

A.C. Circuits Program

Joe Tarnoky

10308742

School of Physics and Astronomy
The University of Manchester

May 2021

Abstract

A program allowing a user to create A.C circuits is created using an object orientated approach in C++. The program allows users to build a library of components and circuits, which can subsequently be connected in series or parallel to form further circuits. This recursive definition of circuits allows circuits of any size or structure to be formed, including nested circuits. The circuit impedance and phase are calculated and outputted, along with a visual representation of the circuit. Upon testing, the code is found to be fully functional, although possible expansions of functionality are also discussed.

1. Introduction

1.1 Theory

An A.C circuit involves the application of an oscillating voltage resulting in an alternating current which periodically switches directions [1]. The main advantage this provides over a D.C circuit is that it is far easier to transform A.C voltages from high to low voltages or vice versa. This allows us to transport current at high voltages over large distances, increasing efficiency.

The impedance of an A.C circuit is equivalent to the resistance in a D.C circuit, but in complex form to account for the phase resulting from an oscillating power supply. The modified Ohm's law is given by

$$V = IZ, \quad (1)$$

where Z is the complex impedance of the circuit, V is the voltage and I is the current. The applied voltage varies as a sinusoidal function of time, meaning the same can be said for the current and impedance. We can therefore express these parameters as complex exponentials:

$$V = |V|e^{i(\omega t + \phi_V)} \quad (2)$$

$$I = |I|e^{i(\omega t + \phi_I)} \quad (3)$$

$$Z = |Z|e^{i(\omega t + \phi)} \quad (4)$$

ϕ_V and ϕ_I are the respective phases of the voltage and current and ω is the angular frequency of oscillations [2]. ϕ is the phase difference between the voltage and current. Z can also be expressed as

$$Z = R + iX, \quad (5)$$

where R is the resistance, and X is the reactance. The phase is given:

$$\phi = \arctan\left(\frac{X}{R}\right) \quad (6)$$

The capacitive and inductive reactances are given respectively by:

$$X_C = \frac{-i}{\omega C} \quad (7)$$

$$X_L = i\omega L, \quad (8)$$

where C is the capacitance in Farads and L is the inductance in Henrys. The phase shifts of a capacitor and an inductor are $\pi/2$ and $-\pi/2$ radians respectively. The impedances of circuits connected in series and parallel are given by:

$$Z_{series} = \sum_{i=1}^n Z_i \quad (9)$$

$$Z_{parallel} = \sum_{i=1}^n \frac{1}{Z_i} \quad (10)$$

1.2 Outline of project

The aim of the project is to produce a program that allows a user to create a library of components which can be used to construct circuits. The user can create resistor, capacitor and inductor components, inputting the relevant values which are stored. Circuits are created by adding two components in series or parallel. The circuits created with these components are also stored in the component library and can subsequently be used to form larger circuits. This recursive circuit construction method allows the creation of all combinations of series and parallel circuits. This includes nested circuits, in which a branch in a parallel circuit contains components added in series. The impedance, impedance modulus and phase are all stored, as is a visual representation of the circuit. The list of all components and circuits can be outputted, along with all relevant information.

2 Code design and implementation

2.1 File structure

The code is split into two source files and a header file to improve user readability. The header file called classes.h contains component and circuit class declarations accompanied by the member function declarations. It also contains all used headers, along with the declaration of the complex template class. The source file classes.cpp contains the implementation of all the class member functions. The source file named interface.cpp contains all functions which relate to the user interface, along with all input validation functions.

2.1 Class hierarchy

A simple class structure is used to exploit the features of object orientated programming and is displayed in figure 1. An abstract base class for components contains pure virtual functions to be inherited by

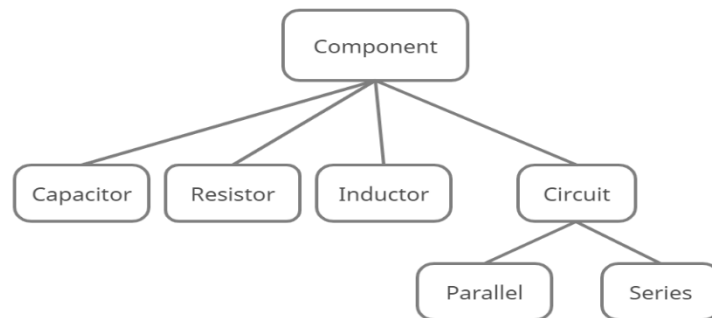


Figure 1: Class hierarchy of code

resistor, capacitor and inductor classes. The impedance is the key member variable of the component class and is stored as a complex number, making use of the standard complex number library. Member functions include an info function to print out relevant information, functions to set and get frequency, and functions to get impedance, phase and the impedance modulus. A draw component function is also included, which outputs a string type illustration of the component. This is later used to draw the full circuit diagram. Component classes take an argument of type double which is used to give either the resistance, capacitance or inductance. Eq. (7) and Eq. (8) are used to calculate the capacitive and inductive reactances.

