Programming a Circuit Simulator with Modified Nodal Analysis

11120960

School of Physics and Astronomy

University of Manchester

Introduction to Object-Oriented Programming in C++ Project Report

May 2025

Abstract

We explore the design of an AC circuit simulator implemented in C++ using Modified Nodal Analysis (MNA). The simulator supports arbitrary closed circuits consisting solely of resistors, inductors, and capacitors connected to a single AC voltage source. Users can construct custom circuits by instantiating the default `Circuit` class and adding components via their nodal locations. We begin by outlining the theory and design of the simulator. We conduct demonstrations for several example circuits, including an RLC series circuit, a Wien bridge, and a 3D cuboid resistor network. All source files required for simulation are accessible by including the `MNA.h` header file found within the `MNA` folder.

Word count: 2454 (Overleaf)

Link to repository: here

1 Introduction

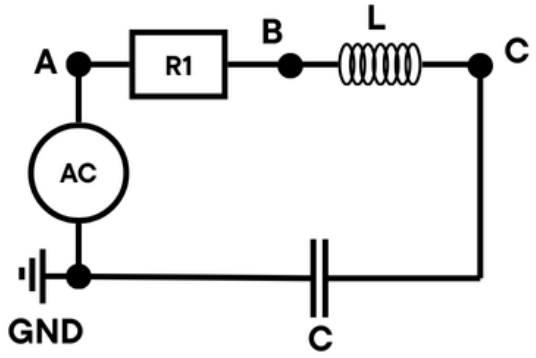
As a final coursework assignment for PHYS30762 at the University of Manchester, students are tasked with one of the following projects to design in C++: A star catalogue (a class hierarchy for storing observational data for astronomical objects), a particle detector (to simulate detecting particles as in the ATLAS or CMS experiment), or an analogue AC circuit simulator. This report explores the latter task.
As a starting point, we could consider circuits that only contain components in series or parallel (known as series-parallel circuits). A standard way to do this would be to create a `Circuit` class that contains a vector of component objects. Each component has a predefined impedance, so when adding this component in series we can just add to the total impedance. At any time the user wants to branch into parallel components, they could create a 'parallel' component object that has an effective impedance depending on the components in the parallel branch. This design would support all series parallel circuits.
A limitation of this setup is there are many interesting circuits worth investigating that cannot be constructed using only series and parallel combinations, as seen in Figure 1(b). In this report, we will explore a more powerful technique used to simulate circuits known as *Modified Nodal Analysis* (MNA), which overcomes this limitation. This method is used by SPICE (Simulation Program with Integrated Circuit Emphasis) – the analogue circuit simulator that is predominantly used in industry, and recognised as an IEEE milestone in 2011 [1].
We will begin by introducing the theory behind MNA, then outlining the design of the project, and finally we will demonstrate the usage of the programme through some interesting examples. All of the source files for the project are located inside the `MNA` folder.

2 Modified Nodal Analysis

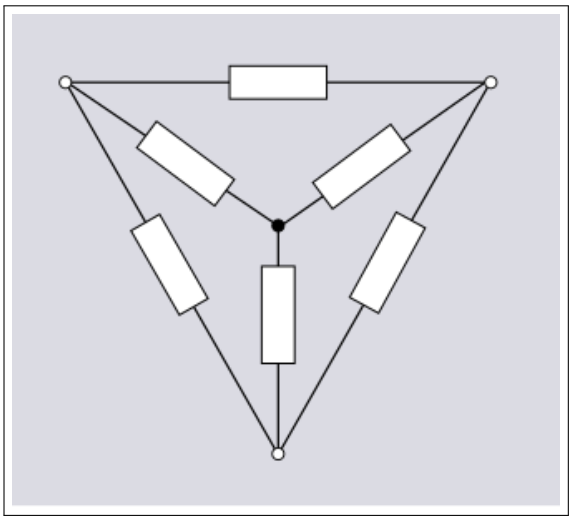
A node is defined as a point of connection between two or more circuit elements [2]. This is perhaps easiest explained by an example. Consider the RLC circuit in Figure 1(a). Node A separates the AC source and the resistor, B separates the resistor and the inductor, and so on. In figure 1(b) for example, the nodes are on the vertices of the triangle and in the centre. Each node has a potential associated with it, and the voltage across a component between two nodes is precisely the difference between the nodal voltages. For example, the voltage across the resistor is $v_B - v_A$. The ground node is set to zero, so we obtain $v_A = V_0$ which is the source voltage. We can use the fact that the current leaving a node matches the current entering it to obtain the following set of equations

