AC Circuits

Frederik Brooke Barnes

Department of Physics and Astronomy

The University of Manchester

Object Oriented Programming in C++

May 2021

Contents

1 Abstract

`AC-Circuits` is a C++ program written to simulate circuits driven by alternating currents (AC) as part of the course Object Oriented Programming in C++. The code allows users to create a library of components, build these components into a circuit, and calculate the electrical characteristics of the circuit. Additionally, users may add these custom circuits to the component library, allowing the nesting of 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.

2 Introduction

AC circuit analysis is a significant and essential and aspect of modern electronics and electrical engineering. The most striking example is the electrical grid — used all over the world to distribute electrical power to billions of people. As well as power generation and distribution, alternating currents are used in a large number of the technologies that make the modern world including audio amplification, wireless communications, and electric motors, to name a few.

Therefore, being able to model the behaviour of an AC circuit is an important capability. In some cases this can be done by hand or heuristically. However, as the complexity of a circuit increases it is often desirable to calculate the properties of an AC Circuit using software. Many software packages exist to perform AC circuit simulation. An early example is SPICE (Simulation Program with Integrated Circuit Emphasis), first released in 1973 [1]. SPICE was originally written in FORTRAN, an early compiled procedural language (although later versions of Fortran included features to support object-orientated programming [2]). Modern versions of SPICE, written in C, are still popular today [3]. Features are extensive and component model libraries exist for a large number of commercially available components as well as the ability to define custom components [4].

The software in this project is limited to impedance calculations of combinations of resistors, capacitors, and inductors. The theoretical details will be discussed in the Section 3. The code, written in C++ using the object-orientated paradigm [5], will be discussed in Sections 4 and 5.

3 Background theory

In the scope of this project, an AC circuit will be defined as an electrical circuit driven by voltage or current source with a sinusoidal signal of single frequency. The source will have two terminals to which any number of components may be attached. Each component will have two connections which must be connected to either the terminal and/or other components.

For direct current (DC), Ohm's law describes the relationship between current, I, passing through an Ohmic conductor and the voltage, V, between its terminals:

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

where R is defined as the resistance of the conductor. An example of an ohmic conductor is the metal film resistor, commonly used in electronics. For AC, resistors will also obey Ohm's law. For a capacitor, the relationship between voltage and current is given by

$$I = C\frac{dV}{dt} \tag{2}$$

where C is the capacitance of the capacitor. From this, one can see that if the voltage is constant, as for DC, then no current will flow through the capacitor and it can be thought of as an open switch. However, if the voltage is changing then the current will be non-zero. Therefore, for an AC voltage of angular frequency ω, $V(t) = V_0 \sin\omega t$, the current through the capacitor will be $I(t) = C\omega_0 V_0 \cos\omega t$. This means the capacitor introduces a $+90°$ phase shift between voltage and current, and $X_C \equiv \frac{|V|}{|C|} = \frac{1}{\omega C}$ where we have defined the reactance, X. For an inductor, the relationship between voltage and current is given by

$$V = L\frac{dI}{dt}, \tag{3}$$

where L is the inductance of the inductor. From this, one can show that the inductor introduces a $-90°$ phase shift between voltage and current, and $X_L = \omega L$ [6].

An AC signal can be represented as a phasor, using complex numbers. For instance, the voltage $V(t) = V_0 \cos(\omega t + \phi_V)$ can be written as $V(t) = \Re(\tilde{V}(t))$, where the phasor is defined as $\tilde{V}(t) \equiv V_0 e^{i+\phi_V}$. By using this representation, Ohm's law becomes

$$\tilde{V} = \tilde{I}Z, \tag{4}$$

where Z is a complex number known as the impedance and is can be written as $Z = R + iX$. Hence, for resistors, Z is real and simply equal to the resistance, $Z_R = R$; for capacitors, $Z_C = -i/\omega C$; and for inductors, $Z_L = i/\omega L$. The magnitude is given by $|Z| = \sqrt{R^2 + X^2}$. As Z is complex, it also contains information about the phase introduced by a component. The phase of the impedance is simply its argument in the complex plane, given by

$$\phi_Z = \operatorname{atan2}(X, R) \equiv \begin{cases} 2\arctan\left(\frac{X}{\sqrt{R^2+X^2}+R}\right) & \text{if } R > 0 \text{ or } X \neq 0, \\ \pi & \text{if } R < 0 \text{ and } X = 0, \\ \text{undefined} & \text{if } R = 0 \text{ and } X = 0. \end{cases} \tag{5}$$

Using this, the $+90°$ and $-90°$ phase relationship between voltage and current for capacitors and resistors is recovered [7].