This use of polymorphism significantly simplifies the code, and it is further exploited in the circuit class. The circuit class is derived from the component class, as each circuit created is treated as a possible component of any subsequent circuit the user may create. Series and parallel circuit classes are derived from the circuit class. The main circuit class contains a complex member variable named circuit impedance, which is inherited by the two derived classes. This variable is then calculated for the respective circuit type using Eq. (9) and Eq. (10).

Series and parallel circuit constructors both take arguments of two component base class pointers, as shown in figure 2.

```
class parallel_circuit : public circuit
{
private:
    component* component1;
    component* component2;
public:
    parallel_circuit(component* c1, component* c2) : component1{c1}, component2{c2}{};
```

Figure 2: constructor of parallel circuit class

This allows member functions within the classes access to the associated quantities of each component needed to calculate the properties of the circuit. Since series and parallel circuits are derived from the component class, circuits can also be taken as arguments, allowing for recursive circuit construction. It is also essential in allowing series and parallel classes to draw component functions, as it allows the draw component function of each circuit to output the two connected components. Components combined in series are illustrated as -component1-component2-, whilst components connected in parallel are represented using the notation P{component1, component2}.

The main circuit class contains a draw circuit function which outputs the circuit connected to a power supply. To account for the varying length of strings outputted by the draw component functions, the * operator is overloaded to allow the output of a character to be repeated a selected number of times. This overload allows the correct number of wires and spaces to be outputted to ensure all wires are correctly positioned. The implementation of this overload in the draw circuit function is shown in figure 3.

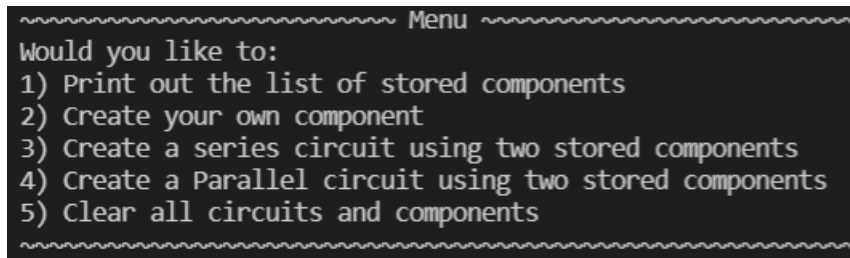
```
std::string circuit::draw_circuit()
{
    std::string wire("-");
    std::string space(" ");
    int length = draw_component().length() - 1;
    std::string circuit = "V" + (wire * length) + "\n" + (space * (length - 1)) + "\n" + draw_component();
    return circuit;
}
```

Figure 3: The draw circuit function used to output circuit illustrations.

As shown above, the final circuit illustration is a single string stream using the “-” character as wires to connect the component to a power supply represented by “V”. The correct number of wires and spaces are found by finding the length of the draw component function output.

2.2 User interface

Upon running the program, the user is presented with five options from a menu, displayed in figure 4.



```
~~~~~ Menu ~~~~~  
Would you like to:  
1) Print out the list of stored components  
2) Create your own component  
3) Create a series circuit using two stored components  
4) Create a Parallel circuit using two stored components  
5) Clear all circuits and components  
~~~~~
```

Figure 4: The program menu displaying all options available to the user

Option 1 prints out the information of all stored components by iterating through a vector of base class pointers. This vector is defined above all user interface functions and takes objects of type `component*`. The vector starts with one resistor, capacitor and inductor, all with predefined parameters. The user is given the choice to order components in terms of when they were created (first to last), or in order of impedance modulus (largest to smallest). This is done using a lambda function to compare the impedance moduli of each component, as shown in figure 5.

```
std::sort(std::begin(component_list), std::end(component_list), [&](component* a, component* b)  
{return a->get_impedance_modulus() > b->get_impedance_modulus();});  
print_list(component_list);
```

Figure 5: Lambda function used to order list

The use of the lambda function improves the readability of the code in two ways: it reorders the list in a concise manner whilst also being conveniently located within the code. As lambdas are anonymous functions, it can be defined precisely where it is used.

The user has the option to use these components to create a circuit, or the option to create additional components to be used in circuit construction. Option 2 asks for user input to choose which component to add, and to give values for resistance, capacitance or inductance. The user is also asked for a frequency input, which can later be adjusted in circuit construction.

Options 3 and 4 print out the full list of stored components. This list is indexed by making use of a `get_index` function which returns the length of the list after a component is added to it. The user is asked to input the indexes of the components they wish to create a circuit with. They are then asked to set the frequency of the circuit, which sets the frequency of all components to the inputted value. The circuit is then outputted, with all calculated values accompanied by the circuit diagram. Option 5 allows the user to clear the component list and start again without having to rerun the program, an option which becomes useful if the list gets too cluttered.

After an action has been completed, the user is offered the option to return to the main program or to quit. This is done by setting a character variable to “y” and instructing the user to type “y” to return to the menu, or any other key to quit. By returning to the main menu, the user can construct progressively larger circuits one step at a time. If the user quits the program, a `clear_list` function iterates through the vector of components and deletes the contents to avoid possible memory leaks.

