C++ for Scientific Computing

Mark Richardson

May 2009

Contents

1 Introduction

The aim of this special topic is to develop C++ code that mirrors the functionality of the scientific computation package, MATLAB. In particular, we will attempt to write code in C++ that enables the user to perform most of the basic, and some of the more complicated matrix and vector operations found in MATLAB. A natural question to ask is *why* should we be interested in replicating the functionality of an existing piece of software? In order to address this, let's briefly look the history of, and what is offered by, respectively, MATLAB and C++.

During the late 1970s, a lecturer at the University of New Mexico, Cleve Moler, wished to give his students access to the linear algebra software libraries, LINPACK and EISPACK, though without them having without them having to learn FORTRAN, the language these software libraries were written in. MATLAB was first developed by Moler in order to allow this access and after going from strength to strength, in 1984 MATLAB was first released as a commercial product. MATLAB has developed a great deal since then and is now widely used in industry and academia for a wide range of problems in scientific computing. It would not be unfair to say that MATLAB is generally considered to be the benchmark tool for computations in linear algebra. MATLAB is popular because of its ease of use. It performs calculations with vectors and matrices, and though MATLAB can perform these computations speedily under certain circumstances, other common programming languages such as C, C++, FORTRAN are generally considered to be faster.

Around the same time at Bell Labs, New Jersey, C++ was being created by Bjarne Stroustrup. C++ was concieved as an extension to the language, 'C', which provided support for classes and consequently, object oriented design. The name of the language, 'C++' is a humorous reference to the command from C for incrementing an integer (to increment an integer, x we would write `x++`). C++ is thus C incremented. C++ has the advantage of having both high and low level features. The object oriented aspect of C++ allows code to be abstracted far away from the machine running the code. However, it is also possible to write code in C++ which accesses the computer's memory directly. C++ has been incredibly successful and is often the language of choice for programmers who wish to develop a piece of professional software.

As powerful as MATLAB undoubtedly is, if we could replicate its functionality in another language, such as C++, we would be free from the shackles of the MATLAB commercial license, and would be able to make any extensions we saw fit. We could even design our own user interface and place it within a separate application. In addition to this, the speed at which code written in C++ executes is incentive enough to make replicating MATLAB commands a worthwhile task.

We will first define a class of matrices. This will be achieved by specifying in our code how the matrix *constructor* should allocate memory to store the details of the matrix, whenever a matrix object is created by the user. In addition to this, we will need to define an algebra for matrix objects, that is we will need to specify exactly what it means (for example) to add, subtract and multiply matrix objects together. Once we have established this basic algebra, we will go on to define a sub-class of vectors. It is then natural to look at trying to implement some of the more sophisticated linear algebra tools. In this project, we will be focussing on solving linear systems of equations. Two methods we shall code for doing this will be the ubiquitous direct method, Gaussian Elimination with partial pivoting, and the Krylov subspace iterative method, Generalised Minimal Residual (GMRES).

2 The Matrix Class

Our class of matrices is defined using two files: a header file, *matrix.h*, and C++ code file, *matrix.cpp*. If we wish to run any code that uses this class, we require a *main.cpp* file. Contained within the `main()` scope within this file, we can place the name of other pieces of code that we wish to be executed. The test code for the matrix class is found on a file named, *use_matrix.cpp*, and is referred to within the `main()` scope in the *main.cpp* file as `use_matrix()`.

The header file of a class defines all the *variables* and *methods* that are associated with the class. There are three member variables for our class of matrices; the number of rows of the matrix (`rows`), the number of columns (`columns`), and a pointer to the first entry of a vector of pointers (`**x`) to double precision numbers. This is the standard way of declaring an array of numbers and entries of the matrix can henceforth be accessed using the declaration, `x[i][j]`, where i and j are integers. The methods of the matrix class are functions that act upon variables of a matrix object, or upon the object itself. They are declared in the *matrix.h* header file and the code that implements them is contained in the *matrix.cpp* file.

Each of the variables and methods must be declared as; `public`, `private` or `protected`. The variables of the matrix class are declared as, `protected`, since later we will define a sub-class of vectors which will require to have access to the variables of the matrix class. The methods and constructors are declared as, `public`.

2.1 Constructors

Any particular instance of the matrix class is known as a matrix *object*. Whenever such an object is created, a special routine, called the *constructor* is executed. The job of the constructor is to allocate memory for the variables of the object. The matrix class has three constructors; a default constructor, a constructor which takes the matrix dimensions as arguments, and a copy constructor. In addition to this, we also declare a *destructor* which deletes the memory of an object once it goes out of scope. The constructor which creates objects by taking two integer arguments is:

```
matrix::matrix(int no_of_rows, int no_of_columns)
{
    // set the variable values
    rows = no_of_rows;
    columns = no_of_columns;
    xx = new double *[no_of_rows];
    for (int i=0; i<no_of_rows; i++)
    {
        // allocate the memory for the entries of the matrix
        xx[i] = new double[no_of_columns];
    }
    for (int i=0; i<no_of_rows; i++)
    {
        for (int j=0; j<no_of_columns; j++)
        {
            // initialise the entries of the matrix to zero
            xx[i][j]=0.0;
        }
    }
}
```

We see that in lines 4 & 5, integer values are assigned to the *rows* and *columns* fields. Line 6 assigns the memory by creating a vector of pointers (corresponding to the number of rows of the matrix). The loop in lines 7-11 then allocates the number of columns for each row. The final loop initialises the content of the matrix to 0.0. The other constructors and the destructor can be found in the appendix.

2.2 Binary Operators

Now that we are able to create matrix objects, we can go about defining what it means to add, subtract, multiply and divide matrices by scalars and other matrices. This is done by *overloading* the binary operators, $+, -, *, /$. Much of this is repetitive, so we shall only describe a few key examples in detail. The full code can be found in the appendix. The following code tells the matrix class what to do if it is asked to add two matrices together.

```
1  matrix operator +(const matrix& A, const matrix& B)
2  {
3      int m = rows(A), n = columns(A), p = rows(B), q = columns(B);
4      if ( m != p || n !=q )
5      {
6          std::cout << "Inconsistent dimensions: Returned first argument";
7          return A;
8      }
9      else
10     {
11         matrix C(m,n);
12         for (int i=0; i<m; i++)
13         {
14             for (int j=0; j<n; j++)
15             {
16                 C.xx[i][j]=A.xx[i][j]+B.xx[i][j];
17             }
18         }
19         return C;
20     }
21 }
```

