Alternating-Current Circuits: An Object-Oriented Implementation in C++

Ricardo Wölker*

Object-Oriented Programming in C++ – *Final Project Report*
Student ID: 8939451
(Dated: May 15, 2017)

A C++-based class hierarchy for the simulation of alternating-current circuits is developed. An abstract base class for generic components is used as an interface, from which the physical circuit components (resistors, capacitors and inductors) are derived. A circuit class computes and stores various properties of the whole circuit. Individual-component and whole-circuit phase changes are output, and the user is able to create their own circuit, with an arbitrary number of components, connected in series or parallel. A simple console output for an ASCII-based plotting method for phasor diagrams is developed. The addition of lossy components is introduced. Recursive definitions to account for nested circuits are implemented. The successful operation of the code is demonstrated using a simple three-component serial circuit.

I. INTRODUCTION

Electrical circuits in which the current repeatedly reverses direction are called *alternating-current* (AC) circuits. Most electrical grids make use of alternating currents, and operate, in the UK, at a voltage of 230 V, and a frequency, f, of 50 Hz.

Simple AC circuits consist of an arrangement of electrical components. The AC supply generates an alternating current at a fixed frequency and a fixed voltage. A simple functional AC circuit can be described as a connection of resistors, capacitors and inductors. Resistors have an associated resistance, R, capacitors have a capacitance, C, and inductors have an inductance, L. For this reason, these setups are sometimes referred to as RLC circuits.

This report outlines a C++ implementation of RLC circuits in an AC environment. An abstract base class is responsible for laying out the features of a generic component, from which derived classes representing physical components are created. The user can create a customised circuit, which can consist of an arbitrary number of components, connected in series or parallel. The code prints out the circuit's properties in a readable format, and plots its phasor diagram.

As inductors and capacitors are physical components made of wire, they too have an internal resistance, which can be quantified by their so-called quality (Q) factor. The inductor and capacitor classes accommodate this Q factor, and, if it is non-zero, adjust the computation of the circuit properties accordingly.

II. THE C++ PROGRAMMING LANGUAGE

C++ is a general-purpose programming language that was first developed by Bjarne Stroustrup in 1983, and grew out of an attempt at adding class functionalities to the C programming language [1, 2]. While the C language was designed mainly to operate as "close to the machine" as possible such that all operations happening in the machine (the physical part of a computer) could be controlled by the programme, the C++ language builds on the features of C with extended problem-solving capabilities in mind. These include "function argument checking, const member functions, classes, constructors and destructors, exceptions, and templates" [2, p. 9].

III. OBJECT-ORIENTED PROGRAMMING

Object-oriented programming is a paradigm of computer programming that sees *objects* as its basic concept. In the words of Bjarne Stroustrup, "object-oriented programming is programming using inheritance" [3]. Implemented in actual languages, it features components such as encapsulation, which describes association of the programme's data with statements used to manipulate the code. The key idea of encapsulation is realised in the form of *classes* in C++, which can store both *data* and *methods* which define operations (on the data). The idea of classes leads to the concept of *polymorphism*, which describes a hierarchy of classes. *Base classes* can define the most general properties of an object, and the specific implementations of this object can inherit these properties. Lower-level classes are said to be *derived classes* if they inherit from a *base class*. If a base class contains *virtual functions* only, it is said to be an *abstract class*, a concept that is used extensively in this report.

