This information is to be attached to all scripts of candidate number 1056127

Candidates with Specific Learning Difficulties

This candidate has been diagnosed with a Specific Learning Difficulty ('SpLD' e.g. dyslexia, dyspraxia, dysgraphia, working memory deficit and attention deficit (hyperactivity) disorder (AD(H)D)).

Please mark the script or submitted work as it stands, but indicate to yourself in your notes that the candidate has a SpLD and record any factors you consider to have a particular bearing on his or her performance. The Board will later consider what account to take of the candidate's condition when adjudicating his or her classification.

'Specific Learning Difficulties' is an umbrella term given to a range of conditions which affect a person's ability to learn. These are commonly characterised by impaired concentration and problems with information processing and recall, and may also cause difficulties with reading, writing and spelling.

Candidates with these conditions may be awarded extra time in examinations to allow them to read the examination paper slowly, consider their responses to the questions, plan out their work, and read it over at the end.

Amongst Oxford students the most common of these disorders is dyslexia, the symptoms of which include:

- omission, repetition, transposition or substitution of words or punctuation
- particular difficulty in interpreting the question
- simplified vocabulary and language structure (to avoid making errors)
- spelling and grammatical errors
- errors in sentence structure, word ordering and organisation
- poor or immature handwriting
- poor short-term memory
- particular difficulties generalising, or acquiring and applying rules

Students with dyslexia often think in non-verbal, non-linear patterns, with the result that their work may appear disjointed. It is recommended that examiners first read the work through quickly in order to obtain an initial sense of the candidate's overall argument and understanding of the question.

Examiners should discount errors in spelling, grammar and sentence structure as these are considered to derive from the candidate's disability (though this does not apply in examinations where to do so would compromise the academic standards of the assessment, or where fitness to practise regulations apply). This is the case regardless of whether candidates have opted to take their examinations with extra time. Where a wordprocessor is used, the spelling and grammar checks are enabled.

Examiners should not make extra allowance for remaining deficiencies in planning, content and logical argument, as this would constitute double compensation (even if extra time has not been taken).

Form Code No.2D

A Class for Matrix Operations in C++

Special Topic: C++ for Scientific Computing

Candidate Number: 1056127

Contents

1 Introduction

In many branches of science, the effective numerical application of theories and their solutions depend on solving systems of linear equations. This has led to the need for efficient implementation of numerical linear algebra methods so that large scale problems can be solved in a practical time frame. The theoretical construction of these algorithms and consequently the number of floating point operations per second (FLOPs) have a great impact on their efficiency, however, significant speed increases can be gained from the choice of programming language and the way that the code is executed by the computer.

Many high-level languages, such as MATLAB, Maple, or Python's NumPy, provide seamless integration of matrices and vectors, along with the many linear algebra techniques that involve them. In most cases, these high-level languages achieve this integration by acting as "wrappers" for linear algebra packages written in low/mid-level languages, as opposed to implementing the algorithms natively in their own syntax. The languages listed above heavily rely on the FORTRAN90 linear algebra libraries LAPACK and BLAS [1]. These libraries have the benefit of being highly optimised, usually for the specific machine architecture they are being executed on; something that is not easily facilitated when working with higher-level languages due to the lack of control of specific hardware features such as CPU caching and memory hierarchy [2].

Our aim is to implement a framework for matrix and vector algebra in C++, a mid-level object-oriented language. The focus will not be on the high performance computing techniques described in the previous paragraph, and instead on the benefits of compiled languages and object-oriented programming (OOP): two definitive features of C++. In particular, how by creating a matrix class structure we build up a wealth of functionality using just the standard C++ operations, without the need for external libraries.

This report serves as a supplement that is complementary to our project as a whole, and its goal is to provide an outline of OOP, our implementation of a matrix class structure, and how we used this structure to build a library of linear algebra algorithms. These algorithms can be applied to many of the problems that occur in scientific studies; including finding the determinant, eigenvalue problems and solving linear systems. We encourage the reader, whilst reading this report, to also glance at the examples given in the source code and the class documentation created for the matrix class, so that a more complete view of our work can be had. These materials are all available on our GitHub and can be accessed by clicking here.

2 Object-oriented programming

In our implementation of a linear algebra package, we make heavy use of OOP to structure our code into simple, readable and reusable pieces. The core idea behind OOP is the definition of classes and objects:

Definition 2.1 (Class [3])**.** A *class* is the formula or blueprint for creating *objects*; they are essentially user defined data types. A class has *attributes* which store information about an instance of the class, though the values of the attribute are not assigned within the class. A class may also contain *methods* which are functions that may only be used on objects of that type.