There is a simple, unsophisticated dimension check in lines 4-8. If the dimensions of the matrix are not identical, then an error message is displayed and the first argument (A) is returned to the user. There are better ways of dealing with such situations (exception handling), but we shall not be discussing them in any great detail here. If the dimensions are consistent, then the code in lines 10-20 is executed. This consists of two nested loops over the rows and columns of the matrix. Quite simply, the sum of the entries of the input matrices (A & B) are added and assigned to the corresponding entry of the output matrix (C). The operator '$-$' is overloaded in a directly analogous way to this. Note that the overloading of '$+$' uses the functions, `rows()` and `columns()`, which are defined as friend functions and return the row and column dimensions. The function `rows()` is defined by:

```
int rows(matrix A);          // function prototype
int rows(matrix A)
{
    return A.rows;           // Returns the protected field, 'rows'
}
```

The unary operators, $+, -$ are defined naturally using similar code to that of the binary operators. To multiply and divide matrix objects by scalars, we use the following code:

```
matrix operator *(const double& p, const matrix& A)
{   // Creates a matrix with the same dimensions as A
    matrix B(A.rows,A.columns);
        for (int i=0; i<A.rows; i++)
        {
            for (int j=0; j<A.columns; j++)
            {
                B.xx[i][j]= p * A.xx[i][j];     // Multiply each entry by p
            }
        }
        return B;
}
```

2.3 Matrix Multiplication

Multiplying two matrices $\mathbf{A} * \mathbf{B}$, where $A \in \mathbb{R}^{m \times n}$ and $\mathbf{B} \in \mathbb{R}^{n \times p}$, results in a matrix, $\mathbf{C} \in \mathbb{R}^{m \times p}$ where the $(i, j)^{th}$ entry of $\mathbf{C}$ is given by:

$$\mathbf{C}_{i,j} = \sum_{k=1}^{n} \mathbf{A}_{i,k} \mathbf{B}_{k,j}$$

For each $1 \leq i \leq m$ and $1 \leq j \leq p$. To overload the '$*$' operator for two matrix arguments, the following code which implements the above expression is used. It involves three nested loops (one for the rows, one for the columns, and one for the sum for each entry):

```
matrix operator *(const matrix& A, const matrix& B)
{   // use assertion to check matrix dimensions are consistent
    assert(A.columns==B.rows);

    // create a result matrix, C with the correct dimensions
    matrix C(A.rows,B.columns);

    // initialise temp (sum variable)
    double temp = 0;

        for (int i=0; i < A.rows ; i++)          // for rows (m)
        {
            for (int j=0; j < B.columns ; j++)   // for columns (q)
            {
                for (int k=0; k < A.columns ; k++)
                {   // dot product step (n sums)
                    temp = temp + A.xx[i][k]*B.xx[k][j];
                }

                C.xx[i][j]=temp;    // set the C matrix values to temp
                temp = 0;           // reset temp
            }
        }
    return C;
}
```

Here, we have used a simple assertion to check if the operation is dimensionally consistent. The assertion ends the code immediately if the statement in the brackets is untrue. Note that this code is general enough for us to automatically have defined matrix-vector multiplication, if we define an m-vector as an $m \times 1$ matrix.

2.4 Achieving MATLAB-like functionality

Now that we have established a basic algebra for matrix objects, we can go about making the code 'feel' more like MATLAB. The two most important ways of doing this are by overloading the assignment operator, '=' and the brackets operator, '()'. In MATLAB if **P** were a matrix, and we were to write `Q=P`, then we would have formed an identical copy of **P**, called **Q**. We need to explicitly tell our matrix class how this should be done. The following code allows us to do this, but only after initially declaring a matrix object that the RHS is being assigned to (this is different to how things work in MATLAB, but unavoidable here).

```
1  matrix& matrix::operator =(const matrix &A)
2  {// Operator returns a matrix equal to the RHS
3
4      // Destruct previous entries
5      for (int i=0; i < rows; i++)
6          {
7              delete[] xx[i];
8          }
9      delete[] xx;
10
11     // Assign new dimensions to be equal to that of the RHS
12     rows = A.rows;
13     columns = A.columns;
14
15     // Allocate the memory as in the constructor
16     xx = new double *[A.rows];
17         for (int i=0; i < A.rows ; i++)
18         {
19             xx[i] = new double[A.columns];
20         }
21
22         // Copy the values across from the RHS
23         for (int i=0; i < A.rows; i++)
24         {
25             for (int j=0; j < A.columns ; j++)
26             {   // Set entries to be the same as the RHS matrix
27                 xx[i][j]=A.xx[i][j];
28             }
29         }
30     return *this;
31 }
```

In all cases, we begin with a matrix object (on the LHS of the '=' sign) that the RHS matrix is being assigned to. Lines 5-9 of the above code delete any data that was there previously (since the user has decided that this object is to be overwritten). This code is very similar to that found in the destructor. Lines 12 and 13 then reset the dimensions of the target matrix to be equal to that of the RHS whilst lines 16-20 reallocate memory just as the constructor did. We now have a matrix with object with the correct dimensions. The last remaining task is to copy across the values from the RHS matrix. This is achieved in lines 23-29.

In MATLAB, it is possible to access and write to elements of a matrix, **A**, by using the command `A(i,j)`, where i and j are the row and column entries of the matrix. To achieve this effect for our matrix class, we use the following code. Note that again we are using a simple check to ensure that the inputted values are within the dimension bounds of the matrix, though this time we are not using an assertion, merely a warning message.

```
double &matrix::operator() (int i, int j)
// Allows reference to the entries of a matrix in
// the same way as MATLAB. Can call or assign values.
{
    if (i < 1 || j < 1)
    {
        std::cout << "Warning: An index may have been too small \n\n";
    }

    else if (i > rows || j > columns)
    {
        std::cout << "Warning: An index may have been too large \n\n";
    }
    return xx[i-1][j-1];
}
```

In order to debug code effectively, we would like to be able to display the contents of matrices easily. For example, we would like to be able to simply write `std::cout << A;` to display the contents of the matrix, A. This can be achieved by overloading the '`<<`' operator in the following way:

```
ostream& operator<< (ostream& output, const matrix &A)
{
    for (int i=0; i<A.rows; i++)
    {   // work down the rows of the matrix

        for (int j=0; j < A.columns; j++)

        {   // send the entry in each column to output, with a space
            output << " " << A.xx[i][j];
        }

        // begin newline in output when the end of the row is reached
        output << "\n";
    }
    output << "\n";
    return output;
}
```

