A Matrix Class in C++ for Numerical Linear Algebra

Ilan Price

July 18, 2017

MSc Special Topic
Course: C++ for Scientific Computing
Lecturer: Dr. Joe Pitt-Francis
Mathematical Institute
The University of Oxford

1 Introduction

This paper presents a Matrix class developed in C++ equipped for numerical linear algebra. In particular, our goal is to create a class which offers a wide variety of tools to analyse and solve linear systems of equations of the form $A\mathbf{x} = \mathbf{b}$, which crop up in a vast array of scientific disciplines and industries.

C++ is a much lower level language than, say, MATLAB, which is already very well designed and optimised to implement numerical linear algebra. Where feasible (and sensible), we develop our Matrix class and associated functions in such a way as to mimic MATLAB's syntax.

We assume a working knowledge of linear algebra, and for the sake of economy we will only broadly sketch the relevant mathematical background and motivation behind particular algorithms in the main body of the text. We omit most of the details and proofs, for which we will refer the interested reader to the appropriate appendices or references.

In Section 2, we outline the structure of the Matrix class and its relationship with an independent Vector class. In Section 3 we explain the implementation of some of the matrix operations and pieces of linear algebra machinery built for the Matrix class. We focus on those which are most interesting from a scientific computing perspective, though we recommend the reader to consult Appendix B for a full summary of the members and friends of the class. Section 4 details the nine linear solvers which we developed for this class. Finally, Section 5 applies these solvers to a model problem, the 2D Poisson equation, and compares their performance both to one another and to their analytically expected performance.

2 The Matrix Class

2.1 Attributes, Constructors & Destructor

Our Matrix class has three private attributes: (i) `mElements`, which is a pointer to an array of pointers, each of which 'points' to where the first element of a single row of matrix entries is stored (with each entry stored as a double), (ii) `mRows` and (iii) `mColumns`, which are integers storing the number of rows and columns in the Matrix. These attributes are private so that we can define the terms on which they are accessed and changed [8].

We override the default constructor and overload it to implement three new con-

structors. One constructor takes the number of rows and columns as parameters and creates a zero Matrix of those dimensions. The second accepts an additional parameter in the form of an initialiser list [3], which contains the elements which will populate the Matrix. This makes it much easier to create and populate Matrices, and is similar to MATLAB's functionality in this regard. The elements of the initialiser list are assigned as Matrix elements row by row, and in the case that the the list is the wrong length, the Matrix will either set the extra unspecified elements as zeros if the list provided is too short, or use only as many elements from the list as needed to fill the Matrix (if the list provided was too long.)

The third constructor is a copy constructor, which accepts a constant reference to a Matrix and creates another Matrix of the same dimensions with the same values in each element.[1]

The following snippet shows examples of how these constructors might be called.

```
Matrix A(3,3); // Creates a 3x3 Matrix A of zeros
Matrix B(3,3,{1,2,3,4,5,6,7,8,9}) // creates a 3x3 matrix B with row 1 = 1,2,3;
    row 2 = 4,5,6; row 3 = 7,8,9
Matrix C(B); //creates a Matrix C which is a copy of B
```

We also override the default destructor so that it frees up memory which was dynamically allocated to a Matrix when that Matrix goes out of scope [8].

2.2 'Const' and 'friend' Declarations

Aside from the constructors and destructor in Section 2.1, the '`setElements`', '`getElements`', and '`applygivens`' functions described in Section 3.1.2 and Section 3.3.2, and the assignment and indexing operators, all other functions and operators described in this paper are declared as '`friend`' in the '`Matrix.hpp`' file (and appropriately prototyped outside the class as necessary). This allows the operators and functions to access private members of the class.[2]

[1]Note that for other functions,'`const& Matrix`' and '`Matrix`' are both possible input paramter choices (though we frequently choose and motivate the former). In the case of the copy constructor, however, we are not free to design it with a '`Matrix`' parameter instead of a '`const& Matrix`'. This is because when a function takes '`Matrix`' as parameter, the function calls the copy constructor to make a copy of that matrix to be used by in the body of the function. Thus, in the case of the copy constructor, it would be trying to call itself.

[2]It would indeed have been possible to write certain functions without needing to be defined as '`friend`', but the decision was made to maintain consistency. Though it breaks encapsulation, given the purposes and use cases of this class there are no significant downsides.

In order to guarantee that our functions and operators do not change the Matrices and Vectors which they take in as parameters, all '`friend`' functions and operators accept '`const`' references to the Matrix or Vector, and thus have prototypes of the form shown below.

```
function/operator_name(const Matrix& A, const Vector& v, ....) // number and
    type of parameters depends on the function
```

Accepting a reference as a parameter instead of a Matrix/Vector itself avoids calling the copy constructor to create a copy of the Matrix/Vector for use within that function, and so is more time and memory efficient. Specifying the reference as '`const`' then ensures that even though actual Matrix/Vector is passed by reference into the function, the function will not alter the contents of that Matrix/Vector.

2.3 Connection with Vector Class

Our Matrix class is built to interact with a Vector class written and supplied by Dr Joe Pitt-Francis [7].

We have made some modifications to the Vector class, for example, adding a '`ones`' and '`linspace`' function. Some of these additions depend themselves on the Matrix class (for example, we have added a '`reshape`' function which reshapes a Vector into a Matrix of specified dimensions). This dependency requires that we use forward declaration in the respective '`.hpp`' files for the code to successfully compile.

The decision to maintain a separate Vector class was a considered and deliberate choice. Though a vector may be considered just a single row or column matrix, there are number of benefits to representing them a separate class. Firstly, it allows one to ignore the 'orientation' of a vector, which is often preferable. Secondly, from a storage and access perspective, storing a column vector as a single column Matrix involves storing one pointer (as well as the value) for each element of the vector, instead of simply storing a single pointer to the first element of the vector, as is the case with the given Vector class.

3 Linear Algebra Functionality

3.1 Operators and Indexing

3.1.1 Binary and Unary Operators

We overload the binary addition and subtraction operators '`+`' and '`-`' to perform element-wise addition and subtraction. We deliberately enforce, by including a helper function '`equalDim(A,B)`', that only Matrices of the same dimensions can be added or subtracted from one another, and an exception is thrown if the dimensions are not equal. An alternative approach might be to pad the 'extra' rows and columns with zeros before adding/subtracting them. However, this is neither a mathematically defined operation, nor do we consider the potential benefit of its inclusion to be substantial. Thus we follow MATLAB's lead and require that the Matrix dimensions agree. We also overload the '`*`' operator to perform standard matrix-matrix multiplication, matrix-vector and vector-matrix multiplication (all of which throw exceptions in the event of a dimensions mismatch), as well as multiplication of a Matrix by a constant, from either side.

We overload the unary operator '`-`' to return the expected result, a Matrix with all elements of the opposite sign. We also overload the '`!`' operator to return the transpose of a given matrix. Finally, we overload the assignment operator '`=`' so that it functions as expected with Matrices.

3.1.2 Indexing and slicing

We overload the '`()`' operator to allow for indexing in a similar style to MATLAB, so that, for example, `A(1,2)` accesses the element in the 1st row and 2nd column (note the indexing from 1, not 0).

There are a few important points to be made about this operator's declaration.

The first is that we need also need to overload this operator as a '`const`' operator. This is because we want to be able to use this method of indexing in the body of other functions which take in, say, '`const& A`' as an argument. One option is to copy the body of this code twice exactly, as '`const`' and non-'`const`' operators in the '`.cpp`' file. Alternatively, to avoid this code duplication, we can follow [6], and define them in the '`.cpp`' file as follows,

```
const double& Matrix::operator()(int i, int j) const {
  // ... code to throw exceptions if i or j are out of range
  return mElements[i-1][j-1]; }
```

```
double& Matrix::operator()(int i, int j) {
  return const_cast<double&>(static_cast<const Matrix &>(*this)(i, j)); }
```

The single line in the non-'`const`' definition is essentially doing three things. First, it is casting away the '`const`' on `()`'s return type. Second it is adding '`const`' to `*this`'s type; and then third it is calling the '`const`' version of `()`, defined just above.

Mimicking MATLAB's slicing notation (e.g. `A(1:3,2:5)`) is more trouble than it is worth in C++. We therefore implement the same functionality with the '`getElements`' and '`setElements`' member functions. '`getElements`' accepts as parameters the start and end row and column numbers of the sub-matrix you would like to copy, and an optional string parameter which allows one to specify that they would like the function to return a Vector instead of Matrix (assuming the slice is from either a single row or column). '`setElements`' accepts the start and end row and column numbers of the sub-matrix you would like to change, as well as the Matrix (or Vector) of values to be assigned. Both methods throw exception in cases where the indexing is out of range or when there is a dimension mismatch.

