**Matrix Initialization**

Building the matrix was no doubt the most computationally expensive part of the project, being able to build a circuit, and solve it for unknown variables in less than a second. The matrix class in Eigen is defined by 6 parameters<link>, out of which 3 are optional<link>, which we opted to leave out. The first parameter defined as “typename scalar” would initialize the type of each coefficient in the conductance matrix, and vector of knowns/unknowns. This allowed coefficients to be of any type defined, such as int, float, double and complex etc. Since conductance is the reciprocal of resistance, this meant we needed a type that allowed accuracy to decimal precision. Types of floats and doubles allowed this but during testing, we opted to go for double as it allows for twice the number of bytes as floats, but also sacrifices speed. Later on, we settled on complex double to allow for AC reactive components and this is written in depth in the “Advanced input management” section.

The next 2 parameters, known as RowsAtCompileTime and ColsAtCompileTime<link> would as suggested, initialize the number of rows and columns of the conductance matrix. This is of 2 types, dynamic and fixed. The Eigen typedef<link> can be used as a shorthand for initializing the matrix size and type, where the Eigen class is defined as Matrix<typename Scalar, int RowsAtCompileTime, int ColsAtCompileTime>, can instead be written as MatrixNt, where N is the number of rows or columns, and t is the type. Since the conductance matrix is built upon Kirchhoff’s current law, which in turn is derived from Maxwell’s equations<link>, a characteristic of the matrix is that it must be invertible, which in turn means it must be a square matrix. Therefore the “N” in the Eigen typedef represents both the rows and columns because the matrix must be square and invertible. The Eigen typedef MatrixXd was the popular choice, since it allowed the coefficients to be of type double, whilst keeping the size of the matrix as dynamic, allowing it to be changed when appropriate. Using a fixed sized matrix would be more efficient than a dynamic one in terms of performance<link>, however it was decided that we would just use a dynamically sized matrix, since most of our matrix/vector sizes can be defined by the maximum number of nodes and independent voltage sources, which would not be known at compile time. This meant that the 3 optional parameters of the Eigen class were not needed, and in the end made code more readable and easier to manipulate.

The most complicated part of this part of the program, was initializing the conductance matrix, because once this was done it was straightforward to define matrices of one dimension, i.e., vectors, and finding the inverse of the conductance matrix also turned out to be straightforward using specialized Eigen methods<link>.

The first obstacle was correctly constructing the size of the conductance matrix so that it appropriately scales to a suitable size defined by the nth node and kth voltage source. To achieve this, the use of a function mat\_Populate() was defined, in which it would populate the coefficients of the 3 matrices, but we will discuss the two vectors later and focus on the main conductance matrix for now.

Using the typedefs for the matrix, and similar typedefs<link> for the vectors, a total of 3 matrices were constructed. The first, Cond, was of type double and dynamic size using the typedef MatrixXd. The 2 vectors, also known as matrices with one column, used the typedef VectorXd which were also of type double and dynamic size. As per the project specification for the netlist template, the first string would be the designator of the component, and the next 2 strings represent the connected terminal nodes of the component. Knowing this, a function node\_max() was defined that would return the largest node in our parsed std::vector<std::vector <std::string>> netList by looping through each line within netList and appropriately assigning the largest node back. This was important as the maximum node was required when defining several other matrices that would construct the conductance matrix and vectors.

Text

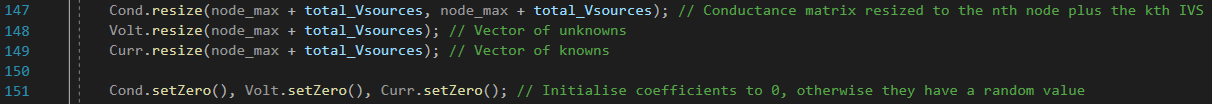
Description automatically generated

The next objective is to identify the number of Independent Voltage Sources, which will be needed to scale the size of the matrix and vectors. For this, a simple iteration of the parsed netList is required to locate every instance of a voltage source, which can be done by checking if the type is equal to “V” and is then counted by another variable, total\_Vsources. Text

Description automatically generated

The main conductance matrix, known as Cond (cond\_Mat in main) in the function, can be resized to size (n x k), where n is the number of nodes and k is the number of voltage sources. This is another method that comes with the Eigen library, allowing the resizing of dynamic and fixed type matrices using .resize()<link>. However, the coefficients within the matrix are still uninitialized, so we require the setZero()<link> method to set all coefficients to zero. Not doing this leads to some random value within the matrix, which in our testing equated to e^+6 which affects results dramatically when solving the matrix for the vector of unknowns.

