C++ Implementation of Electrical Network Library

Special Topic report for the *C++ for Scientific Computing* course taught by Prof. Joseph Pitt-Francis
Trinity term 2021-2022

Candidate Number 1064551

Contents

1 Introduction

This report concerns the development process of a scientific computing project, creating both a linear algebra and an electrical network library, in the C++ programming language [3]. Through the specific implementation, the development practices are discussed.

First, the development and build system are outlined and motivated in Section 2. Then Section 3 outlines the main features of the linear algebra library and the memory management. Subsequently, in Section 4, the theory and implementation of the electrical network solver are presented. Finally, Section 5 conveys the testing, automation and documentation framework before Section 6 provides recommendations and concludes the report.

2 Development and build system

With scientific computing projects ever increasing in size and complexity, the importance of well structured and easily reusable code increases tremendously. To that end, we have set up a structure and build system that is conducive to these goals.

Folder structure One part of this task is setting up an appropriate file and folder structure. The linear algebra library was placed as a dependency of the electrical network library, linked statically. A demonstration file (inside the `demo_code` folder) then imports the electrical network library and uses it. The folder structure is provided in Listing 5.

Build structure To the build system, two libraries are presented (`LinAlg` and `ElectricalNetwork`) and they are included in two places (`ElectricalNetwork` needs `LinAlg` , and `demo_code` needs `ElectricalNetwork`). The CMake build system was chosen: "CMake is an open-source, cross-platform family of tools designed to build, test and package software" [1]. It is platform and compiler independent, and allows easy setup and reuse of code. Specifically in this project, CMake is useful as it allows the libraries to be neatly packaged up, and imported where needed, without having to maintain a complicated Makefile.

Creating a library To create a library in CMake, we first call `add_library`, define the CMake library name and desired linking structure (static, shared or module) and list the source files. The second step is to define the directories to be included and how (public, private or interface), using `target_include_directories`. Consider the CMakeLists file for the LinAlg library in Listing 1.

```
add_library(LinAlg STATIC
            src/LinAlg/linalg.h
            src/LinAlg/matrix.cpp
            src/LinAlg/matrix.h
            src/LinAlg/matrix_constructors.cpp
            src/LinAlg/matrix_constructors.h
            src/LinAlg/vector_constructors.cpp
            src/LinAlg/vector_constructors.h
            src/doctest.h)

target_include_directories(LinAlg PUBLIC "${CMAKE_CURRENT_SOURCE_DIR
   }/src")
```

Listing 1: ../CppScientificComputing/electrical_network/lib/linalg/CMakeLists.txt

As can be seen from Listing 1, the library is defined as being static, and the directories are included using the public form. The static operator implies the libraries are archives of object files for use when linking other targets (hence will be added into the main executable). This is in contrast to a shared library which is kept separate and loaded at runtime, or a module, which may be loaded at runtime. The source was exposed to the user code using the public operand, signifying that it is both compiled with and can be imported by the user code. Note that the code itself is within a folder named src/LinAlg, such that including the code will have the form `#include <LinAlg/matrix.h>`, ensuring that there is no ambiguity where the code is being imported from (in case of multiple libraries).

Including a library To import a library, we inform CMake where to look for a CMakeLists.txt file (using the `add_subdirectory`), when needed, and then link the library using `target_link_libraries`. Finally, to include user code we use the `add_executable` function to define the target (whose name is needed for linking) and add the source files (one of which must contain a `main()` function). For example, consider the demo-code in Listing 2.

```
add_executable(demo_exec main.cpp)

target_link_libraries(demo_exec PRIVATE ElectricalNetwork)
```

Listing 2: ../CppScientificComputing/demo_code/CMakeLists.txt

We didn't have to include any directories because this file itself is included in the `CMakelists.txt` file in the root of the project.

3 Linear Algebra

The electrical network library requires matrix and vector classes, as well as a linear equation solver. These were packaged up as the linear algebra library (named LinAlg), and is where the majority of the development effort was dispensed.

3.1 General structure

The library consists of one central Matrix class with extensive functionality, and several special constructors (in the `matrix_constructors.h` and `vector_constructors.h` files).

The Matrix class overloads various operators, explained in Subsection 3.2, which includes the data management operators explained in Subsection 3.3, and implements various other helper functions, as can be seen from the header definition presented in Listing 6.

3.2 Operator overloading

Both to make the matrix class easier to use, and to implement required functionality, a variety of operators were overloaded.

