Simulations of A.C. Circuits with C++

Cassian Lewis
ID: 10304679
School of Physics and Astronomy
University of Manchester

Abstract

An object oriented programme to simulate A.C. circuits with resistors, inductors and capacitors is created in C++. The user is able to create and modify components before connecting an arbitrary number of them in series and/or parallel. The programme can calculate the total impedance for individual components and the whole circuit, and is also able to print a circuit diagram. It has been rigorously error tested to ensure functionality with erroneous inputs.

Introduction

Alternating current circuits (AC) are those in which the direction of current flow oscillates with time – as opposed to direct current (DC) circuits where this direction is constant. Much of the electricity we use on a day to day basis (from the mains) is AC, since with the use of transformers, transporting this electricity reduces power loss across long distances. Most of our household appliances also run on alternating current.

AC voltage in most circuits can be modelled as a sinusoidal wave with a maximum amplitude and angular frequency [1] ;

$$V(t) = V_{max}\sin(\omega t + \phi)$$

where all the symbols have their usual meanings. The voltage is related to the current and impedance by the equation [2] ;

$$V = IZ$$

which is essentially an extension of Ohm's law with resistance replaced by impedance. This quantity is represented by a complex number containing information about both its magnitude and its phase. Representation in this form allows for relatively easy combination of impedances, which follows the same series and parallel addition rules as resistance. The impedances for ideal resistors, inductors and capacitors are given by [3]; [i]

$$Z_R = R, \quad Z_I = i\omega L, \quad Z_C = \frac{-i}{\omega C}$$

where R, L and C are the resistance, inductance and capacitance respectfully, and ω the angular frequency of the current. It should be noted that these are ideal components, and that in the real world they are prone to parasitic effects (for example the resistance of the circuits wires). These can be modelled as mini-circuits of ideal resistors, inductors and capacitors; for example a real resistor can be considered as a resistor and parasitic inductor connected in parallel with a parasitic capacitor. The total impedance changes accordingly [4]. [ii]

$$Z_R = \frac{R + i\omega L_p}{\left(1 - \omega^2 C_p L_p\right) + i\omega R C_p}$$

Equation 4 reduces to that of an ideal resistor in the case of zero parasitic inductance and capacitance. Similar expressions can be calculated for real inductors and capacitors (although in this case all the components are connected in series).

This programme allows the creation of any number of ideal or real world components, with values of resistance, inductance and capacitance entered at the users discretion (within values specified in the programme). These can then be connected to form a basic series/parallel circuit of arbitrary length, whose information and diagram can be printed.

Code

Structure

The code is split into three files. The first, a header file ('classes.h'), declares the component class and its virtual functions as well as all the other derived classes (including the circuit class). It also declares a template and a function which are used in multiple member functions. The second file ('classes.cpp') includes the class constructors and member function implementations, as well as the body of the aforementioned template and function. The third, the main file, contains all other functions used in the programme and the code for the user interface.

Classes and functions

The core of the programme is an abstract base class for components. This contains (among other things) member data for impedance (a complex number created with a separate class) and angular frequency (a double), plus a number of pure virtual functions including those to print component information and set angular frequency. The classes for resistors, capacitors and inductors are derived from the component base class and thus inherit its member data and functions. These virtual functions are overridden to suit the data of the particular component, and this polymorphism allows for general implementation of these functions later in the code.

These component classes contain non constant member data (since the user may want to modify values) for the resistance, inductance and capacitance, and are thus considered as real components. Three more ideal component classes are derived from these real component classes; in their constructors, the parasitic effects of the real components are set to zero, and thus the equations for their impedance reduce to those found in [3]. Having the ideal components as derived classes of the real ones reduces the code length and is a natural extension of the inheritance of member data which underpins this programme.

Once components have been created by the user, they are stored in a vector of base class (smart) pointers. Combined with the polymorphic member functions this allows, for example, to print a list of component information using an iterator calling the same function, but acting on different component classes. Passing components as pointers also ensures that, if the user wants to modify a component in the library, the corresponding component in the circuit will be modified too.

The circuit class is derived from the base class, and contains extra member data such as voltage and a vector of smart pointers to components. It is constructed with three parameters; voltage, frequency and base component. The class can be considered as a line of components connected in series. It contains member functions to add components to the vector and calculate the impedance of the line of components (which simply adds the individual impedances of all the components in the vector). The components' angular frequency setting function is called (passing the circuits angular frequency as a parameter) whenever they are added to the circuit, so that their individual impedances, which are frequency dependent, can then be calculated.

