# OOP in C++: Modelling and Calculating the Impedance of AC Circuits

*Adam Sitponya*
*10672159*

School of Physics and Astronomy
University of Manchester
May 2023

## Abstract

A C++ program was created to build circuits made of resistors, capacitors and inductors connected arbitrarily and calculate the overall impedance and phase shift. Class hierarchies were defined both for a component and circuit class with derived classes to represent specific components and series or parallel circuits. Class member variables were defined to represent physical quantities such as frequency, impedance and phase shift. A simple user interface (UI) was implemented in the terminal allowing a user to create and store component and circuit objects. The UI also allowed the user to access their library of stored circuits and print out its' associated physical quantities and a simple visual representation using ASCII characters.

## 1 Introduction

### 1.1 A Brief Summary of Circuit Theory

Alternating current (AC) is an electric current flow that periodically varies direction and magnitude. In AC circuits, current ($I$) and voltage ($V$) vary periodically following sinusoidal waveforms. Conventionally, $V$ and $I$ are represented in complex exponential form e.g. $V = V_0 e^{i(\omega t+\phi)}$ where $V_0$ is the amplitude, $\omega$ is the signal frequency and $\phi$ is some phase shift. Complex exponential form is used as it provides convenient expressions for magnitude and phase shift. It also makes it easier to analyse complex circuits as separate voltage and current signals can be added using phasor algebra.

In complex exponential form, $I$ and $V$ are related by:

$$V = ZI \tag{1}$$

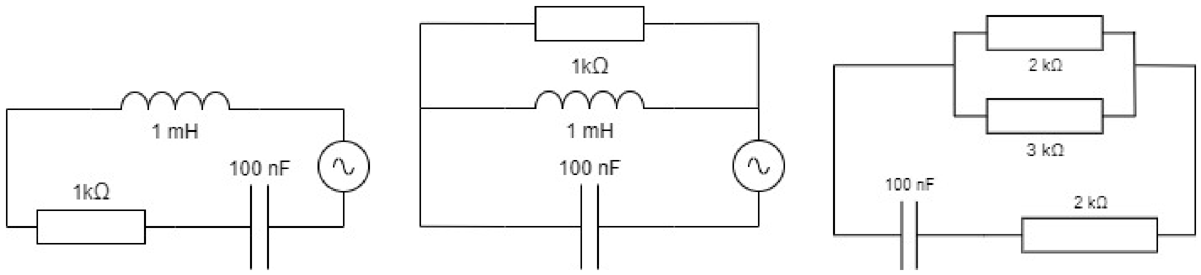
where $Z$ is the impedance measured in Ohms ($\Omega$). Impedance is a complex object which represents the opposition to the flow of current caused by circuit components. $Z = R + iX$ where R is resistance and $iX$ is the reactance. Reactance is the imaginary component of $Z$ which typically encodes opposition to current flow in AC circuits due to capacitance ($C$) and inductance ($L$). $V$ and $I$ are waveforms and as such there is a phase shift between them which is equal to the argument of the impedance, $\Phi = \arctan(R/X)$ [1].

In this project, we consider 3 types of components: resistors, capacitors and inductors. Resistors are passive components which resist the flow of current. Capacitors can store and release electric charge and store energy within an electric field. Inductors store energy within a magnetic field.

Each type of component has an associated impedance:

$$Z_R = R \qquad Z_C = \frac{-i}{\omega C} \qquad Z_L = i\omega L \tag{2}$$

where $R$ is measured in Ohms, $C$ is measured in Farads (F) and $L$ is measured in Henry (H).



(a) A Simple Series Circuit (b) A Simple Parallel Circuit (c) A More Complex Circuit Structure.