Analytical operations Performing element-wise multiplication, addition or subtraction are very similar. To avoid code duplication (which is error prone and hard to maintain), the following helper function was setup:

```
typedef double f_double(const double& a, const double& b);

Matrix combine_two_matrices(const Matrix& m1, const Matrix& m2,
    f_double func)
{
  assert (m1.n_rows == m2.n_rows);
  assert (m1.n_cols == m2.n_cols);

  double** _new_arr;
  _new_arr = new double* [m1.n_rows];
  for (int i=0; i<m1.n_rows; i++)
  {
    _new_arr[i] = new double [m1.n_rows];

    for (int j=0; j<m1.n_cols; j++)
    {
      _new_arr[i][j] = func(m1.read(i, j), m2.read(i, j));
    }
  }
  return Matrix(m1.n_rows, m1.n_cols, _new_arr);
}
```

Listing 3: Combine Two Matrices

such that the overloading operators could then have the following form:

```
inline double unit_add(const double& a, const double& b)
{
  return a+b;
}

Matrix Matrix::operator+(const Matrix& m2) {
  return combine_two_matrices(*this, m2, unit_add);
}
```

where one notes the use of the `inline` keyword, which places a copy of that function everywhere it is called, reducing the function call overhead. The addition and subtraction operators are implemented similarly. A similar helper function was written for the $+$, $-$, and $*$ operations between a matrix and double, and a matrix and integer. Note that the multiplication is performed elementwise, and is not matrix multiplication, which is performed using the overloaded `&` operator. The division operator is overloaded to solve a linear system of the form $Ax = b$ with `Matrix x = A/b`, which uses Gaussian Elimination with pivoting, see Section B for the full code. The algorithms contain rough derivations in the source, see Section 5.

The comparison operators `==` and `!=` were also implemented element-wise, in a similar form as above. Moreover, a `.almost_equal(const Matrix& m2, double tol=1e-15)` function was implemented that compares to a set tolerance.

Ease of use For easily inserting data into a Matrix, the `<<` and `,` operators where overloaded, which enable matrix definition of the form:

```
Matrix A(3, 3);  \\ Creates a 3x3 matrix filled with 0s.

A << 1, 2, 3,
    4, 5, 6,
    7, 8, 10;
```

As part of the data management philosophy, see Subsection 3.3, the raw data is private. Therefore, accessing the data needs to go through helper functions, or the overloading `()` operator (which calls the `get` helper function). The `double& get(int row, int col)` returns a reference, such that it can be used both for reading and writing. When reading the data needs to be performed in way that ensures the source is not modified (which occurs when the variable is a function input with the const keyword), the `double read(int row, int col)` is used.

3.3 Memory management

The main software development challenge was the memory management. To increase the versatility of the Matrix class, the number of interactions with the raw data was minimized. The aim is to make the class as independent of the underlying data structure as possible.

The Big Five Since the introduction of C++ 11 [1] (which is the informal name for ISO/IEC 14882:2011), which introduced move constructor and move assignment operator, the rule of three was broadened to the rule of five [2]. It says that if one defines one of the following, one should define all five:

1. Destructor: `Matrix::~Matrix()`
2. Copy constructor: `Matrix::Matrix(const Matrix &source)`
3. Copy assignment operator: `Matrix &Matrix::operator=(const Matrix &m)`
4. Move constructor: `Matrix(Matrix&& m ) noexcept`
5. Move assignment operator: `Matrix& operator=(Matrix&& h) noexcept`

Given that the Matrix class contains pointers to pointers, it needs to define them all. A simple example is that a destructor is needed otherwise the array's memory will not be deleted, only the `p_arr` pointer. Another reason is during a copy operation, when the default copy is shallow (meaning the pointer address is copied). This leads

[1]See `https://www.stroustrup.com/C++11FAQ.html`

[2]see `https://www.feabhas.com/sites/default/files/2016-06/Rule%20of%20the%20Big%20Five.pdf`

to the compiler trying to delete the same memory twice, when each object goes out of scope (which leads to a segmentation fault).

The difference between the move constructor and the copy constructors lies in the number of number of final objects. A copy constructor takes in a reference to a class instance and returns another, independent (i.e. pointing to different areas in memory) but identically valued instance. A move constructor takes an rvalue variable (temporary, hence the `&&`), and moves all those variables onto the current instance. This requires deleting the `p_arr` after swapping to avoid memory leaks.

The difference between the move and copy assignment operators is again the number of instances at the end. These get called when assigning to an existing variable, either copy when its a named variable or move when it's temporary variable.

Memory management philosophy The only functions that interact with the raw data are the get and read functions (main gateway to data accessing), the memory management functions, the transpose function, and the swap_rows function. All other functions go through get (or the overloaded `()` which calls get) or read when a non-modification of the source is needed.