IV. AC CIRCUIT THEORY: THE COMPLEX IMPEDANCE

Only applicable to *direct-current* (DC) circuits, the concept of *resistance* is not sufficient to describe the dynamics of an AC circuit. To this end, one can extend the notion of resistance to a complex quantity called *impedance*, denoted by Z [4, Ch. 13]. In AC circuits, this is necessary as one needs to account for self-induced voltages in inductors and capacitors. These two effects are called the reactance, X, and contribute to the overall reactance with opposite signs. The reactance is made up of an inductive reactance, X_L, and a capacitative reactance, X_C, such that $X = X_L + X_C$. The overall reactance makes up the imaginary part of Z. Note that X_C is always ≤ 0. The ordinary resistance of a resistor,

* ricardo.woelker@student.manchester.ac.uk

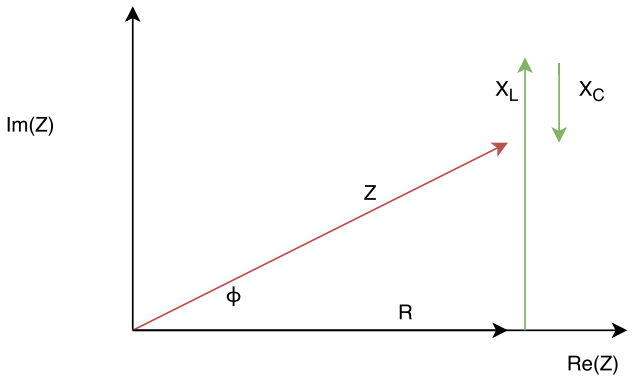

FIG. 1: A phase diagram used to illustrate the complex impedance of an AC circuit.

R, contributes as the real part of Z, so that we have

$$Z = R + iX = R + i(X_L + X_C). \tag{1}$$

From this relation, it follows straightforwardly that the magnitude of the complex impedance, $|Z|$, is simply given by

$$|Z| = \sqrt{R^2 + X^2} = \sqrt{R^2 + X_L^2 + X_C^2 + 2X_L X_C}. \tag{2}$$

A complex quantity, the impedance can be visualised on an Argand diagram, displaying the real part along the x-axis and the imaginary part along the y-axis. This leads further to the notion of a phase angle, ϕ, which is simply defined as the complex argument of Z, or the arctangent of the ratio of its imaginary component and its real component [5]. We thus have

$$\phi = \arg(Z) = \tan^{-1}\frac{\Im(Z)}{\Re(Z)} = \tan^{-1}\frac{X}{R}. \tag{3}$$

A visual representation of the complex impedance on an Argand diagram is called a *phasor diagram*, an illustration of which is shown in Figure 1. For components that are connected in series, the serial impedance, Z_s, is given by the sum of the impedances of the individual components, i.e.

$$Z_s = Z_1 + Z_2 + ... + Z_n, \tag{4}$$

for n components connected in series. If, on the other hand, components are connected in parallel, the overall impedance is defined as the inverse of the reciprocal sum of the impedances. That is, for m components connected in parallel, the equivalent parallel impedance, Z_p, is given by

$$Z_p = \frac{1}{1/Z_1 + 1/Z_2 + ... + 1/Z_m}. \tag{5}$$

A. Component-specific reactances

As discussed above, the resistance of a component contributes simply as the real part of the impedance. The inductive reactance of an inductor, X_L, with inductance L is given by

$$X_L = \omega L, \tag{6}$$

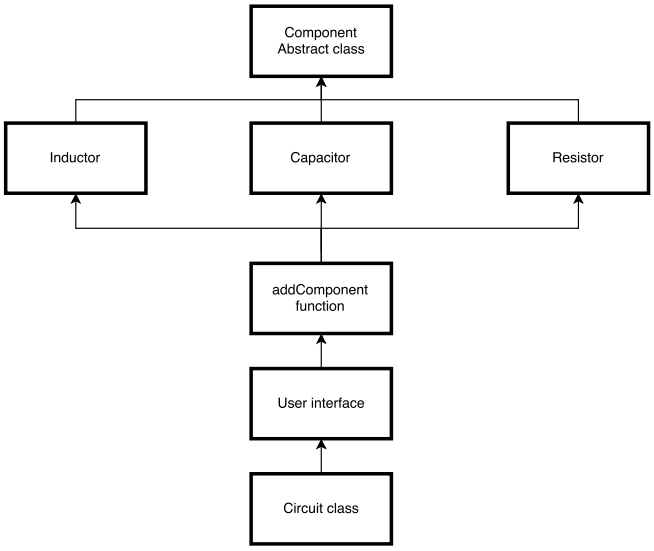

FIG. 2: A class diagram illustrating the class hierarchy used in this code.

where ω is the angular AC frequency of the circuit, i.e. $\omega = 2\pi f$. Similarly, the capacitative reactance of a capacitor, X_C, is defined as

$$X_C = -\frac{1}{\omega C}, \tag{7}$$

where C is the ordinary capacitance.