The vector of knowns, Curr, and the vector of unknowns, Volt, were both initialized and resized within the function mat\_Populate, which was used to initialize the coefficients of matrix cond\_Mat. We used the Eigen typedef VectorXd to define the 2 vectors of dynamic size and coefficients of type double. Since the overall result was to do matrix multiplication between the conductance matrix and the vector of knowns, this required that the vectors multiplying the conductance matrix were multipliable, meaning their rows had to match the number of columns in Cond. This was defined as the number of nodes + the number of independent voltage sources and using eigen method .resize() within mat\_Populate(), we were able to initialize the two vectors correctly. Setting the coefficients of the two vectors to zero using setZero() was important, because it helped to remove random uninitialized terms, but also helped with future calculations. For example, if a node was not connected to any current source, the current into that node in the current vector will be zero. So this saves time as we don’t need to implement further code to change some unknown value to zero in the vector.



**Matrix Calculations and Initializing Coefficients**

Now that we had initialized the sizes of Matrix Cond, and Vectors Volt and Curr, we began to add support for different components that were possible to place in the netlist file.

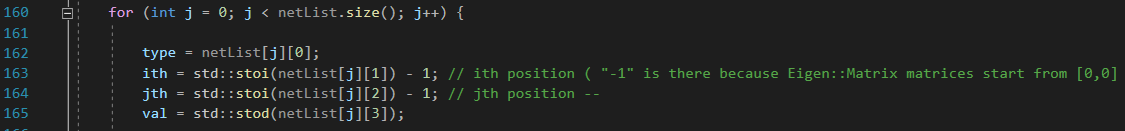
To populate the matrix coefficients, the first parameter that would be required is the vector of vectors netList, as this contained all the information we needed such as the type of component, their node connections and their values. Using the vector of vectors, we can define new variables ith, jth, val and type. Ith and jth are the corresponding [ith, jth] position in the conductance matrix, defined by the first and second node contained in each line within netList. If a component is connected between nodes 1 and 2, its position in cond\_Mat would be [1, 2]. An important aspect of the Eigen library to note is that contrary to linear algebra in mathematics, the matrix starts from the [0, 0] position since indexes in programming are defined by their location in memory, which often starts from zero. In fact, the [1, 2] position will need to be adjusted to [0, 1], which is why ith and jth are defined as the node found in each vector minus 1. A suitable data type would be int since a node cannot be anything but an integer. Val is defined as the 4th element or index 3 in each vector, since the value is prescribed after both the designator and the connected nodes during construction of netlist as well as on the project specification. The instinctive decision would be to assign val as type double, since the spectrum of values for reactive components such as capacitors and inductors can reach well below the micro (10^-6) scale, and double will allow for more precision than float, so we went with that. There is also the first element in the vector, the designator, which will read the type of component. We can leave this as a string type since its only use is to identify the type of component and cannot conveniently be converted into anything else that is useful.

Text

Description automatically generated

Now we started to initialize the contents of matrix Cond. This involved a form of Modified Nodal Analysis, in which the algorithm on this website was especially useful<link>. The first step was to break down the conductance matrix into 4 smaller matrices G, B, C, D. Matrix G would contain the passive and reactive components of our circuit and would be of size (n x n) where n was the number of nodes. A for loop was used to pass each vector in the vector of vectors into the new function, G\_matrix() which constructs the G matrix. To place these components into the matrix, we need the ith, jth nodes, the value of that component, and its type which are defined by their respective positions of each vector in netList.