Definition 2.2 (Object [3])**.** An *object* is an individual instance of a *class*, meaning that the attributes have been initialised and may be assigned specific values related to that object. It has access to the attributes and methods defined by the class.

Along with these definitions, there exist four main concepts related to OOP: Inheritance, Encapsulation, Abstraction, and Polymorphism [4]. They enable code and programs to be safer, more concise, scalable, and user friendly.

Inheritance is an idea that supports reusability of code. It is a procedure in which a *derived class* inherits the attributes and methods of a *base class*. Later, we shall use this feature to create a derived class of vectors from our matrix class.

Encapsulation is the concept of containing all important information about an object within the object itself, hence where ever the object "goes" so too does the information about it. This not only makes keeping track of important information easier, it also allows the programmer to control what information is available to the end user. In C++, this is achieved by the use of *access specifiers* to declare attributes and methods as either *private*, *protected*, or *public*. Encapsulation promotes safe programming as it can prevent end users from accessing or changing variables that might negatively effect the program (i.e. bugs or crashes).

Abstraction can be defined as capturing only those details about an object or class that are relevant to the user's perspective. To promote clarity, we may only want to "show" information to the user that is important to them and hide an unnecessary information and complexity.

Polymorphism means the reuse of code, functions and operators on a variety of different objects. Polymorphisms frequently occur in our code through the use of function overloading, which allows the same function to accept a multitude of input types. For example, we

overload the multiplication operator " $*$ " to accept two matrix objects and perform matrix multiplication.

Lastly, we would like to give a brief explanation as to why C++ was chosen above other languages such as C, FORTRAN, or Python. The first reason is that C++ is fast. It allows for a level of control over computer hardware that many higher level languages like Python do not, which in turn allows a programmer to create code that is better optimised for that hardware. As mentioned in the previous section, it is a compiled language, meaning code is converted to machine code before runtime. This reduces the amount of "extra" operations a processor has to do to run the code a programmer has made, thus increasing the speed of execution. Secondly, C++ has a wealth of OOP functionality that we wish to take advantage of, whereas other languages such as C or FORTRAN are somewhat lacking in this area. These factors along with wide range of numerical libraries already available within C++ make it an attractive choice, hence why it is ours.

3 A matrix class

The foundation of our numerical linear algebra library is the matrix class, with it we can declare matrix objects and functions which act upon them. The attributes, in a mathematical sense, of a matrix: the number of rows, the number of columns, and the values of the elements. These are what become the *protected* attributes of our matrix class, meaning they may only be access from within the class itself, by *friend* functions of the class, and by derived classes such as the vector class which we shall discuss in due course.

The attributes are defined like so

```
const int mRow; const int mCol; double* mData;
```

where we have chosen to declare `mRow` and `mCol` as `const int` as we wish them to be integers that remain unchanged after they are initialised with values. The attribute `mData` is declared as a *pointer*, once the constructor is called, it will point to a memory address of a 1-D array of length `mRow * mCol` that contains the elements of the matrix. There is a simple correspondence between an element's position in a matrix and its position in the array. Recalling that arrays in C++ are 0-indexed, hence the presence of "-1", this relation is given by

$$a_{i,j} = \texttt{mData[mRow * (i-1) + (j-1)]}. \tag{1}$$

The decision to store all elements in a 1-D array rather than a 2-D was made for simplicity, to avoid the use of *double pointers* and to encourage us, the developer, to use "safe" *setters* and *getters* rather than working with the array directly.

3.1 Constructors and destructors

To allocate and deallocate memory for the data of our matrix class, we require constructors and destructors. These do, as one might expect, construct a matrix object when we require one, and destruct the object when we no longer require. Our class has two distinct constructors, the first initialises an array of zeros when given integer inputs of the number of rows and columns desired. The second is the copy constructor, it takes and input of another matrix object and then initialises a new object which is a copy of the input.

Both constructors dynamically allocate memory using the following syntax

```
mData = new double [mRow * mCol];
```

The values of this array are then initialised with either zeros when using the standard constructor, or with the values of the elements in the input matrix when using the copy constructor.

C++ requires manual memory management, meaning that when memory is allocated with `new`, it must also be deallocated with `delete` when it is no longer needed. If this step is forgotten, then problems may arise such as memory leaks or dangling pointers [4]. Fortunately, by implementing a destructor, it is ensured that this happens every time the matrix object "goes out of scope", i.e it is not longer needed. Our destructor simply calls the operator `delete[] mData`, where the square-brackets "`[]`" indicate that all elements in the array should be deleted.

