correct type and within appropriate numerical bounds. The numerical bounds for different type of component properties were selected based on literature values. If user input is invalid, the program throws an exception and allows to repeat the last entry. The error is specified by outputting text in red colour.

Property of component	Minimum value	Maximum value
Frequency	1 Hz	30 GHz
Resistance	10 mΩ	100 GΩ
Capacitance	1 fF	10 kF
Inductance	1 fH	1 kH
Parasitic resistance	0 Hz	2 Hz
Parasitic capacitance	0 F	1 pF
Parasitic inductance	0 H	1 nH

Table 1. Numerical bounds of different properties of components, specified by user input.

2.5 Program structure

To improve the readability of the code, the program is split into two header files and three source code files. The header file named ‘complex_class.h’ includes all declarations of complex number class and the standard library headers used to perform actions with complex numbers. The other header file ‘circuit_and_component_classes.h’ has all the declarations of component and circuit classes. It also includes every necessary standard library header to be able to build circuits. The source code files ‘complex_class.cpp’ and ‘circuit_and_component_classes.cpp’ provide the implementation of class member functions of complex and circuit-based classes respectively. The last source code file ‘analogue_circuits.cpp’ contains all the necessary functions for user interface and input validation.

3. Results

To conclude whether the program works correctly, input validation needed to be tested thoroughly. Numerical, integer and non-numerical inputs were all tested by different invalid entries. These invalid inputs consisted of empty, non-numerical and out of numerical bounds entries. The results of input validation are shown in Figure 7 below:

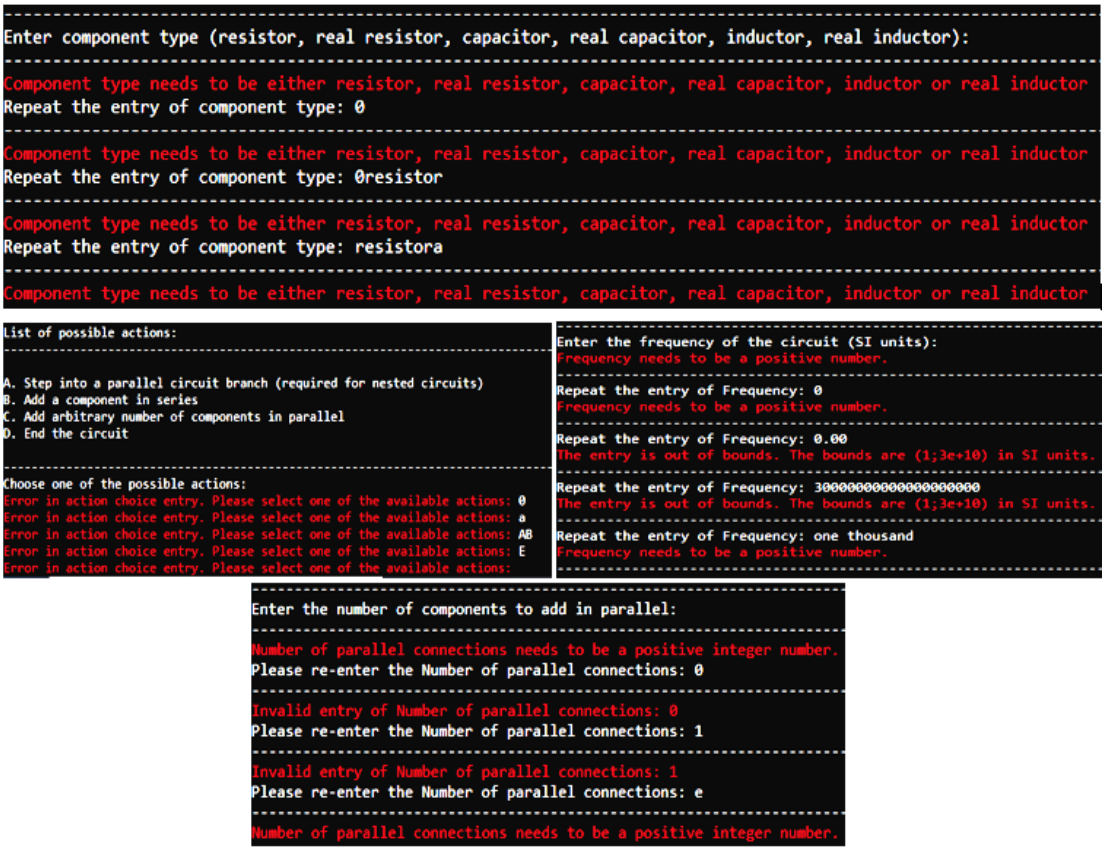

Figure 7. Different cases of user input validation with corresponding error outputs.

As the Figure 7 indicates, input validation works correctly in all cases. This allows to safely test whether the program correctly calculates total impedance of A.C. circuits. This part of testing started with building a simple circuit, consisting of 50 Ohm resistor, 50 μF capacitor and 1 nH inductor connected in series. The expected impedance magnitude and phase shift were 3.18×10^4 Ω and -1.5692 rad respectively. The output of the program is shown in Figure 8.

