PHYS30762 Object-Oriented Programming in C++
School of Physics and Astronomy

OOP in C++ Project Report
A.C. Circuits

Harry Howell
Student ID: *9632467*

May 10, 2019

Abstract

The following report outlines the development and structure of a program to calculate the impedance of an A.C. circuit with an arbitrary list of components. These components can be added in series or in parallel. There is implementation for the addition of resistors, inductors, and capacitors into an overall 'circuits' class.

The code is designed so that any class declarations are held in header files with associated .cpp files to define their functionality and operator overloads. A main.cpp then demonstrates features declared within the classes with data input by the user[a].

[a] *C++ Logo.* `https://github.com/isocpp/logos`. Accessed: 29-04-2019.

1 Introduction

Within A.C. electrical circuits, the phase between input waveform and output waveform of a resistor is zero, i.e. it is purely resistive. However, components such as inductors or capacitors are *reactive* and the phase between output and input sinusoidal signals is frequency-dependent. Keeping track of phase lags and total impedances for large circuits by hand would be arduous. The program written here allows input of resistors, capacitors and inductors with the aim of automatically calculating the impedance of the input circuit, along with keeping track of its total phase lag. From this, such properties as whether the circuit has capacitive or inductive reactance can be inferred [1].

The code itself is written to further exercise aspects learnt in the object-oriented programming course (PHYS30762) such as the use of classes, the STL library and polymorphism.

1.1 Necessary Theory

Phasors and A.C:

In A.C. circuits driven by sinusoidal sources at some frequency, there can be a phase difference between voltage and current, as mentioned. The magnitude of currents are still proportional to the driving voltages - as in Ohm's law - however, the time-variation of the current and voltage, along with their phase differences, are more succinctly represented in complex form.

Voltages and currents, $\boldsymbol{V}$ and $\boldsymbol{I}$ are thus represented using phasors in the form

$$\boldsymbol{V} = V_0 e^{j(\omega t + \phi)} \tag{1}$$

$$\boldsymbol{I} = I_0 e^{j\omega t} \tag{2}$$

with V_0 the voltage amplitude, ω the source angular frequency and ϕ the phase.
Ohm's law is then generalized to

$$\boldsymbol{V} = \boldsymbol{I}\boldsymbol{Z} \tag{3}$$

where $\boldsymbol{Z}$ is the complex impedance of the circuit. This factor describes within it the phase lag between the voltage-current relationship.

Resistors:
For resistors the *impedance* (the combination of resistance and the reactance) is simply equal to its resistance. $\boldsymbol{Z_R} = \boldsymbol{R}$.

Inductors:
For an inductor, the voltage is given by

$$\boldsymbol{V} = L\frac{d\boldsymbol{I}}{dt} = j\omega L\boldsymbol{I} \rightarrow \boldsymbol{Z_L} = j\omega L \tag{4}$$

with L being the inductance.

Capacitors:
Similarly for capacitors of capacitance C

$$\boldsymbol{I} = C\frac{dV}{dt} = j\omega C\boldsymbol{V} \rightarrow \boldsymbol{Z_C} = \frac{-j}{\omega C} \tag{5}$$

with the $-j$ term accounting for the 90° leading phase shift for the current [2].
In the proposed program there will be an arbitrary number of components each of which can be configured in parallel or series with one-another. Equations (6) and (7) describe how the complex impedances of individual components must be added to get the overall impedance depending on their series or parallel configuration.

$$\text{Series:} \quad \boldsymbol{Z} = \boldsymbol{Z_1} + \boldsymbol{Z_2} + \boldsymbol{Z_3} + \cdots \tag{6}$$

$$\text{Parallel:} \quad \boldsymbol{Z} = \frac{1}{\frac{1}{\boldsymbol{Z_1}} + \frac{1}{\boldsymbol{Z_2}} + \frac{1}{\boldsymbol{Z_3}} + \cdots} \tag{7}$$

Each component that is added to our circuit class is done so iteratively. Therefore only code to add one component to the previously calculated impedance is needed.

```
if (comp->getType() == "Capacitor"){
   if (comp->getConfig() == "Series"){
      serCap++;
      // adjust circuit impedance accordingly
      imped += comp->getImpedance(angFreq);
   }
   else{
      parCap++;
      imped = pow((pow(comp->getImpedance(angFreq),-1) + pow(imped,-1)),-1);
   }
}
```

The code immediately above shows how the circuit's impedance is added to in the event that the added component is a capacitor, adjusting depending on whether or not it's configuration is in series or parallel.