The reason for this is to allow easy overloading for implementing a matrix with a different underlying data structure. The implemented data structure consist of `p_arr` which is a pointer to an array of pointers, which point to arrays of doubles. This enables the logical `p_arr[row][col]` indexing. However, it is not necessarily the best option. The main advantage of this data structure is the easy row swapping, as it only involves swapping two pointers rather than copying the arrays. Table 1 compares three options. On top of the presented reasons, the ease of implementation should also be considered. For example, implementing a diagonal sparse matrix will probably be easiest with a single array.

What is therefore required from the Matrix class, is that the core, hard-to-implement functionality (such as the linear solver) is agnostic to the underlying data structure. Replacing the functions that are specific to the data structure (made possible by using the `virtual` keyword), then allows easy implementation of different types of matrices (such as upper diagonal or sparse tridiagonal matrices). The functions that must be overloaded are the big five, the add/remove row/column functions, the get and read functions and the transpose function.

Choosing between the options presented in Table 1 boils down to a trade-off between the cost of re-arranging (low for hash-maps) and data-finding cost (low for the single array). In this report the array or arrays option was chosen as an appropriate

Table 1: Comparison of data storage options

Method	Advantages	Disadvantages
Single array	* Data retrieval is fast (1 operation to calculate the address and 1 memory read).	* Any transformation is expensive, as the data needs to move. * Requires a contiguous space in memory for the entire array to fit in
Array of Arrays	* Swapping, adding or removing rows don't require moving any data and hence is fast	* Column operations still require moving data * Data retrieval is slower as it requires two memory reads.
Hashmap	* No data ever needs to be moved, only the hashmap updated.	* Every data read/write requires an evaluation of the hashmap * Hashmap may take up non-negligible resources

middle ground.

4 Electrical Network

Electrical networks form the building blocks of many of our daily appliances, form the backbone of all digital systems and power our homes. Therefore, being able to model them appropriate is a relevant skill in the modern world. In this report we model a direct current circuit built of lines (with internal resistance) and voltage sources.

A library was implemented to solve for the voltage and current in an electrical grid. We start by deriving a solution, inspired by [2, p37-43], and then explain the C++ implementation.

4.1 Mathematical theory

The variables defining the state of the grid are either nodal values (such as the potential) or edge variables (such as the current). By using Kirchhoff's current and voltage laws, together with Ohms law, we can relate build a well-posed linear system. Consider the network shown in Figure 1.

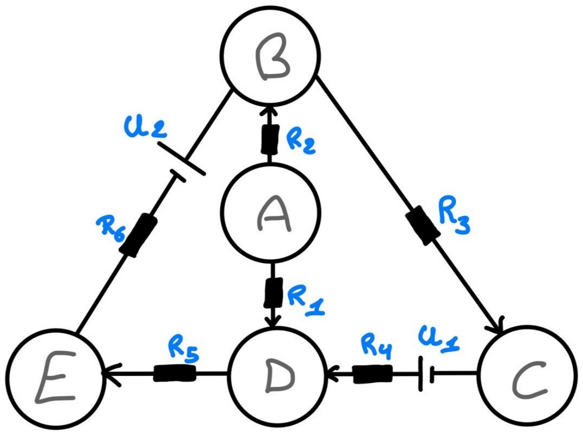

Figure 1: Example electrical network

Let x_i be the voltage at node i, collected into the potential vector $\mathbf{x} = (x_1 m \ldots, x_m)^T$. Similarly, let y_j be the current on line j, collected into the current vector $\mathbf{y} = (y_1, \ldots, y_n)^T$. We then define the voltage drop $e_j = x_{u(j)} - x_{l(j)}$ where $l(j)$ is the node from which line j start, ending at node with index $u(i)$, such that the voltage drop vector is $e = (e_1, \ldots, e_n)^T$. Equivalently, the relationship between the potential vector and the potential difference vector can be written as:

$$e = -Bx \quad \text{where} \quad B_{i,j} = \begin{cases} -1 & \text{if } j = u(i) \\ 1 & \text{if } j = l(i) \\ 0 & \text{otherwise} \end{cases}$$

where $B \in \mathrm{R}^{n \times m}$ is called the incidence matrix. This matrix describes the geometry of the network. Consider a line with an resistance R_j and a voltage source p_j, as seen in Figure 2.