3.2 Setters and getters

As mentioned previously, the attributes `mRow`, `mCol`, `mData` were declared as protected. While we may not want these variables to be accessed and edited directly when working outside the matrix class (or friend functions and derived classes), other parts of our code may still want to "see" what the values of these variables are or in some cases update them. To enable this functionality, we have implemented setters and getters for our protected attributes.

A *setter*, as its name suggests, is a *method* of a class that allows us to set the value of private or protected attributes when outside the class itself. In our case, it is important to be able to change the elements in our matrix, however, we also want to prevent users from incorrectly attempting to access coordinates outside the dimensions of our matrix or the `mData` array. By using a setter, instead directly accessing `mData`, we are able to enforce this condition by presenting an error message if the coordinates entered are outside

the dimensions of our matrix. For `mRow` and `mCol`, we do not create a setter. This adds another layer of protection by ensuring they remain constant after their initialisation.

Getters have been created for all protected attributes so that their value may be retrieved. In particular, for the `mData` getter (and setter), the user can now input the coordinates of the elements to retrieve their values rather than using the relation in Equation (1).

3.3 Arithmetic operators

With a constructor for creating matrices of given dimensions and setters for assigning values to the elements, now we can begin to define operations (or functions) to apply to the matrix object. An obvious first step in this direction are the arithmetic operators. It is at this point that we all see our first use of polymorphisms, in the form of overloading the native C++ arithmetic operators to accept the inputs of matrix objects.

The addition (+), subtraction (−), and multiplication ($\star$) operators were overloaded so that matrix-matrix, matrix-number, and number-matrix operations can be perform with them. We used standard methods to achieve this (for loops to do element-wise addition, subtraction and multiplication, and standard algorithms for matrix multiplication), hence we do not state them explicitly here. The division (/) operator was also overloaded for matrix-number operations.

3.4 Boolean operators

The Boolean operators (== and !=) were also overloaded to allow equality comparison between two matrices. This was achieved by first checking if the dimensions of the two matrices were the same, and then proceeding to check element-wise equality. Overloading these operators proved to be especially useful when testing our code against known results.

3.5 Other functions

A multitude of other functions were created with the matrix class to aid with frequently required operations. These include identity matrix constructor, matrix transpose, getting row and column maxes, printing matrices, and many more. In the next sections, we see that the creation of these functions becomes very useful when implementing more complex linear algebra algorithms for solving linear systems and finding eigenvalues. For a complete list of functions included in our matrix class, we direct the reader to the UML (Unified Modelling Language) class diagram in Appendix A. We have also included some documentation of the matrix which is available via the GitHub link given in the introduction under the file

name: `ref.pdf`. It gives a more complete outline of what each function does within our class.

3.6 The derived vector class

As vectors may be viewed as a special subset of matrices, those with only one row or column, we have elected to make our class of vectors a derived class of the matrix class. This allows us to avoid rewriting the code for functions such as arithmetic operators, transposing, or printing. Instead, we only need to overload a few functions, such as the constructor, and the element setters and getters. What remains are two classes that interact seamless with one another since, at their core, they are the same.

4 Linear algebra algorithms

The main focus of our project was to build, using our matrix class, a library of linear algebra algorithms. In this section, we outline the algorithms we have included and their implementation in pseudocode. It is worth noting that, to be more concise, the pseudocode does not mirror the C++ code exactly, however, the algorithms used are still the same.

4.1 LU decomposition and determinants

The LU decomposition presents itself as a simple starting point when beginning to program linear algebra algorithms since it forms the basis for computing the determinant and inverse of a matrix. It does so by factorising the input matrix $\mathbf{A}$ into a product of a lower triangular matrix, $\mathbf{L}$, and an upper triangular matrix, $\mathbf{U}$. In our code, we implemented LU decomposition (for square matrices) with partial pivoting, this ensures that all square matrices can be factorised and that the algorithm is numerically stable [5]. Thus, the decomposition has the form

$$\mathbf{P}\,\mathbf{A} = \mathbf{L}\,\mathbf{U} \tag{2}$$

where $\mathbf{P}$ is the permutation matrix. By recording the number of rows permuted or exchanged in the decomposition, the determinant can be easily computed using one of the useful properties of triangular matrices: the determinant of a triangular matrix is equal to the product of its diagonals. Therefore, under LU decomposition, the determinant of matrix A is given by

$$\det(A) = (-1)^S \det(L) \det(U) = (-1)^S \left(\prod_i l_{i,i} \right) \left(\prod_i u_{i,i} \right) \tag{3}$$

where S is the number of rows exchanged in the decomposition. Algorithm 1 gives a simplified version of the LU decomposition implemented in our library.