A vector of smart pointers to these circuits are used to store the individual lines of the main circuit, where each element represents a new line. Adding a component in series will append a stored circuit line; it will create a new one if added in parallel. An arbitrary number of components can be added to the main circuit from the component library in such a way; although clearly this method of circuit construction will not allow for more complex, nested circuits.
The function to print the circuit diagram first calculates the maximum number of components on any one line of the main circuit, and uses this value as a maximum width for the diagram. The diagram is then printed line by line using the circuit 'print line' member function which takes this maximum value as a parameter and prints a scaled diagram of the line of components. This function is able to accurately print any circuit which the user is able to construct with the programme.

The total circuit impedance is calculated by adding the individual line impedances in parallel. The magnitude and phase of the circuit impedance, as well as for each component, is printed at the users request. A number of other non-member functions are also present in the code, however most of these are self-explanatory.

Interface

A comprehensive user interface is coded in order to implement the functions. The user is asked to input an integer between 1-10 to select an option, all of which are shown in figure 1.

```
Menu
----
What would you like to do?
     1. View programme information
     2. Create components
     3. Modify components
     4. View component information
     5. Create circuit
     6. Add component in series
     7. Add component in parallel
     8. View circuit information
     9. Delete all data
     10. Terminate the programme
```

Figure 1, the main menu of the programme, displaying the different options available to the user.

The code is nested inside a while loop, which will continue until the user enters the number 10, whereupon the code clears up the data and exits. The other options are accessed via a switch statement which evaluates the users entry. Some of these options will lead to a sub menu; an example of this is shown in figure 2. Once the function has finished running, the code breaks out of the switch statement and the user is returned to the main menu.

```
There are 6 components to choose from:
1. Resistor
2. Ideal resistor
3. Inductor
4. Ideal inductor
5. Capacitor
6. Ideal Capacitor
Please enter a corresponding number from 1-6 to select a component:
```

Figure 2, a sub-menu displayed after selecting the option to create a component (option 2) from the main menu.

Errors and invalid inputs

Most of the invalid inputs are handled with a template, shown in figure 3, that takes two values (low and high) and checks that the input is a valid double between (and including) them. This allows for various ranges of variables to be easily set.

```
template<class type> type valueCheck(type low, type high)
{
    type x;
    std::cin>>x;
    while (x<low || x>high || !std::cin.good()){
        std::cout<<"Invalid entry. Please enter again"<<std::endl;
        std::cin.clear();
        std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
        std::cin>>x;
    }
    return x;
}
```

Figure 3, a template to check the validity of the users input.

Checking the user input for the menu is slightly more complex, and two functions are required. The first, a Boolean, checks whether the entered string contains any characters which are not numbers. If this determines the string to be an integer, then the second checks if it falls within a range of allowed values. These functions ensure that the programme runs smoothly, even if incorrect inputs are entered.

The ranges of the different variables are outlined in table 1.

Parameter	Minimum	Maximum
Supplied voltage	1μV	1GV
Frequency	1mHz	1GHz
Resistance	1μΩ	1GΩ
Inductance	1μH	10kH
Capacitance	1pF	10kF
Parasitic resistance	0Ω	10Ω
Parasitic inductance	0H	10H
Parasitic capacitance	0F	10F

Table 1, a graph of the allowed values for the electrical parameters. Some of these values may lie outside standard ranges; this is to ensure the user has a wider scope when creating components.

Results

The code successfully compiles at run time without error (clearly this requirement is a bare minimum). Testing the programme with a simple ideal RLC circuit yields results consistent with those calculated using [3], as shown in Figure 4. Modification of components changes the circuit accordingly (this is only possible due to the use of pointers).

The information and diagram for a more complex circuit is shown in figure 5.