2.5 Special Functions and Operations

We will now attempt build up a small library of functions that perform tasks that will be useful to use when we attempt to write code for solving linear systems using either Gaussian Elimination or GMRES. To create an $n \times n$ identity matrix in MATLAB, we would write something like, `I=eye(n)`. We can obtain the same functionality with our matrix class by using the following code:

```
matrix eye(int size)
{   // Create a temporary matrix with the same dimensions as A
    matrix I(size,size);

    for (int i=0; i < size; i++)
    {   // set the values on the diagonal to be 1

            I.xx[i][i] = 1;
    }
    return I;
}
```

For the partial pivoting steps in the Gaussian Elimination algorithm (which will be explained in greater detail later), we will need to identify the pivot for a given column of a matrix (this is the largest entry that is either on the diagonal or below the diagonal for a given columns). The code that achieves this takes a matrix and an integer (the column number you wish to find the pivot for) as inputs and returns the row number corresponding to the largest diagonal/sub-diagonal value for that column:

```
1  int find_pivot(matrix A, int column)
2  {
3      // Initialise maxval to be diagonal entry in column, 'column'
4      double maxval=fabs(A(column,column));
5      // Initialise rowval to be column
6      int rowval=column;
7
8          for (int i=column+1; i ≤ A.rows; i++)
9          {
10             if ( fabs(A(i,column)) > maxval)
11             {   // Update maxval and rowval if bigger than previous maxval
12                 maxval = fabs(A(i,column));
13                 rowval = i;
14             }
15         }
16     return rowval;
17 }
```

The code works by initialising the variable 'maxval' to be the diagonal entry of the column that is specified by the user as one of the input arguments. The quantity 'rowval' is initialised to be the same as the input argument, 'column'. In the 'for' loop in lines 8-15, the code cycles down the entries below the sub-diagonal and asks if each entry is larger than 'maxval'. If it is, then 'maxval' and 'rowval' are updated. The returned integer value is thus the row corresponding to the largest diagonal/sub-diagonal entry.

Once this information has been obtained, it needs to be used to obtain a permutation matrix which will switch the two rows in question. A function which accepts two integers (representing the two rows to be swapped) and which returns the appropriate permutation matrix is defined using the code below.

We simply create an appropriately sized identity matrix and manually swap over rows i and j by setting the diagonal entries on those rows to zero and the (i, j) and (j, i) entries to one:

```
matrix permute_r(int n, int i, int j)
{
    // Create nxn identity matrix
    matrix I=eye(n);

    // set to zero the diagonal entries in the given rows
    I(i,i)=0;
    I(j,j)=0;

        // set the appropriate values to be 1
    I(i,j)=1;
    I(j,i)=1;

    return I;
}
```

A function that will be required by the GMRES routine is a function that takes a matrix and two integers as inputs and returns a re-sized matrix with dimensions corresponding to the integer inputs. Any entries that are shared by the inputted and outputted matrix are copied across, whilst any entries that are not shared are either lost (if the new dimensions are smaller than the previous dimensions) or set to zero (if the converse is true). This resizing operation can be achieved with the following code:

```
matrix resize(matrix A, int m, int n)
{
    int p,q;
    matrix Mout(m,n);

    if (m≤A.rows)       // set p as the lowest of the two row dimensions
    {
        p=m;
    }
    else
    {
        p=A.rows;
    }

    if (n≤A.columns)    // set q as the lowest of the two column dimensions
    {
        q=n;
    }
    else
    {
        q=A.columns;
    }

    for (int i=1; i≤p; i++)      // loop across the smallest row dimension
    {
        for (int j=1; j≤q; j++)  // loop across the smallest column dimension
        {
            Mout(i,j) = A(i,j);  // copy across the appropriate values
        }
    }

    return Mout;
}
```

Lastly, we define the method for computing the transpose of a matrix. This will be used in the GMRES code for computing inner products. In MATLAB, once a matrix object, $\mathbf{X}$ exists, its transpose is computed by the command, `X'`. It is not as straightforward to allow this command mean transpose in C++, so we will assign the ’ ’ symbol placed before the object to represent transpose. It is therefore reasonable to class this as a unary operator. We simply create a matrix object with the dimensions reverse and loop through setting values using the transpose relation, $\mathbf{A}^T_{i,j} = \mathbf{A}_{j,i}$.

```
matrix operator ¬(const matrix& A)
{
        // Create a temporary matrix with reversed dimensions
    matrix B(A.columns,A.rows);

        // Set the entries of B to be the same as those in A
    for (int i=0; i < A.columns; i++)
    {
        for (int j=0; j < A.rows; j++)
            {
                B.xx[i][j] = A.xx[j][i];
            }
    }
    return B;
}
```

3 The Vector Sub-Class

We can consider the set of all vectors to be a subset of the set of all matrices. For consistency with the rules of multiplication defined in the matrix class, we must define vectors to have column dimension 1. This convention of course does not conflict with any notions that we naturally have about the dimensions of vectors. All our previous algebra that was defined for matrices can be applied to vector objects. The vector class is instructed to inherit from the matrix class using the following code in the header file:

```
// 'vector' inherits from 'matrix' in a public way
class vector: public matrix
{
        ...
}
```

As in MATLAB, we would like to construct a vector object with the call, `vector v(n)`, to create a column vector of length n. To this we define the vector constructor to run the matrix constructor with n as the first argument and 1 as the second argument.

```
vector::vector(int no_of_elements)
// run the matrix constructor
: matrix(no_of_elements,1)
{ }
```

The default vector constructor simply runs the default matrix constructor. Any instance of the vector class can be passed as an argument into a matrix method and we hence retain all the functionality of the matrix class with vector objects. Mathematically, there is no real *need* to have defined this subclass, the reason it has been added is to make the interface more natural to the user. It also allows us to overload the brackets operator for vectors so that elements can be accessed using the call, `v(i)`, where i is some entry of the vector, `v`. The code for doing this is almost identical to the corresponding matrix method, so we shall not include it here, though it can, along with all the other code from this project, be found in the appendix.

A problem that arises when defining the vector sub-class in this way is that the methods written for the matrix class always return matrix objects. In other words, though a user may pass a vector object as an argument into a matrix method, if that method returns an object, then that object will be a matrix, not a vector. This did not seem to be a problem at first, however when writing the code for GMRES, which uses inner products and other vector operations, it became frustrating to program. The solution to this was to replicate (or overload) the necessary methods for the vector subclass. Thus, the binary, unary and assignment operators have been overloaded and for simple vector operations, the returned object is also a vector. Again, the code for these methods is very similar to the corresponding matrix versions, so they will not be included here.

