Analogue Circuits Coding Project

Tomas Welsh

Abstract

A computer program in C++ was written to calculate the impedance of A.C. circuits with arbitrary numbers of resistors, capacitors and inductors connected in series or parallel. The class hierarchy created lets a coder copy and move circuits, get frequency, impedance, magnitude of impedance and phase difference of circuits and their components. It contains functions to add generic units in serial and parallel, including subcircuits, as well as functions to print out circuits in a logical manner. The code lets a user create a library of different units, which can then be added to with specific components or circuits a user creates, and respective impedances can be calculated.

1 Introduction

In a circuit, alternating voltages and currents vary sinusoidally with time. They are characterised by an angular frequency ω, a phase, which corresponds to the offset of the sinusoid, and an amplitude. Therefore we can write the voltage, for example, as

$$V(t) = V_0 \cos(\omega t + \phi_V), \tag{1}$$

with V_0 the maximum value of the voltage, i.e the amplitude, and ϕ_V the phase. We can then represent this voltage as the real part of the complex number $\tilde{V} = V_0 e^{i(\omega t + \phi_V)}$; this is known as a phasor. The currents everywhere in a circuit vary sinusoidally at the same angular frequency but the phase of a current is not necessarily the same as that for the voltage. Hence, we can characterise currents very similarly as $\tilde{I} = I_0 e^{i(\omega t + \phi_I)}$, where I_0 and ϕ_I are the respective amplitude and phase.

We shall now define the complex characteristic of analogue circuits - the impedance - very similarly to [1] using the capacitor as an example. Consider a capacitor with capacitance C with a voltage across it defined by the phasor $\tilde{V} = V_0 e^{i\omega t}$. It can be shown that the current leads the voltage by a phase angle $\pi/2$, and that the amplitude of the current is $I_0 = \omega C V_0$. The phasor of the current is then $\tilde{I} = \omega C V_0 e^{i(\omega t + \pi/2)}$. Since $e^{i\pi/2} = i$, we can write

$$\tilde{I} = i\omega C V_0 e^{i\omega t} = i\omega C \tilde{V} \implies \tilde{V} = \frac{1}{i\omega C}\tilde{I}. \tag{2}$$

This equation has the form of Ohm's law with the resistance R replaced by the complex number $1/i\omega C$. We can similarly show that the ratio between $\tilde{V}$ and $\tilde{I}$ for inductors is $i\omega L$, for an inductance L, and trivially R for resistors. The ratio $\tilde{V}/\tilde{I}$ is called the complex impedance and is denoted by Z. The complex impedance hence satisfies the equation

$$\tilde{V} = Z\tilde{I}. \tag{3}$$

The phase of Z is then the difference in phase between the voltage and the current and the magnitude is the real impedance, i.e resistance.

Just as for resistance, the complex impedance of a series circuit Z_S is the sum of the individual components' impedances, i.e

$$Z_S = \sum_{i \in I} Z_i, \tag{4}$$

where Z_i denotes the impedance of an individual component of the circuit. Similarly, the complex impedance of a parallel circuit Z_P is the inverse of the sum of the inverse of each individual's impedance. This can be written more clearly as

$$\frac{1}{Z_P} = \sum_{i \in I} \frac{1}{Z_i}. \tag{5}$$

Using Figure 1 below we shall give a demonstration of how to calculate the impedance of a circuit by evaluating the most nested parts first; the code implemented the exact same methods. First consider branch A; we have a resistor and capacitor in series. Therefore, to find the impedance of the branch we must add the individual impedances 4 and 1/(i x 50 x 0.0025) = -8i, which gives 4 - 8i. Now we can consider the impedance of branch A and branch B in parallel. Branch B has impedance i x 50 x 0.4 = 20i and so the total impedance of the parallel part of the circuit is 1/(1/(4 - 8i) + 1/(20i)), which evaluates to 10 - 10i. Finally we then consider the series circuit C, which is composed of the two resistors and the parallel circuit. The impedance of the total circuit will then be 5 + 8 + (10 - 10i) = (23 - 10i)Ω. Note that units have been omitted until the end.