Figure 1: Examples of different circuit structures using resistors, capacitors and inductors. a) and b) are simple series and parallel arrangements. c) shows a more complex arrangement which can be thought of as a series circuit containing a nested parallel circuit

Circuits are composed of electrical components that can be connected in series or parallel. For circuits, the overall impedance is determine by combining the impedances of the individual components. The method for combining depends on how the components are connected. For components combined in series and parallel, the overall impedance is determined by:

$$Z_s = \sum_i Z_i \qquad \frac{1}{Z_p} = \sum_i \frac{1}{Z_i} \tag{3}$$

where $Z_i$ is the impedance of the $i^{th}$ component [1]. $Z_s$ and $Z_p$ are the impedance of a series and parallel circuit respectively. Fig. 1 shows examples of simple and complex circuit structures. This project aims to correctly represent complex circuit structures and accurately calculate the overall impedance.

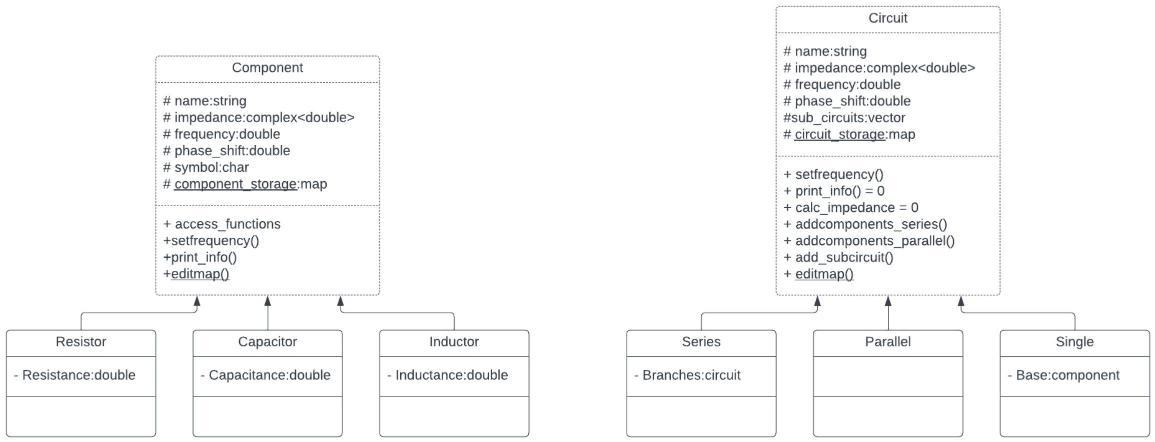
### 1.2 Project Outline

The purpose of this project was to develop a C++ program thats allows users to create and store component objects and use them to build arbitrary circuits. We focused on resistors, capacitors and inductors and implementing the equations discussed in section 1.1. Class hierarchies were defined for component and circuit objects and they were stored in maps that were declared as static data members of the corresponding class. A simple terminal-based user interface was designed to allow a user to construct and view circuits with a simple representation being printed using ASCII characters.

## 2 Code Design

### 2.1 Class & File Structure

2 class hierarchies were defined in this program, the 1st for component objects and the 2nd for circuit objects. The hierarchies are illustrated in Fig 2.



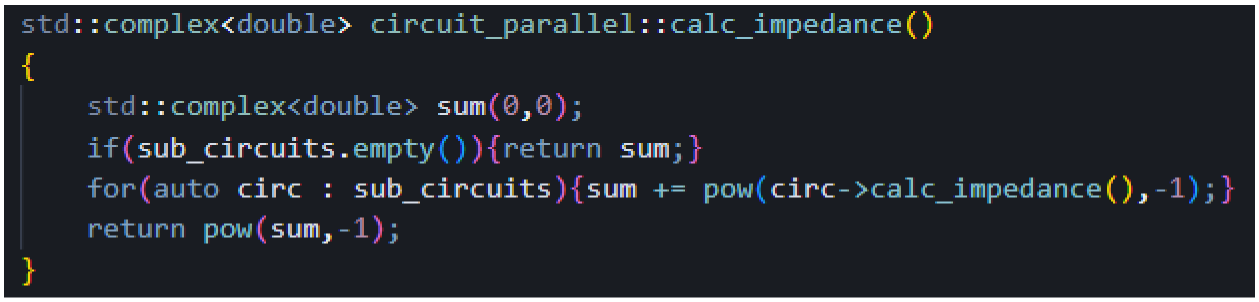
(a) Component class hierarchy

(b) Circuit class hierarchy

Figure 2: UML diagrams for the 2 main class hierarchies defined in this program. Some access functions and map editing functions are not listed.