Algorithm 1 LU Decomposition - Finds P, L, U in $PA = LU$

```
procedure [P, L, U] = LU(A)
    int m = A.mRow;                                  ▷ No. of rows A
    Matrix P = eye(m);
    Matrix L = eye(m);
    Matrix U = A.copy();
    for int k = 1 to m do
        int i = argMax(U(k:m,k));
        if i != k then
            swap(P(1:m,k), P(1:m,i));                ▷ Swap rows of P, L, U
            swap(L(k:m,k), L(k:m,i));
            swap(U(k:m,k), U(k:m,i));
        L(k+1:m,k) = U(k+1:m,k) / U(k,k);            ▷ Update elements
        for int j = k to m do
            U(k+1:m,k) = U(k+1:m,j) - L(k+1:m,k) * U(k,j);
    return [P, L, U];
```

4.2 Solving linear systems with LU decomposition

Once the LU decomposition can be computed, solving systems of linear equations becomes a more practicable task due to the ease at which triangular matrices may be inverted. By writing the linear system as

$$\mathbf{PAx} = \mathbf{LUx} = \mathbf{Pb}, \tag{4}$$

the computation can be broken up into two steps. First, solving a system involving only a lower-triangular matrix and the second involving only an upper-triangular. This is achieved by letting $\mathbf{Ux} = \mathbf{y}$ and solving, via forward substitution,

$$\mathbf{Ly} = \mathbf{Pb}. \tag{5}$$

Similarly, once $\mathbf{y}$ is found, the following is solved, now by back substitution,

$$\mathbf{Ux} = \mathbf{y}. \tag{6}$$

Hence, the solution $\mathbf{x}$ is determined. The algorithms for both types of substitution are implemented with relative ease, using our matrix class. In Algorithms 2 and 3 we provide the pseudocode for forward and backward substitution.

Algorithm 2 Forward substitution

```
procedure y = FORWARDSUB(L,c)
    int m = L.mRow;
    Vector y(m);
    for int i = 1 to m do
        double temp = c(i);
        for int j = 1 to i-1 do
            temp -= L(i,j) *
y(j);
        y(i) = temp / L(i,i);
    return y;
```

Algorithm 3 Backward substitution

```
procedure x = BACKWARDSUB(U,b)
    int m = U.mRow;
    Vector x(m);
    for int i = m to 1 do
        double temp = b(i);
        for int j = i+1 to m do
            temp -= U(i,j) *
x(j);
        x(i) = temp / U(i,i);
    return x;
```

4.3 QR factorisation

Trefethen and Bau state in their chapter on the subject that

> "One algorithmic idea in numerical linear algebra is more important than all the others: QR factorisation" [5].

This might suggest that a natural next step in our construction of a linear algebra library is to tackle the QR factorisation. Similarly to the LU decomposition, the factors on their own may not seem awe inspiring, however, once the factorisation is computed it may be used in solving a variety of interesting problems.

Definition 4.1 (QR factorisation)**.** The QR factorisation is the decomposition of an input matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ into the orthogonal matrix $\mathbf{Q} \in \mathbb{R}^{m \times m}$ and an upper triangular matrix $\mathbf{R} \in \mathbb{R}^{m \times n}$ such that $\mathbf{A} = \mathbf{QR}$.

To maintain numerical stability, we implemented the Householder QR factorisation, which uses Householder reflectors to construct orthogonal matrices that when successively applied to $\mathbf{A}$ result in an upper triangular matrix $\mathbf{R}$. The Householder reflector for a vector $\mathbf{x} \in \mathbb{R}^n$ is given by

$$\mathbf{H} = \mathbf{I}_n - 2\,\frac{\mathbf{v}\mathbf{v}^{\mathrm{T}}}{\mathbf{v}^{\mathrm{T}}\mathbf{v}} \quad \text{and} \quad \mathbf{v} = \operatorname{sign}(\mathbf{x}_1)\|\mathbf{x}\|_2\,\mathbf{e}_1 + \mathbf{x} \tag{7}$$

where $\operatorname{sign}(\mathbf{x}_1)$ is the sign of the first element of $\mathbf{x}$ (by convention if $\mathbf{x}_1 = 0$ then $\operatorname{sign}(\mathbf{x}_1) = 1$) and $\mathbf{e}_1 = (1, 0, \ldots, 0)^{\mathrm{T}}$ is the first standard basis vector [5]. By successively finding the householder reflector for each column of $\mathbf{A}$ (from the diagonal down) we obtain the algorithm for QR factorisation. Our implementation is given in Algorithm 4.