V. CODE IMPLEMENTATION

The code is broken up into a `main` function, in which various features of the code are demonstrated, and several additional *header* and *source* files which contain the individual classes. A default circuit consisting of nine components is presented. For this circuit, the individual components' properties are output, as well as the overall impedance, Q factor, phase shift and phasor diagram. The various classes are divided into *header files* (`.h`) containing declarations and *source files*, containing the actual definitions (`.cpp`).

A. Class hierarchy

The code design is rooted in an abstract class called `component`, which only contains pure virtual functions. Three physical component classes, `resistor`, `capacitor`, and `inductor`, inherit the methods from the abstract class and define them accordingly. The `main` function creates components from the derived classes via the `circuit::addComp` function. The code thus makes use of polymorphism in that concrete derived classes are derived from an abstract class containing purely virtual functions only. Figure 2 illustrates the overall class hierarchy.

B. Checking user input

In order to ensure that only sensible user input is accepted, the string literals prompted via the

`std::getline` and `std::cin` method are converted into their supposed format. This routine is placed inside a loop that is exited only once the input is sensible. Listing 1 demonstrates the implementation of such a loop.

```
cout << "Enter the circuit's AC frequency in Hz:\n";
string inputFreq;
bool freqIn = true;
while(freqIn){
getline(std::cin, inputFreq);
if( std::atof(inputFreq.c_str()) > 0) {
cout << "Your frequency was stored.\n";
freqIn = false;}
else { cerr << "Input invalid, must be a double > 0. Try
     again.\n"; }}
userCircuit->setF(std::atof(inputFreq.c_str()));
```

Listing 1: A function prompting for the circuit frequency and checking for sensible input.

The user is prompted to input the circuit's AC frequency, for which `string inputFreq` is created. A `bool freqIn` is declared, which is responsible for keeping the prompting loop alive until it is set to `false`, which occurs once the user input is deemed sensible. Here, the circuit's frequency must be greater than or equal to zero, which is checked in the argument of the if statement via `std::atof(inputFreq.c_str())>0)`. Finally, the method `circuit::setF` is called and sets the circuit's AC frequency.

C. Error handling

The code makes use of the C++ exception handling. This provides a method for reacting appropriately to exceptional circumstances. The portion of the code prone to errors is placed inside a `try` block, which, when an error occurs, `throws` a user-specifiable `exception`. This is subsequently caught by a `catch` block. The code below in Listing 2 illustrates this feature in the handling of the complex division by zero in the calculation of the circuit impedance.

```
try{ Z = complexOne/recipSum; }
catch(...){
cout<<"Error: Reciprocal sum division by zero."<<endl
    ;
exit(-1);}
```

Listing 2: Exception handling in the case of complex division. If `complexOne` ($= 1 + 0i$) is divided by zero, the exception will be caught by the `catch` block.

D. The addComp function

The `circuit` class is equipped with an `addComp` function, which is able to create any of the three available components, making full use of the polymorphic capabilities of the code. It thus provides a user interface, with which the user, or the main programme, can communicate with the derived component classes. The `circuit::addComp` function is summarised in Listing 3.

```
void circuit::addComp(
  const string& type,
  double a, double b, char d ){
 if( type == "resistor" ){
  comp.push_back(new resistor(a,b,d));}
 else if ( type == "capacitor" ){
  comp.push_back(new capacitor(a,b,d)); }
 else if ( type == "inductor" ){
  comp.push_back(new inductor(a,b,d));}
 else{ cout...; } }
```

Listing 3: The `circuit::addComp` function which creates dynamically creates components.