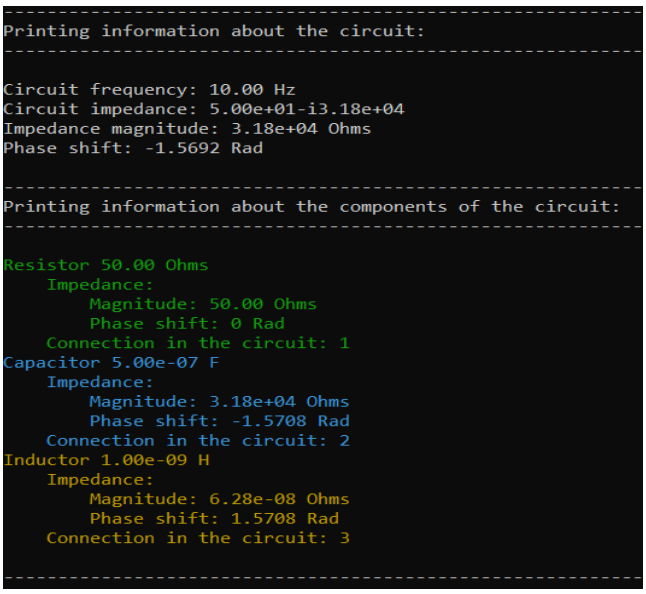

Figure 8. Program output of a circuit, consisting of a resistor, capacitor and inductor connected in series.

Figure 8 shows that the program correctly calculates the total impedance of the circuit and provides an accurate list of circuit components and their properties. However, to properly test the performance of the program, a complicated nested circuit was built involving the use of both ideal and non-ideal components. In total 8 components were used to build the circuit, involving combinations of parallel and series connections. The expected impedance magnitude and phase shift were 21 Ω and 0.0016 Rad respectively. The circuit diagram and results of the program are indicated in Figure 9 and Figure 10.

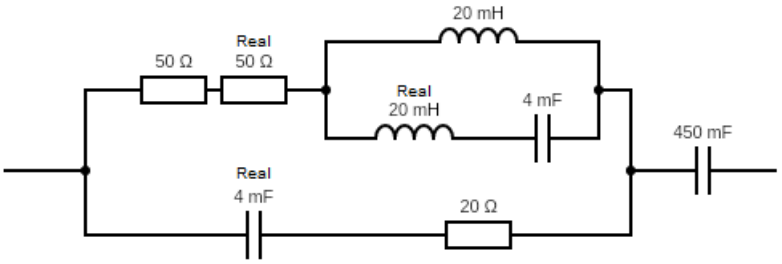

Figure 9. Diagram of a circuit which was used to test the program in extreme cases.

```
----------------------------------------------------------------------------------------
Printing information about the circuit:
----------------------------------------------------------------------------------------

Circuit frequency: 1.00e+07 Hz
Circuit impedance: 2.10e+01+i3.26e-02
Impedance magnitude: 21.00 Ohms
Phase shift: 0.0016 Rad

----------------------------------------------------------------------------------------
Printing information about the components of the circuit:
----------------------------------------------------------------------------------------

Resistor 50.00 Ohms
    Impedance:
        Magnitude: 50.00 Ohms
        Phase shift: 0 Rad
    Connection in the circuit: 1 1 1
Resistor 50.00 Ohms, parasitic inductance 4.70e-10 H, parasitic capacitance 6.50e-12 F
    Impedance:
        Magnitude: 49.99 Ohms
        Phase shift: -0.0198 Rad
    Connection in the circuit: 1 1 2
Inductor 0.02 H
    Impedance:
        Magnitude: 1.45e+06 Ohms
        Phase shift: 1.5708 Rad
    Connection in the circuit: 1 1 3 1 1
Inductor 0.02 H, parasitic resistance 1.80 Ohms, parasitic capacitance 3.20e-13 F
    Impedance:
        Magnitude: 5.15e+04 Ohms
        Phase shift: -1.5708 Rad
    Connection in the circuit: 1 1 3 2 1
Capacitor 4.00e-03 F
    Impedance:
        Magnitude: 3.98e-06 Ohms
        Phase shift: -1.5708 Rad
    Connection in the circuit: 1 1 3 2 2
Capacitor 4.00e-03 F, parasitic resistance 1.00 Ohms, parasitic inductance 6.50e-10 H
    Impedance:
        Magnitude: 1.00 Ohms
        Phase shift: 0.0408 Rad
    Connection in the circuit: 1 2 1
Resistor 20.00 Ohms
    Impedance:
        Magnitude: 20.00 Ohms
        Phase shift: 0 Rad
    Connection in the circuit: 1 2 2
Cpacitor 0.45 F
    Impedance:
        Magnitude: 3.54e-08 Ohms
        Phase shift: -1.5708 Rad
    Connection in the circuit: 2

----------------------------------------------------------------------------------------
```

Figure 10. Program output of a circuit, presented in Figure 9.

The output of the program agrees with the expected values. Further testing was performed by connecting existing circuits in series and parallel which provided correct results as well. Thus, it can be concluded that all program features behave correctly.

4. Discussion and conclusion

The testing of the program approved that the main objectives of the project were accomplished. The program outputs correct total impedances of created circuits, as well as properties of circuits' components. Moreover, the program extended the minimum project definition provided by the lecturer by introducing non-ideal components to A.C. circuits, recursive definitions and ability to create nested circuits. The use of smart pointers and standard template library advanced features allowed efficient storage of circuit components. Features of object-oriented programming, such as polymorphism and inheritance were utilised correctly and efficiently throughout the code. After circuit building is finished, the program deletes the created objects and does not show any signs of memory leakage.

The program could be extended by introducing visual representation of the circuits. This would help the user to check whether the desired circuit has been built. Furthermore, other types of components, such as transistors and diodes could be added to the program. However, this would require another method of component position representation. Perhaps the most useful extension would be to introduce voltage and circuit calculations of individual circuit components. This would provide complete information about circuits. In terms of efficiency, the code is lengthy, however most functions were introduced to allow the user to create nested circuits.

References

[1] Pease R, editor. Analog circuits. Newnes; 2008 Jul 2.

[2] Williams J, editor. The art and science of analog circuit design. Elsevier; 1998 Aug 24.

Word count without references and figure captions: 2381