The processing of G\_matrix() was straightforward, and we began with adding support for resistors since they were the easiest passive component to implement. A simple nested conditional was used to check if the type of component was “R”, verifying that it is a resistor. This conditional was defined by the variable ‘type’, which would be equal to the first string in the netList written as netList[0]. If the component is indeed a resistor, another conditional within would check to see if the resistor is connected to the reference (ground) node or not, since this connection would affect the position of the resistor’s conductance in the matrix. If a component was connected to ground, then its conductance would be placed along the diagonal of the matrix using the node that it is connected to. So, a resistor between ground and node 2, would be stored in the [2, 2] position ([1, 1] in Eigen) with its conductance equal to 1/val. This is because the reference node is not included in the conductance matrix, so it is not included in the sub matrix G. If connected between 2 non-reference nodes, then the negative conductance of the resistor would be taken and stored in matrix position corresponding to the ith and jth node. The reason why we take the negative conductance involves simple nodal analysis equations. If a resistor is connected between two nodes, 1 and 2. The voltage drop across the resistor would be (V1-V2)/R and rewriting -V2/R as (-1/R)\*V2 shows that we do indeed have a negative conductance here. Since G\_matrix() was called in a for loop within mat\_Populate(), it would be updated with every passive component in the vector of vectors, so the G matrix would have been initialized and its coefficients populated by the end of the for loop.

Text

Description automatically generated

We then started on capacitors and inductors. In DC, capacitors charge up to the voltage they are connected to, and as a result hinder the flow of current. Therefore, the conductance of a capacitor tends to infinity like a voltage source, and so it can be treated like one and ultimately ignored from the G matrix, since there are no voltage sources in G. With inductors, since there is no voltage drop across them, they act as a short circuit and so we used the normal impedance of an inductor(similar to a resistor) This does lead to some inaccuracy problems when integrating AC circuits, as the larger the inductance, a greater margin of error was visible in our results which we resolved later on.

Text

Description automatically generated

The next sub matrix of Cond is called B, and its size is defined as (n x k), where n is the number of nodes and k is the number of independent voltage sources.



The B\_matrix() function constructs this matrix, only populates it with 1, 0 or -1, and is found within the same for loop that G\_matrix() is in, with the only exception that it falls under a conditional to check if the type == “V”. The reasoning behind this was because matrix B only shows the position of independent voltage source(s) in the circuit and the sources’ values would be stored in the vector of knowns. An independent voltage source is allocated a position based on whether it is grounded or not, and allocated a value based on the direction of positive and negative terminals. If a source was grounded then it will only appear once, since the reference node is not defined in the matrix, and it will be positioned on the ith/jth row, depending on which node is the reference node i.e. = -1(we defined earlier why minus 1 is used when referring to matrix positioning). If the ith node is ground, this means that the jth node receives a negative voltage as defined by the netlist specification where the first node mentioned is positive and the second node is negative, and so the position [jth, Vsource\_pos] is equated to -1. Here, Vsource\_pos is set as the kth voltage source. If one voltage source is passed in, its position is 0. But if a second and/or third source is in the circuit, Vsource\_pos keeps track of the kth source, and the kth column of matrix B is updated. If an independent voltage source is connected between 2 non-reference nodes, this equates to the KCL equation Vi - Vj, so the ith node is allocated a 1 and the jth node a -1, and the appropriate column is chosen based on how many voltage sources there are. If this was the second voltage source out of 2, then these values would be stored in the second column etc.

Text

Description automatically generated

The C matrix is ultimately the transpose of the B matrix, since the connection between nodes i and j are equal to the connection between nodes j and i. The transpose is found using the Eigen method .transpose(), flipping rows and columns and storing into C.



The D matrix is of size (k x k), where k is the number of independent voltage sources, and it is initialized using the setZero() method. The D matrix serves no purpose other than to complete the full Cond matrix by making it square and invertible.



Using the Eigen header library, one can initialize a matrix using “<<”. This can be used to set a matrix with constant integers, float types or double types, or it can be used to initialize a matrix using other matrices. As mentioned before, Cond is made up of 4 smaller matrices G, B, C & D, so writing “Cond << G, B, C, D;” yields the same result, and we end up with a square, invertible matrix of size (n + k) x (n + k). This completes the construction of the conductance matrix, and later on we will improve it to work with AC signals but for now, it works well with DC passive components. Matrix Cond was passed into the function mat\_Populate by reference, so there is no need to return our matrix type to the main, since it will update the conductance matrix globally, and hence further calculations on Cond take place in the main().