$$\begin{cases} \text{at node } A: & (v_A - v_B)Y_R + I_s = 0 \\ \text{at node } B: & (v_B - v_A)Y_R + (v_B - v_C)Y_L = 0 \\ \text{at node } C: & (v_C - v_B)Y_L + v_C Y_C = 0 \end{cases} \quad \Longrightarrow \quad \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & Y_R & -Y_R & 0 \\ 0 & -Y_R & Y_R + Y_L & -Y_L \\ 0 & 0 & -Y_L & Y_L + Y_C \end{bmatrix} \begin{bmatrix} I_s \\ v_A \\ v_B \\ v_C \end{bmatrix} = \begin{bmatrix} V_0 \\ 0 \\ 0 \\ 0, \end{bmatrix} \tag{1}$$

where Y stands for the admittance, defined as the inverse of the impedance of a component. By numerically solving this system, we obtain the nodal voltages and the source current. This highlights how MNA works, and in a more general setting, the conductance matrix $\mathbf{A}$ is obtained as follows: Let $\mathbf{A} = \mathbf{0}$ be a



(a) An RLC circuit.



(b) A non-series-parallel circuit.

Figure 1

square matrix with size equal to the number of nodes, where we start indexing the matrix elements at 0. We enumerate each of the nodes starting with 0 for the ground node. Suppose a component is between nodes i and j has admittance Y_{ij}. If $i \neq 0$, then add Y_{ij} to $\mathbf{A}_{ii}$, and similarly, if $j \neq 0$, add Y_{ij} to $\mathbf{A}_{jj}$. If both i and j are non-zero, subtract Y_{ij} from $\mathbf{A}_{ij}$ and $\mathbf{A}_{ji}$. That is, we add either

$$\begin{bmatrix} \ddots & & \\ & Y_{ij} & \\ & & \ddots \end{bmatrix} \quad \text{or} \quad \begin{bmatrix} \ddots & & & \\ & Y_{ij} & \cdots & -Y_{ij} \\ & \vdots & \ddots & \vdots \\ & -Y_{ij} & \cdots & Y_{ij} \\ & & & \ddots \end{bmatrix}. \tag{2}$$

After doing this for all components, we then consider the AC source with voltage V_0 between the ground node and node k. We simply add 1 to both A_{0k} and A_{k0}, and this completes the construction of the conductance matrix. The nodal voltages and source current are then obtained by solving

$$\mathbf{A} \begin{bmatrix} I_s \\ v_1 \\ \vdots \\ v_n \end{bmatrix} = \begin{bmatrix} V_0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}. \tag{3}$$

Thus, by locating the components of a circuit via nodes, we can "stamp" their contributions to the conductance matrix, and then the properties of the circuit can be computed. For more information on the theory behind MNA, see [3].