There are only a few extra methods which are unique to the matrix class, `mat2vec()` and `norm()`. They have both been added to be used in the GMRES code. The first method, `mat2vec()`, takes a matrix object as an argument and returns a vector object. The reason for the existence of such a method is similar to the reasons described above for overloading the binary, unary and assignment operators for the vector class. It is needed if, for example, we wish the result of a matrix-vector multiplication to be a vector. The code accepts a matrix input (which will usually be an $m \times 1$ matrix), creates a vector object with the appropriate dimensions, and copies across the entries. A matrix with more than one column can be passed into the method. In this case, the returned object will just be the first columns of that matrix. The code for this method is:

```
vector mat2vec(matrix A)
{
    // create vector with same no of rows as A and one column
    vector v(rows(A));

    for (int i=1; i ≤ rows(A); i++)
    {   // copy only the values in the first column
        v(i)=A(i,1);
    }
    return v;
}
```

The last function that we will need in order to program the linear solvers is a way to compute the p-norm of a vector. In the header file, the function is declared in the following way:

```
friend double norm(vector v, int p=2);
```

The function takes two arguments as inputs; the vector we wish to find the p-norm of, and an integer, p. The statement of second argument, `int p=2`, allows the function to be called with just the first argument, in which case the default value, $p = 2$, is used. The following code is adapted from the similar method given in the C++ for scientific computing course on the MSc Mathematical Modelling and Scientific Computing, 2009.

```
double norm(vector v, int p)
{
    // define variables and initialise sum
    double temp, value, sum = 0.0;

    for (int i=1; i ≤ rows(v); i++)
    {   // floating point absolute value
        temp = fabs(v(i));
        sum += pow(temp, p);
    }

    value = pow(sum, 1.0/((double)(p)));

    return value;
}
```

4 Gaussian Elimination with Partial Pivoting

The ubiquitous direct method for solving the linear system, $\mathbf{Ax} = \mathbf{b}$, is *Gaussian Elimination.* This method is so important that it is usually first learned in a certain form by secondary school children when they try to solve (usually two or three variable) simultaneous equations.

For the general $n \times n$ case, the goal is to perform a series of operations on the linear system so that we end up with the equation, $\mathbf{Ux} = \mathbf{y}$, where $\mathbf{U}$ is an upper triangular matrix and $\mathbf{y}$ is a known vector. This system can then be solved easily using back-substitution to obtain $\mathbf{x}$. This process can be interpreted in terms of a series of matrix multiplications applied to the left of the original equation, $\mathbf{Ax} = \mathbf{b}$. The basic Gaussian Elimination algorithm begins with the first column of the matrix and attempts to set to zero all the sub-diagonal entries of that column by multiplying them by a suitable multiple of the number in the diagonal entry. Once this has been achieved for that column, the same is applied to the next column, and so on until the matrix has been reduced to an upper-triangular form. Formally stated, the basic GE algorithm is:

> **for** j = 1,2,...,n-1 **do**
>   **for** i = j+1,j+2,...,n **do**
>   calculate multiplier $l_{ij} = \frac{a_{ij}}{a_{jj}}$
>   row i $\leftarrow$ row i - l_{ij} $\times$ row j
>   **end for**
> **end for**

Each outer loop step (over j) can be represented as a matrix multiplication. To demonstrate this, consider the simple 4×4 linear system, $\mathbf{Ax} = \mathbf{b}$:

$$\begin{pmatrix} 8 & 3 & 4 & 6 \\ 2 & 4 & -1 & 2 \\ 1 & -4 & -6 & -9 \\ 4 & -3 & -4 & -5 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix}$$

We can perform the first step of the GE algorithm by forming an identity matrix and placing the row multiplier multiplied by -1 on the appropriate sub-diagonal entry. It is easy to see what the multipliers are in this case. Multiplying $\mathbf{A}$ on the left by this matrix produces:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ -\frac{1}{4} & 1 & 0 & 0 \\ -\frac{1}{8} & 0 & 1 & 0 \\ -\frac{1}{2} & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 8 & 3 & 4 & 6 \\ 2 & 4 & -1 & 2 \\ 1 & -4 & -6 & -9 \\ 4 & -3 & -4 & -5 \end{pmatrix} = \begin{pmatrix} 8 & 3 & 4 & 6 \\ 0 & 3.25 & -2 & 0.5 \\ 0 & -4.375 & -6.5 & -9.75 \\ 0 & -4.5 & -6 & -8 \end{pmatrix}$$

This concept can be implemented successively for each column of the matrix. In order to make the sub-diagonal entries zero, we multiply each matrix on the left by the new matrix for each column. This process will eventually produce an upper triangular matrix, and indeed we will actually obtain an LU factorisation of A. This step in the GE algorithm is implemented with the code from lines 13-21, in the implementation given below. For each column of the matrix, we first create a new identity matrix (line 13). The `for` loop in lines 15-20 then calculates and stores the multiplier in the appropriate sub-diagonal entry. Line 21 then computes the matrix multiplication for that step of the loop. The implementation for GE w. partial pivoting is given overleaf:

```
matrix operator /(const matrix& b, const matrix& A)
{
    int n = A.rows;
    // create empty matrices, P & L, U & Atemp
    matrix P, L, Utemp = eye(n), Atemp=A;

    for (int j=1; j < n ; j++)
    {   // create permutation matrix, P
        P = permute_r(n,find_pivot(Atemp,j),j);
        // update U & Atemp
        Utemp = P*Utemp;
        Atemp = Utemp*A;
        L = eye(n);

        for (int i=j+1; i <= n ; i++)
        {   // check for division by zero
            assert(fabs(Atemp(j,j))>1.0e-015);
            // compute multiplier and store in sub-diagonal entry of L
            L(i,j)= -Atemp(i,j)/Atemp(j,j);
        }
        Utemp = L*Utemp;
        Atemp = Utemp*A;
    }

    matrix U = Utemp*A;
    matrix y = Utemp*b;
    matrix x(n,1);                      // Create result vector

    // Solve Ux=y by back substitution:
    // Compute last entry of vector x (first step in back subs)
    x(n,1)=y(n,1)/U(n,n);
    double temp = 0;

    for (int i=n-1; i>=1; i--)
    {
        temp = y(i,1);
        for (int j=n; j>i; j--)
        {
            temp = temp - U(i,j)*x(j,1);
        }
        x(i,1)=temp/U(i,i);
    }
    return x;
}
```

Working with exact arithmetic, the basic GE algorithm (assuming that we have 'enough' time and computer power available) will be able to successfully solve all nonsingular linear systems. However, we live in a world of floating point computations. It is very well known that basic GE is in general unstable when implemented on a computer. This is due to the existence floating point rounding errors which can often propagate through the algorithm.