3.2 Matrix Decompositions

3.2.1 LUP factorisation

Given a square matrix A, we might wish to factorise this into a lower-triangular matrix L and an upper-triangular matrix U, such that $A = LU$. This can be performed by slightly modifying the Gaussian elimination algorithm. However, given that this algorithm breaks down if there are zeros on the diagonal (and for the sake of numerical stability), we implement partial pivoting first, which involves swapping the rows of matrix A at each step as and when necessary. This is equivalent to left-multiplying by a permutation matrix P, and then factorising such that $PA = LU$.

The algorithm loops over the columns of a copy of the Matrix A, called `U` in the code snippets below. This matrix U will become upper-triangular as the algorithm zeros all the elements beneath the diagonal. For each column, the function first ensures that the diagonal element is larger in absolute value than all those below it. If in, say, column k, the element in row j ($j > k$) has the largest absolute value out of the column entries below the diagonal, then the algorithm swaps the j^{th} row with the k^{th} row. This row permutation is also applied the L matrix as each stage, and is

iteratively stored in a permutation matrix P.

The heart of the LU factorisation algorithm at step k involves performing row operations to zero the elements of the k^{th} column beneath the diagonal, and simultaneously populating our matrix L. The basic algorithm is

1: **for** $i = k + 1, \ldots, n$ **do**
2: L(i,k) = A(i,k)/A(k,k)
3: Row i = Row i - L(i,k)*Row k
4: **end for**

The function needs to return three Matrices (L, U, and P), which itself is not trivial in C++. In order to achieve this our function makes use of the Boost Tuple library [4], because C++ does not have built in tuple-structures.

The Boost library and tuple structure feature in our function as follows,

```
#include "boost/tuple/tuple.hpp"
...
boost::tuple< Matrix, Matrix, Matrix> LU(const Matrix& A) {
   ...
   boost::tuple<Matrix,Matrix,Matrix> LUP = boost::make_tuple(L, U, P);
   return LUP; }
```

and the individual matrices are accessed by

```
boost::tuple<Matrix, Matrix, Matrix> LUP = LU(A);
Matrix L = LUP.get<0>();
Matrix U = LUP.get<1>();
Matrix P = LUP.get<2>();
```

The snippets of code showing the implementation of partial pivoting and the necessary row-operations can be found in Appendix 6.

3.2.2 QR factorisation

Any real matrix $A \in \mathbb{R}^{m \times n}$ can be factorised into an orthogonal matrix $Q \in \mathbb{R}^{m \times m}$ (with $Q^{-1} = Q^{\top}$) and an upper triangular matrix $R \in \mathbb{R}^{m \times n}$, such that $A = QR$ [11].

There are a number of ways to compute the QR decomposition, including the Gramm-Schmidt process and using Givens rotations[11].

The QR function in our Matrix class makes uses of a method using Householder matrices [11]. The method involves iteratively constructing Householder reflections which, when applied to the Matrix A, zero all the elements below the diagonal in a

particular column.

We give a brief outline of the algorithm here as it is necessary for understanding its implementation in C++, as shown in the code excerpt below.

First we create a Householder matrix $H_1 \in \mathbb{R}^{m \times m}$, such that

$$H_1 A = \begin{bmatrix} \alpha_1 & \cdots \\ 0 & A_1 \end{bmatrix}. \tag{1}$$

Next, we create $\hat{H}_2 \in \mathbb{R}^{m-1 \times m-1}$ such that

$$\hat{H}_2 A_1 = \begin{bmatrix} \alpha_2 & \cdots \\ 0 & A_2 \end{bmatrix}, \qquad \text{and then} \qquad H_2 \in \mathbb{R}^{m \times m} = \begin{bmatrix} 1 & 0 \\ 0 & \hat{H}_2 \end{bmatrix}. \tag{2}$$

Continuing with this process for n steps gives us our QR factorisation, with

$$R = H_n \dots H_2 H_1 A = \begin{bmatrix} \alpha_1 & \cdots & \cdots \\ 0 & \ddots & \\ 0 & 0 & \alpha_n \end{bmatrix}, \tag{3}$$

and

$$Q = (H_n \dots H_2 H_1)^{-1} = H_1^\top \dots H_n^\top = H_1 H_2 \dots H_n. \tag{4}$$

The main crux of the algorithm is the iterative construction of the correct Householder reflection, by

$$\hat{H}_i = I - \frac{2}{w^\top w} w w^\top, \tag{5}$$

where $w = u - v$, with u being the first column of A_i and and $v = [||u||_2, 0, \dots, 0]^\top$.

This is implemented by the following procedure in our `QR` function.

```
// while looping over i from to n
Matrix Ai = R.getElements(i,Rrows,i, Rcolumns);
Vector u = Ai.getElements(1,Ai.mRows,1,1,"Vector");
Vector v(length(u)); v(1) = norm(u,2);
Vector w = u-v;
Matrix Hi = eye(Rcolumns);
Matrix Ii = eye(B.mRows);
Matrix temp2 = outer(w,w);
if (norm(w,2)>1e-15) {
   Matrix Hhat = Ii - (2/(w*w))*temp2;
   Hi.setElements(i,Rcolumns,i,Rcolumns,Hhat); }
```

Note that the '`if`' statement before the construction of the Householder reflection is crucial. This deals with those cases where all entries below the diagonal are already zero (for example, if one computed the QR factorisation of an upper-triangular matrix.) In such cases $w = 0$, and so our function would otherwise end up dividing by zero in calculating $\hat{H}_k$. Instead, we want $\hat{H}_k$ in this case simply to be the identity.

We need our '`QR`' function to return two matrices, and in order to achieve this we take advantage of the '`std::pair< , >`' structure from the standard library [3]. Within our function we create and return this pair with

```
std::pair <Matrix,Matrix> QR(const Matrix& A){
...
   std::pair<Matrix,Matrix> qr = std::make_pair(Q,R);
   return qr;}
```

and the Matrices are accessed by

```
std::pair<Matrix, Matrix> qr = QR(A);
Matrix Q = qr.first;
Matrix R = qr.second;
```

3.2.3 Eigenvalues and Eigenvectors

The algorithm we use to calculate the eigenvalues and eigenvectors of a square matrix A is known as the QR algorithm [11]. The algorithm is very simple: at each step k (i) QR factorise the Matrix A, and then (ii) set $A = RQ$. The diagonal elements of A converge[3] to the eigenvalues of A, and the corresponding eigenvectors are given by the columns of $\Pi_k Q_k$.

We implement this algorithm with the following excerpt from our '`eig`' function.

```
...
do {
Atemp = Anew;
qr = QR(Anew); Q = qr.first; R = qr.second;
Anew = R*Q; evecs = evecs*Q;
}while((std::abs(Atemp(1,1) - Anew(1,1))) > 1e-15 //... also check last diag entry
```

We design our '`eig`' function so that it returns a Vector of eigenvalues and a Matrix of eigenvectors as a '`std::pair<Vector, Matrix>`', accessed in a similar way

[3] Convergence of the QR algorithm is only guaranteed for a matrix when either (i) $|\lambda_1| > |\lambda_2| > \ldots |\lambda_n|$, or (ii) all its elementary divisors are linear [13]. We thus consider only such matrices.

to that described for the Q and R matrices in Section 3.2.2.

We consider the algorithm as having converged when the difference between the first and last diagonal elements of A_k and A_{k+1} is less than 1e-15 (which corresponds to machine ε), and this has performed well in all test cases.

As eigenvalues and eigenvectors can only be calculated for square matrices, our function throws an exception if the matrix supplied is not square.[4]

```
if (dim(1) != dim(2)) {throw Exception("dimension error",
"computing eig(), matrix supplied not square");}
```

3.3 Matrix Transformations

3.3.1 Hessenberg Matrix

Given a square Matrix A, the '`hess`' function returns an upper Hessenberg matrix that is similar to A (that is, it has the same eigenvalues as A).

The algorithm resembles that used for QR factorisation, but is slightly more involved analytically and computationally. Let

$$A = \begin{bmatrix} a_1 & v_1^\top \\ u_1 & B_1 \end{bmatrix}, \quad \text{and let} \quad H_1 = \begin{bmatrix} 1 & 0 \\ 0 & K_1 \end{bmatrix}, \quad \text{then} \quad H_1 A H_1 = \begin{bmatrix} a_1 & v_1^\top K_1 \\ K_1 u_1 & K_1 B_1 K_1 \end{bmatrix}. \tag{6}$$

Choosing K_1 to be a Householder matrix such that $K_1 u_1 = \alpha_1 \mathbf{e_1}$, we have

$$A^{(2)} = H_1 A H_1 = \begin{bmatrix} a_1 & b_1 & \hat{v}_2^\top \\ \alpha_1 & a_2 & v_2^\top \\ 0 & u_2 & B_2 \end{bmatrix}. \tag{7}$$

Inductively this gives $A^{(n-1)} = H_{n-2} \dots H_1 A H_1 \dots H_{n-2}$, which is upper Hessenberg.