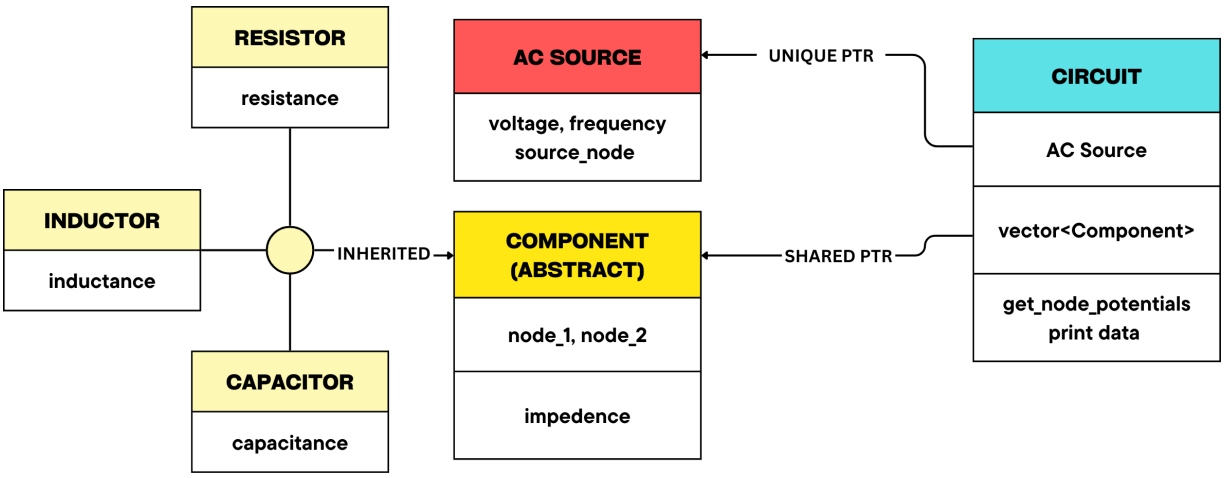


Figure 2: UML for the MNA circuit simulator.

3 MNA implementation

This section will explore the design of the MNA circuit simulator. The user can get access to the entire `MNA` folder by including the `MNA.h` header file. The `helper.h` file supports the folder with general helper functions. This includes defining $\pi = 3.14\ldots$ at compile time, and simplifying the usage of complex numbers. The line

```
1     using complex = std::complex<double>;
```

is written so that anytime the user would like to work with complex doubles, they can just write `complex`. The << operator is overloaded to nicely print phasors:

```
1  template<typename T> // templated function
2  std::ostream& operator<<(std::ostream& os,
3      const std::complex<T>& complex_nr)
4  {
5    double magnitude = std::abs(complex_nr);
6    double phase = std::arg(complex_nr); // in radians
7
8    os<<std::setprecision(4); // nr sig figs
9    os<<magnitude<<" cis "<<(phase / PI)<<" pi";
10   return os;
11 }
```

Lastly, the `hline()` and `double_hline()` functions print horizontal lines, where the `inline` keyword tells the compiler that it should substitute in the code for the function directly when it is called to reduce overhead. A UML diagram for the MNA programme is illustrated in Figure 2. The `Circuit` class contains a vector of shared pointers to `Component` objects. The pointers are shared so that the user can access and modify the components outside of the vector, which is useful for analysis. Similarly, the `Circuit` class contains a unique pointer to an `AC_Source` object. The AC source is always between the ground node and another node. Thus the class `AC_Source` only contains one node named `source_node`. The default node is $+1$, so that when the default `Circuit` constructor is called, the AC source is already implemented into the circuit.

Additionally, the `AC_Source` class contains the private members `V0`, `frequency` to indicate the starting phasor and frequency of the source.

The abstract `Component` class contains the private members `type` and `label` to identify the type of component – resistor, inductor, or capacitor – and a label to uniquely identify the component. The integers `node_1` and `node_2` indicate the nodes between which the component lies. The inherited classes contain an extra private member – `Resistor` contains `resistance` for example – and from this value, the `get_admittance()` function is overridden. The default nodes are $-1, -1$ to indicate an invalid location in the circuit.

```
class Component // abstract class
{
private:
  std::string type;
  int node_1;
  int node_2;
  std::string label;
  public:
  // ... rule of 5 ... //
  // ... setters and getters for the members ... //
  // get_admittance will be overriden by different component types
  virtual complex get_admittance(const double& frequency) const = 0;
  complex get_impedence(const double& frequency) const;
};
```

For the derived classes from `Component` – for example the resistor – the parametrised constructor takes the following arguments:

```
// label is "R" if unfilled
Resistor(const int& new_node_1, const int& new_node_2,
const double& new_resistance, const std::string& new_label="R");
```