Since the impedance has the form $a+bi$, the phase-shift for the component - or the entire circuit - can be worked out simply by taking the arctan of *imaginary*/*real* components.

```
double circuit::getPhaseDiff(){
   double phase = atan(imped.imag()/imped.real());
   return phase;
};
```

2 Code Design and Implementation

2.1 Structure of code

Figure 1 shows the hierarchy for the file system.

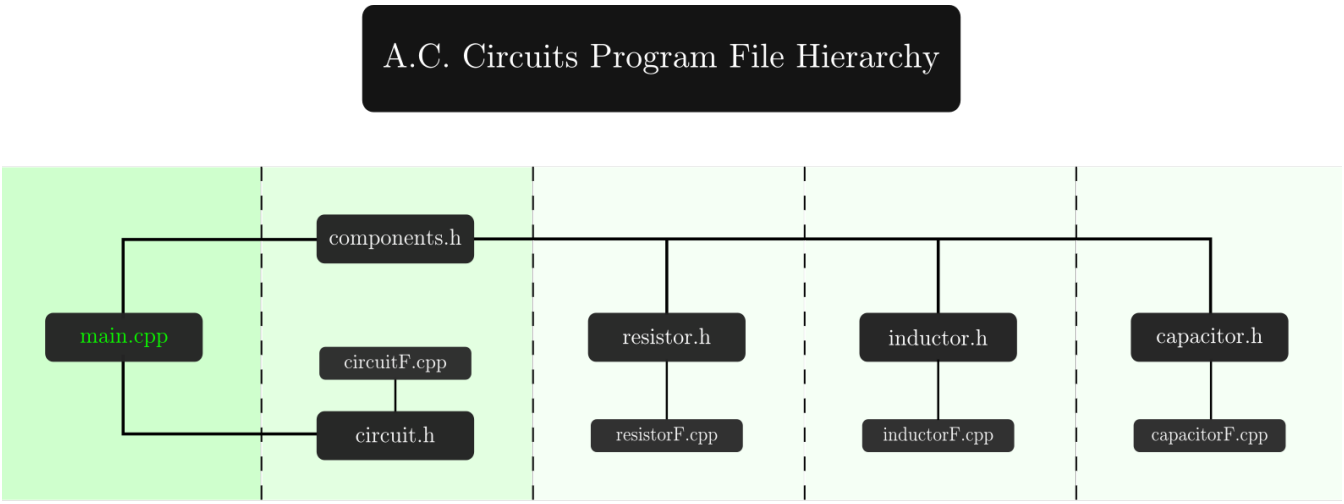

Figure 1: File hierarchy for A.C. circuits program. Header files declare classes and their member functions whilst source files define the functionality.

The `components.h` file acts as an abstract class for the derived classes `resistor.h`, `inductor.h` and `capacitor.h`. Within these classes, constructors, destructors and access functions, along with member functions are all defined. The private data members for the derived classes consist of a pair for component characteristics and a double to hold the angular frequency of electrical signal passing through it.

Separate from this object, the circuit class, defined in `circuit.h` contains within it the following protected data: a list of pointers to components within the circuit, a complex double holding the total impedance, the angular frequency of the source e.m.f. and counts for how many resistors, capacitors or inductors are are added in series or in parallel.

When choosing an array to store the component library, different STL containers were considered. The considerations were a simple array of base class pointers, a dynamic vector array or a list container. The chosen library container was a list of components; this sequence container will perform mathematical operations and sorting faster than a vector array, and with the use of pairs, each element can contain a double for the component value (be it resistance, inductance or capacitance) and a string for whether or not it is added in series or parallel (‘s’ or ‘p’).

Details of which member functions belong to which class and their broad functionality are outlined in the tables overleaf.