The excerpt of code below from within the main loop in the '`hess`' function shows how this is implemented at each iteration. Notice once again that in order to avoid dividing by zero in the construction of our Householder reflection, we must check whether or not the column in question was already all zero beneath the sub-diagonal, in which case H_k should be the identity matrix.

[4] Perhaps a better way of handling this error case would be to return the singular values and vectors of the matrix, instead of the eigenvalues and vectors, and to print a message explaining that this has been done. However, as we have not yet included a function for computing the singular value decomposition of a matrix, our current approach will suffice.

```
// snippet from inside loop: looping over i up to n-1
Matrix Bi = Ahes.getElements(i,n,i, n);
Vector ui = Bi.getElements(2,Bi.mRows,1,1,"Vector");
Vector vi(length(ui)); vi(1) = norm(ui,2);
Vector wi = ui-vi;
Matrix Ki = eye(Bi.mRows-1);
if (norm(wi)>1e-14) {
   Matrix temp2 = outer(wi,wi);
   Ki = Ki - (2/(wi*wi))*temp2;}
Matrix Hi = eye(n);
Hi.setElements(i+1,n,i+1,n,Ki);
Ahes = Hi*Ahes*Hi;
```

3.3.2 Givens Rotations

A Givens rotation matrix $G(i, j, \theta)$ has entries $g_{k,k} = 1$ for $k \neq i, j$, $g_{i,i} = \cos\theta$, $g_{j,j} = \cos\theta$, $g_{j,i} = -\sin\theta$, $g_{i,j} = \sin\theta$ for $i > j$, and 0 elsewhere [11]. When a Givens rotation matrix multiplies another matrix A, only rows i and j of A are affected. With this in mind, Matrix multiplication (which is $O(n^3)$) is a very inefficient way of applying a Givens rotation. Instead, we include two functions which much more efficiently calculate and apply Givens rotations.

The '`givens`' function accepts two parameters a and b, and returns a '`std::pair<double, double>`' with the values $\cos\theta$ and $\sin\theta$ such that

$$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}. \tag{8}$$

The '`applygivens`' method accepts four parameters: the rows i and j which will be affected in the matrix being rotated, and the values for $\cos\theta$ and $\sin\theta$. Instead of creating and multiplying by a full Givens rotation Matrix, the method simply loops over the columns of the Matrix, and appropriately edits the values in the affected rows, as shown in the snippet below.

```
void Matrix:: applygivens(int i, int j, double c, double s) {
   for (int k = 1; k<=mColumns; k++) {
      double newi = c*mElements[i-1][k-1] - s*mElements[j-1][k-1];
      double newj = s*mElements[i-1][k-1] + c*mElements[j-1][k-1];
      mElements[i-1][k-1] = newi; mElements[j-1][k-1] = newj; } }
```

In our MINRES and GMRES functions, which both involve multiple Givens rotations (see Section 4.2.1), avoiding this unnecessary Matrix multiplication cuts computational time by more than an order of magnitude.

3.4 Testing

For all operators and functions listed in this section, an effort was made to fully test the functionality of our code. This includes testing that exceptions are thrown as expected, and that our functions work as expected on appropriate 'special' matrices, like singular, Hessenberg, or identity matrices (depending on the function). We direct the reader to an abridged summary of a sample of the test cases for the functions in this section and their results in Appendix A.

4 Linear solvers

In this section we describe the various methods developed for solving linear systems of equations of the form $A\mathbf{x} = \mathbf{b}$.

4.1 Direct solvers

4.1.1 Back and Forward-substitution

Systems of the form $L\mathbf{x} = \mathbf{b}$ and $U\mathbf{x} = \mathbf{b}$, where L is upper-triangular and U is lower-triangular, can be solved directly by the simple back and forward-substitution algorithms respectively, each of which we have implemented as independent functions.

These functions are sufficient on there own in those rare cases where the coefficient matrix is already lower or upper triangular, but feature more often as one step in a more general method.

We implement the standard back-substitution algorithm as follows,[5]

```
// Back substitution snippet: standard algorithm, when bandwidth not specified
for (int i=n;i>0;i--) {
   if (A(i,i)==0) { ... // code which throws exception}
   x(i) = b(i);
   for (int j = (i+1); j<=n; j++) {
      x(i) = x(i) - A(i,j)*x(j);}
   x(i) = x(i)/A(i,i); }
```

[5]Forward-substitution is very similar in its implementation and hence we omit the code here.

In the general case, as is clear from the code snippet above, the computational complexity of standard back and forward-substitution is $O(n^2)$. However, in the case of 'banded' triangular matrices, which have only a few non-zero diagonals, this is unnecessarily expensive and can be reduced to $O(cn)$, where c is the bandwidth, by avoiding the many multiplication-by-zero operations which the standard algorithm would compute. Our back and forward-substitution functions are therefore designed to take an optional extra, positive integer parameter, to specify this 'bandwidth' of the triangular matrix.

```
// Back substitution declaration in .hpp file, outside of class definition
Vector backsubst(const Matrix& A, const Vector& b, int bwidth = 0);
```

The functions default to standard back and forward-substitution, but when the bandwidth is specified, it ignores all those elements (which are zeros) above the uppermost or lower-most non-zero diagonal.

```
// Back substitution snippet: when bandwidth > 0
for (int i=n;i>0;i--) {
   ... // Throw exception if matrix is singular
   x(i) = b(i);
   for (int j = (i+1); j<=(i+bwidth - 1); j++) {
      if(j<=n) { x(i) = x(i) - A(i,j)*x(j);} }
   x(i) = x(i)/A(i,i); }
```

We expect the primary error case to be when the coefficient Matrix is singular, and thus there is no solution to the system of equations. It is simple to prove that in the case of a triangular matrix, it is singular if and only if one or more of its diagonal elements is 0. In such cases our function throws an exception and outputs the appropriate error message.

4.1.2 Using LU decomposition

For a dense, non-singular square matrix A, the system $A\mathbf{x} = \mathbf{b}$ can be solved by using the LU decomposition of A by noting that solving $PA\mathbf{x} = P\mathbf{b}$ yields the same solution. The algorithm is straightforward: (i) calculate the P, L, and U, such that $PA = LU$, (ii) let $U\mathbf{x} = \mathbf{y}$ and solve $L\mathbf{y} = P\mathbf{b}$ by forward-substitution, and (iii) solve $U\mathbf{x} = \mathbf{y}$ by back-substitution.

This solver is implemented as an overloading of the '`/`' operator, as shown in the code below, and is implemented by the command '`Vector x = A/b`'.

```
// Overlaoded '/' operator to solve by modified Gauss Eliminiation/LU
    factorisation
Vector operator/(const Matrix& A, const Vector& b) {}
   boost::tuple<Matrix, Matrix, Matrix> LUP = LU(A);
   Matrix L = LUP.get<0>(); Matrix U = LUP.get<1>(); Matrix P = LUP.get<2>();
   Vector newb = P*b;
   Vector y = forwardsubst(L,newb);
   Vector x = backsubst(U,y);
   return x; }
```

While this is an exact method, applicable to all systems for which there is a solution, it is also a very computationally expensive. We therefore include a number of other, more efficient methods as well.

4.2 Krylov Subspace Methods

Given some matrix $A \in \mathbb{R}^{n \times n}$ and some vector $\mathbf{r} \in \mathbb{R}^n$, Krylov subspaces are nested vector spaces defined as

$$\mathcal{K}_k(A, \mathbf{r}) = \text{span}\{\mathbf{r}, A\mathbf{r}, A^2\mathbf{r}, \dots A^{k-1}\mathbf{r}\}. \tag{9}$$

Given a linear system $A\mathbf{x} = \mathbf{b}$, and an initial guess $\mathbf{x}_0$ (with an initial residual $\mathbf{r}_0 = \mathbf{b} - A\mathbf{x}_0$), Krylov subspace methods work by iteratively calculating the vector $\mathbf{x}_k \in \mathbf{x}_0 + \mathcal{K}_k(A, \mathbf{r}_0)$ which minimises the residual $\mathbf{r}_k$ in some norm. Krylov subspace methods are in fact exact solvers, since eventually $\mathbf{x}$ will be contained in $\mathcal{K}_k(A, \mathbf{r}_0)$ for large enough k. However, these methods are also often used as iterative methods, choosing to accept a solution $\mathbf{x}_k$ once the residual drops below as certain tolerance. Our approach in writing these as functions for our Matrix class will be to terminate the algorithm once the residual drops below 1e-15.