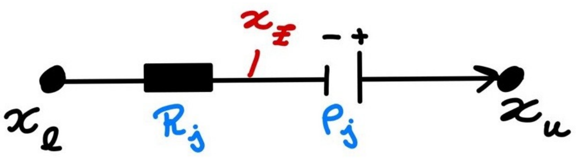

Figure 2: Diagram of the model of a line.

Applying Ohm's law and a potential difference, we find:

$$\begin{cases} x_z = x_l - R_j y_j \\ x_z + p_j = x_u \end{cases} \quad \rightarrow \quad e_j = x_u - x_l = R_j y_j - p_j$$

which, in matrix form becomes

$$\mathbf{y} = C(e+p) = C(p - Bx) \tag{1}$$

where $C = \text{diag}(1/R_1, \dots, 1/R_n)$. Note that p_j is positive if the polarity from - to $+$ corresponds to the positive direction. Then we apply Kirchhoff's current law, which states that the net current outflow from a node is zero:

$$Ay = 0 \quad \text{where} \quad A_{i,j} = \begin{cases} -1 & \text{if } i = u(j) \\ 1 & \text{if } i = l(j) \\ 0 & \text{otherwise} \end{cases}$$

from which we can see that $A = B^T$. We then have the following system of equations:

$$\begin{cases} e = -Bx \\ y = C(e+p) \\ 0 = B^T y \end{cases} \quad \rightarrow \quad B^T C(p - Bx) = 0 \quad \rightarrow \quad B^T CBx = B^T Cp \tag{2}$$

which is a linear system for x. Once x is obtained, we can map that to the currents using $y = C(p - Bx)$.

Existence and uniqueness of solution Having derived the linear system of Equation 2, we ought to determine the existence of a solution and its uniqueness. One option is to determine whether $M = B^T CB$ is positive definite. If so, the linear system $Mx = y$ will have a unique solution for every y. With $\langle \cdot, \cdot \rangle$ the Euclidean scalar product, form:

$$\langle x, Mx \rangle = \langle Bx, CBx \rangle$$

One can conclude that M is:

1. positive semidefinite is C is positive semidefinite.

2. positive definite if C is positive definite and B has a non-trivial kernel.

For most real networks, which have a positive nonzero resistance, C is positive definite. However, it can be shown that the incidence matrix B has a non-trivial kernel (=nullspace) [2, p42]. Therefore, we conclude that Equation 2, if it has a solution, it will be non-unique. Physically this can be understood as the potentials only being defined relative to each other. This is usually resolved by defining one nodes as a reference and defining its potential as being zero.

However, due to the non-trivial kernel, it is not directly clear that the system of equations has a solution at all.

Theorem 4.1: Solvability

Let $C \in \mathrm{R}^{n,n}$ be symmetric and positive definite and $B \in \mathrm{R}^{n,m}$. Then it holds for $M = B^T C B$:

1. $\mathrm{Null}(B) = \mathrm{Null}(M)$
2. For every $b \in \mathrm{R}^n$, the system of equations $Mx = B^T b$ has a solution.

To prove part 1. of Theorem 4.1, we first consider that $\mathrm{Null}(B) \subset \mathrm{Null}(M)$ (for every $Bx = 0$, $B^T C B x = 0$ also holds). On the other hand, for $x \in \mathrm{Null}(M)$:

$$0 = \langle x, B^T C B \rangle = \langle Bx, CBx \rangle$$

As C is positive definite, we conclude $Bx = 0$, and hence $x \in \mathrm{Null}(B)$. This holding for all x implies $\mathrm{Null}(B) = \mathrm{Null}(M)$.

To prove part 2., we note that the equation is solvable if $B^T b \perp \mathrm{Null}(M^T)$ (by the Fredholm alternative). As M is symmetric, for any $x \in \mathrm{Null}(M^T) = \mathrm{Null}(M) = \mathrm{Null}(B)$, we have:

$$\langle B^T b, x \rangle = \langle b, Bx \rangle = 0$$

which proves the assertion.

In a practical setting, we may modify the equation to have a unique solution. We do this by choosing one of the nodes as a reference, with all potentials being relative to it. This choice is imposed either by setting that voltage to zero, or by dropping the corresponding column in the B matrix.

