A.C. Circuits

Sam Brind
Student ID: 8926382
School of Physics and Astronomy
The University of Manchester

May 16, 2016

Abstract

A program to simulate AC circuits was created using the C++ language and standard library. The program was designed to connect an arbitrary amount of user created components in series and/or parallel and then calculate the total circuit impedance. The program was thoroughly tested and found to be functional but further improvements are discussed.

1. Introduction

1.1. Theory

Alternating current (AC) is a current that periodically alternates the direction of charge flow. Most power supplies, including the United Kingdom mains supply, utilise AC currents and it is also useful for applications such as audio processing. Using an AC supply for a circuit allows interesting components such as capacitors and inductors to be connected in the circuit as well as standard resistor components.

An essential quantity for describing an AC circuit is the circuit impedance. Impedance is an extension of resistance to AC circuits using complex numbers. This can be seen by the relation between voltage and current [1],

$$V = IZ$$

where V and I are periodic voltage and current respectively and Z is the impedance. From direct comparison to Ohm's law it can be seen impedance plays a similar role to resistance but impedance can be a complex number. Due to this similarity, impedance follows the same rules as resistance for addition of components in series and in parallel. [1]

The impedance for an ideal resistor is given by,

$$Z_R = R \tag{1}$$

where R is the resistance of the resistor. The impedance of an ideal capacitor and ideal inductor are given by,

$$Z_C = \frac{-i}{\omega C} \quad \text{and} \quad Z_I = i\omega L \tag{2}$$

respectively, where C is the capacitance of the capacitor, L is the inductance of the inductor and ω is the angular frequency of the current flowing through the components. [1]

In reality, these components will contain parasitic effects that become relevant at high frequencies. These non-ideal components can be represented as a circuit containing a resistor, capacitor and inductor where two of these components are parasitic effects. An example of a parasitic effect is the inductance of a resistor's wires. Figure 1 shows the three circuit diagrams for these non-ideal components. [2]

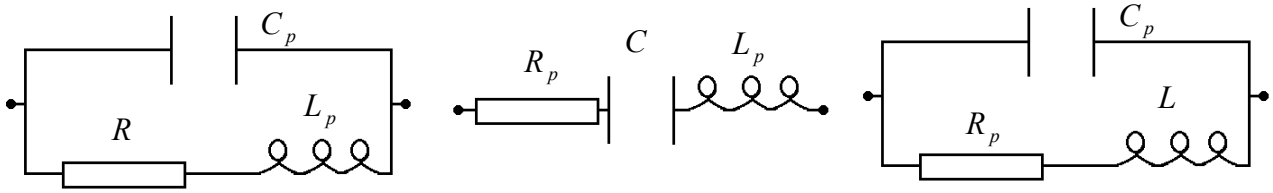

Figure 1. The circuit diagrams for non-ideal components. From left to right: non-ideal resistor, capacitor and inductor. The symbols have their usual meanings but the subscript p indicates a parasitic effect.

From these diagrams, the impedances of non-ideal components can be derived using Eq. 1, Eq. 2 and the rules for series and parallel impedance addition. This gives,

$$Z_R = \frac{R + i\omega L_p}{(1 - \omega^2 C_p L_p) + i\omega R C_p} \quad , \quad Z_C = R_p + i\left(\omega L_p - \frac{1}{\omega C}\right)$$
$$\text{and} \quad Z_L = \frac{R_p + i\omega L}{(1 - \omega^2 C_p L) + i\omega R_p C_p} \tag{3}$$

as the impedances of a non-ideal resistor, capacitor and inductor respectively. [2]

1.2. Project definition

The purpose of this project was to produce a C++ program to model AC circuits. Specifically, the ability for the user to create an arbitrary number of ideal or non-ideal components: resistors, capacitors or inductors. Allow them to then connect these components in series and/or in parallel to construct a circuit of arbitrary length. Output from the program would then be a list of the component impedances, the total circuit impedance and a circuit diagram.