4.2.1 GMRES and MINRES

GMRES [9] works by calculating, at each iteration, one more A-orthonormal basis vector for $\mathcal{K}_k(A, \mathbf{r}_0)$, and finding $\mathbf{x}_k$ as the linear combination of the first k basis vectors found, that minimises the L_2-norm of the residual $\mathbf{r}_k$.

The algorithm for generating this basis is known as the Arnoldi algorithm [1], shown in full in Algorithm 2 in Appendix C, and each step of which is implemented in our GMRES function by the snippet of code below.

```
...
```

```
while (norm_r > Tol) {
   Vector v_k = V.getElements(1,n,k,k,"Vector");
   Vector w = A*v_k;
   for (int j = 1; j<=k; j++) {
      Vector v_j = V.getElements(1,n,j,j,"Vector");
      double h = v_j*w;
      H(j,k) = h;
      w = w - h*v_j; }
   double hplus = norm(w);
   H(k+1,k) = hplus;
   if (hplus>=1e-15) {
   Vector v_kplus1 = w/hplus;
   V.setElements(1,n,k+1,k+1,v_kplus1);
   ...}
...
```

After the k^{th} iteration, the Arnoldi algorithm has produced a matrix system of the form

$$AV_k = V_{k+1}\hat{H}_k, \tag{10}$$

where $\hat{H}_k$ is upper Hessenberg, and V_k is a matrix whose columns are the basis vectors $\mathbf{v}_i$, $i = 1, \ldots, k$, of $\mathcal{K}_k(A, \mathbf{r}_0)$.

After some manipulation, we can show that the $\mathbf{x}_k \in \mathcal{K}_k(A, \mathbf{r}_0)$ which will minimise $||\mathbf{r}_k||_2$ is $\mathbf{x}_k = \mathbf{x}_0 + V_k\mathbf{y}$, where $\mathbf{y}$ solves the linear least squares problem

$$\min_{y \in \mathbb{R}^k} ||||\mathbf{r}_0||_2\mathbf{e_1} - \hat{H}_k\mathbf{y}||_2, \tag{11}$$

and $\mathbf{e_1} = [1, 0, 0 \ldots, 0]^\mathsf{T} \in \mathbb{R}^{k+1}$.

One of the challenges in coding GMRES is that the Matrices involved keep changing size every iteration ($\hat{H}_k \in \mathbb{R}^{k+1\times k}$ at step k, etc.). Fortunately, as explained in Section 4.2, we know that k will never increase above n, the number of rows and columns in A. We therefore begin by setting up $n \times n$ matrices, and then simply access the necessary portion of these Matrices at each step using the '`getElements`' and '`setElements`' functions. While for extremely large n this may cause memory issues, it allows for a very clean implementation, and avoids having to create new matrices of different sizes and copy the existing elements across, at the start of each iteration, before the bulk of the algorithm can be executed.[6]

[6]Moreover, in cases of limited memory, Restarted GMRES (GMRES(k)) can be implemented with this function to limit the required memory [11].

Solving the linear least squares problem in (11) involves computing a QR factorisation of $\hat{H}_k$. Though we already have a `QR` function which could successfully do this, it would be unnecessarily expensive in this case. Firstly, $\hat{H}_k$ is upper Hessenberg and can be made to be upper triangular with the application of k Givens rotations. Secondly, this factorisation needs to happens at every step, and hence at each step k, we can store the Givens rotation necessary to zero the k^{th} sub-diagonal, meaning that we only need to calculate and apply one more Givens rotation per iteration. This is a crucial element of an efficient GMRES implementation.

Furthermore, even though our stopping criterion involves $||\mathbf{r}_k||$, which is normally calculated as $\mathbf{b} - A\mathbf{x}_k$, we need not actually calculate x_k at each iteration. It can be easily shown that $||r_{k+1}||_2 = \sin\theta||r_k||_2$, where θ is the angle in the Givens rotation applied in the k^{th} step. We therefore save a lot of matrix multiplication and back-subsitution at each step by simply checking the updated residual value, and solving for $\mathbf{y}$ and calculating $\mathbf{x}$ only when the residual is sufficiently small.

The code extract below shows the process by which we calculate and store the necessary Givens rotation at each step. The full code for our GMRES implementation is included in Appendix 6.

```
while (norm_r > Tol) {
...
   std::pair<double,double> g = givens(Hhat(k,k),Hhat(k+1,k));
   double gcos = g.first; double gsin = g.second;
   G.applygivens(k,k+1,gcos,gsin);
   norm_r = std::abs(norm_r*gsin);}
   ...
```

It can also be shown that the once no more A-orthogonal basis vectors are possible (and so when $h_{k+1,k} = 0$), then the solution $\mathbf{x}_k$ will be the exact solution $\mathbf{x}$, since the basis will span the entire possible space in which the exact solution $\mathbf{x}$ exists. On this iteration, $\hat{H}_k$ is already upper triangular, and thus no additional Givens rotation is needed. Moreover, there will be no additional vector $\mathbf{v}_{k+1}$ to calculate and store. Thus, our function checks the value of h_{k+1}, breaking the loop iterating over k when this value is close to machine zero ('`if (hplus<1e-15) {break;}`').

If the Matrix A is symmetric, then the Matrix $\hat{H}_k$ will also be symmetric, and since it is upper Hessenberg, this means that $\hat{H}_k$ will also be tridiagonal. Given that we know this fact from the outset, it is unnecessary to individually calculate all the zeros above the superdiagonal in $\hat{H}$ with the Arnoldi algorithm. The MINRES algorithm takes advantage of the tridiagonal nature of $\hat{H}$ by replacing the Arnoldi algorithm

with the Lanczos algorithm [10]. The Lanczos algorithm is shown in Algorithm 3 in Appendix C, and our implementation thereof is in Appendix F.

Furthermore, if A is symmetric, then when we eventually QR factorise $\hat{H}_k$ with Givens rotations, the upper triangular matrix R will have bandwidth 3. We take advantage of this by using specifying this bandwidth when we solve for $\mathbf{y}$ with our back-substitution method in our '`MINRES`' function.

```
Vector y = backsubst(H.getElements(1,k,1,k), r_e1.getValues(1,k), 3);
```

The following convergence result for GMRES on symmetric positive definite matrices is known [5],

$$\frac{||\mathbf{r}_k||_2}{||\mathbf{r}_0||_2} \leq \left(\frac{\kappa^2 - 1}{\kappa^2}\right)^{k/2} \tag{12}$$

where $\kappa = \lambda_{\max}/\lambda_{\min}$, with $\lambda_{\max}$ and $\lambda_{\min}$ being the maximum and minimum eigenvalues of A.[7] We will check that our implementation satisfies this convergence bound in Section 5.

4.2.2 Conjugate Gradient

If the matrix A is symmetric positive definite, then the conjugate gradient (CG) method is an efficient Krylov subspace method for solving the linear system.

CG works by calculating the $\mathbf{x}_k \in x_0 + \mathcal{K}_k(A, \mathbf{r}_0)$ such that the A-norm of the residual $\mathbf{r}_k$ is minimised at each iteration. We refer the interested reader to [11] for an excellent exposition of the conjugate gradient method. For our purposes we will take as a starting point the algorithm given in Algorithm 1 in Appendix C.

The code below shows our implementation of the bulk of the algorithm (after the necessary initialisations).

```
Vector CG(const Matrix& A, const Vector& b) {
  // ... initialise x0, r0, p0 = r0; temp1 as A*x,
  do {
     temp1 = A*p;
     alpha = (temp2)/(p*temp1);
     x = x + alpha*p;
     temp3 = temp2;
     r = r - alpha*temp1;
     temp2 = r*r;
     beta = temp2/temp3;
```

[7] Note that for poorly conditioned matrices this convergence result is rather unhelpful.

```
    p = r + beta*p; } while (norm(r)>1e-15);
  return x; }
```

Though it is possible to write this algorithm in slightly fewer lines, the use of '`temp`' variables minimises the number of Matrix-Vector products and Vector inner products. For example, at each iteration in the `do-while` loop, $A\mathbf{p}$ and $\mathbf{r}^\top\mathbf{r}$ are only calculated once each, though they are used twice per iteration in the algorithm.

A convergence result for CG says that at iteration k,

$$\frac{||\mathbf{x}-\mathbf{x}_k||_A}{||\mathbf{x}-\mathbf{x}_0||_A} \leq \min_{p\in\Pi_k, p(0)=1} \max_j |p(\lambda_j)| \tag{13}$$

$$\leq 2\left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^k, \tag{14}$$

where Π_k is the set of polynomials of degree $\leq k$, and $\kappa = \lambda_{\max}/\lambda_{\min}$, with $\lambda_{\max}$ and $\lambda_{\min}$ being the maximum and minimum eigenvalues of A. We confirm that our implementation satisfies this convergence result in Section 5.