We have defined how the conductances are appropriately placed within a matrix based on their node connections, and that if they have one connected to ground, we just add them to the main diagonal. However, one important factor to note is that the main diagonal doesn’t just contain conductances that are connected to ground, but all the conductances that are connected to that node. If 3 resistors were connected between nodes 1 to 2, 1 to 3 and 1 to 4, we have a total of 3 resistors connected to node 1 but none to ground. This means we will have a conductance at positions [1, 2], [1, 3] and [1, 4] but also the total conductance at position [1, 1] will be equal to the sum of these conductances. To do this we defined a new function cond\_Diagonal, that would simply look at the conductances in one row of the matrix, sum them up and place the total into the main diagonal.

Text

Description automatically generated

Now that the conductance matrix has been initialized and its coefficients given a value, we can move onto the two vectors. The first vector, Volt, does not need to be filled with values yet, since it behaves as our vector of unknowns with unknown voltages and currents. The second vector, Curr, is our vector of unknowns and is straightforward to update. We already constructed the size of Curr as ((n+k) x 1), the nth node and kth voltage source, meaning the number of values in this vector will equal to the sum of nodes and sources.

Kirchhoff’s Current Law states that the sum of currents entering/leaving a node is 0, so if a node is not connected to any current sources, we can assume that the current is 0 at that node. Therefore, a KCL equation at node 1 would mean the first entry in the current vector is 0. However, when a node is connected to a current source, we must look at the way it is defined in the netList. Following the project guidelines, the first node mentioned in the “in” terminal and the second node is the “out” terminal. This means when looking at the currents going out of a node, as we have learnt to do so this year, if the “in” terminal is facing the current node, this means the current is flowing out of the node and fulfills the nodal equation = 0. However, this does not mean a zero is stored in the current vector, but instead the negative current will be stored in the vector at the nth position, since this is the same as rearranging a KCL equation for unknowns only. A positive current on the left of an equation, means a negative current on the right, so we place the opposite value of the current into the vector of knowns. Likewise, if a current source is floating, then the corresponding nodes connected to each terminal will be added to nth row of the vector, but with opposite signs since one will be entering the node, and one will be leaving as defined by their nodal equations. This processing was defined within a new function curr\_Matrix().

Text

Description automatically generated

**Matrix Decomposition**

Now that both the conductance matrix coefficients and the current vector coefficients have been initialized, it was time to solve for the voltage vector. Our first thoughts were to use the Eigen method .inverse()<link> because we needed the inverse of the matrix Cond to be multiplied with the current vector on the right. However, as well as this, the eigen header offers a more versatile approach with methods that can be used to directly solve for some unknown vector, and they are each rated by their accuracy, performance, and limitations<link>.

The first method we considered using was ColPIvHouseholderQR(), since it boasted relatively fast decomposition speeds whilst also maintaining high accuracy. The tradeoff was that for larger matrices (circuits with more than 16 nodes), the speed of decomposition dramatically falls and so presents a challenge to efficiency for those larger circuits. Though we did not expect to evaluate circuits with so many nodes, it was clear that the algorithm would not be best for such scenarios.

Another method we considered was LLT(), as it was presented to have one of the fastest decomposition times within the entire list of methods. The limitation to this method was that it required values within the matrix to be positive definite, and we have shown that our matrix contains both positive and negative conductance values, so the LLT() method will not work. This also eliminated the use of LDLT(), which is just as fast as LLT() decomposition but also boasts a similar drawback whereby the coefficients of the matrix must be positive or negative semidefinite, meaning they can only either be positive or negative, whilst our matrix is both. It was worthwhile discussing the use of these methods, because if we had components that were only connected to ground and not between nodes, the matrix would be positive only, but this can only be used for very basic small circuits, and the extra logic that would be required to perform this decomposition specifically to small circuits was not very appealing.

This left the final method we analyzed partialPivLU(), which offered very fast decomposition times of both small and large matrices, as well as a fairly accurate decomposition but not as good as other methods. However, since we were limited to using type double, the difference of accuracy was hardly noticeable and barely affected our results. partialPivLU() does have a drawback in that its matrices must be invertible. We have already discussed why our matrix must be invertible because of its relationship with Kirchhoff’s Current Law derived from Maxwell’s equations<link>, so our matrices will always fulfill the condition of invertibility. When using decomposition methods, they must be defined in the format, MatrixNt.partialPivLU.solve(VectorN). To move from A\*x=b to x=A^-1\*b, the first matrix defined is the matrix that will be made into its inverse, and the second matrix that will be multiplied with the inverse matrix is placed within the parameters of the solve() member function. This would give us all the node voltages within the voltage vector, and hence the solution has been found and can be used for AC analysis.