The fix for this issue is to use 'partial pivoting'. This strategy involves permuting the rows of the matrix at each stage of the computation. We ensure that for each column that is currently being manipulated, the largest diagonal or sub-diagonal entry (in absolute terms) is moved to the diagonal entry. When the largest number is already on the subdiagonal, no change is made. As was the case before, this goal can be accomplished using matrix products, in particular, we can use permutation matrices. To illustrate, let's perform this step of the algorithm for the above 4×4 system.

Recall that we had already achieved our goal for the first column. Our matrix is now:

$$\begin{pmatrix} 8 & 3 & 4 & 6 \\ 0 & 3.25 & -2 & 0.5 \\ 0 & -4.375 & -6.5 & -9.75 \\ 0 & -4.5 & -6 & -8 \end{pmatrix}$$

Our attention is now focussed on the diagonal and sub-diagonal entries of the second column of this matrix. We see that the largest value in absolute terms (-4.5) is in the $(4,2)$ entry of the matrix. The goal is to move this number to the diagonal of the same column $(2,2)$. We accomplish this using a permutation matrix to interchange the second and fourth rows:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} 8 & 3 & 4 & 6 \\ 0 & 3.25 & -2 & 0.5 \\ 0 & -4.375 & -6.5 & -9.75 \\ 0 & -4.5 & -6 & -8 \end{pmatrix} = \begin{pmatrix} 8 & 3 & 4 & 6 \\ 0 & -4.5 & -6 & -8 \\ 0 & -4.375 & -6.5 & -9.75 \\ 0 & 3.25 & -2 & 0.5 \end{pmatrix}$$

This part of the algorithm in implemented in lines 8-11 of the code above. We utilise the two methods, `permute r(.,.,.)` and `find pivot(.,.)` described earlier to find the pivot in the given column and then create the corresponding permutation matrix. The actual matrix multiplication occurs in line 11.

Throughout this code, the temporary matrix objects `Utemp` and `Atemp`. `Utemp` is the matrix result of all the matrix operations that have been carried out up to a given point in the algorithm. `Atemp` is `Utemp` multiplied by A for a given point in the algorithm (see line 22). The reason for needing `Atemp` is to allow us to calculate the (step specific) multipliers for the elimination step (line 19).

Once this transformation to an upper-triangular matrix has taken place, the last task the algorithm accomplishes is to solve the triangular system using back substitution. This is found in lines 31-42 of the above code. We start at the bottom of the matrix and solve by working upwards, substituting in all our previously computed unknowns to work out the next unknown.

5 GMRES

The work required to implement Gaussian Elimination (even without partial pivoting) is around $\frac{2}{3}n^3$ flops[1]. For large systems, we may look to an iterative solver as an alternative as for certain classes of matrix (e.g. sparse), as they can often converge to an acceptable level of precision with far less work.

During this section, we shall implement the now famous Generalised Minimal Residual algorithm (GMRES). The paper describing this method was first published by Youcef Saad and Martin Schulz in 1986[2]. It has since been one of the most cited papers in all of applied mathematics.

GMRES is a Krylov subspace method which seeks to minimise the residual ($\mathbf{r} = \mathbf{b} - \mathbf{AX}$) when measured using the vector 2-norm. Given an initial guess vector, $\mathbf{x_0}$, and corresponding initial residual, $\mathbf{r_0}$, the k^{th} Krylov subspace, $K_k(\mathbf{A}, \mathbf{r_0})$ is defined as:

$$K_k(\mathbf{A}, \mathbf{r_0}) = \text{span}\{\mathbf{r_0}, \mathbf{Ar_0}, \mathbf{A^2r_0}, \ldots, \mathbf{A^{k-1}r_0}\}$$

For the k^{th} iterate, Krylov subspace methods attempt to find:

$$\mathbf{x}_k \in \mathbf{x}_0 + K_k(\mathbf{A}, \mathbf{r_0})$$

In general, $\{\mathbf{r_0}, \mathbf{Ar_0}, \mathbf{A^2r_0}, \ldots, \mathbf{A^{k-1}r_0}\}$ is particularly bad basis for the Krylov subspace, $K_k(\mathbf{A}, \mathbf{r_0})$, so usually some kind of orthogonalisation process is required. In the case of GMRES, the *Arnoldi* iteration is used (for MINRES, Lanczos). The Arnoldi iteration generates an orthonormal basis for the k^{th} Krylov subspace, using a Gramm-Schmidt style iteration. The Arnoldi iteration is outlined in the following pseudo-code:

Guess $\mathbf{x_0}$, $\mathbf{r_0} = \mathbf{b} - \mathbf{Ax_0}$, set $\mathbf{v_1} = \frac{\mathbf{r_0}}{||\mathbf{r_0}||_\mathbf{2}}$
for l = 1,2,... **do**
  $\mathbf{w} = \mathbf{Av_l}$
  for j = 1,2,...,l **do**
    $h_{j,l} = \mathbf{v_j^T w}$
    $\mathbf{w} = \mathbf{w} - h_{j,l}\mathbf{v_j}$
  end for
  $h_{l+1,l} = ||\mathbf{w}||_\mathbf{2}$
  $\mathbf{v_{l+1}} = \frac{\mathbf{w}}{h_{l+1,l}}$
end for

This process can also be represented in matrix form. Defining the $n \times k$ matrix $\mathbf{V_k} = [\mathbf{v_1}, \mathbf{v_2}, ..., \mathbf{v_k}]$, and the $(k+1) \times k$ upper Hessenberg matrix, $\mathbf{\tilde{H}}$, who's entries are created in the Arnoldi process as:

$$\mathbf{\tilde{H}_k} = \begin{pmatrix} h_{1,1} & h_{1,2} & h_{1,3} & \cdots & h_{1,k-1} & h_{1,k} \\ h_{2,1} & h_{2,2} & h_{2,3} & \cdots & h_{3,k-1} & h_{2,k} \\ 0 & h_{3,2} & h_{3,3} & \cdots & h_{3,k-1} & h_{3,k} \\ \vdots & \vdots & \ddots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & h_{k,k-1} & h_{k,k} \\ 0 & 0 & 0 & \cdots & 0 & h_{k+1,k} \end{pmatrix}$$