2.3 Validation

All user inputs are heavily restricted to avoid any crashes. This is fully achieved using two validation functions: one to validate the user interface options and one to validate values such as frequency. Each function takes a string type argument which is the prompt given to the user, e.g., “Choose the index of your first component”. The option validation function takes a second argument of integer type. This integer dictates the number of available options the user has and is used to define the range of accepted

inputs. The use of the second argument generalises the function, allowing it to be used multiple times throughout the user interface. The validate value function permits only positive doubles.

2.4 Error handling

Most potential for divide by zero errors is negated in the validation. However, if two components are being added in parallel and they have equal and opposite impedances, then the denominator of Eq. (10) will be zero. In preparation for this unlikely occurrence, try, throw and catch statements are implemented to prevent the program from breaking.

3 Results

3.1 Run through

All features of the program were tested in a run which used a combination of default and created components to create a circuit which combined components both in series and parallel. The list of components following the run is shown in figures 6 and 7.

```
Here are your components:
~~~~~
Component 1
Resistor:
Resistance = 330  $\Omega$ 
Phase = 0 rad
Frequency = 50 Hz
-R-
~~~~~
Component 2
Capacitor:
Capacitance = 1e-06 F
Capacitive reactance = (0,-3.18e+03)  $\Omega$ 
Phase = 1.57 rad
Frequency = 50 Hz
-C-
~~~~~
Component 3
Inductor:
Inductance = 0.5 H
Inductive reactance = (0,157)  $\Omega$ 
Phase = -1.57 rad
Frequency = 50 Hz
-L-
```

Figure 6: list of default components

```
~~~~~
Component 4
Capacitor:
Capacitance = 1e-05 F
Capacitive reactance = (0,-318)  $\Omega$ 
Phase = 1.57 rad
Frequency = 50 Hz
-C-
~~~~~
Component 5
Series circuit:
Impedance of circuit is (330,-318)  $\Omega$ 
Impedance modulus is 458  $\Omega$ 
Phase: -0.767 rad
-C---R-
~~~~~
Component 6
Parallel circuit:
Impedance of circuit is (60.4,187)  $\Omega$ 
Impedance modulus is 196  $\Omega$ 
Phase: 1.26 rad
P{-C---R-, -L-}
```

Figure 7: list of components and circuits created in run through

The values outputted were checked using Eqs. (6-10) and are all found to be correct. Figures 6 and 7 show what is displayed when choosing to print out the component list. This option presents a circuit as a component, as shown in figure 7. A full circuit diagram is displayed when the circuit is created. An illustration of this feature is shown in figure 8, in which components 5 and 6 are connected in series. Figure 9 displays an example of a larger circuit, displaying how recursive circuit definitions allow for more complex circuit construction.

```

Here is your new series circuit
Series circuit:
Impedance of circuit is (390,-132)  $\Omega$ 
Impedance modulus is 412  $\Omega$ 
Phase: -0.325 rad
V-----
|
P{-C---R-, -L-}--C---R-

```

Figure 8: Components 5 and 6 connected in series

```

Here is your new series circuit
Series circuit:
Impedance of circuit is (502,45.9)  $\Omega$ 
Impedance modulus is 504  $\Omega$ 
Phase: 0.0912 rad
V-----
|
P{P{-L---R-, -C-}, -L---R-}-P{-C-, -R-}

```

Figure 9: Two parallel components connected in series

3.2 Validation testing

All user inputs were thoroughly tested to ensure only appropriate responses could be given. An example of the value validation is shown in figure 10.

```

What frequency (in Hz) would you like your circuit to operate at?
r3
Please enter a positive number
What frequency (in Hz) would you like your circuit to operate at?
-7
Please enter a positive number
What frequency (in Hz) would you like your circuit to operate at?
0.9y
Please enter a positive number

```

Figure 10: Value validation function ensuring appropriate frequency inputs

Testing included blank spaces, special characters and unavailable option numbers/indexes. Testing found no instances of invalid inputs getting past validation.

4 Discussion and conclusion

The results above demonstrate the functionality of the code, allowing the user to create series-parallel circuits of any size. The program is user friendly and fully protected against faulty inputs. The use of polymorphism throughout brings clarity to the code, demonstrating the advantages of an object orientated approach.

Whilst the program is fully functional, there are many extensions which could be implemented. External libraries such as matplotlib-C++ could be used to output phasor diagrams, giving a graphical representation of the circuit impedance. Functions to create different components such as non-ideal components or diodes could be implemented. However, this would require significant adjustments and additions to the circuit calculation functions. It could also be useful to enable users to edit specific component in a circuit and output the new circuit, with all values recalculated. The circuit diagram contains all the necessary information, and at smaller sizes is neat and compact. However, at larger circuit sizes involving multiple parallel components the parallel notation can become unclear, as seen in figure 6. To address this, a way to output a full illustration could be devised to replace the shorthand notation.

5 References

- [1] T. R. Kuphaldt, "Lessons In Electric Circuits, Volume II – AC", 2007, pp. 1-6
- [2] R. A. Freedman, H. D. Young, "University Physics with Modern Physics", 2016, pp. 1054-1057