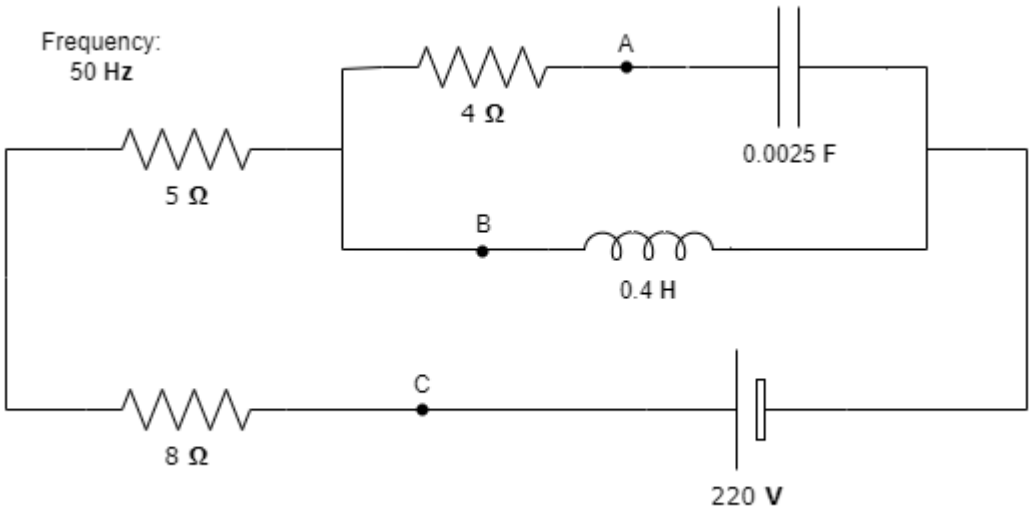

Figure 1: Diagram of an AC circuit with ideal components.

2 Code Design and Implementation

The code, as can be seen in the splitting of the header files, was constructed in two parts: the unit class hierarchy and then the user interface. To begin the design of the former, the class hierarchy itself was constructed on paper. It was always apparent we would need an abstract base class so that we could utilise polymorphism, for example, when creating a list and applying a method element-wise. We chose to call this base class 'unit'. This was because the immediate derived classes were chosen to be 'circuit' and 'component' so the word 'unit' provided an appropriate generic name. These derived classes were constructed so that a circuit would hold a vector of unit pointers, which would represent the components of the circuit. The choice of units here was so that circuits could hold subcircuits. The circuit class was then split into derived classes 'parallel circuit' and 'series circuit' as each one has a different implementation of how to calculate the impedance. 'component' was split into three ideal components: 'capacitor', 'resistor', 'inductor' that would each hold one data-member - characteristic - which would represent their unique attribute, for example this would would be inductance for an inductor. The term characteristic was chosen as all three components had the same structure and so we could generalise and put 'characteristic' in their base class as opposed to having a unique data-member for each.

All desired methods were then written down on paper. This included functions like get_phase and get_frequency as well as more complex functions like print_func. With the class construction, we then decided where each method needed to go. This was done by considering the furthest derived classes, which would be the only things instantiated, and choosing where they stopped sharing the implementation of a method. For example, get_impedance had to be defined differently for every derived class so had to be written out five different times while get_frequency had the same implementation everywhere so could be implemented in our base class 'unit'. Note that to utilise polymorphism all functions not implemented in the base class still had to written there as virtual functions.

To design the user interface, we constructed a story line of how a user should interact with the code. This included points where there were different options and so there were multiple avenues a user could take to get to the end of the story line. The idea of the code was for a user to create a library of components to then build circuits from. Keeping in mind that we wanted the ability for a user to make circuits out of subcircuits, after making a circuit they should be offered the opportunity to add the circuit they had created to their list. It was realised that constructing components at the start from scratch proved laborious so an option for the user to create a random starter list of components with a size of their choice was included. Also, a user should be able to set the frequency of the circuit they were creating so that option had to be included in the story line. See Figure 2 below

for a diagrammatic representation of the constructed story line.