2. Code design and implementation

2.1. File structure

The project is split into three files for better readability: two source code files and a header file. The header file (Classes.h) contains all class declarations as well as all declaring the standard library headers used within the whole project. One source file (Classes.cpp) contains the implementation of class member functions and the other (Project.cpp) contains the main program code and functions to take user commands and call the appropriate class functions, effectively the code for a user interface.

2.2. Classes

The basis of the program is a simple class structure. An abstract base class was constructed for components, with the component impedance being the only member data. The standard complex number library functionality is used to store impedance as a complex object. The class also contains virtual member functions to print component details, set or get component frequency, get impedance and get the impedance magnitude or phase. Resistor, capacitor and inductor classes are then derived classes that all publicly inherit from the base component class. Apart from overriding the virtual functions, the only change from the base class is additional member data for resistance, capacitance, inductance and frequency where appropriate. Real (non-ideal) components then publicly inherit from these classes (i.e. real_resistor inherits from resistor). Real components contain additional data members for parasitic quantities and they also override the set frequency and print functions. This class structure allows polymorphism to be used throughout the whole program and simplify the code. Hence, all pointers in the program will refer to the component base class irrespective of whether the pointed object is a resistor, capacitor, inductor or real version of these components.

There is also a class for circuits, containing member data of voltage, frequency, impedance, a text description string and a vector storing pointers to the circuit's components. The two main member functions are to add components in series or parallel and this is done by passing a vector containing component pointers as a parameter. There is also a function to print the circuit details.

Most class functions are self explanatory, especially where standard maths functions have been used to return impedance phase and magnitude. Resistor impedance is set within the constructor whereas capacitor and inductor impedances are set from within the set frequency function because they are frequency dependent. There are two print functions for the components. One outputs a one line summary of the component and the other outputs the summary and the component's impedance magnitude and phase. The component adding functions iterate over the component pointers passed to the function. For addition in series, each component has its frequency set to the circuit frequency, the pointer is pushed back onto the circuit's component vector, impedance is added to the circuit impedance and likewise the component character representation is appended to the circuit text description. For addition in parallel, a similar process is applied but the impedance is summed for all the components in series, representing the total impedance of the line of components, and this is then added in parallel to the circuit impedance. A new line character is also added, prior to component characters, onto the circuit text description. This is simply to separate lines for use when printing the circuit diagram from the text description.

The circuit print function first prints the circuit properties: frequency, voltage, current and impedance. It then iterates over the vector of components and calls the base class print function on each component. The circuit diagram is then printed from the circuit text description. Firstly, the longest line is determined. This maximum length is then used to determine a scale for each line and the component characters are printed with a scaled amount of wires.

2.3. Main program

While the classes contain all the necessary underlying code for the program, an extra set of code is needed to build a user interface that allows the user to access this functionality. This UI is based around a simple menu showing the available commands, as shown in Figure 2.

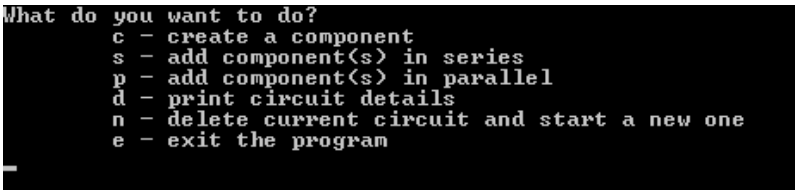

Figure 2. The main menu for the program, showing all the features available to the user.

Create component takes valid user input properties and calls the appropriate class constructor. The pointer to this component is added to a static vector called the component library. This library is used to store user created components for the lifetime of the program. There is also a static pointer to store the users current circuit. Smart pointers are used to simplify the code by removing manual garbage collection. Component library consists of shared pointers because they need to be copied when being added to a circuit and consequently to the circuits component vector. However, the current circuit pointer is a unique pointer due to the program only allowing one circuit at a time so the pointer has unique possession of a single object.