E. The impedance-computation function

The impedance-computation function is a method of the `circuit` class. All circuit components are looped over, and components that are connected in series are sent to another function responsible for the computation of the serial impedance. Components that are connected in parallel are sent to the same function individually. The overall impedance is computed as the inverse of the reciprocal sum of all the serial impedances. In order to accommodate nested AC circuit, a `bool n` is introduced which flags whether the component is in a nested circuit or not. Whenever a series of components is flagged with `n==true`, a sub-circuit is created, whose impedance is computed recursively. This feature is illustrated in Listing 4.

```
1  if( doNested ){
2  shared_ptr<circuit> circSub (new circuit(*this)); //
       make local copy for the operations
3  shared_ptr<vector<complex<double>>> nestedImpCircuit;
4  shared_ptr<circuit> nestedOut; // create sub-circuit
       for nested
5  bool doneNested = false;
6  for( auto d = circSub->comp.begin(); d != circSub->
       comp.end(); ++d ){
7  if( (*d)->getNested() == true ){
8  nestedOut->addComp((*d)->name(),(*d)->getVal(),(*d)->
       getQ(),(*d)->getConn());
9  doneNested = true;}}
10 if( doneNested ){
11 nestedOut->setF(this->getF());// give it same frequ
12 nestedOut->computeZ(false); // recursively submit
       function, but not nested
13 nestedImpCircuit->push_back(nestedOut->getZ());
14 *nestedImpCircuitImport = *nestedImpCircuit;}} //
       assign permanently
```

Listing 4: The `computeZ` function includes a recursive definition, which takes effect when a component is flagged with a `true` nesting option.

A local copy of `this` is created via a `shared_ptr` (see Section VG) in order to avoid memory leaks within the nested block. The components that are flagged as being nested via `getNested` are added to another locally allocated circuit (`nestedOut`), whose impedance is recursively created and later on copied into the parent function (line 14). The impedance is stored finally in the circuit's *member data* as a `complex<double>` object.

F. Output and the « overload

In order to allow for a user-friendly and intuitive output using the shift operator (`std::ostream::operator<<`), it is overloaded as a friend operator for the `circuit` class. It `std::couts` to the console general information about the circuit,

such as the number of components, the phase angle, the AC frequency, the complex impedance and its magnitude, the overall Q factor, as well information about all constituent components, including type, impedance, Q factor, phase angle, and connection type (series, parallel). An example output is shown in Listing 5.

```
This circuit has 4 components:
Element 1: resistor with R = 0.300 Ohms in ser.,
  Z = (0.300,0.000) Ohms, phi = 0.000
Element 2: capacitor with C = 0.920 nF in ser.,
  Z = (0.000,-3459890.067) Ohms, phi = -0.500 Pi, Q =
      3.200
Element 3: inductor with L = 0.930  H in ser.,
  Z = (0.000,292.168) Ohms, phi = 0.500 Pi, Q = 1.800
Element 4: inductor with L = 0.020  H in par.,
  Z = (0.000,6.283) Ohms, phi = 0.500 Pi, Q = 2.300
The impedance is 0.001 + 0.571 i Ohms
Its magnitude is 0.571 Ohms
Its AC frequency is 50.000 Hz
Its Q factor is 0.005
Its phase difference is 1.569 rad
```

Listing 5: The `computeZ` function includes a recursive definition, which takes effect when a component is flagged with a `true` nesting option.

In order to display the complex impedance in a more human-readable format, this code makes use of compact *ternary if* statements, an example of which is shown in Listing 6. If the imaginary part of the complex impedance is greater than or equal to zero, a $+$-sign is inserted, otherwise nothing is inserted, and the $-$-sign will come from the negative double output from the `complex<double>` object. Ternary have the prototype syntax `condition ? do_if_true : do_if_false`.

```
cout<<Z.real()<<(Z.imag()>=0?"+":"")<<Z.imag()<<" i
    Ohms\n";
```

Listing 6: A ternary if statement allowing for a more readable formatting of the complex impedance.