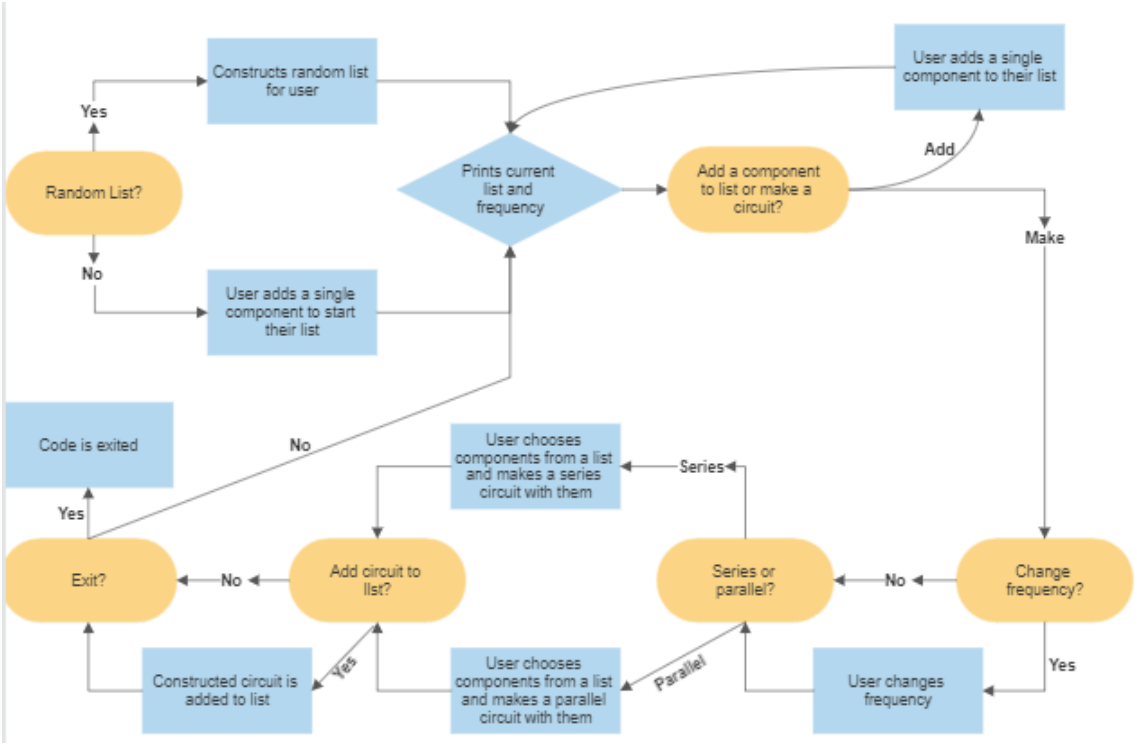

Figure 2: Flow chart of the user's experience with the code. Note that the user starts at the question 'Random List?'

The desire for minimal code duplication led to some relatively complex implementation. In particular, the preference for all fundamental components to have the exact same structure bar the implementation of the get_impedance function led to the creation of a template designed to produce classes of the fundamental components. The template was structured so that there would be no need for specific constructors or copy and move semantics, for example, as well as an ability to inherit generic functions, like get_characteristic. Post-creation, however, there was the realisation that different fundamental components would need a way to identity themselves for printing so parameterised constructors had to be introduced and also that there was no need to define our own copy/move constructors as the classes were very simple and so the default constructors a compiler creates would prove sufficient as well as more efficient. Therefore, in the end, the template structure as opposed to a generic base class was only useful in providing a quick way to have a clone function for each of the fundamental components. It was, however, thought of no consequence to leave the template structure in and it does lend itself to adding class specific functions to the code efficiently.

In contrast to the component classes, we had to specify our own copy and move semantics for the circuit class as one of the data members was a vector of pointers. This led to the creation of the clone function as if we were to deep copy a vector of base class pointers we would need to have a specific way to copy each derived class in the spirit of polymorphism. The clone function was also useful in the main code when copying over circuits made of unique pointers. The copy constructor of the circuit class could then create a new vector of unit pointers by iterating over the old one and using the clone function to create a new 'units' data member for the new object. Listing 1 gives the implementation of how the clone function creates a new unique pointer of a derived class and how the copy constructors then uses it. The move constructor was much simpler to create as the compiler deals well with pointers by using std::move but this still had to be defined along side the copy constructors.

Listing 1: Clone function and copy constructor example.

```
std::unique_ptr<unit> parallel_circuit::clone() const
{ return std::make_unique<parallel_circuit>(*this); }
```

```
circuit::circuit(const circuit& other) {
    for (const auto& unit_ptr : other.units) {
    units.push_back(unit_ptr->clone());}
}
```

When designing the random_list_generator function, it became apparent that smart pointers were required. This was because the random_function could not be modular without them as we could not include deleting raw pointers inside of it or then we would not be able to return anything. Moreover, smart pointers made creating the main code generally a lot easier as we did not have to worry about deleting things at appropriate points. It was decided that the frequency should be a static data member of the whole class system that could be viewed as an environment that all circuits were operating at. This seemed logical as when adding circuits together they would have to be at the same frequency. The choice of a static member was because all derived classes should be able to access the frequency while we did not want to let anything outside the class access the frequency; also, we would be avoiding the problems of global variables, like name collisions or setting up test cases. Thus, both print_list and random_list_generator had to be friend functions so that they could set or print out the frequency.

To construct a printing system, each derived class needed its own print_func. The implementation was very different between a circuit and a component as the circuit had to print all the things inside of it while the component just had to print what it was, eg. a capacitor, and what value its characteristic took. Due to circuits being inside circuits, the print_func had to take on a recursive definition. To keep track of the level of nesting and so create a more visually pleasing printing we introduced a 'level' parameter that was added to for each recursion of the print_func. This can be seen in Lisitng 2. The print_list function was then easy to create as one just had to iterate through the user's list and apply the print_func method to each element, ensuring that the level parameter was set to 1 at the beginning. To be able to print something different between a capacitor and resistor, for example, the identifier data member was created to hold specific information about each class. This was done using a std::map as it was thought that more attributes to the identifier may be added at a later date and so the container format worked well and also mapping allowed for easier indexing.