Algorithm 4 QR factorisation - Finds Q, R in $A = QR$

```
procedure [Q, R] = QR(A)
    int m = A.mRow;   int n = A.mCol;          ▷ No. of rows and cols A
    Matrix Q = eye(m);  Matrix R = A.copy();
    for int k = 1 to n do
        Vector x = zeros(m,1);   Vector v = zeros(m,1);
        x(k:m) = R(k:m,k);
        v = x;   v(k) = v(k) + sign(x(k)) * norm(x,2);
        if norm(x,2) != 0 then
            Matrix H = eye(m) - 2*v*transpose(v) / norm(v,2)^2;
            R = H*R;                    ▷ Pre-multiply by Householder reflector
            Q = Q*H;                    ▷ Post-multiply by Householder reflector
    return [Q, R];
```

4.4 Least squares with QR factorisation

The first application of the QR factorisation that we explored was for solving least squares problems. The aim is to solve the following problem:

Definition 4.2 (The least squares problem)**.** Given a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ and a vector $\mathbf{b} \in \mathbb{R}^m$ for $m \geq n$, we wish to find the vector $\mathbf{x} \in \mathbb{R}^n$ such that $\mathbf{x}$ minimises $\|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2$. We write the solution to the least squares problem as

$$\hat{\mathbf{x}} = \arg\min_{\mathbf{x}} \|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2 \tag{8}$$

The solution is found by considering the QR factorisation of $\mathbf{A} \in \mathbf{R}^{m \times n}$ where $m \geq n$ and $\text{rank}(\mathbf{A}) = n$. This yields,

$$\overbrace{\mathbf{A}}^{m \times n} = \overbrace{\mathbf{Q}}^{m \times m}\overbrace{\mathbf{R}}^{m \times n} = \begin{bmatrix} \mathbf{Q}_1 & \mathbf{Q}_2 \end{bmatrix} \begin{bmatrix} \mathbf{R}_1 \\ \mathbf{0} \end{bmatrix} = \overbrace{\mathbf{Q}_1}^{m \times n}\overbrace{\mathbf{R}_1}^{n \times n}. \tag{9}$$

The factorisation $\mathbf{A} = \mathbf{Q}_1\mathbf{R}_1$ is referred to as the *reduced* QR factorisation. The least squares is solved by substituting the QR factorisation into Equation (8) and recalling that multiplying a vector by an orthogonal matrix does not change its 2-norm. Hence,

$$\hat{\mathbf{x}} = \arg\min_{\mathbf{x}} \|\mathbf{Q}^{\mathrm{T}}(\mathbf{A}\mathbf{x} - \mathbf{b})\|_2 \tag{10}$$

$$= \arg\min_{\mathbf{x}} \|\mathbf{Q}^{\mathrm{T}}(\mathbf{Q}\mathbf{R}\mathbf{x} - \mathbf{b})\|_2 \tag{11}$$

$$= \arg\min_{\mathbf{x}} \|\mathbf{R}\mathbf{x} - \mathbf{Q}^{\mathrm{T}}\mathbf{b}\|_2 \tag{12}$$

The full system is overdetermined since rank$(\mathbf{R}) = n$, therefore there exists no solution $\mathbf{x}$ where $\mathbf{Rx} - \mathbf{Q}^{\mathrm{T}}\mathbf{b} = \mathbf{0}$. However, by using the reduced QR factorisation, and that $\mathbf{R}_1$ is an invertible upper triangular matrix, the solution to

$$\mathbf{R}_1\mathbf{x} - \mathbf{Q}_1^{\mathrm{T}}\mathbf{b} = \mathbf{0} \tag{13}$$

can be computed using back substitution (or any other linear system solver) [5]. The solution to (8) is then the solution to the least squares problem. This results in a relatively simple implementation, provided functions for QR factorisation and back substitution already exist. The coded solution is given in Algorithm 5.

Algorithm 5 least squares - Finds $\hat{\mathbf{x}} = \arg\min_{\mathbf{x}} \|\mathbf{Ax} - \mathbf{b}\|_2$

```
procedure x = LSQ(A,B)
    int m = A.mRow;   int n = A.mCol;          ▷ No. of rows and cols A
    Matrix Q(m,n); Matrix R(n,n);              ▷ Initialise Matrices Q and R
    [Q,   R] = qr(A, reduced = true);          ▷ Get reduced QR
    Vector x(m) = backwardSub(R, transpose(Q) * b);
    return x;
```

4.5 QR algorithm and eigenvalues for symmetric matrices

The second application of QR factorisation we explore is its use in determining eigenvalues; a problem that occurs frequently in many areas of science and engineering. First, we discuss the simple QR algorithm before considering various adjustments that can be made to improve the computational efficiency and convergence.