The component adding functions print the component library for the user to see, take a valid list of user input pointers to the component library and then run the add series or add parallel functions on the current circuit. Print circuit details simply calls the current circuit's print function. Exit program will close the program. Creating a new circuit will call the circuit function. This function is called initially to start the program, takes user values for voltage and frequency, creates a circuit object, assigns the current circuit pointer to the new object and then calls the menu function. This repetitive loop that contains the whole program is summarised in Figure 3.

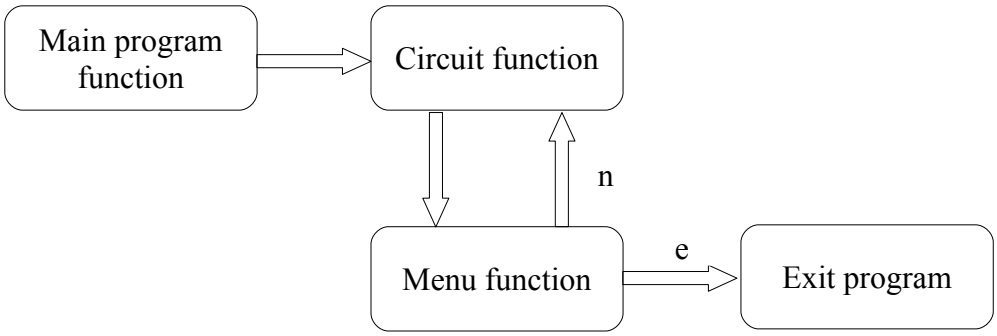

Figure 3. A flow diagram, summarising the structure of the main program.

2.4. Validation

User input numerical quantities are subject to two types of validation, essential and range checking. Essential validation makes sure that the user enters the correct data type and that numerical values are positive, these are the minimum requirements for the program to run without error. Range checking makes sure the value of the number is within a range of reasonable values. Table 1 shows the set ranges for this program.

Electrical property	Minimum allowed value	Maximum allowed value
Frequency	0.01Hz	1GHz
Voltage	1μV	0.1GV
Resistance	1nΩ	100GΩ

Capacitance	1fF	10kF
Inductor	100fH	10kH
Parasitic resistance	0Ω	1Ω
Parasitic capacitance	0F	10pF
Parasitic inductance	0H	10nH

Table 1. The range of allowed user inputs for the electrical properties

3. Results

3.1. Standard run through of code

The first step to testing the code is trying it out with standard values. This was done by constructing an ideal RLC circuit (which consists of simply a resistor, capacitor and resistor in series). Ideal components were created successfully and are shown in Figure 4, when the component library is printed. All three components were then added in series and the circuit details printed. Using Eq. 1 and Eq. 2 the expected impedance is 10.048295Ω at a phase angle of -0.098083 radians. It can be seen from Figure 5 that the program outputs the correct impedance, all circuit components and the circuit diagram.

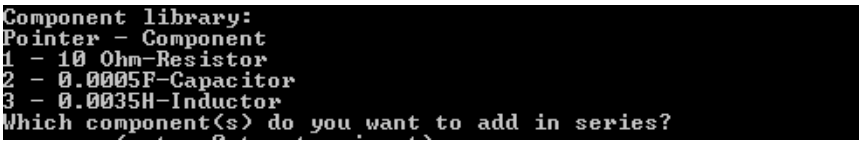

Figure 4. The printed component library showing a standard resistor, capacitor and inductor.