Listing 2: The print_func implementation for the circuit class.

```
void circuit::print_func(int level, bool full_output)
  {
    std::cout << identifier["name"];
    if (full_output) {
      std::cout << " with impendance magnitude " <<
      this->get_impedance_magnitude() << " and phase " <<
      this->get_phase() << ". It has elements:" << std::endl;
    }
    else { std::cout << " with elements:" << std::endl; }

    std::vector<std::unique_ptr<unit>>::const_iterator iter;
    for (iter = units.begin(); iter != units.end(); iter++){
        std::cout << std::string(level, '-');
        (*iter)->print_func(level + 1, full_output);}

    std::cout << "//////////end of circuit//////////" << std::endl;
  }
```

In disagreement with the project description of the analogue circuit, we did not have impedance as a data member of each instantiated class but instead as a method. This was because having it as a data member meant that functions like add_unit or set_characteristic would required unnecessary extra functionality to alter the impedance data member. Having impedance instead as a method meant that we could do whatever we liked to an object and only worry about the impedance when we directly wanted it, making the code simpler and usually more efficient. The only possible drawback is that if one wanted to get the impedance from an object many times without changing any attributes of the object then the method would be less efficient as the impedance has to calculated each time. This, however, is unlikely to happen.

All code was split into header and implementation files according to its function. This was to increase readability as well as compiler efficiency. Note that one had to be careful when splitting code. For example, friend functions still had to be defined again at the bottom of the header file, virtual functions did not need to included in the implementation file and the keyword override did not need to be used again in the implementation file. The override keyword was used to ensure when writing code that the virtual function existed that the coder was referring to. This proved especially useful when splitting code between files as sometimes the implementation file was not seeing the function in the header, which the override keyword would pick up. Additionally, different namespaces were used for each header file. This was to avoid name collisions; words like 'unit' could definitely suffer from this. Note that two functions were included in the main file as they used both of the header files in their construction but it was decided it was too little code to create a separate header file for. Also, implementation of templates was written in the header files themselves as templates are not compiled into binary code until they are instantiated with a specific data type and so the compiler needs to see both the declaration and the definition of the template at the same time.

3 Results

As described in the previous section with aid of Figure 2, the user follows a process to create circuits from a list of components. These circuits can then be saved to create new, bigger circuits ad infinitum. When a user creates a specific circuit the magnitude of impedance and phase of each component of the circuit plus the circuit itself is displayed and the impedance is printed at the bottom; see Figure 3 for an example. One can see in the implementation of the full_output Boolean argument in Listing 2 that the magnitude of impedance and phase of a circuit can be chosen to be printed; this was similarly implemented for components. Full output was only printed when the user created a new circuit. Otherwise, just the characteristic of each component was printed.

```
You have contructed a Series Circuit with impendance magnitude 25.0799 and phase -0.410127. It has elements:
-Resistor: 8 Ohms; impedance magnitude: 8, phase: 0
-Resistor: 5 Ohms; impedance magnitude: 5, phase: 0
-Parallel Circuit with impendance magnitude 14.1421 and phase -0.785398. It has elements:
--Series Circuit with impendance magnitude 8.94427 and phase -1.10715. It has elements:
---Resistor: 4 Ohms; impedance magnitude: 4, phase: 0
---Capacitor: 0.0025 F; impedance magnitude: 8, phase: -1.5708
//////////end of circuit//////////////
--Inductor: 0.4 H; impedance magnitude: 20, phase: 1.5708
//////////end of circuit//////////////
//////////end of circuit//////////////
This circuit has impedance: (23,-10)
```

Figure 3: Code output when the code created the circuit in Figure 1.

4 Conclusion

Beyond the current form of the code, there are many implementations and functionalities that could be included. For the former, we would like to investigate the use of shared pointers instead of unique ones. The idea would be to create a set of components, specifically not pointers, that would act as a base for all circuits to be constructed from. In effect each circuit would then be a system of pointers to the components that were linked in a way to reflect the structure of the circuit. The most desirable function to incorporate would be one that prints diagrams of circuits. The current format works well for seeing a list of circuits with subcircuits using indentation but it would be desirable for a user to be able to actually see the circuit they created. Furthermore, the code could include the option to add non-ideal components, like transistors or diodes. The template structure outlined above might lend itself well to creating these, although the non-ideal nature of these components might mean one has to create a separate class.

References

[1] P. Cunane. The use of complex numbers in ac theory: Supplementary notes. *PHYS10180C University of Manchester*, 2017.