4.3 Splitting methods

Splitting methods for solving linear systems of equations are iterative methods which share a common form, in that

$$\mathbf{x}_{k+1} = F\mathbf{x}_k + F\mathbf{b}, \tag{15}$$

where the iteration matrix F is some combination of different parts of the coefficient matrix A. The methods terminate when the residual $\mathbf{b} - A\mathbf{x}_k$ is smaller than some specified tolerance. Splitting methods differ by how they split A into different parts, and how they construct the iteration matrix F. However, all share the same convergence condition that $\mathbf{x}_k$ will converge to the exact solution as $k \longrightarrow \infty$ if the spectral radius of the iteration matrix is less than 1.

We implement three different splitting methods, known respectively as Jacobi, Gauss-Seidel, and Successive Over-Relaxation. A brief explanation of each method, as well as their C++ implementation, can be found in Appendix D. We omit this from the main text body as their implementation is fairly straightforward.

The convergence of iterative splitting methods like Jacobi, G-S and SOR depends on the spectral radius of the iteration matrices specific to each method. In particular, we expect the error of each iterate, $||\mathbf{x}-\mathbf{x}_k||_2$, to converge at least as fast as ρ^k, as k increases (where ρ is the spectral radius of the iteration matrix in question). We confirm that our code satisfies the expected convergence results in Section 5.

5 The Model Problem

5.1 The Poisson Equation

Perhaps the most famous elliptical partial differential equation is Poisson's equation,

$$-\nabla^2 u = f. \tag{16}$$

This problem is often referred to as the 'model problem', and is used to illustrate and compare the performance of various numerical solvers. We will solve this equation in $\mathbb{R}^2$, with domain $\Omega = [0,1]^2$. x and y are discretised with intervals of length Δx and Δy respectively, such that $x_i = i\Delta x$ and $y_i = i\Delta y$. Approximating u_{xx} and u_{yy} with a finite difference scheme, and assuming homogenous Dirichlet boundary conditions, the problem can be represented as a linear system of the form $A\mathbf{u} = \mathbf{f}$, where A is a symmetric positive definite block-tridiagonal matrix which we assemble with our Kronecker product '`kron`' function. The derivation and form of the matrix A and vectors $\mathbf{u}$ and $\mathbf{f}$ are included in Appendix E.

We will use this problem as the baseline for confirming and comparing the functionality of the linear solvers we have developed to interact with our Matrix and Vector classes (though we have tested each with many other examples as well.)

5.2 Convergence

We now use our various linear solvers to solve our model problem. For these simulations we set $n = 20$ and $h = 1/n = 0.05$. Our 20×20 grid on the unit square means that our Matrix A has dimension 400×400.

All plots below were produced in MATLAB, having included code in our C++ functions to appropriately write our results to '`.csv`' files using the '`fstream`' class, which were then read in by MATLAB.[8]

First, we compare the convergence of the splitting methods with their analytically predicted convergence rates. For this problem, $\rho_{Jacobi} = 0.9888$, $\rho_{GS} = 0.9778$, the optimal value of w for SOR is $w_{opt} = 1.7406$ and $\rho_{SOR} = 0.7406$. Figure 1 shows that all methods converge exactly as expected.

Next we examine the convergence of the Conjugate Gradient and GMRES methods. In Figure 2a we plot the residual of Conjugate Gradient, GMRES and MINRES at each iteration, which demonstrates their convergence, and shows as expected that

[8]It is crucial to use the '`std::setprecision`' command when writing to file, as the default precision will not be sufficient for accurate convergence plots in MATLAB.

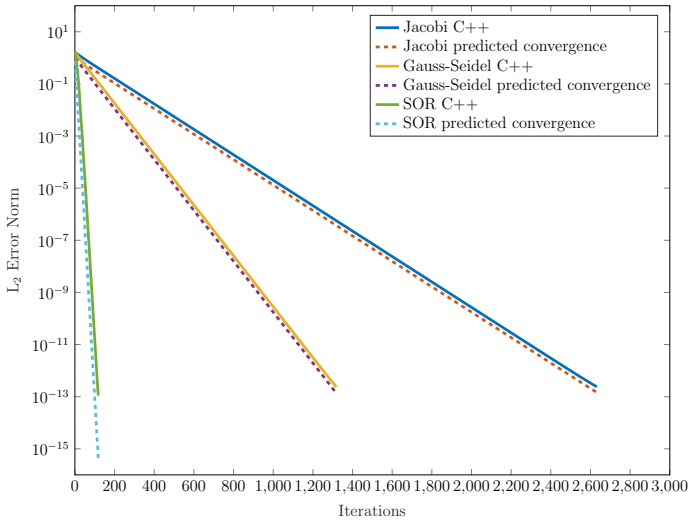

Figure 1: Convergence of splitting methods for solving linear systems, compared with their expected rates

GMRES and MINRES residuals precisely match up and that all three outperform the splitting methods. Comparing their convergence with the upper bounds stated in Section 4.2.2 and Section 4.2.1, Figure 2b confirms that our implementation of CG and GMRES satisfies the convergence results (13) and (12).

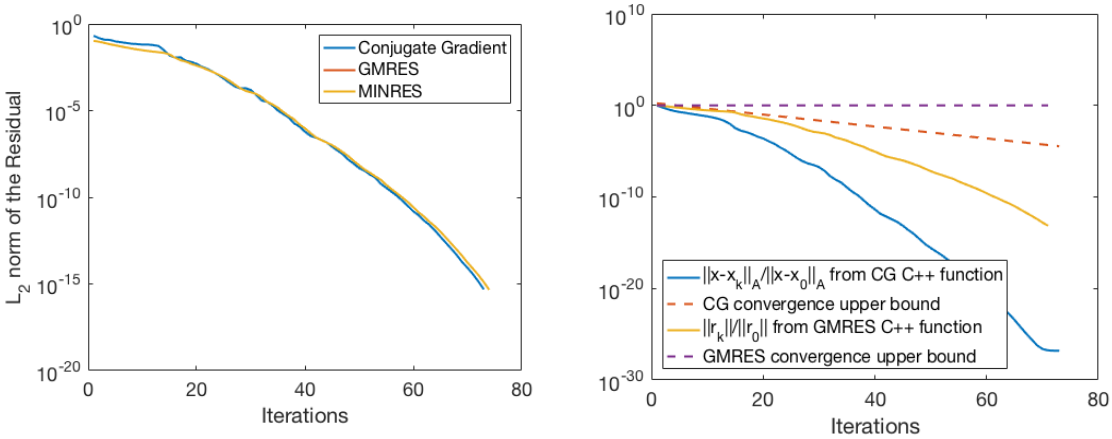

Figure 2: Convergence of Krylov subspace methods for solving linear systems

5.3 Computational Time

Having compared the number of iterations necessary for convergence of each method, we are also interested in the computational time taken by each method.

To track computational time, we instantiate a '`std::clock_t`', and then save the time taken as follows

```
// must #include <cstdio>, <ctime> and <chrono>
std::clock_t jacobistart = std::clock();
Vector x_j = jacobi(A,b);
double jacobitime = (std::clock() - jacobistart) / (double)CLOCKS_PER_SEC;
//... and so on for each method
```

Table 1 shows the CPU times recorded for each method as implemented in C++ on our model problem.

Method	**Computational Time (seconds)**
LUP Factorisation ('/')	71,3565
Jacobi	8.5782
Gauss-Seidel	4.20507
SOR	0.375897
Conjugate Gradient	0.213055
GMRES	0.148258
MINRES	0.136955

Table 1: Time taken for linear solvers to calculate the solution to the model problem. Simulations were run on a MacBook Air, 1,6 GHz Intel Core i5, 8 GB 1600 MHz DDR3.

The results we see are generally as expected, with the Krylov Subspace methods outperforming the iterative methods.[9]

6 Conclusion

In this paper we have described the implementation of a Matrix class and associated functions for solving linear systems of equations in C++. We have extensively tested our functions and solvers, and compared their performance when applied to Poisson's equation on the unit square. Our solvers perform as predicted analytically, and our functions cope as expected with the various edge cases tested.

[9]The initial results of this time comparison test showed GMRES and MINRES performing extremely poorly, even though their iteration count was low. This motivated a investigation using '`callgrind`', an automatic profiling tool [12] which, among other things, allows one to see which the most expensive operations in one's code are. This was instrumental in directing the amendments made to the GMRES and MINRES implementation, both of which now perform extremely well.