The component class is an abstract class with data members that are linked to the physical quantities such as impedance and frequency discussed in section 1.1. The circuit class is also an abstract class containing similar members. The data members symbol and name are declared in the base classes and assigned values in the derived classes to be used when printing out information about the specific components/circuits. The base component class contains a member function to set the impedance and each derived class defines this function according to the formulae in Eqn. 2. Each derived component class includes a characteristic value $(R/C/L)$.

For circuits, 3 derived classes are defined:series, parallel and single. The single class has a data member base which is a component object. The base circuit class defines a vector labelled sub-circuits which stores pointers to other circuit objects;only the series and parallel derived classes use this vector. The series and parallel circuits calculate their impedance by applying the formulae in Eqn, 3 whilst the single class takes the impedance of the base component object. The formulae in Eqn. 3 can then be calculated recursively by iterating over the sub-circuits vector.



```
std::complex<double> circuit_parallel::calc_impedance()
{
    std::complex<double> sum(0,0);
    if(sub_circuits.empty()){return sum;}
    for(auto circ : sub_circuits){sum += pow(circ->calc_impedance(),-1);}
    return pow(sum,-1);
}
```

Figure 3: Redefined virtual function in the parallel circuit class which calculate the impedance of the circuit.

Fig. 3 demonstrates how we calculate the impedance of a parallel circuit. The code iterates over the sub-circuit vector updating the sum variable in each pass. This function works recursively by calling itself for each circuit object in the sub-circuit vector. The recursive loop ends for single circuit objects which simply return the impedance of a single component. There is a similar implementation for the series circuit class. The series circuit class has an extra member called branches. This was included to allow a user to add parallel branches to a base series circuit.

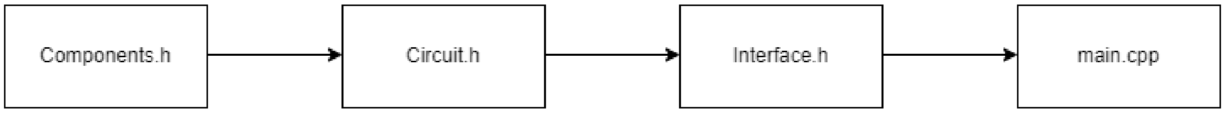
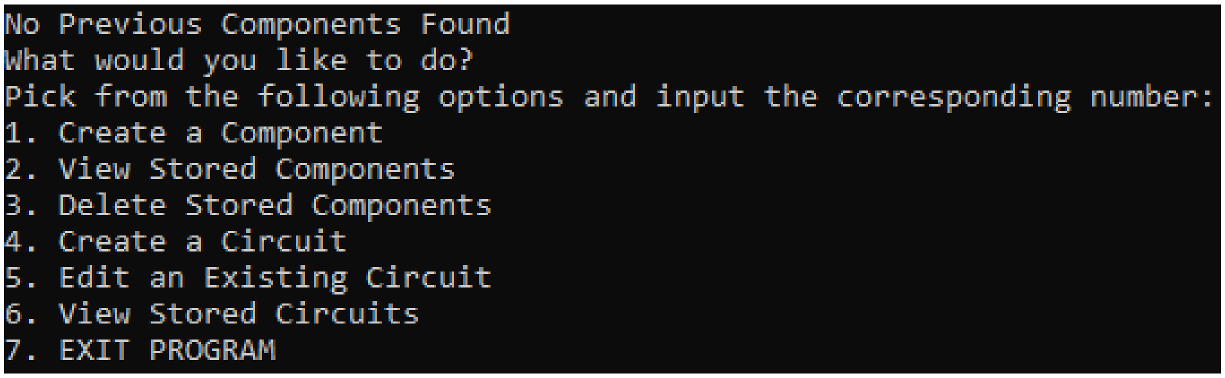


Figure 4: Header file chain illustrated.