This gives the user the option of labelling the components. As mentioned, the `Circuit` class contains a vector of shared pointers `Component` objects. This uses runtime polymorphism, so that when virtual functions are called on these components, the appropriate overridden method for the actual component type is executed. While the rule of five is implemented for each class in Figure 2, the intended usage of the programme is to call the default `Circuit` constructor and then add the components separately using `add_component` and `set_ac_source` (see Section 4.1 for a demonstration). The constructors of the components and the AC source are carefully designed to support this method, with the intention of making the usage of the `Circuit` class relatively straightforward. The `Circuit` header file is shown below:

```
class Circuit
{
private:
  int num_nodes = 0; // used in solve_circuit()

  std::vector<std::shared_ptr<Component>> components;

  // stores correspondence between nodes as strings and as ints
  std::unordered_map<std::string, int> node_ids{{"GND", 0}};
```

```
  std::unordered_map<int, std::string> id_to_node{{0, "GND"}};

  std::unique_ptr<AC_Source> ac_source;

  // converts string to int and stores the data
  int get_node_id(const std::string& node_name);

  // solve the MNA system (confusing so in private)
  std::vector<complex> solve_circuit() const;

public:
  // ... rule of 5 ... //

  // set AC Source with parameters
  void set_ac_source(const std::string& new_node,
    const double& new_V0, const double& new_freq);

    // Add component (has to be in header file)
  template <typename T>
  void add_component(const std::string& n1, const std::string& n2,
  const double& value) // value is the resistance, inductance etc.
  {
    components.push_back
    (
      std::make_shared<T> // resistor inductor or capacitor
      (
      // from str -> int and stored in the node id maps
      get_node_id(n1), get_node_id(n2), value
      )
    );
  }
  // Add component (has to be in header file)
  template <typename T>
  void add_component(const std::string& n1, const std::string& n2,
  const double& value, const std::string& label)
  {// same function but adds a label}

  // using solve_circuit():
  complex get_src_current() const;
  std::vector<complex> get_node_potentials() const;

  void print_data();
};
```

The maps `node_ids` and `id_to_node` store the correspondence between the nodes as strings (easier for the user to interact with) and integers (to index the nodes for computations). The `set_ac_source` function is overloaded with its private members to simplify setting the AC source for the user. The most important function in this class is `add_component`. This is a templated function so that each type of component can

be added individually using the same method. These derived component objects are then stored in the `components` vector. This function is overloaded to handle the arguments (`string`, `string`, `double`, `string`) or just (`string`, `string`, `double`), so that the user has the option whether to label the component. Here, the `double` argument refers to the material property of the component (a capacitor has a certain capacitance, for example).

The function `solve_circuit()` solves the linear system as given in (3):

```
std::vector<complex> Circuit::solve_circuit() const
{
  // ... potential error checks ... //
  // the size of the matrix A
  int N = num_nodes + 1;

  // Create zero matrix A and zero vector z using LinAlg
  std::vector<std::vector<complex>> A = LinAlg<complex>::createZeroMatrix(N);
  std::vector<complex> z = LinAlg<complex>::createZeroVector(N);

  // Fill the matrix A based on components (0th row and col stay emtpy)
  for(const std::shared_ptr<Component>& comp : components)
  {
    int i = comp->get_node_1();
    int j = comp->get_node_2();
    complex y = comp->get_admittance(get_frequency());
    if(i != 0) A[i][i] += y;
    if(j != 0) A[j][j] += y;
    if(i != 0 && j != 0)
    {
      A[i][j] -= y;
      A[j][i] -= y;
    }
  }

  // Modify A for the source node (fill the 0th row and col)
  int source_node = ac_source->get_source_node();
  if(source_node >= 0 && source_node < N)
  {
    A[0][source_node] = 1.0;
    A[source_node][0] = 1.0;
    z[0] = get_V0();  // first element of z (rest of elements are 0)
  }

  // Solve the system using LU decomposition
  std::vector<complex> output = LinAlg<complex>::LU_solve_x(A, z);
  return output;
}
```

This report will not go into detail on how the system is numerically solved, but essentially the `LU_solve_x` function solves the system via LU decomposition with partial pivoting [4] (LU decomposition is a com-

putationally better numerical method than matrix inversion). The full implementation of `LU_solve_x` is located in the `LinAlg.h` header file. Recall that the first element of the solution to (3) is a current instead of a voltage, and this current is in the direction pointing towards the source, instead of away from it. This can be confusing for the user, so the `solve_circuit` function is made private, and the functions `get_node_potentials` and `get_src_current` are designed for usage instead:

```
// get source current
complex Circuit::get_src_current() const
{
  // the zeroth element is actually defined as going into the AC source
  return -1.0 * solve_circuit()[0];
}

std::vector<complex> Circuit::get_node_potentials() const
{
  std::vector<complex> solution = solve_circuit();
  solution[0] = 0.0; // set the GND to zero (was initially -I_src)
  return solution;
}
```