When components are placed in series, the total impedance is found by finding the sum of each component's impedance,

$$Z_{\text{total}} = \sum_i^N Z_i, \tag{6}$$

where the sum is over all N components. When placed in parallel, the total impedance is calculated by

$$Z_{\text{total}} = \left(\sum_i^N \frac{1}{Z_i}\right)^{-1}. \tag{7}$$

Therefore, if one has a circuit of many different resistors, capacitors, and inductors in series and parallel, it may be decomposed into smaller sub-circuits which can be treated as components with a general form for Z. Conversely, if a circuit is constructed one component at a time by sequentially adding components in series or parallel to the whole circuit, the impedance of the new circuit can be found simply by taking the old impedance and adding it to the new component using Equation 6 or 7, as appropriate. A limitation of constructing circuits in this way is that not all possible configurations of components are allowed. For instance, the arrangement of components shown in Figure 1 resistors shown below would not be possible to construct but if one could store

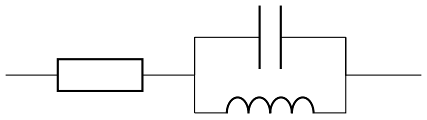

Figure 1: Arrangement of resistor, capacitor, and inductor not possible by sequential addition of components in parallel and series without nesting.

the parallel combination of inductor and capacitor as a new component, then this new component could be added to the resistor in series. Thus, by nesting sub-circuits, this sequential method can be used to create more general circuits and to calculate the circuit's impedance easily.

This method of circuit construction and impedance calculation is used in the program discussed in the following sections.

4 Functionality

The user interface of the program is based on a text-based menu in the terminal. Each menu offers a range of options for the user which may be input by text. Once a valid option is given, the corresponding action is taken in by the program or the next menu is shown. The possible routes a user may take is shown in a flow diagram in Figure 2.

The intended method of using the program would consist of a user adding several resistors, capacitors, and inductors to the component library. Following this, circuits can be created using components in the library or with newly made components. These are added sequentially to the circuit in series of parallel, as described in Section 3. During this process, the user may choose to 'Show info' about the circuit, such as its components and its impedance. Additionally, the user may choose to 'Draw', which will create a text-based image of the circuit wiring in the terminal. At any stage, the user may set the frequency of the circuit and the program will ensure that when the user asks to 'Show info' then the correct impedance will be displayed. Following the creation of one or more circuits, these will be added to the library. These circuits will then be available for use in new circuits made when components are added from the library.

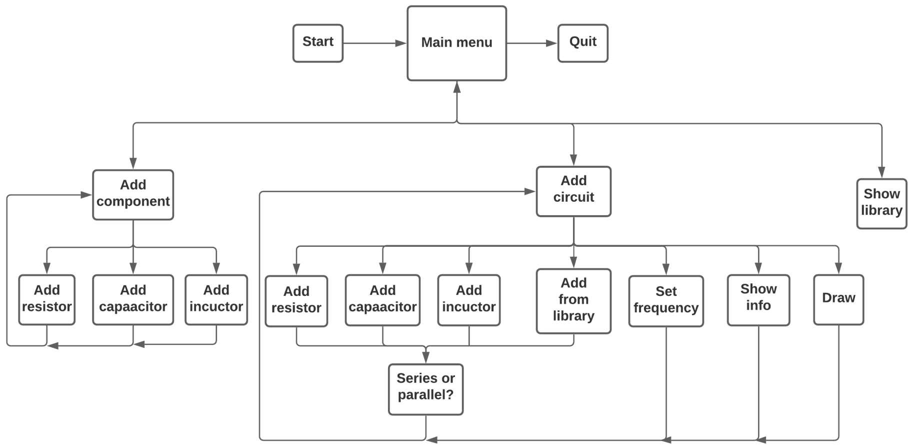

Figure 2: Flow diagram illustrating how a user may navigate the program. Note that some menus allow the user to cancel, which is shown by a backward arrow.

Below is an example of the output when after the user has added a number of custom components to the component library, added them to a circuit, and asked the program to 'show info' about the circuit:

```
Circuit Impedance: 0.00206733+i0.615051 Ohm.
        Frequency: 1 Hz
        Magnitude: 0.615055 Ohm
        Phase: 1.56744 rad
              Circuit components:
              R: 1 Ohm resistor
              C: 12 Farad capactior
              L2: 0.1 Henry inductor
              R3: 200 Ohm resistor
```

Once this circuit was created, the user then asked for it to be drawn. The result is output as shown below:

```
Circuit diagram:
-----R---L2----
  |    |    |
  --C---    |
  |         |
  ----R3-----
```

5 Code features

Component library

The component library is implemented as a custom type `component_library` defined using `typedef`. This type is a `std::vector` (from the C++ Standard Template Library) which contains shared pointers to objects of the `component` class. The `component` class is a base class which with children for resistors, capacitors, and inductors. Thus, this code demonstrates the resource allocation is initialisation (RAII) idiom as well as polymorphism.