A simple QR algorithm

Before presenting what is deemed to be a more practical implementation of the QR algorithm, we consider its simplest formulation. At its core, the QR algorithm exploits the property that the matrix $\mathbf{A} \in \mathbb{R}^{m\times m}$ with QR factorisation $\mathbf{A} = \mathbf{QR}$ is similar to $\mathbf{Q}^{\mathrm{T}}\mathbf{AQ}$, meaning that $\mathbf{A}$ and $\mathbf{Q}^{\mathrm{T}}\mathbf{AQ}$ have the same eigenvalues. Furthermore, we also note that $\mathbf{Q}^{\mathrm{T}}\mathbf{AQ} = \mathbf{RQ}$. Using this property, John Francis, the algorithm's inventor, noticed that if this process is iteratively applied, then the result tends to an upper triangular matrix whose eigenvalues are its diagonal entries [5]. These are also the eigenvalues of $\mathbf{A}$ since the upper triangular matrix is similar to $\mathbf{A}$. This process is best explained in the pseudocode of Algorithm 6.

Algorithm 6 Simple QR - Finds the eigenvalues of symmetric matrix $\mathbf{A}$

```
procedure eigVals = qrAlgSimple(A)
    for int k = 1 to max_iters do
        Matrix Q,  Matrix R = qr(A);          ▷ get QR factorisation of A
        A = R * Q;                            ▷ Recombine in reverse order and update A
    eigVals = diag(A);                        ▷ gets vector of diagonal values of A
    return eigVals;
```

Improvements

The simple QR algorithm suffers from slow convergence and a high computational complexity of $\mathcal{O}(n^3)$ FLOPs per iteration. In particular, the algorithm will not converge if two eigenvalues are very close together [6]. In [5] there are three suggested remedies to these problems:

1. Before each iteration, reduce $\mathbf{A}$ to upper Hessenberg form to reduce computational complexity to $\mathcal{O}(n^2)$.
2. Compute the QR factorisation on shifted matrix $\mathbf{A} - \mu\mathbf{I}$ to improve convergence.
3. If an eigenvalue is found, "deflate" the problem by splitting $\mathbf{A}$ into submatrices.

Reduction to the upper Hessenberg form

Reduction to the upper Hessenberg form entails transforming the matrix $\mathbf{A}$ to an almost upper triangular matrix, or more precisely, to a matrix that has non-zeros only in the upper triangle and the first subdiagonal. The reduction is achieved in a similar fashion to the QR factorisation, by successively applying Householder reflectors. This results in an upper Hessenberg matrix $\mathbf{H}$ remains similar to $\mathbf{A}$, and thus they have the same eigenvalues. $\mathbf{H}$ may then be used in the QR algorithm, which reduces the computational complexity of the algorithm from $\mathcal{O}(n^3)$ to $\mathcal{O}(n^2)$.

Choosing an effective shift

In an effort to improve the QR algorithm's rate of convergence, we shift the matrix $\mathbf{A}$ at each iteration. This results in calculating the QR factorisation of the matrix $\mathbf{A} - \mu\mathbf{I}$. Thus, the main steps in Algorithm 6 become

```
mu = chooseShift(A);
Matrix Q,  Matrix R = qr(A - mu * eye(m));
A = R * Q + mu * eye(m)
```

where the variable `m = A.mRow`, i.e. the dimensions of $\mathbf{A}$.

Lastly, the question arises: how to choose the shift effectively? To which the answer is: the *Wilkinson shift*. It uses the last two-by-two submatrix on the diagonal, given by

$$\mathbf{A}_{m-1:m,m-1:m} = \begin{bmatrix} a_{m-1,m-1} & a_{m-1,m} \\ a_{m,m-1} & a_{m,m} \end{bmatrix} = \begin{bmatrix} a & b \\ b & c \end{bmatrix}$$

The formula for the Wilkinson shift, which gives an estimate of the eigenvalue, is given by

$$\mu = c - \frac{\text{sign}(\delta)\, b^2}{|\delta| + \sqrt{\delta^2 + b^2}} \tag{14}$$

where $\delta = (a - c)/2$ and we specify that $\text{sign}(0) = 1$.

Deflation

The deflation is implemented by reducing the size of the matrix $\mathbf{A}$ once an eigenvalue has been found. The criterion for this is when the last subdiagonal $a_{k,k-1}$ is sufficiently close to zero, where $k = \dim(\mathbf{A})$; the current dimension of $\mathbf{A}$. When this occurs, then one eigenvalue is approximately known to equal $a_{k,k}$. This allows us to use the $(k-1) \times (k-1)$ principle minor of $\mathbf{A}$ to find the other eigenvalues. Each time an eigenvalue is found, the number of operations per iteration decreases since calculations are being performed on smaller and smaller matrices.