Future development of this class will involve adding extra linear algebra functionality. Some such additions would be fairly straightforward (for example, amending the `LU` function to return the determinant of the Matrix). Others, like computing the singular value decomposition, will be more involved. Furthermore, there might be a better approach to slicing Matrices which allows access to sub-sections of a Matrix without having to make a copies of them.

Finally, as a large amount of scientific computing deals with sparse matrices, an important future step will be to develop a separate Sparse Matrix class, to complement the class presented in this paper.

References

[1] Walter Edwin Arnoldi. The principle of minimized iterations in the solution of the matrix eigenvalue problem. *Quarterly of applied mathematics*, 9(1):17–29, 1951.

[2] James W Demmel. *Applied numerical linear algebra*. SIAM, 1997.

[3] Nicolai M Josuttis. *The C++ standard library: a tutorial and reference*. Addison-Wesley, 2012.

[4] Jaakko Jrvi. Boost C++ libraries. http://www.boost.org/doc/libs/1_64_0/libs/tuple/doc/tuple_users_guide.html. As seen on 2017/06/10.

[5] Jörg Liesen and Petr Tichỳ. Convergence analysis of krylov subspace methods. *GAMM-Mitteilungen*, 27(2):153–173, 2004.

[6] Scott Meyers. *Effective C++: 55 specific ways to improve your programs and designs*. Pearson Education, 2005.

[7] Joe Pitt-Francis. C++ for Scientific Computing: 2016-2017. http://www.cs.ox.ac.uk/people/joe.pitt-francis/C++ScientificComputing/. As seen on 2017/05/22.

[8] Joe Pitt-Francis and Jonathan Whiteley. *Guide to scientific computing in C++*. Springer Science & Business Media, 2012.

[9] Youcef Saad and Martin H Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on scientific and statistical computing*, 7(3):856–869, 1986.

[10] Yousef Saad. *Iterative methods for sparse linear systems*. SIAM, 2003.

[11] Lloyd N Trefethen and David Bau III. *Numerical linear algebra*, volume 50. Siam, 1997.

[12] Stanford University. Guide to callgrind. https://web.stanford.edu/class/cs107/guide_callgrind.html. As seen on 2017/06/22.

[13] James Hardy Wilkinson. *The algebraic eigenvalue problem*, volume 87. Clarendon Press Oxford, 1965.

Appendix A: Abridged Function Testing Summary			
Function	**Test Case**	**Expected Results**	**Result**
`LU`	Nonsingular Square Matrix	Return correct L and U matrices	✓
	Square Singular Matrix (which can be LU factorised)	Return correct L and U matrices (though at least one singular)	✓
	Identity Matrix	Return L and U both as identity matrices	✓
	Non-square Matrix	Return correct L and U matrices	✓
`QR`	Nonsingular Square Matrix	Return correct Q and R matrices	✓
	Singular Square Matrix	Return correct Q and R matrices	✓
	Identity Matrix	Return Q and R both as identity matrices	✓
	Non-square Matrix	Return correct Q and R matrices	✓
`eig`	Nonsingular Square Matrix	Return correct eigenvalues and eigenvectors	✓
	Singular Square Matrix	Return correct number of zero eigenvalues	✓
	Identity Matrix	Return all eigenvalues = 1, and canonical basis as eigenvectors	✓
	Non-square Matrix	Throw exception because Matrix is not square	✓
`applygivens`	a and b both non-zero	Return $\cos\theta$ and $\sin\theta$ such that applying the Givens rotation to [a,b] zeros b	✓
	a non-zero, b = 0	Return $\cos\theta = 1$, $\sin\theta = 0$ matrix	✓
`hess`	Dense Square Matrix	Return upper Hessenberg matrix with the same eigenvalues	✓
	Upper triangular Matrix	Return the matrix unchanged	✓
	Upper Hessenberg Matrix	Return the matrix unchanged	✓

Appendix B: Matrix Class Summary		
Attributes	**Type**	
`mElements`	double**	
`mRows`	int	
`mColumns`	int	
Constructors	**Parameters**	**Function**
(1) `zeros`	(int r, int c)	Constructs ($r \times c$) Matrix of zeros
(2) `fill`	(int r, int c, initialiserlist elements)	Constructs ($r \times c$) Matrix, with elements initialise (row-wise) with the contents of the initialiser list
(3) `copy`	(Matrix A)	Constructs a Matrix with the same elements as Matrix A
Destructor	**Parameters**	**Function**
∼`Matrix`	None	Frees dynamically allocated memory when Matrix goes out of scope
Methods	**Parameters**	**Function**
`getElements`	(int r1, int r2, int c1, int c2)	returns the sub matrix with rows r1 to r2, and columns c1 to c2 (= M(r1:r2, c1:c2) in MATLAB notation), as a Matrix
	(int r1, int r2, int c1, int c2, string "vec")	returns a Vector either, (i) containing the elements in column c1 (= c2) from rows r1 to r2, or (ii) the elements in row r1 (= r2) from columns c1 to c2 (= M(r1, c1:c2) or M(r1:r2,c1) in MATLAB notation). Either r1=r2 or c1=c2.
`setElements`	(int r1, int r2, int c1, int c2, Vector v)	sets either the elements in rows r1 to r2, in column c1 (= c2), or the elements in columns c1 to c2, in row r1 (= r2), to the values in the Vector v.
	(int r1, int r2, int c1, int c2, Matrix A)	sets the elements in rows r1 to r2, columns c1 to c2, to the values in the Matrix A.
`applygivens`	(int i, int j, double c, double s)	Applies Givens rotation affecting rows i and j defined by $\cos\theta$ = c and $\sin\theta$ = s.

Operators	**Parameters**	**Function**
+, -	(Matrix A, Matrix B)	Binary operator, performs matrix addition (C = A+B) and subtraction (C = A-B), element wise, and returns the Matrix C
*	(Matrix A, Matrix B)	Binary operator, performs matrix multiplication C = AB and returns the Matrix C. Does not allow multiplication in the case of mismatched dimensions
	(double c, Matrix B) or (Matrix B, double c)	Binary operator, performs scalar-matrix multiplication C = c*B or C = B*c, and returns Matrix C
	(Vector v, Matrix B) or (Matrix B, Vector v)	Binary operator, performs vector-matrix multiplication u = v*B or u = B*v, and returns the Vector u. Does not allow multiplication in the case of mismatched dimensions
/	(Matrix A, Vector b)	Binary operator, solves Ax=b by Gaussian Elimination, and returns the Vector x
!	(Matrix A)	Unary operator, returns B = $A^\top$, the transpose of Matrix A
-	(Matrix A)	Unary operator, returns B = -A
=	(Matrix A)	Assignment Operator
( )	(int r, int c)	Used for indexing, provides access to element in row r, column c. Can be used for assignment and retrieval.
<<	(std::ostream, Matrix)	Outputs a Matrix in a readable format, with each row on a new line, entries separated with commas, and the matrix surrounded by square brackets

Friend Functions	**Parameters**	**Function**
`ewisemult`	(Matrix A, Matrix B)	performs element-wise multiplication of Matrices A and B (A.*B in MATLAB notation), returns the resulting Matrix.
`outer`	(Vector u, Vector v)	calculates the outer-product of Vectors u and v, and returns the resulting the Matrix

`kron`	(Matrix A, Matrix B)	returns the kronecker product of A and B, C = A $\otimes$ B
`size`	(Matrix A)	returns a Vector whose first and second elements are the number of rows and columns in A respectively
`eye`	(int n)	returns and $n \times n$ identity matrix
`diag`	(Vector v)	returns a square diagonal Matrix, with the Vector v along leading diagonal
	(Vector v, int loc)	returns a square diagonal Matrix, with the Vector v along the diagonal specified by loc (-1 for the sub-diagonal, etc.)
	(Matrix A)	returns a Vector whose elements correspond to the diagonal elements of Matrix A
`hess`	(Matrix A)	returns the resulting matrix after reducing Matrix A to Hessenberg form
`LU`	(Matrix A)	calculates the LU factorisation of A, and returns the Matrices L and U, as well as the permutation matrix P. Return format is boost::tuple<U, P>.
`QR`	(Matrix A)	calculates the QR factorisation of A and returns the matrices Q and R. Return format is std::pair<Q, R>
`eig`	(Matrix A)	calculates the eigenvalues and eigenvectors of A, and return a Vector e of the eigenvalues, and a Matrix V whose columns are the corresponding eigenvectors. Return format is std::pair<e,V>
`givens`	(double a, double b)	returns a `std::pair` containing $\cos\theta$ and $\sin\theta$ for the Givens rotation which, when applied to [a,b], will zero b.
`backsubst`	(Matrix A, Vector b, int width = 0)	Solves the system Ax=b, where A is upper triangular, by back-substitution. The width parameter can be used to specify the bandwidth of A, if applicable, for efficiency. Vector x