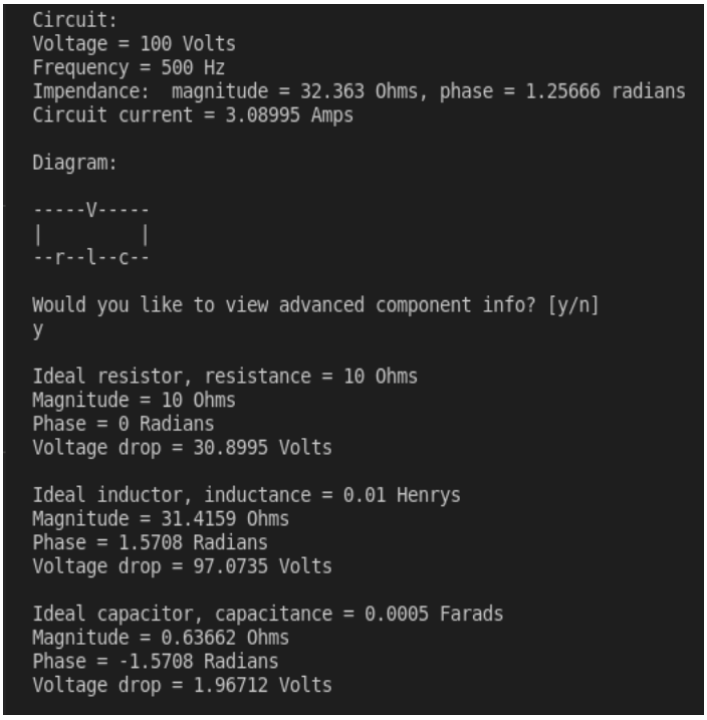

Figure 4, the output of the print function for a series RLC circuit with a 10 Ohm resistor, 10mH inductor and 0.05mF capacitor. Note that the supplied voltage does not equal the sum of the voltage drops across individual components; this is due to the differences in phase and the fact that voltages are added in quadrature in RLC circuits.

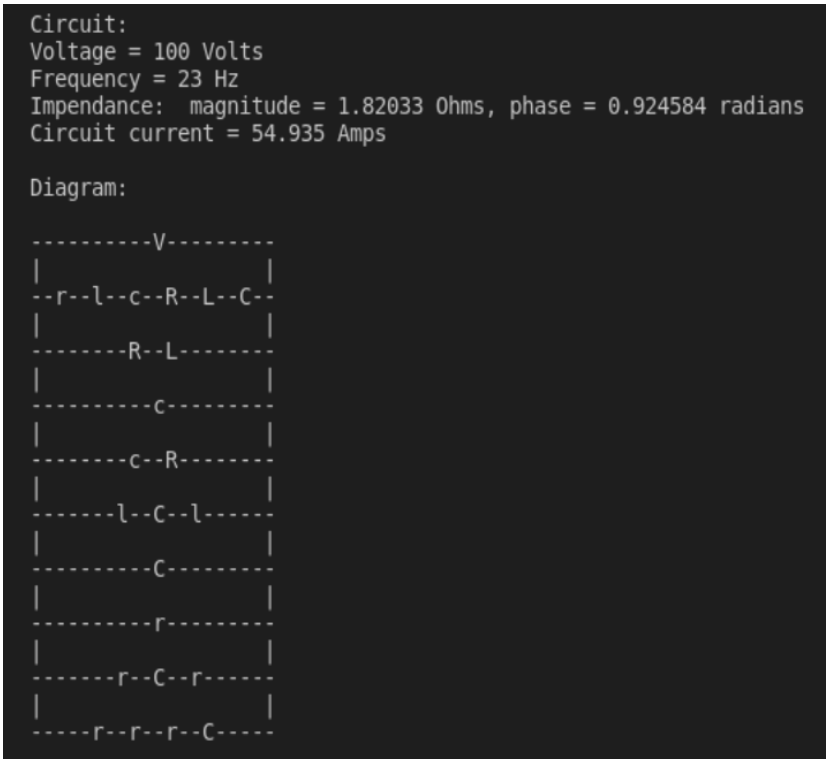

Figure 5, the basic information and diagram of a more complex circuit with a number of ideal and real world components connected in series and parallel.

Discussion

This programme fulfils all the basic requirements set out in the project specification. Additional functionality includes the ability to create non-ideal components (of which ideal component classes are derived from), modify components and print an accurate diagram of any stored circuit, as well as calculate the current and voltage drop through each component. It also includes a robust user interface and complete input validation functions.

With more time this programme could be improved in a number of ways. The design of the circuit class does not allow for more complex, nested circuits. It could be expanded to include member data such as a vector of pointers to subcircuits, and member functions to connect these in series or parallel, as well as a recursive function to calculate the impedance. This class design could raise problems in printing the circuit diagram, which would perhaps have to be displayed in a simpler form (such as a tree diagram of individual subcircuits). Nevertheless this would allow for virtually any circuit to be created.

Added functionality could also include; increasing the number of components to include transistors and diodes, using graphics packages to display the circuit using the traditional symbols for components, printing current-time graphs and including options for damping the circuit.

References

[i] Agarwal, Anant; Lang, Jeffrey H. *Foundations of Analog and Digital Electronic Circuits*, Morgan Kaufmann (2005) p.713

[ii] Eric, Bogatin. *Signal and Power Intensity*, Pearson (2018), p.105