G. Smart pointers

Smart pointers are a new feature within C++ 11 which act like ordinary pointers but have advanced memory management capabilities [6]. In particular, this code makes extensive use of *shared pointers* (`shared_ptr`), which represent *shared ownership*. That is, when multiple pieces of code require access to the same pointer, the `shared_ptr` pointer keeps track of the number of times it is referenced, and gets deleted as soon as the count goes to zero. This is useful in particular in the recursively defined impedance function, where it is able to efficiently prevent memory leaks.

VI. LOSSY COMPONENTS

The simplified assumption that inductors and capacitors have no internal resistance is often not valid for physics applications. Like all electrical circuit components, they are made of (copper) wire, which has an associated intrinsic electrical resistance.

A. Q factor of an inductor

The Q factor of an inductor is the ratio of its reactance and its effective internal serial resistance. It is given by the relation

$$Q_{\mathrm{L}} = \frac{\omega L}{R_{\mathrm{L}}}, \tag{8}$$

where R_{L} is the effective internal series resistance of the inductor.

B. Q factor of a capacitor

Similarly, the Q factor of an inductor is defined as the ratio of its reactance and the capacitative resistence, i.e.

$$Q_{\mathrm{C}} = \frac{X_{\mathrm{C}}}{R_{\mathrm{C}}}. \tag{9}$$

C. Overall circuit Q factor

The quantification of the Q factor of the overall circuit follows straightforwardly from the Q factors of the individual components. It is defined as the inverse of the reciprocal sum of all the components in series, regardless of whether the individual components are connected in series or parallel. It is thus given by

$$Q_{\mathrm{circuit}} = \frac{1}{\sum_i \frac{1}{Q_i}}, \tag{10}$$

where i is an index running over all components in series. Here, Q can denote either Q_{C} or Q_{L}.

VII. PLOTTING OF THE PHASOR DIAGRAM

The circuit class has a plotting method called `circuit::plotPhasorDiagram`, which provides Cartesian plotting functionalities based on ASCII characters [7] that are output in the console window. At the heart of the plotting function lies a computation of the impedance's slope on a phasor diagram, m, which is given by

$$m = \tan \arg(Z). \tag{11}$$

The plotter loops over a grid of size 80 by 25, and plots a dot whenever a linear equation related to the slope is satisfied. Otherwise, it plots dashes. An example output of the plotter can be seen in Figure 3.

VIII. FUNCTIONAL TEST

In order to assess if the impedance calculator works correctly, a test circuit consisting of a resistor, a capacitor and an inductor in series was considered. Its impedance and phase angle were calculated by hand as well as using an online calculator [8], and compared with the result of the code presented in this report. The diagram for the circuit in question is shown in

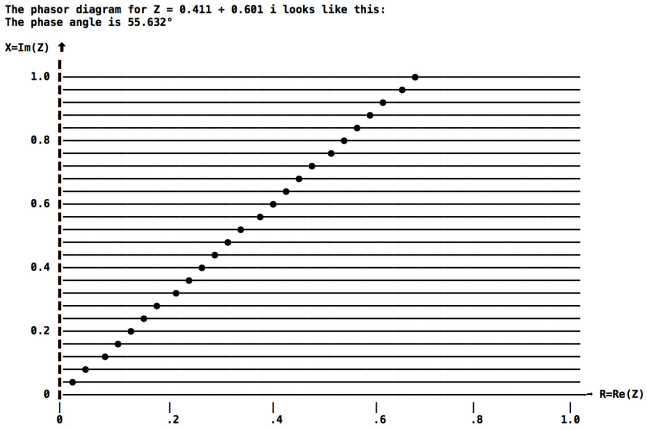

FIG. 3: An example output generated from the phasor-diagram plotting function. The function plots dots and dashes as ASCII characters in the console, implementing a graphical output.