4.2 Implementation

Building the network To allow the library end-user to easily build a network, both Line and Node classes were implemented. The intended workflow is:

```
Node node_1;
Node node_2;
Node node_3;
Node node_4;
Node node_5;

double u_1 = 4;
double u_2 = -2;

Line line_1(&node_1, 1, 0,   &node_4);
Line line_2(&node_2, 1, 0,   &node_3);
Line line_3(&node_3, 1, u_1, &node_4);
Line line_4(&node_1, 1, u_2, &node_2);
Line line_5(&node_4, 1, 0,   &node_5);
Line line_6(&node_5, 1, 0,   &node_2);

Line* line_arr[] = {&line_1, &line_2, &line_3, &line_4, &line_5, &
    line_6};

\\ Constructor inputs: (line_arr, n_lines, n_nodes, n_reference)
Network main_network(line_arr, 6, 5, 4);

Matrix x(5, 1);
main_network.solve(x);

Matrix y(6, 1);
main_network.map(x, y);
```

First the nodes are defined. Then lines are constructed, from a starting node, the line impedance, the imposed voltage difference (if any) and the ending node. These lines are then combined into an array of lines, which along with the number of lines and of nodes, is used to construct the network. Calling `network.solve(x)` with an empty vector `x` is the solves for the current and fills x. If the line currents are also desired, these can found by mapping from the voltage using `network.map(x, y)`.

Logic implementation To solve for the voltage (and potentially current), the B and C matrices must be constructed, as well as the p vector. These are all constructed by iterating over the `line_arr`, which holds pointers to the Line instances, which

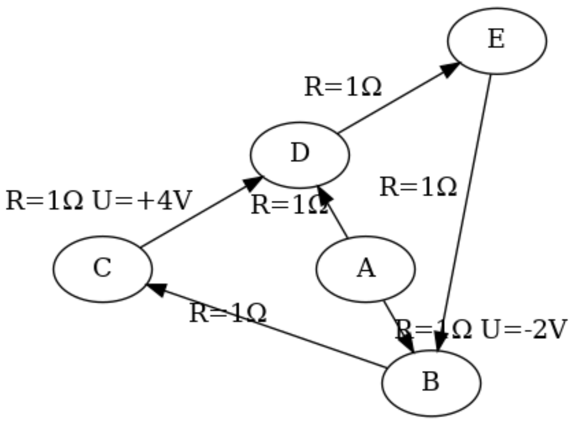

Figure 3: Example electrical network visualization generate automatically using graphviz.

themselves hold pointers to the nodes.

Note that, in this implementation, no special destructors, move or copy operators are needed. This is because all instances are automatically deleted by the compiler when they go out of scope.

One caveat to note is that, to build the B matrix, one has to number the nodes, to be able to connect each node to a column. For this, `int GLOBAL_n_nodes = 0;` variable is defined in `node.cpp`, which is then assigned to the node, and incremented each time a constructor is called.

Assuming the nodes are numbered from `0` to `n_nodes` (the input parameter to the network specifying the number of nodes) works only if just one network is built. However, even just running the unit test will build a network. Hence, we need to assume the nodes are number from `GLOBAL_n_nodes - n_nodes` to `n_nodes`. To access this in the `network.cpp` file, this variable was declared as `extern` in `node.h`, which was then (indirectly) included in `network.cpp`.

Graph visualization For the user to verify that they have correctly implemented a problem, a visualisation option was setup. The function `.visualize(const std::string filename)` exports the graph structure to a 'network.dot' file and then uses graphviz (must be available on the target system) to create a png file with the given filename. An example network is shown in Figure 3.

4.3 Code Verification

To verify that the algorithm described in Subsection 4.1 has been properly implemented, we follow an example presented on page 43 of [2], and compare the results.

This example consists of the network shown in Figure 1 (and Figure 3) with all resistances $R_j = 1$. By choosing the fifth node as the reference node (and hence dropping the fifth column from B), we find:

$$B^TCB = B^TB = \begin{bmatrix} 2 & -1 & 0 & -1 \\ -1 & 3 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ -1 & 0 & -1 & 3 \end{bmatrix} \quad \text{and} \quad B^TCb = B^Tb = \begin{bmatrix} 2 \\ -2 \\ -4 \\ 4 \end{bmatrix}$$

Solving this system of equations, and mapping to the currents using Equation 1, we find:

$$x = \begin{bmatrix} 1 \\ -1 \\ -2 \\ 1 \end{bmatrix} \quad \text{and} \quad y = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

This verification test was implemented as a unit test. Note that the order of the x and y vectors may differ with the chosen ordering of the nodes and lines.

5 Testing and Documentation

5.1 Unit and integration testing

To improve the development speed and final code quality, the test driven development approach was followed, with the Doctest unit testing framework. This implies that, while coding major functionality, a unit test was also created. This unit test is then a de-facto requirement for what the code should do, and hence indicates when a function is appropriately implemented. Another advantage is that this framework indicates when a breaking change occurs.