The circuit and class hierarchy are contained in separate header files. Another header file is also defined to contain function declarations relating to the user interface. Each header file has an associated .cpp file where the functions are defined. The header files are included in a chain. The interface header file includes the circuit header which in turn contains the components header. The interface header is the only one included in the main program which prevents duplication (header guards are also included). This is demonstrated in Fig. 4.

## 2.2 User Interface

When the application launches the user is presented with a simple menu in a terminal window as shown in Fig. 5. The create component and create circuit options take user inputs to call a class constructor. A pointer is then saved to a map which is a static member of the component/circuit class. Stored components and circuits can be viewed by calling static member functions of the corresponding class.

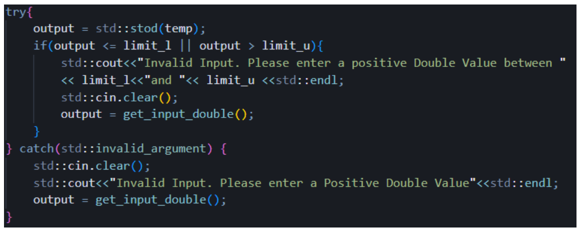


```
No Previous Components Found
What would you like to do?
Pick from the following options and input the corresponding number:
1. Create a Component
2. View Stored Components
3. Delete Stored Components
4. Create a Circuit
5. Edit an Existing Circuit
6. View Stored Circuits
7. EXIT PROGRAM
```

Figure 5: The main menu of the program. The program always prints out this menu after completing any action.

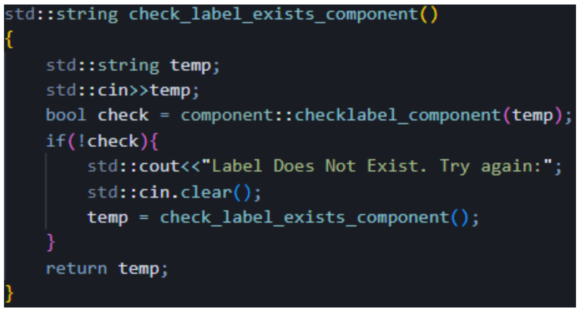
## 2.3 Input Validation & Exception Handling

The user interface regularly relies on user input so appropriate checks are put in place to ensure user inputs are valid. This is essential to prevent unrealistic values of physical quantities and segmentation faults if the user tries to access some object which does not exist. Users often have to input positive double values so a dedicated function was defined for reading in such values.
As Fig. 6a) shows, a try catch block is used. User inputs are read into a string, which the program attempts to cast to a double using ”std::stod()”. This can cause an exception to occur



```cpp
try{
    output = std::stod(temp);
    if(output <= limit_l || output > limit_u){
        std::cout<<"Invalid Input. Please enter a positive Double Value between "
        << limit_l<<"and "<< limit_u <<std::endl;
        std::cin.clear();
        output = get_input_double();
    }
} catch(std::invalid_argument) {
    std::cin.clear();
    std::cout<<"Invalid Input. Please enter a Positive Double Value"<<std::endl;
    output = get_input_double();
}
```

(a) Getting double input function.



```cpp
std::string check_label_exists_component()
{
    std::string temp;
    std::cin>>temp;
    bool check = component::checklabel_component(temp);
    if(!check){
        std::cout<<"Label Does Not Exist. Try again:";
        std::cin.clear();
        temp = check_label_exists_component();
    }
    return temp;
}
```

(b) Checking labels function.

Figure 6: Functions defined for checking user inputs to prevent incorrect physical quantities or memory faults.