As opposed to just adding components in series or parallel, the user now has more freedom with the network of the circuit. This comes hand in hand with more room for the user to construct ill-defined circuits. To combat this, many error checks are used. Firstly, the resistors cannot have resistance less than or equal to zero, and likewise for the rest of the components. This avoids the user from creating a short circuit (one might ask if the user can create just a wire between two nodes, but that is not possible as the nodes can only be initialised when they have a component between them). The AC source must have frequency strictly positive, again to avoid dividing by zero, and also a non-zero value for `V0`. Before solving the circuit, the function `nodes_invalid` is called to check that none of the nodes are equal to -1, which is a signature for invalid nodes. Additionally, the `is_empty` function checks that there is at least one component, so that there is a system to solve. For the component objects themselves, they cannot have the same node, and their nodal values cannot be less than -1 (something is either connected to a non-trivial node, the ground, or it is invalid). Furthermore, if these errors don't get detected, the `LinAlg::LU_solve_x` function will check for singular matrices.

4 Demonstration of usage for the MNA programme

The previous section was likely difficult to decipher. Hopefully however, the complexity behind the design will allow for a smoother user experience. In this section we demonstrate the usage of class `Circuit` through some interesting examples. The user can gain access to all of the `MNA` source files by including `MNA.h` located inside the `MNA` folder. The examples which we will explore are located in the `Examples` folder.

The design of the programme was intended to allow the user to investigate a circuit by creating a `Circuit` object with the desired components and then calling `get_node_potentials()`, or `print_data()` to do analysis. In practice, this is more easily done if the user creates their own class for a circuit which publicly

inherits `Circuit`. This makes the analysis more organised and thus easier for the user to interact with.

4.1 RLC circuit

We start by building the class `RLC` which publicly inherits `Circuit`. We write the default constructor to match Figure 1(a):

```
class RLC : public Circuit
{
public:
  RLC() : Circuit()
  {
    set_ac_source("A", 1.0, 1.0); // between GND and A
    add_component<Resistor>("A", "B", 1.0);
    add_component<Inductor>("B", "C", 1.0);
    add_component<Capacitor>("C", "GND", 1.0);
  } // ... class not finished
```

We will investigate how the average power dissipated through the resistor varies with the source frequency at different resistance values. We will also do this for the phase lag of the current behind the AC source, and compare our results with the expected relationship:

$$P_{\text{avg}} = \frac{V_0^2 R}{2\left[R^2 + \left(\omega L - \frac{1}{\omega C}\right)^2\right]} \quad \text{and} \quad \phi = \tan^{-1}\left(\frac{\omega L - \frac{1}{\omega C}}{R}\right) \tag{4}$$

First, we define a function to set the resistance more easily. We gain control of the `Resistor` object by using `dynamic_pointer_cast` from a `Component` object.

```
void set_resistance(const double& new_R)
{
  auto resistor=std::dynamic_pointer_cast<Resistor>(get_components()[0]);
  resistor->set_resistance(new_R);
}
```

Next, we do the frequency sweep for the average power. The full function can be found in the `RLC` file in the `Examples` folder. To highlight the main idea, we construct the `frequency_grid` vector of different frequencies we want to simulate and do the same with the resistances. The simulation is then computed and saved to an output file as follows:

```
for(double freq : frequency_grid)
{
  outFile<<freq<<" "; // add the frequency to the first column
  for(double R : resistance_grid)
  {
    set_resistance(R);
    set_frequency(freq);
    double I_tot = std::abs(get_src_current());
    double power = 0.5 * I_tot * I_tot * R; // = I_rms^2 R
    outFile<<power<<" "; // add the power for each R value
```