Doctest was chosen specifically for two main reasons: its speed and running type and the location where it's written. The speed and runtype enable the unit tests to be performed at runtime, at a programmatically chosen time (probably before any application code). This ensures that any breaking changes are found out instantly, and don't require the developer to remember to run the unit tests (as one might do just before committing changes). Note that if a certain file isn't linked into the final executable (as there is no function that directly or indirectly calls it from main), then the unit tests in that file won't run either, ensuring no irrelevant unit tests are run.

The other special feature of Doctest is that the tests are written right into the source files. This means no extra files are needed, and the tests can be written right next to the function definition. By reducing the barrier to writing good unit tests, this framework simplifies test driven development. As a result, this project has 12 unit tests with 60 total assertions. This approach was highly beneficial, especially when refactoring, especially when learning a completely new language.

5.2 Gitlab CI pipeline

Version control and Continuous Integration (CI) pipelines are invaluable tools for more advanced development projects. Using git, for version control, ensure that there is always a working version of the code available, and if used with a hosting service (such as Github or Gitlab) the code is also backup and can be easily shared publicly. Moreover, collaboration on larger is really only feasible using a hosted version control system. The CI pipeline is automatically performing actions on the code, such as building it, checking for memory leaks and even measuring code quality. This project uses Gitlab to run the pipeline, defined in `.gitlab-ci.yml`.

The CI pipeline is run inside of a docker container, with an image built with CMake, gcc, valgrind and more. First, in the build stage, CMake is installed, it is run to generate the Makefiles, then it attempts to build the code, saving the final binary as an artifact. This artifact can both be downloaded by the user and be accessed by the next step of the pipeline. The next three parallel steps are running the unit tests (as doctest is used, this simply consists of running the executable), linting using cpplint, and memory checking using valgrind.

```
# Based on a template from:
# https://docs.gitlab.com/ee/development/cicd/templates.html

image: danger89/cmake


stages:
```

```yaml
  - build
  - test

build:
  stage: build
  script:
    - mkdir build
    - cd build
    - cmake ..
    - cmake --build .
    - ls
  artifacts:
    paths:
      - build/demo_code/demo_exec


unit-test-job:
  stage: test
  script:
    - ./build/demo_code/demo_exec

valgrind-job:
  stage: test
  script:
    - valgrind --leak-check=full  ./build/demo_code/demo_exec

lint-job:
  stage: test
  script:
    - cpplint electrical_network/src/ElectricalNetwork/*
   electrical_network/lib/linalg/src/LinAlg/*
```

Listing 4: ../CppScientificComputing/.gitlab–ci.yml

5.3 Documentation

To document the code, the Doxygen style was used, combined with inline comments, which include rough derivations when appropriate. As an example, consider the `solve_upper_triangular` function:

```cpp
/*!
* This function solves a linear system Ax=b for an upper triangular
   matrix A.
```

```cpp
*
* @param m_A: The A matrix (Must be upper triangular, won't be
  modified)
* @param m_b: The b matrix (won't be modified)
* @param x: (a reference to ) the vector in which the result is to
  be stored.
*/
void solve_upper_triangular(const Matrix& m_A, const Matrix& m_b,
  Matrix& x)
{
  // As the matrix is upper triangular, we shall start iterating
  from the bottom.
  for (int row=m_A.n_rows-1; row>=0; row--)
  {
    // The equation we need to solve is:
    // a_{row, row} x_{row} + a_{row, row+1} x_{row+1} + .. a_{row,
    n} x_{n} = b_{row}
    // Hence we first assign x_{row} = b_{row}
    x(row, 0) = m_b.read(row, 0);

    // Then subtract all a_{row, j} x_{j} for j in (row+1, n)
    for (int col=row+1; col<m_A.n_rows; col++)
    {
      x(row, 0) -= m_A.read(row, col) * x(col, 0);
    }
    // And finally divide by a_{row, row}
    if (m_A.read(row, row) == 0){
      throw Exception( "Matrix not uniquely solvable.");
    }
    x(row, 0) = x(row, 0) / m_A.read(row, row);
  }
}
```

The advantage of using Doxygen style documentation is that every modern C++ editor will render it when asked, for any instances of the function, speeding up development. Another advantage is that the documentation is right in the code, making it easier to keep it up to date and relevant, while still maintaining the option to export it into a documentation pdf or html.