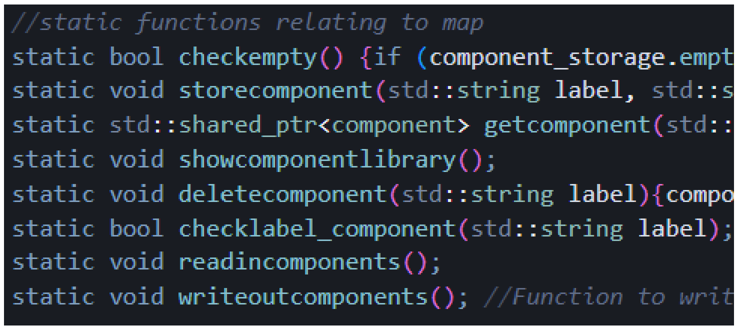
which leads to an error message being printed and the function being re-called. The function also has allowed limits for the double value. Since the inputs are meant to be physical quantities, there are reasonable limits on the values they can take. For example, frequency and resistance must be greater than 0. In general, we find that real capacitors have capacitance up to the range of $\mu$F and inductors have inductance up to the range of mH so an upper limit argument is included to ensure inputted values are realistic.

Throughout the program, the user inputs labels to retrieve stored objects from a map. Maps are associative containers that store key-value pairs. Each key is unique and leads to a particular value. It is important that the program only attempts to access keys that are already part of the map as this could otherwise lead to segmentation faults. It is also important that the user does not try to assign a value to the same key twice as the originally stored value will be overwritten on the second pass. To prevent this, check functions as shown in Fig.6 b) are used to retrieve a valid label input. A static member function of the component class returns a bool which returns true if the map already contains a specific key. In Fig.6b), the program wants to refer to an existing component, so the function goes on to ask for an existing key if the user input is not already a valid key. A similar function was defined for the case when the program wants the user to input a new key.

## 2.4 Memory Management & Storing Objects

Smart pointers are implemented throughout this program to manage memory allocation for the stored components and circuits. User created objects are instantiated and stored as shared pointers. This removes the need to call the delete operator when an object needs to be freed from memory. Shared pointers also keep a reference counter which keeps track of the number of pointers pointing to an object at any time. This prevents objects being prematurely deleted in one part of the code despite being needed in some other part. This is useful as the same stored component can be copied into multiple circuits in this program.

The program allows the user to store components and circuits and refer back to them in other functions. Component objects are stored by including a static map object in labelled component storage in the component class. Static members are only declared once and are shared for all instances of the class. A map was chosen to store objects as the keys can be defined as strings and the user can easily refer to components. The static map was included as a protected member of the component class as this prevents it from being directly modified. To allow addition and viewing of stored components, static member functions were included as part of the class. These are seen in Fig. 7.



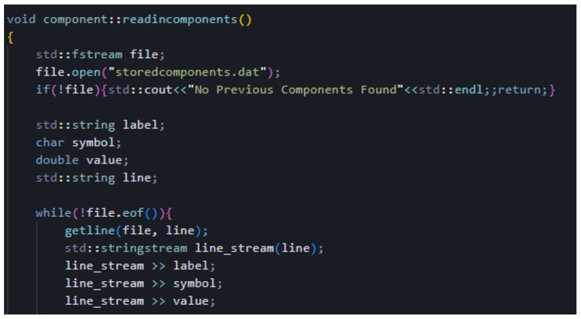
```
//static functions relating to map
static bool checkempty() {if (component_storage.empt
static void storecomponent(std::string label, std::s
static std::shared_ptr<component> getcomponent(std::
static void showcomponentlibrary();
static void deletecomponent(std::string label){compo
static bool checklabel_component(std::string label);
static void readincomponents();
static void writeoutcomponents(); //Function to writ
```

Figure 7: Static member functions that interact with the component storage map and allow user to add and read from it.