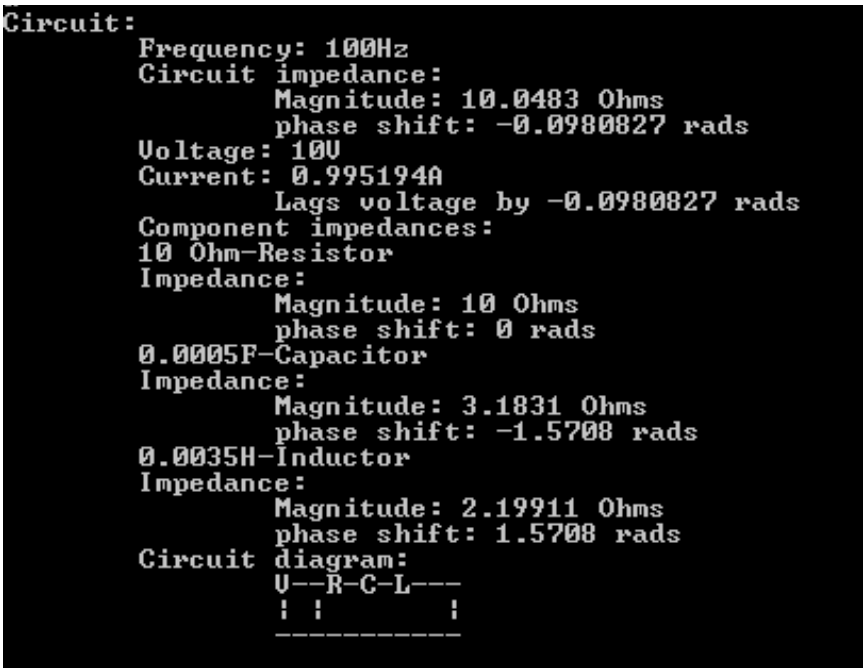

Figure 5. Printed circuit details from a standard RLC test using the components shown in Figure 4.

3.2. Extreme run through of code

The code was then tested by trying out more unusual configurations such as using non-ideal components and adding a large amount of components in parallel. To check the non-ideal components, a high frequency is needed to notice the parasitic effects. Figure 6 shows an example circuit where ten ideal and non-ideal components have been added in series and parallel. The calculation of circuit impedance using Eq. 1, Eq. 2 and Eq. 3 gives an expected impedance of 0.117306Ω at a phase of 0.556739 radians. Figure 6 shows that the program calculates the value very well, considering there are rounding values in the above calculation.