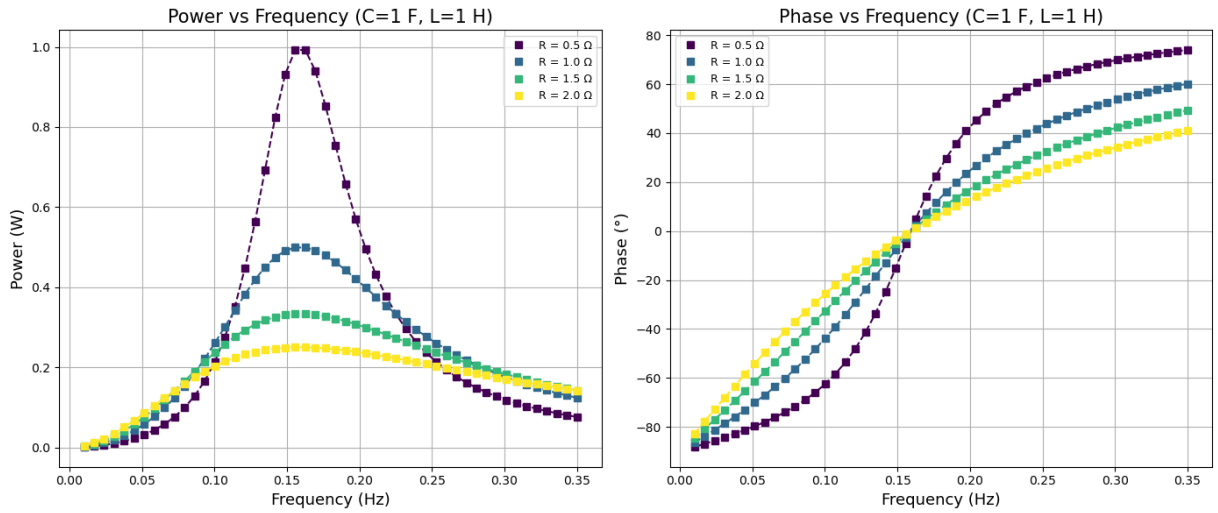


Figure 3: Results for the frequency sweeps for an RLC circuit at different resistances. The dashed line is the predicted curve, which is in accordance with the numerical simulation.

```
  }
  outFile<<""<<std::endl;
}
```

Now we do the same for the phase:

```
for(double freq : frequency_grid)
{
  outFile<<freq<<" ";
  for(double R : resistance_grid)
  {
    set_resistance(R);
    set_frequency(freq);
    // get the current leaving the AC source
    complex I_0 = get_src_current();
    // the 'lag' of the current is positive so we multiply by -1
    double phase = -1.0*std::arg(I_0);
    outFile<<phase<<" "; // write the phase to the ouptut file
  }
    outFile<<""<<std::endl;
}
```

The results for the simulations are plotted in Figure 3, which match our predictions in (4).

4.2 Wien bridge

Figure 4(a) shows a Wien bridge, a popular bridge circuit that was originally developed by Max Wien in 1891 [5] to measure impedances. In this example, we will investigate how the voltage between the bridge depends on frequency and resistance. One can deduce from Kirchoff's voltage law that that the bridge

should be balanced for [6]

$$\frac{R_2}{R_1} = \frac{R_4}{R_3} + \frac{C_1}{C_2} \quad \text{and} \quad f = \frac{1}{2\pi\sqrt{R_3 R_4 C_1 C_2}}. \tag{5}$$

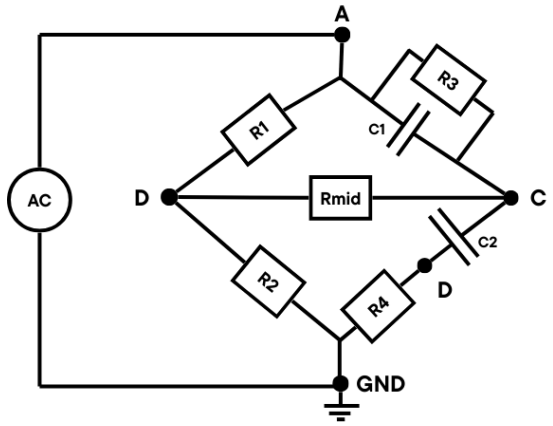
We follow a similar approach to before:

```
class WienBridgeCircuit : public Circuit
{
public:
  WienBridgeCircuit() : Circuit()
  {
    set_ac_source("A", 1.0, 1.0); // Between nodes GND and A
    // Left Arm
    add_component<Resistor>("A", "B", 1, "R1");
    add_component<Resistor>("B", "GND", 1, "R2"); // will vary
    // Right arm
    add_component<Capacitor>("A", "C", 1.0, "C1");
    add_component<Resistor>("A", "C", 1.0, "R3");
    add_component<Capacitor>("C", "D", 1.0, "C2");
    add_component<Resistor>("D", "GND", 1.0, "R4");
    // Bridge
    add_component<Resistor>("B", "C", 1.0, "R_mid");
  } // ... class not finished
```