We can then write the Arnoldi process as:

$$\mathbf{AV_k} = \mathbf{V_{k+1}\tilde{H}_k}$$

Note that $\mathbf{V_{k+1}}$ is simply the matrix of orthogonal basis vectors for the $(k+1)^{th}$ Krylov subspace. Returning to our task, find $\mathbf{x}_k \in \mathbf{x}_0 + K_k(\mathbf{A}, \mathbf{r_0})$, we note that since the columns of $\mathbf{V_k}$ are, by construction, an orthonormal basis for $K_k(\mathbf{A}, \mathbf{r_0})$, this is equivalent to writing:

$$\mathbf{x_k} = \mathbf{x_0} + \mathbf{V_k y} \qquad \text{for some coefficient vector,} \qquad \mathbf{y} \in \mathbb{R}^k$$

$$\begin{aligned} \text{Now, since} \qquad \mathbf{x} - \mathbf{x_k} &= \mathbf{x} - \mathbf{x}_0 + \mathbf{V_k y} \\ \Rightarrow \qquad \mathbf{A}(\mathbf{x} - \mathbf{x_k}) &= \mathbf{A}(\mathbf{x} - \mathbf{x_0} - \mathbf{V_k y}) \\ \Rightarrow \qquad \mathbf{b} - \mathbf{A x_k} &= \mathbf{b} - \mathbf{A x_0} - \mathbf{A V_k y} \\ \Rightarrow \qquad \mathbf{r_k} &= \mathbf{r_0} - \mathbf{A V_k y} \end{aligned}$$

Now, let us suppose that we wish to minimise the k^{th} residual in the 2-norm. Our problem is thus: find $\mathbf{y} \in \mathbb{R}^k$ such that $||\mathbf{r_0} - \mathbf{A V_k y}||_2$ is minimised. From the first step in the Arnoldi process, we defined the first orthonormal basis vector, $\mathbf{v_1}$ using $\mathbf{r_0} = ||\mathbf{r_0}||_2 \mathbf{v_1}$. We now make the innocuous seeming observation that:

$$\mathbf{v_1} = \mathbf{V_{k+1} e_1} \qquad \text{where} \qquad \mathbf{e_1} = (1, 0, 0, ..., 0)^T \in \mathbb{R}^{k+1}$$

$$\text{therefore} \qquad \mathbf{r_0} = ||\mathbf{r_0}|| \, \mathbf{V_{k+1} e_1}$$

Recalling also the relation, $\mathbf{A V_k} = \mathbf{V_{k+1} \tilde{H}_k}$, we have:

$$\begin{aligned} ||\mathbf{r_k}||_2 = ||\mathbf{r_0} - \mathbf{A V_k y}||_2 &= \Big|\Big| ||\mathbf{r_0}|| \, \mathbf{V_{k+1} e_1} - \mathbf{V_{k+1} \tilde{H}_k y} \Big|\Big|_2 \\ &= \Big|\Big| \mathbf{V_{k+1}} \left(||\mathbf{r_0}||_\mathbf{2} \, \mathbf{e_1} - \mathbf{\tilde{H}_k y} \right) \Big|\Big|_2 \\ &= \Big|\Big| ||\mathbf{r_0}||_\mathbf{2} \, \mathbf{e_1} - \mathbf{\tilde{H}_k y} \Big|\Big|_\mathbf{2} \end{aligned}$$

The last line is true because the orthogonality of the columns of $\mathbf{V_{k+1}}$ means that mutliplying by that matrix does not affect the vector 2-norm. Let us remind ourselves of the dimensions of each of the quantities involved. $||\mathbf{r_0}||_\mathbf{2} \, \mathbf{e_1}$ is a vector of length $k+1$, $\mathbf{\tilde{H}_k}$ is a $(k+1) \times k$ matrix, and $\mathbf{y}$ is a vector of length k. This is therefore a (Hessenberg) linear least squares problem. The standard way of solving such problems is by using Givens rotation matrices, (since Givens rotation matrices are orthogonal, this in essence leads to a QR factorisation of the Hessenberg matrix).

We are now ready to state the full GMRES algorithm:

For arbitrary $\mathbf{x_0}$, $\qquad \mathbf{r_0} = \mathbf{b} - \mathbf{A x_0}$, $\qquad$ set $\mathbf{v_1} = \frac{\mathbf{r_0}}{||\mathbf{r_0}||_\mathbf{2}}$

for k = 1,2,... **do**
do step k of the Arnoldi process
(this gives us a new vector, $\mathbf{v_{k+1}}$ and a new last column of the matrix, $\mathbf{\tilde{H}_k}$
solve the Hessenberg linear least squares problem:

find $\mathbf{y} \in \mathbb{R}^n$ s.t. $\Big|\Big| ||\mathbf{r_0}||_\mathbf{2} \, \mathbf{e_1} - \mathbf{\tilde{H}_k y} \Big|\Big|_\mathbf{2}$ is minimised