Static member functions are associated with the class itself rather than with a specific object and can be called without an acting on an object by using the scope resolution operator "::". Static member functions can interact with static data members so these functions can interact with the map and are called during the program when the user attempts to either create new components or access stored components. A similar static map along with analogous static member functions is included in the circuits class.

## 2.5 Saving Beyond Run-time

As mentioned previously, the user can create and store component objects while running the program. As an additional feature, the program can save the component information to a .dat file before exiting. Static functions are used for this and are shown in Fig. 8.

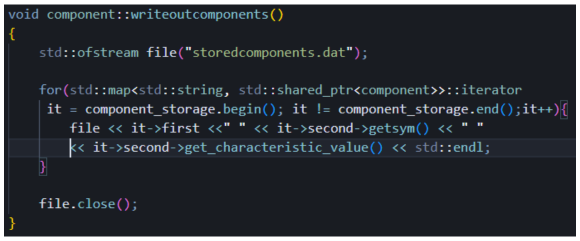


```
void component::readincomponents()
{
    std::fstream file;
    file.open("storedcomponents.dat");
    if(!file){std::cout<<"No Previous Components Found"<<std::endl;;return;}

    std::string label;
    char symbol;
    double value;
    std::string line;

    while(!file.eof()){
        getline(file, line);
        std::stringstream line_stream(line);
        line_stream >> label;
        line_stream >> symbol;
        line_stream >> value;
```

(a) Reading in components function.



```
void component::writeoutcomponents()
{
    std::ofstream file("storedcomponents.dat");

    for(std::map<std::string, std::shared_ptr<component>>::iterator
     it = component_storage.begin(); it != component_storage.end();it++){
       file << it->first <<" " << it->second->getsym() << " "
       << it->second->get_characteristic_value() << std::endl;
    }

    file.close();
}
```

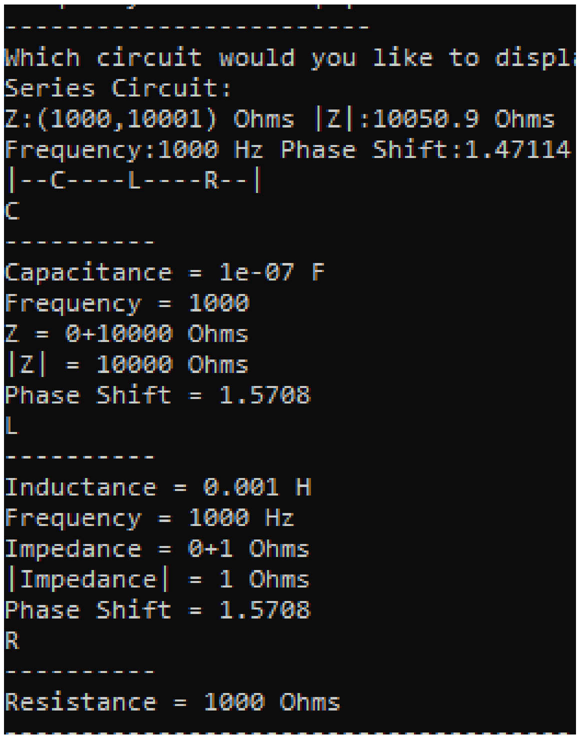
(b) Writing out components function

Figure 8: Functions defined for reading in components from a .dat file and writing out to the file at the end of the program.

The program reads from and writes to a file named "storedcomponents.dat" using a filestream object. The reading in function (Fig. 8a)) first checks if the storedcomponents file exists as a protection in case the file was corrupted or deleted. The file is then read in line by line into a stringstream which splits up the line into the important information. From this, appropriate constructors and the map is updated. Before the program exits, the writing out function (Fig 8b)) is called which iterates over the storage map. For each component, the label, symbol(either R,C or L) and the characteristic value is written to the .dat file with spaces acting as delimiters.