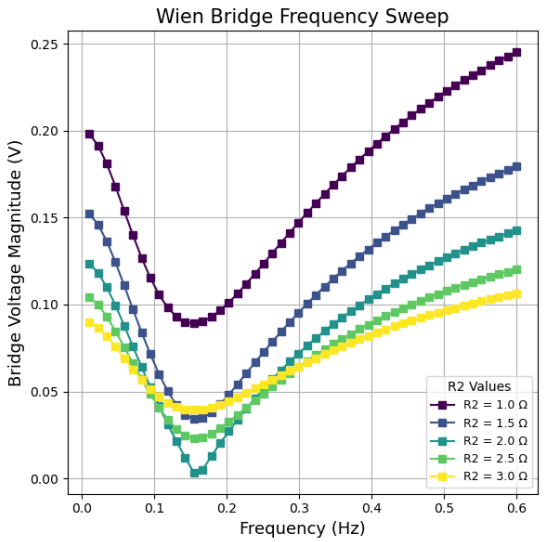
We create the function `voltage_frequency_sweep` to run the simulation. We construct the frequency grid and the resistance grid and then do the following:

```
for(const double& freq : frequency_grid)
{
  outFile<<freq<<" "; // add the frequency to the first column
  // set the frequency
   set_frequency(freq);
  // loop over resistance values and compute the voltage for each one
  for(const double& R2 : resistance_grid)
  {
    // set the value of R2
    Res_2->set_resistance(R2);
    // get the node potentials
    std::vector<complex> potentials = get_node_potentials();
    // bridge output = V(B) - V(C)
    std::shared_ptr<Component> Rmid = get_components()[6];
    int b_id = Rmid->get_node_1();
    int c_id = Rmid->get_node_2();
    complex v_out = potentials[b_id] - potentials[c_id];
    // add the voltage for this R2 value
    outFile<<std::abs(v_out)<<" ";
  }
  outFile<<std::endl; // new line for the next frequency
}
```

We then write the results to another output file to store the data. The results are shown in Figure 4(b).



(a) A Wien bridge

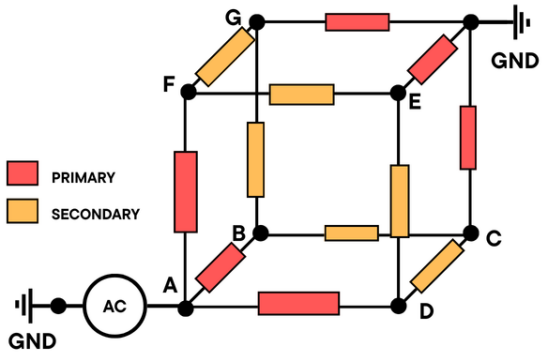


(b) Absolute value of bridge voltage vs. frequency for different R_2 values. The results verify that balancing is only achieved if (5) is satisfied.