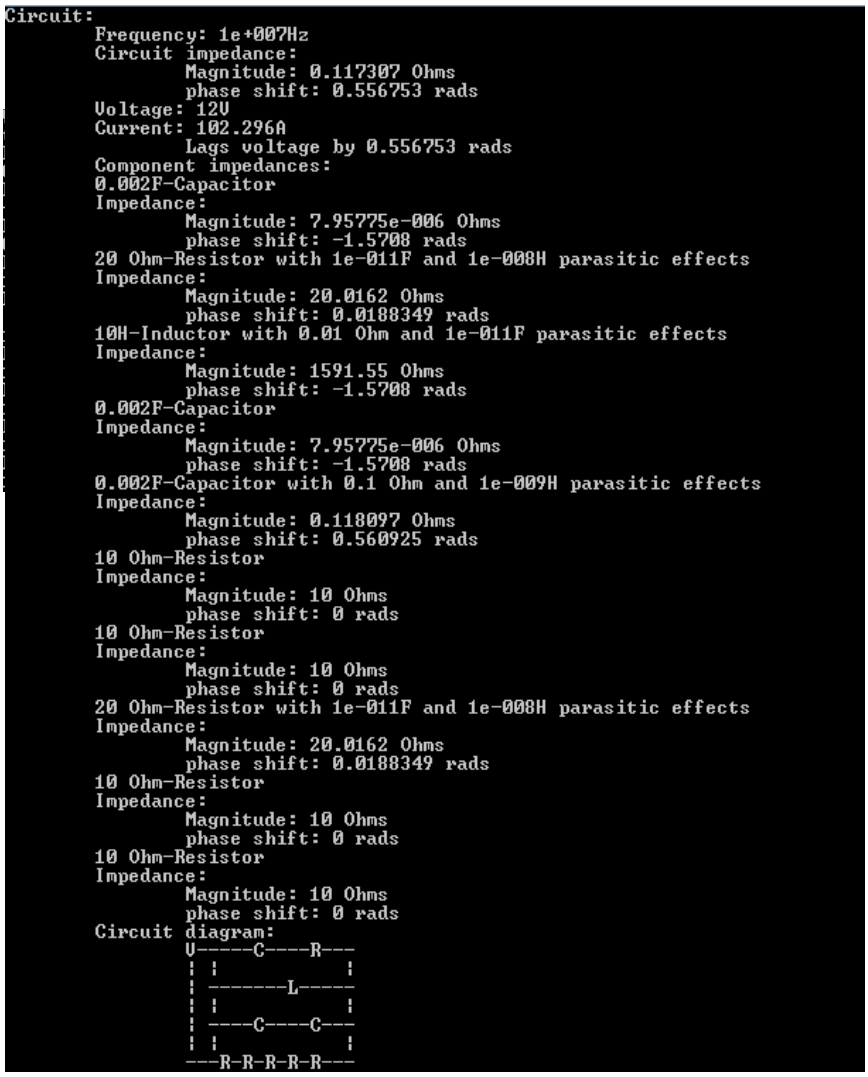

Figure 6. Printed circuit details for a large circuit involving ten components, four of which are non-ideal components.

3.3. Invalid user inputs

All of the system validation was checked to make sure it was correctly stopping all invalid user inputs. This meant that invalid data types, blank lines, negative numbers and values outside the ranges shown in Table 1 were entered into every user input. A typical example of this is shown in Figure 7. The system was found to catch all invalid data but allowed good values.

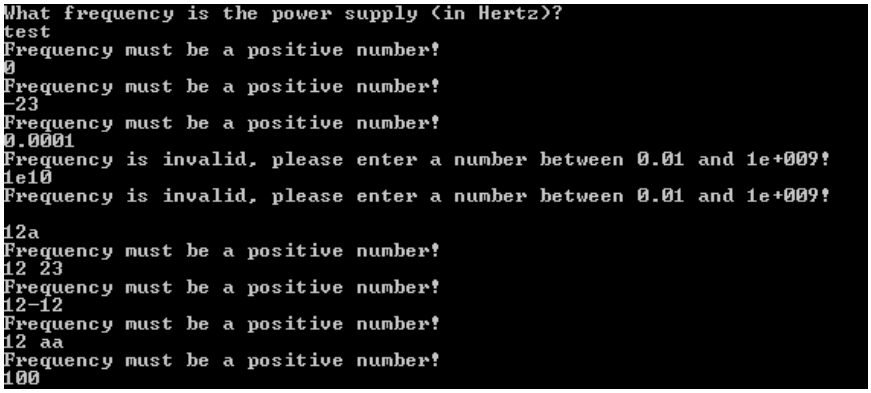

Figure 7. Validation testing of frequency input at the start of creating a new circuit.

4. Discussion and conclusion

From the results shown above and further testing it seems that the program fulfils the objectives outlined in the project definition. The system creates components and allows them to be connected through a circuit object. The details of this circuit can then be printed out and the impedance values are calculated correctly. The system also calls the correct amount of destructors on exiting the program and seems to have no memory leak problems. This shows the system is feasible and could be used in its current state for AC circuit calculations.

However, if there was further time for this project there are multiple ways the code could be extended. There is the general thought that perhaps the code could be made more efficient. Performance is currently good so this is not currently a huge concern but could be important if a large amount of new functionality is added to the program or a specialist user needed to add a very large numbers of components together. A simple thing that may improve user experience is to convert small and large number into their prefixed units when outputting values, for example 1nF instead of 1e-09F. More complex components could also be added to the program such as diodes and transistors. These are components that alter current flow, which would require the current classes to be severely modified to take this into account as no concept of current is currently programmed into the classes. A bigger improvement to the code would be an extension of the code to allow multiple circuits. There could then be a circuit library as well as a component library. The program could then become very general and allow addition of components to circuits and vice versa and connecting circuits to each other. Although, this would need to adjust the circuit to ensure there is only one power supply otherwise the behaviour would become very messy due to the interference from having two different frequencies within the same circuit.

References

[1] U. Krey and A. Owen, Basic Theoretical Physics, Springer, 2007

[2] R. Pease, Analog Circuits, Elsevier science, 2008

Number of words (excluding references): 2249