`forwardsubst`	(Matrix A, Vector b, int width = 0)	Solves the system Ax=b, where A is lower triangular, by forward-substitution. The width parameter can be used to specify the bandwidth of A, if applicable, for efficiency. Returns the Vector x
`CG`	(Matrix A, Vector b)	Solves the system Ax=b, where A is symmetric positive definite, using the conjugate gradient method. Returns the Vector x
`GMRES`	(Matrix A, Vector b)	Solves the system Ax=b, using the GMRES method. Returns the Vector x
`MINRES`	(Matrix A, Vector b)	Solves the system Ax=b, where A is symmetric, using the MINRES method. Returns the Vector x
`jacobi`	(Matrix A, Vector b)	Solves the system Ax=b, using Jacobi's method. Returns the Vector x
`GS`	(Matrix A, Vector b)	Solves the system Ax=b, using the Gauss-Seidel method. Returns the Vector x
`SOR`	(Matrix A, Vector b, double w)	Solves the system Ax=b, using the Successive Over-Relaxation method, with the relocation parameter w. Returns the Vector x
`EqualDim`	(Matrix A, Matrix B)	Helper function which returns `true` if size(A) = size(B), and returns `false` otherwise
`CanMultiply`	(Matrix A, Matrix B)	Helper function which returns `true` if the number of columns in Matrix A equals the number of rows in Matrix B, and returns `false` otherwise

Additions to Vector Class

Functions	**Parameters**	**Function**
`ones`	(int length)	Returns a Vector ones of the specified length
`linspace`	(double start, double end, int points)	Returns a Vector of n equispaced points beginning with 'start' and ending with 'end'
`reshape`	(Vector u, int r, int c)	Returns an r × c Matrix whose elements are taken columnwise from the Vector u

Table 4: A summary of the contents of the Matrix Class. Note that '`const&`' specifications have been omitted from parameter descriptions.

Appendix C: Algorithms

Algorithm 1 Conjugate Gradient Algorithm

$\mathbf{r}_0 := \mathbf{b} - \mathbf{A}\mathbf{x}_0$
$\mathbf{p}_0 := \mathbf{r}_0$
for $|\mathrm{r}_k| >$ Tol **do**
    $\alpha_k := \frac{\mathbf{r}_k^\mathsf{T}\mathbf{r}_k}{\mathbf{p}_k^\mathsf{T}\mathbf{A}\mathbf{p}_k}$
    $\mathbf{x}_{k+1} := \mathbf{x}_k + \alpha_k\mathbf{p}_k$
    $\mathbf{r}_{k+1} := \mathbf{r}_k - \alpha_k\mathbf{A}\mathbf{p}_k$
    if $|\mathbf{r}_k| <$ Tol **then**
        Break
    end if
    $\beta_k := \frac{\mathbf{r}_{k+1}^\mathsf{T}\mathbf{r}_{k+1}}{\mathbf{r}_k^\mathsf{T}\mathbf{r}_k}$
    $\mathbf{p}_{k+1} := \mathbf{r}_{k+1} + \beta_k\mathbf{p}_k$
    k := k + 1
end for
return $\mathbf{x}_{k+1}$

Algorithm 2 Arnoldi Algorithm

Guess $\mathbf{x}_0$ (say = $\mathbf{0}$)
$\mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0$
$\mathbf{v}_1 = \frac{\mathbf{r}_0}{||\mathbf{r}_0||}$
for k = 1, 2, 3... **do**
    $\mathbf{w} = \mathbf{A}\mathbf{v}_k$
    for j = 1,2, ... k **do**
        $h_{j,k} = \mathbf{v}_j^\mathsf{T}\mathbf{w}$
        $\mathbf{w} = \mathbf{w} - h_{j,k}\mathbf{v}_j$
    end for
    if $||\mathbf{w}||_2 = 0$ **then**
        Break
    end if
    $h_{k+1,k} = ||\mathbf{w}||_2$
    $\mathbf{v}_{k+1} = \frac{\mathbf{w}}{h_{k+1,k}}$
end for

Algorithm 3 Lanczos Algorithm

Guess $\mathbf{x}_0$ (say = $\mathbf{0}$)
Take $\gamma_1 = 0$ and '$\mathbf{v}_0$' = $\mathbf{0}$
$\mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0$
$\mathbf{v}_1 = \frac{\mathbf{r}_0}{||\mathbf{r}_0||}$
for k = 1, 2, 3... **do**
    $h_{k-1,k} = \gamma_k$
    $\mathbf{w} = \mathbf{A}\mathbf{v}_k$
    $h_{k,k} = \delta_k = \mathbf{v}_k^\mathsf{T}\mathbf{w}$
    $\mathbf{w} = \mathbf{w} - h_{k,k}\mathbf{v}_k - \gamma_k\mathbf{v}_{k-1}$
    $h_{k+1,k} = \gamma_{k+1} = ||\mathbf{w}||_2$
    if $||\mathbf{w}||_2 = 0$ **then**
        Break
    end if
    $\mathbf{v}_{k+1} = \frac{\mathbf{w}}{h_{k+1,k}}$
end for

Appendix D: Splitting methods

Jacobi's Method

Jacobi's method [2] is an iterative method for solving linear systems of the for $\mathbf{Ax} = \mathbf{b}$. Taking $\mathbf{M} = \text{diag}(\mathbf{A})$ and $\mathbf{N} = \mathbf{A} - \mathbf{M}$, we calculate the '$k+1$th approximation $\mathbf{x}_{(k+1)}$ at every step with

$$\mathbf{x}_{(k+1)} = -M^{-1}N\mathbf{x}_{(k)} + M^{-1}\mathbf{b}. \tag{17}$$

For the purposes of understanding how the algorithm is coded, it may be more helpful to consider the formula to calculate eich element of $\mathbf{x}^{(k+1)}$

$$x^i_{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j\neq i} a_{ij}x^j_{(k)}\right), \quad i = 1, 2, \ldots, n. \tag{18}$$

The Jacobi iteration matrix is $T_J = -M^{-1}N$.

```
Vector jacobi(const Matrix& A, const Vector& b) {
int n = length(b);
Vector x(n);
Vector xnext(n);
Vector r = b - A*x;
double Tol = 1e-14;
int iter = 0;
while (norm(r) > Tol & iter < 10000) {
for (int i = 1; i<=n; i++) {
   double sum = 0;
   for (int j=1; j<=n; j++) {
      if (i != j) { sum = sum + A(i,j)*x(j);}
      else if (i==j) {}}
   xnext(i) = (b(i) - sum)/A(i,i);}
   r = b - A*xnext;
   x = xnext;";
   iter++; }
if (iter>=10000) {throw Exception("convergence error",
"jacobi failed to converge");}
return xnext;}
```

Gauss-Seidel (G-S)

Gauss-Seidel [2] is very similar to Jacobi's method, except in that it always uses the most recently updated elements of $\mathbf{x}$ available when performing the update. This

means that the algorithm for calculating the ith element of $\mathbf{x}_k$ is given by

$$x^i_{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x^j_{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x^j_{(k)} \right), \quad i = 1, 2, \ldots, n. \tag{19}$$

This corresponds to splitting $A = L + D + U$, with

$$(L + D)\mathbf{x}^{(k+1)} = -U\mathbf{x}^{(k)} + \mathbf{b}, \tag{20}$$

making the iteration matrix in this case $T_{GS} = -(L + D)^{-1}U$. It can be proven that the spectral radius $\rho(T_{GS}) = [\rho(T_J)]^2$.

```
Vector GS(const Matrix& A, const Vector& b)
{
int n = length(b);
Vector x(n);
Vector xnext(n);
Vector r = b - A*x;
double Tol = 1e-14;
int iter = 0;
while (norm(r) > Tol & iter < 10000) {
   for (int i = 1; i<=n; i++) {
      double sum1 = 0; double sum2 = 0;
      if (i!=1) {
         for (int j=1; j<i; j++) {
            sum1 = sum1 + A(i,j)*xnext(j);}}
      if (i!=n){
         for (int j=i+1; j<=n; j++) {
            sum2 = sum2 + A(i,j)*x(j);}}
      xnext(i) = (b(i) - sum2 - sum1)/A(i,i);}
   r = b - A*xnext;
   x = xnext;
   iter++; }
   if (iter>=10000) {throw Exception("convergence error",
   "GS failed to converge");}
   return xnext; }
```

Successive Over-Relaxation (SOR)