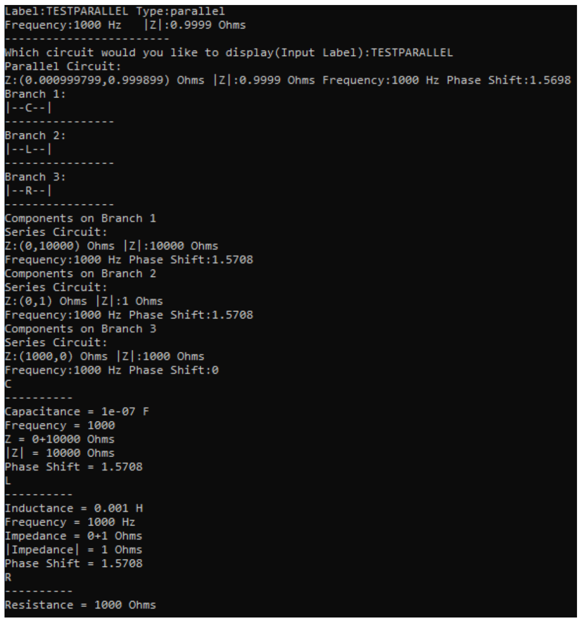
## 3 Results

The main objective of this program is to calculate the impedance of circuits. This was tested by constructing the circuits found in Fig. 1 with the corresponding values and checking the program output against the theoretical result. The program output for both the simple series and simple parallel circuits are shown in Fig. 9.



```
------------------------
Which circuit would you like to displa
Series Circuit:
Z:(1000,10001) Ohms |Z|:10050.9 Ohms
Frequency:1000 Hz Phase Shift:1.47114
|--C----L----R--|
C
----------
Capacitance = 1e-07 F
Frequency = 1000
Z = 0+10000 Ohms
|Z| = 10000 Ohms
Phase Shift = 1.5708
L
----------
Inductance = 0.001 H
Frequency = 1000 Hz
Impedance = 0+1 Ohms
|Impedance| = 1 Ohms
Phase Shift = 1.5708
R
----------
Resistance = 1000 Ohms
```

(a) program output for the circuit defined in Fig. 1a).



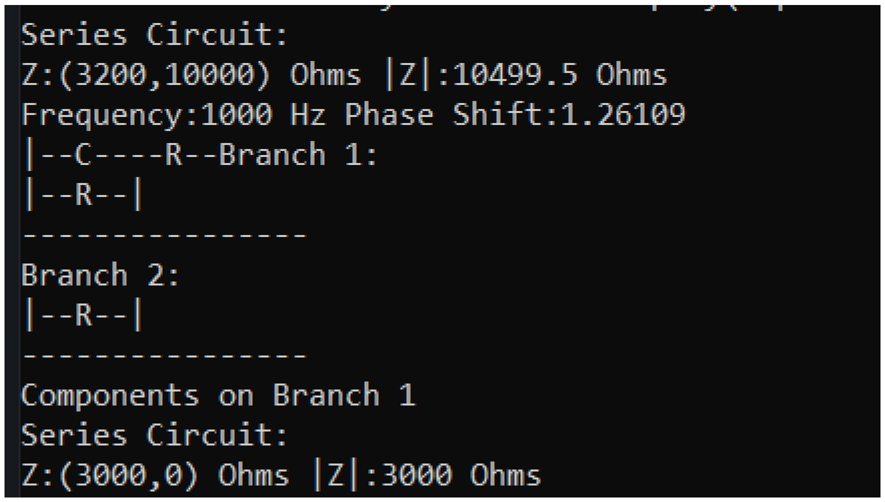
```
Label:TESTPARALLEL Type:parallel
Frequency:1000 Hz   |Z|:0.9999 Ohms
------------------------
Which circuit would you like to display(Input Label):TESTPARALLEL
Parallel Circuit:
Z:(0.000999799,0.999899) Ohms |Z|:0.9999 Ohms Frequency:1000 Hz Phase Shift:1.5698
Branch 1:
|--C--|
----------------
Branch 2:
|--L--|
----------------
Branch 3:
|--R--|
----------------
Components on Branch 1
Series Circuit:
Z:(0,10000) Ohms |Z|:10000 Ohms
Frequency:1000 Hz Phase Shift:1.5708
Components on Branch 2
Series Circuit:
Z:(0,1) Ohms |Z|:1 Ohms
Frequency:1000 Hz Phase Shift:1.5708
Components on Branch 3
Series Circuit:
Z:(1000,0) Ohms |Z|:1000 Ohms
Frequency:1000 Hz Phase Shift:0
C
----------
Capacitance = 1e-07 F
Frequency = 1000
Z = 0+10000 Ohms
|Z| = 10000 Ohms
Phase Shift = 1.5708
L
----------
Inductance = 0.001 H
Frequency = 1000 Hz
Impedance = 0+1 Ohms
|Impedance| = 1 Ohms
Phase Shift = 1.5708
R
----------
Resistance = 1000 Ohms
```