The practical QR algorithm

In Algorithm 7, we present the QR algorithm used in our C++ code, some simplifications have been made in the way matrix objects are overwritten and reassigned to keep the pseudocode concise.

Algorithm 7 Practical QR - Finds the eigenvalues of symmetric matrix **A**

```
procedure eigVals = qrAlg(A, tol = 1e-10)
    int m = A.mRow;
    Vector eigVals(m);   Matrix A_old = A.copy();
    for int k = m to 2 do
        Matrix A = hessenburgReduction(A_old(1:k,1:k));
        while abs(A(k,k-1)) > tol do          ▷ Check eigenvalue convergence
            double mu = wilkinsShift(H);
            Matrix Q,   Matrix R = qr(A - mu * eye(k));
            A = R * Q + mu * eye(k);
        eigVal(k) = A(k,k);                   ▷ Assign converged eigenvalue to eigVal
        A_old(1:k,1:k) = A;     ▷ Save A to A_old for deflation in next iteration
    eigVal(1) = A_old(1,1);
    return eigVals;
```

5 Tests and Examples

5.1 Tests

To check the validity and accuracy of our matrix class, we created a set of functions to test individual aspects of it, such as operators and helper functions. This involved comparing the results of our programmed functions with hard-coded known results. These tests can be found in the "`MatrixTest.cpp`" file, and may called using the *namespace* "`MatrixTests::`". While we shall not explicitly list them here, we can confirm that all of our functions past our basic tests. However, we note that the tests we employed were in no way exhaustive, hence, we wish to suggest some improvements going forward.

If this numerical linear algebra library was to be developed further, we would recommend the use of a standardised unit testing framework such as Google Test or CTest. This would allow for a more formulaic style of testing which we would expect to be more robust and scalable. A more efficient testing style, instead of using hard-code results, would perhaps have been to use one of the many linear algebra libraries than already exist in C++, such as Armadillo or Eigen. With these libraries, we would have been able to generate far more tests with ease, particularly when working with larger matrices.

5.2 Examples

For the algorithms discussed in Section 4, we created another namespace: "`Examples::`". Within this namespace, there exists an example for each of algorithm, their purpose is to illustrate how to use these algorithms within our framework. We encourage the reader to view the all examples in the file "`Examples.cpp`" to gain a complete view of what our library is capable of. In Figure 1, we have included the example made for our "`lsq`" function, used to solve least squares problems.

```
/* Find the solution to the least squares problem
  min ||A x = b||_2 where
          [ 2   1  -1] [x1]   [1]
          [-3  -1   2] [x2]   [2]
          [-2   1   2] [x3] = [3]
          [ 5   7  13] [x4]   [4]
          [10  20  30] [x5]   [5]      */
  // Declare matrix A
  Matrix A(5,3);
  A(1,1) =  2; A(1,2) =  1; A(1,3) = -1;
  A(2,1) = -3; A(2,2) = -1; A(2,3) =  2;
  A(3,1) = -2; A(3,2) =  1; A(3,3) =  2;
  A(4,1) = -5; A(4,2) =  7; A(4,3) = 13;
  A(5,1) = 10; A(5,2) = 20; A(5,3) = 30;
  // Declare vector b
  Vector b(5);
  b(1) = 1; b(2) = 2; b(3) = 3; b(4) = 4; b(5) = 5;
  // Find Least squares solution
  Vector x(3);
  x = lsq(A, b);
  Output: x = [ -0.331 ]
              [  0.319 ]
              [  0.060 ]
```

Figure 1: The example of the `lsq` function from the `Examples.cpp`.

6 Conclusion

In this report, we began by introducing object-oriented programming (OOP), defining its two main structures, classes and objects, and outlining its four key ideas: inheritance, encapsulation, abstraction, and polymorphism. We also discussed why we elected to use the programming language C++ for our project, the main reasons being; speed of execute due to being a compiled language and native OOP integration.

In Section 3, we presented our matrix class, which is the foundation of our numerical linear algebra library. The importance of constructors and destructors was discussed, as well as, how we ensure correct memory management. We explained the need for setters and getters to ensure control and safety when information is exchanged between our program and the end user, embodying the idea of encapsulation. We further discussed our use of function overloading of some standard arithmetic operator to incorporate matrix objects, allowing for a more native syntax to be used.

The main focus of our project was described in Section 4, where we present the mathematical underpinnings of the numerical linear algebra algorithms we were able to implement using our matrix class. We provide explanations and pseudocode for algorithms such LU Decomposition, QR Factorisation, and the methods that are derived from them. In the tests and examples section, we discussed the use of a testing namespace to check that our library was operating correctly. We also acknowledge that many improvements could be made in this area, our recommendation being that in the future the use of unit testing framework and comparison to standard linear algebra libraries would help improve the robustness and accuracy of our code.