Member function	Description
`void info()`	Print member data for the component
`string getConfig()`	Returns Series or Parallel configuration
`complex<double>getImpedance()`	Returns complex Impedance in format (real, imaginary)
`double getAbsImpedance()`	Returns magnitude of complex impedance
`double getPhaseDiff()`	Returns phase difference between voltage and current
`void setFreq(const double &Freq)`	Set electrical signal frequency through component
`string getType()`	Returns type of component as string
`double getFreq()`	Returns present signal frequency through component

Table 1: Table showing which functions are common to all components across the three derived component classes. One other access function to return the value of each component is included in the three classes. E.g. this function is named `double getInductance()` for the inductor and similar for the remaining two.

Member function	Description
`void addComp(component* comp)`	Add single pointer to component to list of components
`void addCompList(vector<string>)`	Add multiple pointers to components to list
`void setFreq(double &fr)`	Set electrical signal frequency for circuit source
`void info()`	Print member data for the circuit
`void graphic()`	Print condensed graphic for circuit
`complex<double>getImpedance()`	Returns total Impedance for circuit
`double getAbsImpedance()`	Returns magnitude of complex impedance for circuit
`double getPhaseLag()`	Returns phase lag between voltage and current for circuit
`double getFreq()`	Returns source signal frequency in circuit
`double size()`	Returns # of components in the circuit

Table 2: List of member functions available for use within the circuit class.

Function	Description
`double cinVerify(double &val)`	Function to verify user input for component double
`vector<double>cinVerify(string)`	Overload for multiple component string input
`template<class T>T cinVerify(..)`	Overload for strings and chars
`vector<string>cinVerify(string)`	Overload for multiple component double input
`template<class T>void vecOut(..)`	Function to print vector components

Table 3: Functions declared outside of the circuit class definition but within the same header file. `cinVerify()` functions check the input data by the user at different points within `main.cpp`. This is further discussed in section 3. `vecOut()` was written to avoid repetition in `main.cpp`.

3 Results

The usage of the class structure and their associated methods is summarised in figure 2 below.