6 Conclusion and Recommendations

Recommendations Depending on the application certain improvements can be made to improve the value and trustworthiness of the two libraries. For example, if one wants to use the linear algebra class to repeatedly solve $Ax = b$ for different b, it would be valuable to implement LU decomposition. Other improvements are more general, such as increasing the number of unit tests, implementing different storage options (such as explained in 3.3) or implementing templating for the Matrix data type. Another example would be to add more constructor classes (currently implemented are diagonal, Gaussian random, and identity matrices, as well as linear spacing vector). While most error handling code in this project used assertions(with the one exception being the 'Matrix is singular' exception), as they were not recoverable, implementing a more rigorous error handling framework is crucial to appropriately handling the application state as the codebase grows. If the library is to be used in a parallelized environment, implementing semaphores for every function that reads/writes to the array will be required to avoid race conditions.

Conclusion This report outlined the development process of a scientific computing project. Much focus was placed on the process, the choices and trade-offs performed, as opposed to the finished product. This involved considerations of setting up a productive and scalable development process, reducing the barriers to writing good documentation, effective unit tests, discovering memory leaks, and adherence to coding standards.

The main visible outcomes from this course are two libraries, for electrical network and linear algebra. However, the author also believes to have made an effective first attempt at learning C++, and improved on their coding practices. Most importantly, is that the code is of quality code, that it is unit tested, well structured, devoid of memory leaks, and developed using a scalable framework.

A Folder structure

The Folder structure is as follows:

```
CppScientificComputing/
|-- electrical_network/
|   |-- lib/
|   |   `-- linalg/
|   |       |-- src/
|   |       |   |-- LinAlg/
|   |       |   |   |-- linalg.h
|   |       |   |   |-- matrix.cpp
|   |       |   |   |-- matrix.h
|   |       |   |   |-- matrix_constructors.cpp
|   |       |   |   |-- matrix_constructors.h
|   |       |   |   |-- vector_constructors.cpp
|   |       |   |   `-- vecotr_constructors.h
|   |       |   `-- doctest.h
|   |       |-- CMakeLists.txt
|   |       `-- README.md
|   |-- src/
|   |   |-- ElectricalNetwork/
|   |   |   |-- network.h
|   |   |   |-- network.cpp
|   |   |   |-- lines.h
|   |   |   |-- lines.cpp
|   |   |   |-- nodes.h
|   |   |   `-- nodes.cpp
|   |   `-- doctest.h
|   |-- CMakeLists.txt
|   `-- REAMDE.md
|-- demo_code/
|   |-- CMakeLists.txt
|   `-- main.cpp
|-- CMakeLists.txt
|-- .gitlab-ci.yml
`-- README.md
```

Listing 5: Folder Structure

B Code Samples

Matrix class header The following code snippet consists of the Matrix class definition in the header file `matrix.h`, located at CppScientificComputing/electrical_network/lib/linalg/src/LinAlg/matrix.h

```
class Matrix
{
private:
  // Related to pushing data into the array.
  int mInsertIndex = 0;
  void push_new_value(const double& a);
  void create_filled_matrix(int n_rows, int n_cols, double value);
  double** p_arr;
public:
  int matrix_number;

  // As opposed to the size, this will be updated when you transpose
    the matrix.
  int n_rows;
  int n_cols;

  // Constructors
  virtual explicit Matrix(int size);
  virtual Matrix(int rows, int columns);
  virtual Matrix(int rows, int columns, double val);
  virtual Matrix(int rows, int columns, double ** source_arr);

  // Copy overloading & assignement
  virtual Matrix(const Matrix &obj);
  virtual Matrix& operator=(const Matrix& m);

  // Examples of move constructor and assignment operator.
  virtual Matrix(Matrix&& m ) noexcept ;
  virtual Matrix& operator=(Matrix&& h) noexcept;

  // Destructor
  virtual ~ Matrix();

  // General helper functions
  void print();

  virtual void swap_rows(int row_num_a, int row_num_b);

  virtual double& get(int row, int col);
  virtual double read(int row, int col) const;

  virtual Matrix transpose();

  // These overloadings are all element-wise.
  Matrix operator + (const Matrix& m2);
  Matrix operator + (const int &val);
  Matrix operator + (const double &val);
  Matrix operator - (const Matrix &m2);
  Matrix operator - (const int &val);
  Matrix operator - (const double &val);
  Matrix operator * (const Matrix &m2);
  Matrix operator * (const int &val);
  Matrix operator * (const double &val);
  bool operator == (const Matrix& m2);
  bool operator != (const Matrix& m2);

  // Special accessing mechanism
  double& operator() (int i, int j);

  // Matrix multiplication
  Matrix operator & (const Matrix& m2) const;

  // Special inserting option.
  Matrix& operator << (const double& val);
  Matrix& operator , (const double& val);

  bool almost_equal(const Matrix& m2, const double& tol=1e-15);

  // The overloading of '/' is such that you can solve Ax=b by
    typing x = A/b.
  // Hence the '/' operator solve a linear system.
  Matrix operator / (const Matrix& m2) const;

  virtual void drop_column(int col);
  virtual void drop_row(int row);
  virtual void add_row(int row);
  virtual void add_column(int col);
};
```