(b) program output for the circuit defined in Fig. 1b).

Figure 9: Terminal output for simple series and parallel circuits.

Fig. 1a) shows a simple series circuit of a resistor, capacitor and inductor. By applying Eqn. 2, for a signal frequency of 1 kHz, we expect $|Z| = 10050\ \Omega$ and $\Phi = 1.47$. As Fig. 9a) shows, the program successfully calculates this value, and also displays a simple diagram using ASCII characters. The properties of the individual components are also printed out. For a parallel circuit as shown in Fig. 1b), we expect $|Z| = 1\ \Omega$ and $\Phi = 1.56$. The program does calculate this, although floating point error may have affected the outputted value for $|Z|$.

A more complex nested circuit structure was the next to be tested. We consider a capacitor and resistor in series, followed by two resistors in parallel as shown in Fig. 1c). This is a slightly more complicated calculation as the 2 parallel branches combined impedance must be calculated using the parallel impedance formula which then is combined with the other components using the series impedance formula. The expected values are $|Z| = 10500\ \Omega$ and $\Phi = 1.26$. The circuit was created in the program by first making a parallel circuit object with the 2 resistor branches. This was then added to a series circuit object containing the capacitor and 3rd resistor. The output is shown in Fig 10. The output matches our expectations and this proves the programs' ability to accurately represent more complex circuit structures.



```
Series Circuit:
Z:(3200,10000) Ohms |Z|:10499.5 Ohms
Frequency:1000 Hz Phase Shift:1.26109
|--C----R--Branch 1:
|--R--|
----------------
Branch 2:
|--R--|
----------------
Components on Branch 1
Series Circuit:
Z:(3000,0) Ohms |Z|:3000 Ohms
```

Figure 10: Terminal output for the nested circuit shown in Fig. 1c).

## 4 Discussion & Conclusion

We have successfully proven the program's ability to accurately build and analyse simple and more complex circuit structures. Defining impedance calculation functions recursively allowed for generality in a way that any arbitrary structure that can be thought of as a mix of series and parallel sub-structures can be inputted by a user. A potential extension could be optimising the impedance calculations by adjusting the way circuits are represented. One possible way is to represent the circuits as a graph where the nodes are components and the edges represent the connections. Graph algorithms could then be implemented to move through the circuits and find the overall impedance [2].

A more visually appealing way to display the circuits could be devised. This would make more complex circuit structures easier for the user to follow. External libraries could potentially be used for this. Elsewhere, the program currently is able to save stored components beyond run-time but does not save the stored circuits. Information about the stored components can easily be written to any simple file format however this is less straightforward for circuits, especially if the structure involved multiple branches and nested components.

**Latex Word Count: 2323**

## References

[1] P. Horowitz and W. Hill, *The Art of Electronics*. USA: Cambridge University Press, 3rd ed., 2015.

[2] L. Toscano, S. Stella, and E. Milotti, "Using graph theory for automated electric circuit solving," *European Journal of Physics*, vol. 36, p. 035015, mar 2015.