Successive Over-Relaxtion [2] is a variation of the Gauss-Seidel method, but which converges faster. Using the same Matrix splitting as described above in Section 6,

and introducing a parameter $w > 1$ which is known as the relaxation factor, we can represent our system $Ax = b$ as

$$(D + wL)\mathbf{x} = w\mathbf{b} - [wU + (w-1)D]\mathbf{x} \tag{21}$$

Like G-S, SOR calculates x_{k+1} using the most recently updates x elements. In matrix notation each iterate is given by

$$\mathbf{x}^{(k+1)} = (D + wL)^{-1}\big(w\mathbf{b} - [wU + (w-1)D]\mathbf{x}^{(k)}\big) = L_w\mathbf{x}^{(k)} + \mathbf{c}, \tag{22}$$

and this means that at each iteration, each element is updated according to the rule:

$$x_i^{(k+1)} = (1-w)x_i^{(k)} + \frac{w}{a_{ii}}\left(b_i - \sum_{j<i} a_{ij}x_j^{(k+1)} - \sum_{j>i} a_{ij}x_j^{(k)}\right), \quad i = 1, 2, \ldots, n. \tag{23}$$

The key difference between G-S and SOR is the relaxation parameter w. It can be proven that for a given matrix, the optimal w value which yields the fastest convergence is given by

$$w = \frac{2}{1 + \sqrt{1 - \rho(T_{GS})}}. \tag{24}$$

It has further been proven that when w is set to this optimal value, the spectral radius of the iteration matrix is given by $\rho(T_{SOR}) = w - 1$.

```
Vector SOR(const Matrix& A, const Vector& b, double w) {
int n = length(b);
Vector x(n);
Vector xnext(n);
Vector r = b - A*x;
double Tol = 1e-14;
int iter = 0;
while (norm(r) > Tol & iter < 10000) {
   for (int i = 1; i<=n; i++) {
      double sum1 = 0; double sum2 = 0;
      if (i!=1){
         for (int j=1; j<i; j++) {
            sum1 = sum1 + A(i,j)*xnext(j); }}
      if (i!=n){
         for (int j=i+1; j<=n; j++) {
            sum2 = sum2 + A(i,j)*x(j);}}
      xnext(i) = (1-w)*x(i) + (b(i) - sum2 - sum1)*(w/A(i,i));}
   r = b - A*xnext;
```

```
    x = xnext;
    iter++; }
if (iter>=10000) {throw Exception("convergence error",
"SOR failed to converge");}
return xnext; }
```

Appendix E: Deriving the matrix equation of the model problem

Poisson's equation is

$$-\nabla^2 u = f. \tag{25}$$

. We will solve this equation in $\mathbb{R}^2$, with domain $\Omega = [0,1]^2$. x and y are discretised with intervals of length Δx and Δy respectively, such that $x_i = i\Delta x$ and $y_i = i\Delta y$.

Denoting the value of the function u at position (x_i, y_j) as $u_{i,j}$, u_{xx} and u_{yy} are approximated with a finite difference scheme,

$$\frac{\partial^2 u_{i,j}}{\partial x^2} \approx \frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{\Delta x^2}, \tag{26}$$

$$\frac{\partial^2 u_{i,j}}{\partial y^2} \approx \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{\Delta y^2}. \tag{27}$$

Assuming homogeneous Dirichlet boundary conditions, we can use these finite difference approximations of the partial derivates to represent the problem for the funciton values on the interior of the domain with a linear system $A\mathbf{u} = \mathbf{f}$, with $A \in \mathbb{R}^{(n-1)^2 \times (n-1)^2}$ defined as

$$A = \begin{bmatrix} B & -C & 0 & \cdots & \cdots & 0 \\ -C & B & -C & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots & \\ \vdots & & 0 & -C & B & -C \\ 0 & \cdots & \cdots & 0 & -C & B \end{bmatrix}, \quad B = \begin{bmatrix} 4 & -1 & 0 & \cdots & \cdots & 0 \\ -1 & 4 & -1 & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots & \\ \vdots & & 0 & -1 & 4 & -1 \\ 0 & \cdots & \cdots & 0 & -1 & 4 \end{bmatrix},$$

where $B \in \mathbb{R}^{(n-1)\times(n-1)}$ and C is an $(n-1) \times (n-1)$ identity matrix. Note that $\mathbf{u}$ and $\mathbf{f}$ are vectors of the form

$$\mathbf{u} = [u_{1,1}, u_{2,1}, \ldots, u_{n-1,1}, u_{1,2}, \ldots u_{n-1,2}, \ldots u_{n-1,n-1}] \tag{28}$$

$$\mathbf{f} = [f_{1,1}, f_{2,1}, \ldots, f_{n-1,1}, f_{1,2}, \ldots f_{n-1,2}, \ldots f_{n-1,n-1}] \tag{29}$$

Appendix F: Other Code

Partial Pivoting as part of the LUP factorisation function

```
// Partial Pivoting as part of the LUP decomposition function
double max = U(k,k);
int index = k;
for (int i = k; i<=m; i++) {
if (std::abs(U(i,k)) > std::abs(max))  {
index = i;
max = U(i,k);} }
pivot = eye(m);
if (index != k) {
Vector temp = U.getElements(k,k,1,n,"vector");
U.setElements(k,k,1,n,U.getElements(index,index,1,n,"Vector"));
U.setElements(index,index,1,n,temp);
Vector temp2 = L.getElements(k,k,1,m,"vector");
L.setElements(k,k,1,m,L.getElements(index,index,1,m,"Vector"));
L.setElements(index,index,1,m,temp2);
pivot(index,index)=0; pivot(k,k) = 0;
pivot(index,k) = 1; pivot(k,index) = 1;
P = pivot*P; }
```

Calculating $\mathbf{v}_k$ and $\hat{H}_k$ in the MINRES function

```
while (norm_r > Tol) {
... // Iteration count is being stored in the variable k
Vector w = A*v_k;
double delta = v_k*w;
H(k,k) = delta;
if (k>1) {
H(k-1,k) = gamma;
w = w - delta*v_k - gamma*V.getElements(1,n,k-1,k-1,"Vector");}
else {  w = w - delta*v_k; }
gamma = norm(w);
H(k+1,k) = gamma;
```

Row Operations as part of LUP Factorisation

```
// Zero the kth sub-diagonal column, and update matrix L
```

```
Matrix Lk = eye(n);
for (int i = k+1; i<=n;i++) {
Lk(i,k) = U(i,k)/U(k,k);
for (int j=k+1;j<=n;j++) {
U(i,j) = U(i,j) - Lk(i,k)*U(k,j); }
U(i,k) = 0; }
L = L*pivot*Lk;
```

Importantly, we must remember to apply the permutation necessary for that column ('`pivot`' in the code snippet above), to our Lk matrix before using it to update our L matrix.

GMRES

```
Vector GMRES(const Matrix& A, const Vector& b) {
   int n = length(b);
   Vector x(n);
   Vector temp1 = A*x;
   Vector r = b - temp1;
   double norm_r = norm(r);
   Vector v = r/norm_r;
   Matrix V(length(v),n);
   Matrix H(n+1,n);
   Vector r_e1(n+1);
   r_e1(1) = norm_r;
   Matrix G = eye(n+1);
   V.setElements(1,n,1,1,v);
   double Tol = 1e-15;
   int k=0;
   while (norm_r > Tol) {
      k = k+1;
      Vector v_k = V.getElements(1,n,k,k,"Vector");
      Vector w = A*v_k;
      for (int j = 1; j<=k; j++) {
         Vector v_j = V.getElements(1,n,j,j,"Vector");
         double h = v_j*w;
         H(j,k) = h;
         w = w - h*v_j; }
      double hplus = norm(w);
      H(k+1,k) = hplus;
      if (hplus<1e-15) {break;}
      else {
```

```
            Vector v_kplus1 = w/hplus;
            V.setElements(1,n,k+1,k+1,v_kplus1);
            Matrix Hhat = H.getElements(1,(k+1),1,k);
            Hhat = G.getElements(1,k+1,1,k+1)*Hhat;
            std::pair<double,double> g = givens(Hhat(k,k),Hhat(k+1,k));
            double gcos = g.first; double gsin = g.second;
            G.applygivens(k,k+1,gcos,gsin);
            norm_r = std::abs(norm_r*gsin);}
        }
        r_e1.setValues(1,k+1,G.getElements(1,k+1,1,k+1)*r_e1.getValues(1,k+1));
        H.setElements(1,k+1,1,k,
            G.getElements(1,k+1,1,k+1)*H.getElements(1,(k+1),1,k));
        Vector y = backsubst(H.getElements(1,k,1,k),r_e1.getValues(1,k));
        Vector xfinal = temp1 + V.getElements(1,n,1,k)*y;
        return xfinal;
    }
```