**Advanced Input Management**

Now that our circuit has been defined for DC circuits, it was time to implement AC logic. The first thing to look at when it comes to AC circuits is the frequency, omega. This means that we must bring complex numbers into the matrix now, since the admittance of capacitors and inductors in AC are jωC and 1/jωL. Eigen allows the use of complex type matrices<link> and can be defined using the Eigen typedef Eigen::MatrixXcd. Here, we are defining a matric of dynamic size, and its coefficients are of type complex double, which is what “c” and “d” in the typedef represent. Because our matrix will now involve complex numbers, this means all 3 matrices Cond, Volt, and Curr had to be initialized as complex double types. As we have discussed how Cond is made up of 4 smaller matrices G, B, C & D, these matrices also needed their typedef initializations to be updated. Although it is only sub matrix G that would hold the imaginary values of reactive components, whereas the other 3 matrices making up Cond do not, it is vital that all matrices that make up Cond are of the same type otherwise there will be initialization errors when constructing Cond.

Text

Description automatically generated

When constructing matrices of type complex double, they now can store two values, one in the real component and one in the imaginary component. First, we need to distinguish to the matrix when to update its imaginary value and when to update its real part. To do this, we used a Boolean variable called is\_AC. The function of this variable to always assume that a circuit is in DC until a component shows otherwise. In the for loop defined for counting the number of independent voltage sources into total\_Vsources, by parsing the netlist and counting every mention of a line that starts with “V”, we can add the boolean variable here and add a conditional to check whether a source is DC or AC. The distinction between an AC and DC source in SPICE, is that DC simply has one real value, whereas an AC source has value equal to its magnitude and phase. This is further emphasized with the value written as AC(mag, arg). So we can do a simple check to see if the value of a source starts with “A”, it is an AC source, and if it does not start with AC but a number, it is DC. When an AC source is found, bool is\_AC is set to true because we have identified that the circuit is indeed AC.

Text

Description automatically generated

This can then be passed on into the construction of the sub matrix G, along with the frequency step as these two variables are required when working with AC circuits. The conditional was simple, if the variable is\_AC was true, and we the current component is of reactive type, i.e. capacitor or inductor, then we require the use of omega and the imaginary value in the [ith, jth] position. For capacitors, the value had to be changed from being capacitance alone, to being 1/jωC. This is the impedance of a capacitor, but the conductance matrix is filled with conductance, so we need the reciprocal of its value, jωC, its admittance. We do not need to write jωC directly into the [ith, jth] position of the matrix, but only need to take ωC and store that into the imaginary component at the correct position. For example, a capacitor connected between nodes 1 and 2(0 and 1 in eigen) and when the circuit is AC, will be entered into the matrix using the code, G(0, 1).imag() = ωC. So when accessing this position again, we can read it as 0 + jωC, since we only changed the value in the imaginary and left the real part as 0.

The same logic can be applied for inductors, as they also need to have their values changed when looking at AC circuits. The impedance of an inductor is jωL, so the admittance (reciprocal) is equal to 1/jωL. This presents a slight issue with how complex variables work in C++, because we want to store 1/ωL into the imaginary component, if we just stored it normally, it would be equal to j/ωL which is not equal to 1/jωL. However, using some complex arithmetic knowledge from our mathematics course this year, we know that the reciprocal of an imaginary number, 1/j, is equal to itself multiplied by -1, -j. And we can just store (-1)/ ωL into the imaginary part.

The value of an AC voltage source is its amplitude, so therefore the only difference between a voltage source in DC and AC is the way the value is read from the netlist. However, the value that is stored into the current vector, is still done in the same way so it is not difficult to implement that with normal voltage sources in DC.

There were also some slight adaptations required to the cond\_Diagonal function, namely because we have both real and imaginary parts, so the main diagonal must contain the sum of all real/imag conductances connected to a specific node. Using the .real() and .imag() methods made this relatively easy as it allowed us to sum up each part separately.

Text

Description automatically generated