then, $\mathbf{x_k} = \mathbf{x_0} + \mathbf{V_k y}$
end for

The C++ code to implement GMRES is given below. An explanation of the lines of code can be found after this.

```
vector GMRES(matrix A, matrix b, matrix x0, double tol)
{
    // determine initial residual, r0 in vector form
    vector r0 = mat2vec(b - A*x0);

    // need this in least square part later
    double normr0 = norm(r0);
    double residual=1.0;
    vector v= r0/normr0;

    // declare and intitialise variables
    int k=1;
    matrix J, Jtotal=eye(2),H(1,1), Htemp, HH;
    matrix bb(1,1), c, cc, tempMat, V, Vold, hNewCol;
    vector w, vj(rows(v));

    bb(1,1)=normr0;

    // initialise matrix V (matrix of orthogonal basis vectors)
    V=v;

    while (residual>tol)
    {
        H=resize(H,k+1,k);

        // Arnoldi steps (using Gram-Schmidt process)
        w = mat2vec(A*v);

            for (int j=1; j≤k; j++)
            {
                for (int i=1; i≤rows(V); i++)
                {
                    // set the vector vj to be jth column of V
                    vj(i)=V(i,j);
                }

                // the next two lines calculate the inner product
                tempMat = ¬vj*w;
                H(j,k)= tempMat(1,1);
                w = w - H(j,k)*vj;
            }

        H(k+1,k)=norm(w);

        v=w/H(k+1,k);

        // append an additional column to matrix V
        V=resize(V,rows(V),k+1);

        for (int i=1; i≤rows(V); i++)
        {
            // copy entries of v to new column of V
            V(i,k+1)=v(i);
        }

      ///////////////// Least squares step //////////////////////

        if (k==1)
        {   // First pass through, Htemp=H
            Htemp=H;
        }
```

```
62
63          else
64          {
65              // for subsequent passes, Htemp=Jtotal*H
66              Jtotal=resize(Jtotal,k+1,k+1);
67              Jtotal(k+1,k+1)=1;
68              Htemp=Jtotal*H;
69          }
70
71       // form next Givens rotation matrix
72       J = eye(k-1);
73       J = resize(J,k+1,k+1);
74
75       // set values to eliminate the h(k+1,k) entry
76       J(k,k)=Htemp(k,k)/pow(pow(Htemp(k,k),2)+pow(Htemp(k+1,k),2),0.5);
77       J(k,k+1)=Htemp(k+1,k)/pow(pow(Htemp(k,k),2)+pow(Htemp(k+1,k),2),0.5);
78       J(k+1,k)=-Htemp(k+1,k)/pow(pow(Htemp(k,k),2)+pow(Htemp(k+1,k),2),0.5);
79       J(k+1,k+1)=Htemp(k,k)/pow(pow(Htemp(k,k),2)+pow(Htemp(k+1,k),2),0.5);
80
81          // combine together with previous Givens rotations
82          Jtotal=J*Jtotal;
83          HH=Jtotal*H;
84
85          bb=resize(bb,k+1,1);
86          c=Jtotal*bb;
87
88          residual=fabs(c(k+1,1));
89
90          k++;
91      }
92
93      std::cout<< "GMRES iteration converged in " << k-1 << " steps\n\n";
94
95      // Extract upper triangular square matrix
96      HH=resize(HH,rows(HH)-1,columns(HH));
97      cc=resize(c,rows(HH),1);
98
99      // solve linear system
100     matrix yy = cc/HH;
101
102     vector y = mat2vec(yy);
103
104     // chop the newest column off of matrix V
105     V=resize(V,rows(V),columns(V)-1);
106
107     vector x= mat2vec(x0+V*y);
108     return x;
109 }
```

The code takes in three primary arguments: the matrix, $\mathbf{A}$, the RHS vector , $\mathbf{b}$, and the initial guess vector, $\mathbf{x_0}$. In addition, there is an optional argument, `tol`, or tolerance, which is set to a default value of $1e-6$, as in MATLAB.

We begin, in line 4 by computing the initial residual vector, $\mathbf{r0}$. Note that we employ `mat2vec()` to ensure that the result is a vector. Line 7 declares a double number `normr0` which is equal to $||\mathbf{r_0}||_2$. We also initialise the double number, `residual=1.0`, in order to enter the subsequent while loop. Line 9 declares the first orthogonal basis vector, $\mathbf{v_1}$. We will store the vectors, $\mathbf{v_i}$, in the matrix, $\mathbf{V}$, so that as we pass through the algorithm, we can just re-use the same vector, v, for the current $\mathbf{v_i}$.

Lines 12-15 declare and initialise all the other matrices and vectors that the algorithm uses. Line 17 sets the first entry of the vector, `bb` to be equal to `normr0`, representing the term $||\mathbf{r_0}||_2\mathbf{e_1}$. Note that we can increment the size of this vector as required during the course of the algorithm by using the command, `resize()`. Line 20 initialises the matrix, $\mathbf{V}$, storing $\mathbf{v_1}$ as its first column.

We now enter the main GMRES loop which utilises: `while (residual>tol)`. Line 24 uses the command, `resize()`, to set the upper Hessenberg matrix, $\mathbf{H_k}$, to the correct dimensions for the current pass (recall that `resize()` retains the previous entries if it increases the dimensions). Lines 27-45 simply implement the Arnoldi algorithm as described previously and hence computes the entries for the newest (k_{th}) column of $\mathbf{H_k}$. A small problem was that when calculating the inner product in lines 38 & 39, the computation is a matrix-matrix multiplication. Therefore, the returned object is a matrix, not a double. We have to manually extract the $(1,1)$ entry of the resultant matrix, `tempmat`, in order to get the required result. This could well have been placed into a function, but since there was only two lines of code involved, I decided that doing it this way would be easier. Line 45 gives the latest basis vector, $\mathbf{v_{k+1}}$ generated by the Arnoldi process. Lines 48-54 then add an additional column onto the matrix, $\mathbf{V}$ and stores the new vector in it by copying across the values.

We now proceed to the least squares step. Recall that this involves converting the Hessenberg matrix to an upper triangular matrix and then solving the truncated linear system (if the residual is small enough). In order to make the Hessenberg matrix upper triangular, we use Givens rotation matrices. During this part of the algorithm, we will use three additional matrices; `J`, the Givens rotation matrix for the *current* step; `Jtotal`, the running product of all the previous Givens rotation matrices; and `Htemp=Jtotal*H`, which is the effect that all the previous rotations have upon the current Hessenberg matrix. Proceeding in this way allows us to only require one Givens rotation matrix at each step. The computation of `Htemp` allows us to work out what the next Givens rotation matrix should be (`Htemp` is always 'almost' upper-triangular - the only nonzero sub-diagonal entry is $\mathbf{Htemp}_{k+1,k}$. Lines 58-79 accomplish this task.

Once we have found the latest Givens rotation matrix, we compute the `Jtotal` in line 82 and then form the upper triangular matrix, `HH=Jtotal*H`, in line 83. This is effectively our QR factorisation of the matrix, $\mathbf{H_k}$. To elucidate: **Jtotal** is orthogonal (the product of k Givens rotation matrices) and **HH** is upper triangular. We have therefore factorised $\mathbf{H}$ in the following way:

$$\mathbf{H} = (\mathbf{Jtotal^T})(\mathbf{HH})$$

$$\text{i.e.} \qquad \mathbf{H} = \mathbf{QR} \qquad \text{where} \qquad \mathbf{Q} = \mathbf{Jtotal^T} \qquad \text{and} \qquad \mathbf{R} = \mathbf{HH}$$