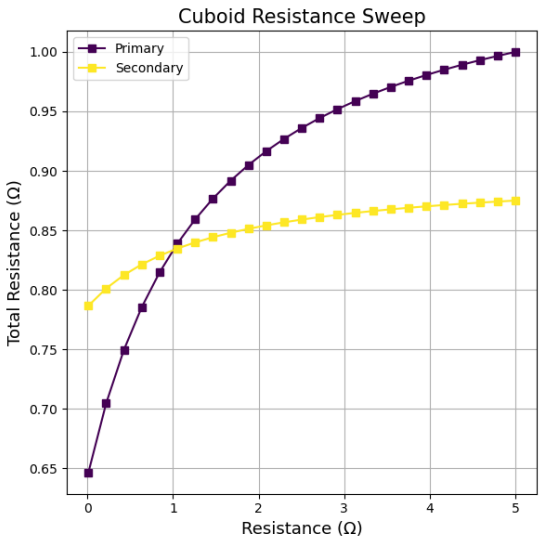
4.3 Cuboid of resistors

A famous problem in physics textbooks is to find the total resistance of a cuboid where each edge has resistance R. The solution $R_{\text{tot}} = 5R/6$ is obtained by a symmetry argument, but if we tune the resistance of one of the edges slightly, this symmetry is spoiled. By inspection of figure 4(c), we can see that geometrically, there are only two types of edges in this problem, which we will term 'primary' and 'secondary'. One could argue that the response to tuning the secondary edges will be less than tuning the primary edges, and we will verify this now. We start again by constructing the cuboid circuit:

```
class Cuboid : public Circuit
{
public:
  Cuboid() : Circuit()
  {
    set_ac_source("A", 1.0, 1);
    // bottom
    add_component<Resistor>("A", "B", 1.0, "Rab"); // primary
    add_component<Resistor>("B", "C", 1.0, "Rbc");
    add_component<Resistor>("C", "D", 1.0, "Rcd");
    add_component<Resistor>("D", "A", 1.0, "Rda");
    // top
    add_component<Resistor>("G", "GND", 1.0, "Rhg");
    add_component<Resistor>("G", "F", 1.0, "Rgf");
    add_component<Resistor>("F", "E", 1.0, "Rfe");
    add_component<Resistor>("E", "GND", 1.0, "Regnd");
    // middle
    add_component<Resistor>("C", "GND", 1.0, "Rcgnd");
```



(c) A cuboid of resistors.



(d) Total resistance vs edge resistance for both primary and secondary edges. The total resistance is less sensitive to changes of the secondary resistance.

```
    add_component<Resistor>("B", "G", 1.0, "Rbg"); // secondary
    add_component<Resistor>("D", "E", 1.0, "Rde");
    add_component<Resistor>("A", "F", 1.0, "Raf");
  } // ... class not finished
```

We then follow a similar procedure from the previous two sections, where the total resistance is computed by constructing the following function:

```
double get_total_resistance() const
{
  // safely convert I_src to double (it should be > 0)
  double I_source = get_src_current().real();
  // voltage should also be > 0
  double voltage = get_V0().real();
  // compute the total resistance
  return voltage / I_source;
}
```

The results are plotted in Figure 4(d), confirming our intuition that the total resistance is less sensitive to the secondary edges than the primary edges. We also see the total resistance is $5R/6$ when the R values of the cuboid are symmetric, which is reassuring as it matches the theoretical solution.

5 Conclusion

The MNA programme should work for all closed circuits consisting only of resistors, inductors and capacitors connected to a single AC source. As demonstrated in Section 4, the user can design any arbitrary circuit of this type by calling the default `Circuit` constructor and then adding the components

via their nodal locations.

There are many additional features that the programme could implement. One idea would be to incorporate additional component types, such as the diode, which would allow for the construction of many new interesting circuits such as the full-wave rectifier [7]. This wasn't included in this project as the current in the circuits is always assumed to be sinusoidal, which we cannot do with non-linear components like the diode (this is also why we're restricted to just one AC source, although technically other sources could be added but they would need matching frequencies). However, with MNA already implemented, this is not too far of an extension. Instead of solving Equation (3), we would need to solve something of the form $Av + B\dot{v} = c$, where the inductors and capacitors no longer have standard frequency dependent impedances, so their contributions would be stamped in B rather than A. This could be done by designing a coupled ODE solver. One could then graphically display the data for each component with time, as the data would then be time-dependent.

Additionally, one could create an interface that allows the user to visualise the circuit. From a practical perspective, this is a more suitable for software beyond C++. However, with the nodes already implemented, graphically displaying the circuit could be done by fixing the nodes on a 2D grid, and then separating them if the density of the components between them exceeds a threshold value.

References

[1] L. W. Nagel and D. Pederson, "Spice (simulation program with integrated circuit emphasis)," Tech. Rep. UCB/ERL M382, Apr 1973.

[2] F. S. Cheever, "Modified nodal analysis." `https://cheever.domains.swarthmore.edu/Ref/mna/MNA2.html`, n.d. Accessed: 2025-05-01.

[3] R. Khazaka, "Modified nodal analysis." ECSE 597 Circuit Simulation and Modeling.

[4] M. W. Reid, "Pivoting for lu factorization." `http://buzzard.ups.edu/courses/2014spring/420projects/math420-UPS-spring-2014-reid-LU-pivoting.pdf`, 2014. University of Puget Sound, Math 420 Project.

[5] M. Wien, "Messung der inductionsconstanten mit dem optischen telephon"," *Annalen der Physik*, vol. 280, no. 12, pp. 689–712, 1891.

[6] F. Terman, *Radio Engineers' Handbook*. New York: McGraw-Hill, 1943.

[7] "Full wave rectifier and bridge rectifier theory." `https://www.electronics-tutorials.ws/diode/diode_6.html`. Accessed: 2025-05-03.