Listing 6: matrix.h

Gaussian Elimination with Pivoting The following is a code snipped from the Matrix.cpp file, and contains the functions required for Gaussian Elimination with pivoting. For completeness, the `operator()` overloading is also added.

```
/*!
* Solves, using Gaussian Elimination with pivoting,
* the system Ax=b for x, and returns x.
*
* @param A: The Matrix
* @param b: The Vector
* @return Matrix x.
*/
Matrix solveGE(const Matrix &A, const Matrix &b)
{
  // Quickly checking that we have the right sizes.
  assert (A.n_cols == b.n_rows);
  assert (b.n_cols == 1);
  assert (A.n_rows == A.n_cols);

  // Given that we don't want to modify the original matrices, we
    need to make the
  // modifications on copies of the matrices:
  Matrix m_A(A);
  Matrix m_b(b);

  // Then we reduce the system to upper triangular,
  ReduceToUpperTriangular(m_A, m_b);

  // and solve the upper triangular system.
  Matrix x(A.n_cols, 1);
  solve_upper_triangular(m_A, m_b, x);

  return x;
}


/*!
* This function solves a linear system Ax=b for an upper triangular
   matrix A.
*
* @param m_A: The A matrix (Must be upper triangular, won't be
   modified)
* @param m_b: The b matrix (won't be modified)
* @param x: (a reference to ) the vector in which the result is to
   be stored.
*/
void solve_upper_triangular(const Matrix& m_A, const Matrix& m_b,
   Matrix& x)
{
  // As the matrix is upper triangular, we shall start iterating
   from the bottom.
  for (int row=m_A.n_rows-1; row>=0; row--)
  {
    // The equation we need to solve is:
    // a_{row, row} x_{row} + a_{row, row+1} x_{row+1} + .. a_{row,
   n} x_{n} = b_{row}
    // Hence we first assign x_{row} = b_{row}
    x(row, 0) = m_b.read(row, 0);

    // Then subtract all a_{row, j} x_{j} for j in (row+1, n)
    for (int col=row+1; col<m_A.n_rows; col++)
    {
      x(row, 0) -= m_A.read(row, col) * x(col, 0);
    }
    // And finally divide by a_{row, row}
    if (m_A.read(row, row) == 0){
      throw Exception( "Matrix is singular");
    }
    x(row, 0) = x(row, 0) / m_A.read(row, row);
  }
}

/*!
* Reduces [A, b]  using row Elementary transformations to A upper
   triangular.
* @param A
* @param b
*/
void ReduceToUpperTriangular(Matrix& A, Matrix& b){
  // This function iterates over rows and columns (sub-diagonal
   entries only). For a given
  // (_row, _col), it performs the operation over the entire row,
   iterating over _icol.

  // holder variable for the row reduction;
  double _h_reductions;
```

```

// holder variables for the partial pivoting.
int _h_index_max;
double _h_value_max;

// Don't need to perform any calculations on the final column,
 hence until A.n_cols-1
for (int _col=0; _col<A.n_cols-1; _col++)
{
  // ----------------------------------------
  // --------- PARTIAL PIVOTING -------------
  // ----------------------------------------
  _h_index_max = _col;
  _h_value_max = A.read(_col, _col);

  for (int _irow=_col+1; _irow<A.n_rows; _irow++){
    // We look for the index containing the largest value.
    if (A.read(_irow, _col) > _h_value_max)
    {
      _h_value_max = A.read(_irow, _col);
      _h_index_max = _irow;
    }
  }
  // Then we swap the rows;
  A.swap_rows(_col, _h_index_max);
  b.swap_rows(_col, _h_index_max);


  // ----------------------------------------
  // --------- ROW REDUCTION -------------
  // ----------------------------------------
  for (int _row=_col+1; _row<A.n_rows; _row++)
  {
    _h_reductions = A.read(_row, _col) / A.read(_col,_col);

    if (A.read(_row, _col) == 0){
      continue;
    }

    for (int _icol=_col; _icol<A.n_cols; _icol++)
    {
      A(_row, _icol) -= _h_reductions*A.read(_col, _icol);
    }
```

```
      b(_row, 0) -= _h_reductions* b(_col, 0);

    }
  }
}
```