The code created through the course of this project still invites many improvements, particularly when it comes to structure and scalability. However, we believe it provides a concrete example, especially to those who have only used high-level languages such as MATLAB or Python, of how to create a scientific computing library from the ground up. We hope that it encourages more students, researchers, and academics to do the same.

References

[1] LAPACK, *LAPACK Frequently asked questions*. Netlib.org, 2008. Available at http://www.netlib.org/lapack/faq.html.

[2] K. Goto and R. A. v. d. Geijn, "Anatomy of high-performance matrix multiplication," *ACM Transactions on Mathematical Software (TOMS)*, vol. 34, no. 3, pp. 1–25, 2008.

[3] *Learning Object-Oriented Programming [electronic resource]*. 1st edition ed., 2015.

[4] J. Pitt-Francis and J. Whiteley, *Guide to scientific computing in C++*. Undergraduate topics in computer science, second edition. ed., 2018.

[5] L. N. Trefethen and D. Bau, *Numerical linear algebra*. Philadelphia: SIAM, 1997.

[6] P. Arbenz, "Lecture notes in numerical methods for solving large scale eigenvalue problems," January 2018.

A UML diagram of the Matrix class

Please turn to the next page to see the diagram.

Matrix
double* mData
const int mRow
const int mCol
const int mSize
—- Constructors and Destructors ————-
+ Matrix(int row, int col)
+ Matrix(int n)
+ Matrix(const Matrix& m)
+ ~Matrix()
—- Setters and getters ———————
+ double& getValue(int row, int col) const
+ void setValue(int row, int col) const
+ int getNumRow() const
+ int getNumCol() const
—- Operators —————————
+ friend Matrix operator+(const Matrix& m1, const Matrix& m2)
+ friend Matrix operator+(const Matrix& m, const double& a)
+ friend Matrix operator+(const double& a, const Matrix& m)
+ friend Matrix operator-(const Matrix& m1, const Matrix& m2)
+ friend Matrix operator-(const Matrix& m, const double& a)
+ friend Matrix operator-(const double& a, const Matrix& m)
+ friend Matrix operator*(const Matrix& m1, const Matrix& m2)
+ friend Matrix operator*(const Matrix& m, const double& a)
+ friend Matrix operator*(const double& a, const Matrix& m)
+ friend Matrix operator/(const Matrix& m, const double& a)
+ Matrix& operator=(Matrix& m)
+ friend Matrix& operator-(Matrix& m)
+ friend bool operator==(const Matrix& m1, const Matrix& m2)
+ double& operator() (int i, int j)
+ friend bool operator()(const Matrix& m1, const Matrix& m2)

Figure 2: The UML diagram outlines the structure of the matrix class, if first box are the attributes. In the second the methods, member, and friend functions are given. The symbol preceding each entry defines if it is private (-), protected (#), or public (+).

Matrix

—- Helpers ————————————

```
+ double rowMax(int row, int col = 0, bool abs_true = true) const
+ double colMax(int col, int row = 0, bool abs_true = true) const
+ int argRowMax(int row, int col = 0, bool abs_true = true) const
+ int argColMax(int col, int row = 0, bool abs_true = true) const
+ void swapRow(int row1, int row2) const
+ void swapCol(int col1, int col2) const
+ void swap(int row1, int col1, int row2, int col2) const
+ friend int* size(const Matrix& m)
+ friend void print(const Matrix& m)
+ friend std::ostream& operator<<(std::ostream& output, const
Matrix& m)
```

—- Linear Algebra helpers ————————-

```
+ Matrix T() const
+ friend Matrix transpose(const Matrix& m)
+ double norm(int p) const
+ double norm(std::string p) const
+ friend double norm(Matrix& m, int p)
+ friend double norm(Matrix& m, std::string p)
+ friend const Matrix eye(int row, int col)
+ friend const Matrix eye(int n)
+ friend const Matrix rand(int row, int col)
+ friend const Matrix rand(int n)
```

—- Linear Algebra algorithms ————————

```
+ friend Matrix gaussianElimination(const Matrix A, const
Matrix b)
+ friend std::tuple<Matrix, Matrix, Matrix, int> lu(const
Matrix& A)
+ double det() const
+ friend double det(const Matrix& A)
+ friend std::tuple <Matrix, Matrix> qr(const Matrix& A)
+ friend Matrix hessenbergReduction(const Matrix& A)
+ friend Matrix eigenVal(const Matrix& A, double tol = 1.0e-10)
```

Figure 3: A continuation of the UML diagram from Figure 2.