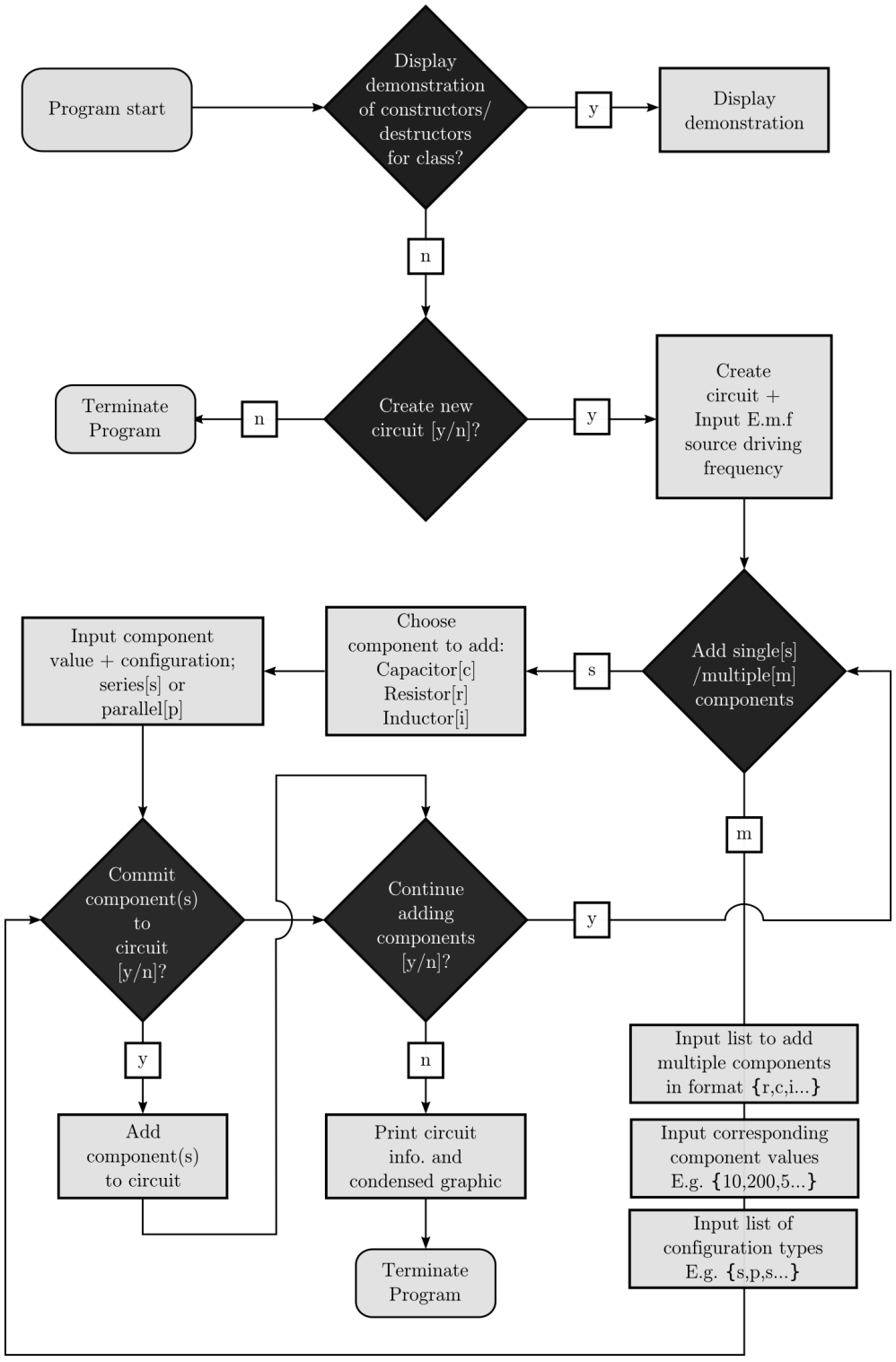

Figure 2: Simple flow-chart outlining the functionality of `main.cpp`. Note: at each input by the user, as outlined, the `cinVerify()` functions outlined in table 3 check the user's input to make sure it is of the correct format.

3.1 Safeguarding against user input errors

If the program is to have the user input the component values and configuration in the circuit, then the input value types must be checked. This is done with the use of the `cinVerify` functions which are overload for different input types. A summary of each of the checker functions is outlined.

`double cinVerify(double& val):` Checks that the input is of type double and is greater than 0. Used for the input of the *single component* values.

`vector<double> cinVerify(string& val, vector<double>& valList, double val1Length):` Takes user input `string &val` - which should be a list convertible to doubles, separated by commas - checks if the elements of the list can be converted to doubles, converts and appends to `valList` if appropriate and throws an error if not; asks user to re-enter the list if this is the case. The function makes use of exception handling and is used for checking the list of component values entered by the user when choosing to create *multiple components*. Also checks that the output list is the same length as that for the previously-declared multiple component list (see figure 2).

`template <class T> T cinVerify(T& val, vector <T>& array):` Function used to check that the input value from the user `T& val` is found in the vector `vector <T>& array`. This is used to check whether input values are from the choices lists {`"s","p"`}, {`"s","m"`}, {`"y","n"`} and {`"r","c","i"`} shown to be used in figure 2. Left as a template function as could be true for type char also.

`vector<string> cinVerify(string& val, vector<string>& array, vector<string>& compList):` Function to take user list for multiple component declaration. Similarly, checks that elements of the entered list are members of `vector<string>& array` then appends to and returns `compList` if this is true. Again, if this is not the case, asks user to re-input the list of components.

4 Discussion and Summary

One major flaw is that once components have been committed to the circuit, there is no implementation within `main.cpp` to edit the individual component values afterwards. This includes the ability to add/remove components. In order to change new components, a new circuit would have to be redefined completely, and with a circuit of say 100 components, the want to change maybe 2 of these 100 may not warrant the complete redefinition of the entire circuit. This feature would need to be added in future. The ability to combine circuits themselves would also be a improvement for the user. Implementation of this may be combined with the use of smart pointers such that when certain circuits go out of scope, the memory being used is automatically freed. Two more relatively simplistic additions to the code may be to include further checks on the user inputs and throw warnings if the input component values are infeasible, and also to append even more component classes such as LDR's, transistors, thermistors etc... to create even more complex circuits.

In summary, the code written allows input for an arbitrary amount of electrical components and automatically calculates useful properties including the total impedance, however, the code would benefit from a wider library of component classes to choose from along with improved user accessibility and editing capabilities.

References

[1] Theodore Korneff. *Introduction to electronics*. Elsevier, 2012. Chap. 2.

[2] Paul Horowitz and Winfield Hill. *The Art of Electronics*. Cambridge Univ. Press, 1989, pp. 44–47.