Input verification

When the user inputs information, exception handling is used in order to verify the input. An example of this is the `get_resistor()` function in the `menu` class. As well as catching any unforseen problems using `catch(...){}`, the code also identifies if the input is not physically realistic (e.g. negative) and will throw the input to a catch which will inform the user the input is not positive and print the value of their input.

Input handling

Menus are implemented as functions of the `menu` class. In a menu there are a fixed number of options a user may choose by inputting a string. The options available to the user are defined using `enum`. For example,

```
enum class menu::get_is_series_option {series=1, parallel=2};
```

sets the options available when the user is choosing whether to add a component in series or parallel as integers. This allows strings that the user may enter to correspond to particular menu items. In this example,

```
void menu::register_get_is_series_options()
{
  get_is_series_options["s"] = get_is_series_option::series;
  get_is_series_options["p"] = get_is_series_option::parallel;
}
```

is a function used to convert allowed input strings ”s” and ”p” to particular options. A `switch()` statement based on the user’s input is then used to control the flow of the program:

```
switch (get_is_series_options[user_input])
{
    case get_is_series_option::series:
        return true;
        break;
    case get_is_series_option::parallel:
        return false;
        break;
    //If input string invalid, remind user
    default:
        std::cout << "Please enter s or p"<<std::endl;
        break;
}
```

when extended to many menu options, this makes input handling more readable and all of the input variables can be edited in the `enum class`.

Templates, algorithms, and Lambda functions

In the course of adding a new component, the code will update the string used to represent the component in a circuit diagram. It does so by appending a number to it according to how many of that component already exists. For instance, resistors have the symbols ”R1”,”R2”,”R3”, etc. To this end, a `template` function was created to count how many elements have the same type as a derived class in a vector of shared base class pointers. This function is shown below:

```
//Counts number of elements in a vector
//which match the type of the input item
template <class T, class T2>
int count_type (T item, std::vector<std::shared_ptr<T2>> vector)
{
  //get type_index of type of interest
  std::type_index const item_type = typeid(item);
  //count elements where typeid matches
  int item_count = std::count_if(begin(vector), end(vector), [&](auto element)
  {
    return item_type == typeid(*element);
  });
  std::cout<<"count:"<<item_count;
  return item_count;
}
```

This function makes use of the standard C++ library header `<typeinfo>` to get the `std::type_index` of interest. Then, `std::count_if` from the standard C++ library header `<algorithm>` is used. The criteria for counting is set by a Lambda function where a comparison is made between the relevant type indices.

6 Conclusion

This report has given an overview of the theory behind AC circuits and impedance, how this was applied in the `AC-Circuits` program, and the code features of employed during the software’s development. The program itself is a useful tool for sketching simple circuits and for calculating impedance. Furthermore, the program is a demonstration of the object-oriented programming paradigm and makes use of several advanced C++ features.

References

[1] L. W. Nagel and D. Pederson, "Spice (simulation program with integrated circuit emphasis)," EECS Department, University of California, Berkeley, Tech. Rep. UCB/ERL M382, Apr 1973. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/1973/22871.html

[2] J. C. Adams, Ed., *The Fortran 2003 handbook: the complete syntax, features and procedures.* Dordrecht, Netherlands ; London: Springer, 2009, oCLC: ocn195730642.

[3] R. Pratap, V. Agarwal, and R. K. Singh, "Review of various available spice simulators," in *2014 International Conference on Power, Control and Embedded Systems (ICPCES)*, 2014, pp. 1–6.

[4] S. Luan, G. Selli, J. Fan, M. Lai, J. Knighten, N. Smith, R. Alexander, G. Antonini, A. Ciccomancini, A. Orlandi, and J. Drewniak, "Spice model libraries for via transitions," in *2003 IEEE Symposium on Electromagnetic Compatibility. Symposium Record (Cat. No.03CH37446)*, vol. 2, 2003, pp. 859–864 vol.2.

[5] J. L. Stanley B. Lippman, *C++ primer*, 3rd ed. Addison-Wesley, 1998. [Online]. Available: http://gen.lib.rus.ec/book/index.php?md5=23d37c3b9786d21985b6061d1625d279

[6] R. Powell, *Introduction to electric circuits*, ser. Essential electronics series. Arnold, 1995. [Online]. Available: http://gen.lib.rus.ec/book/index.php?md5=0d929fa35326c4b993ebf9e70d699098

[7] P. Horowitz and W. Hill, *The art of electronics*, 2nd ed. Cambridge University Press, 1989. [Online]. Available: http://gen.lib.rus.ec/book/index.php?md5=af96a44498a24e1dfc6e010bd5bfb637