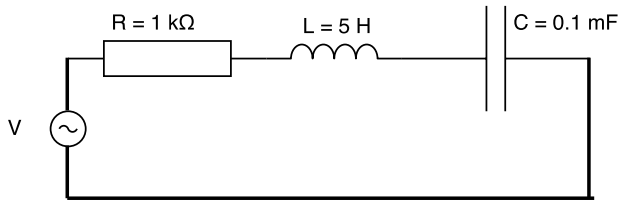

FIG. 4: A test circuit consisting of a resistor ($R = 1$ kΩ), an inductor ($L = 5$ H) and a capacitor ($C = 0.1$ mF). Its impedance is 1.835 kΩ with a phase angle of 56.99°.

Figure 4. Both the online calculator and the manual calculation yield an impedance of 1.835 kΩ with a phase angle of 56.99°. These values are identical to what is determined by the code, as can be seen from the output in Listing 7. We thus conclude that the code is functional by way of example.

```
This circuit has 3 components:
Element 1: resistor with R = 1000.000 Ohms in ser., Z =
    (1000.000,0.000) Ohms, phi = 0.000
Element 2: capacitor with C = 100000.000 nF in ser.,
    Z = (0.000,-31.831) Ohms, phi = -0.500 Pi
Element 3: inductor with L = 5.000 H in ser., Z =
    (0.000,1570.796) Ohms, phi = 0.500 Pi
The circuit has a complex impedance of 1000.000 +
    1538.965 i Ohms
Its magnitude is 1835.324 Ohms
The circuit has an AC frequency of 50.000 Hz
Its phase difference is 56.985 degrees
```

Listing 7: An example circuit and the impedance calculation. This circuit is labelled `c2` in the code submitted.

IX. CONCLUSION AND FUTURE WORK

An object-oriented implementation of AC circuits consisting of potentially lossy capacitors, resistors and inductors in C++ was demonstrated. The code's functionalities have been demonstrated and included polymorphism and inheritance, the computation of the overall circuit impedance and Q factor, nested circuits and a recursive impedance function, a user input/output interface, a console-based plotter, exception handling and several other advanced C++ features. Advanced C++ features used included smart pointers, ternary ifs, exception handling, header files, and abstract classes. The code's functionalities were demonstrated successfully by comparing its results for a simple three-component serial circuit with those of a manual calculation and an online tool.

Future work on this AC circuit code could focus on implementing a graphical user interface (GUI), which would allow for a graphics-based creation of an AC circuit. A library such as `GNUPLOT` [9] would further facilitate a graphically more appealing way of plotting the phasor diagram. It might further be appropriate to extend the recursive definition of the impedance-computation function to allow for an arbitrary degree of nestedness, e.g. a nested circuit within a nested sub-circuit within another nested sub-sub-circuit and so forth.

There is plenty of scope for this code to be extended to calculate and illustrate the resonant phenomena in driven AC circuits, including an assessment of the resonance curve and the power dissipated.

[1] B. Stroustrup, Software: Practice and Experience **13**, 139 (1983).
[2] B. Stroustrup, *The C++ programming language* (Pearson Education India, 1995).
[3] B. Stroustrup, IEEE software **5**, 10 (1988).
[4] A. Agarwal and J. Lang, *Foundations of analog and digital electronic circuits* (Morgan Kaufmann, 2005).
[5] P. Horowitz and W. Hill, *The art of electronics*, Vol. 2 (Cambridge university press Cambridge, 1989).
[6] H. Sutter, Dr. Dobb's Journal (2002).
[7] C. E. Mackenzie, *Coded-Character Sets: History and Development* (Addison-Wesley Longman Publishing Co., Inc., 1980).
[8] C. R. Nave, Department of Physics and Astronomy, Georgia State University. Retrieved March **16**, 2003 (2005).
[9] J. Racine, Journal of Applied Econometrics **21**, 133 (2006).