Returning to the formal statement of the least squares problem, we have, for step k:

$$\begin{aligned}\left|\left|\,||\mathbf{r_0}||_2\,\mathbf{e_1} - \tilde{\mathbf{H}}_\mathbf{k}\mathbf{y}\right|\right|_2 &= ||\,||\mathbf{r_0}||_2\,\mathbf{e_1} - \mathbf{QRy}||_2 \\ &= \left|\left|\mathbf{QQ^T}\,||\mathbf{r_0}||_2\,\mathbf{e_1} - \mathbf{QRy}\right|\right|_2 \\ &= \left|\left|\mathbf{Q}\left(\mathbf{Ry} - \mathbf{Q^T}\,||\mathbf{r_0}||_2\,\mathbf{e_1}\right)\right|\right|_2 \\ &= \left|\left|\mathbf{Ry} - \mathbf{Q^T}\,||\mathbf{r_0}||_2\,\mathbf{e_1}\right|\right|_2\end{aligned}$$

Framing this last statement in terms of the names used in the C++ code, we have:

$$\left|\left|\mathbf{Ry} - \mathbf{Q^T}\,||\mathbf{r_0}||_2\,\mathbf{e_1}\right|\right|_2 = ||(\mathbf{HH}) * (\mathbf{y}) - (\mathbf{Jtotal}) * (\mathbf{bb})||_2$$

Line 86 in the code makes the definition `c=Jtotal*bb`. Since we have a least squares problem with, effectively a $k \times k$ matrix, to find a k dimensional solution, $\mathbf{y}$ with a RHS vector, $\mathbf{c}$ of dimension $k+1$, we know that the k^{th} residual will be the $k+1^{th}$ entry of the vector, $\mathbf{c}$. This is what is computed in line 88. Note that we do not at this stage solve the linear system, this would be un-necessary work. Instead we wait to exit the `while` loop (i.e. we wait until the convergence criterion has been reached), and then solve the upper triangular linear system to compute the solution.

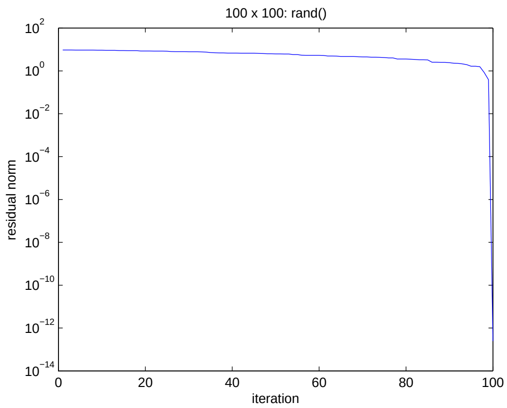
The last part of the code simply solves the linear system and returns the solution as the output of the function. Line 96 and 97 truncate the matrix, `HH` and the vector, `c`. These objects are then passed to the backslash operator (Gaussian Elimination) in line 100, to find the solution. Lastly, in line 105 we remove the newest column from the matrix of orthonormal basis vectors, then in line 107, we project the solution, `y` and add it to the initial guess to form our solution vector, which is the returned object.

6 Testing the code

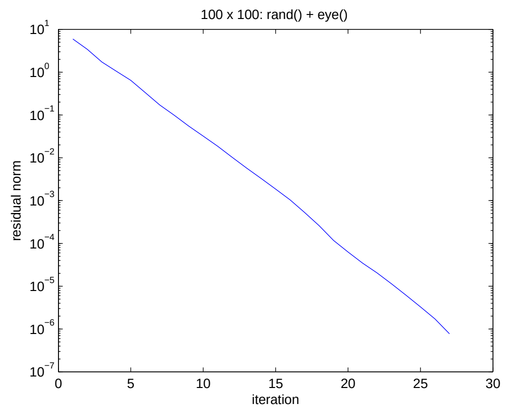
Whilst programming the matrix class and the linear solvers, I used test code to check that what was being produced was expected. The test code for the matrix and vector classes and for the Gaussian Elimination function is on the file named, 'use_matrix.cpp'. The tests I ran all produced the correct results and this file can be found in the appendix. The way I checked the Gaussian elimination solver was simply to multiply the result vector by the matrix and check that you ended up with the original RHS vector.

To test GMRES, I wrote three files called 'use_GMRES1.cpp', 'use_GMRES2.cpp' and 'use_GMRES3.cpp'. The first two are not really that interesting - they just compute the GMRES solution for small systems. 'use_GMRES3.cpp' builds a large matrix and fills it with uniformly distributed random numbers, using the `rand()` command. By modifying the GMRES routine ('GMRESout.cpp') to print the residual at each iteration to a data file, we can produce some convergence plots by loading the data into MATLAB.

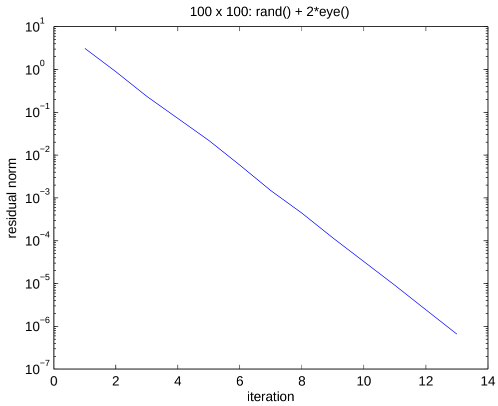
We will solve several 100×100 systems. Filling the entries with scaled random numbers makes for an ill-conditioned matrix and we thus expect convergence to take many iterations (100 or more). We can improve the conditioning of the matrix by adding multiples of the identity matrix, thus shifting the eigenvalues. This should have a dramatic effect upon convergence and if the GMRES method has been programmed correctly, we should be able to observe it on the convergence plots. First, the unshifted and badly conditioned matrix.



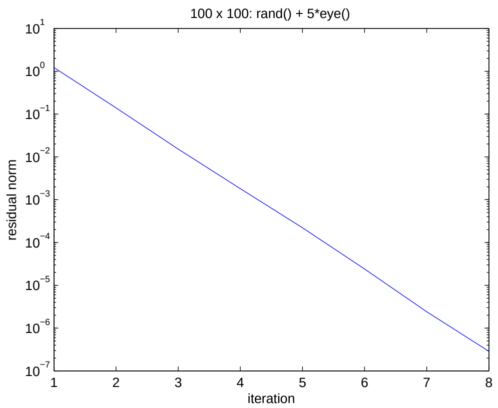
As expected, convergence (to our default tolerance of 1e-6) takes 100 iterations. Next, the same matrix with the identity matrix added on:



A dramatic improvement. Convergence now occurs in 27 iterations and is more uniform. Next, two times the identity added on:



Convergence here occurs in 13 iterations. Lastly, we try adding 5 times the identity on:



7 Extensions

There are many directions in which we could extend this work. Starting with GMRES, more features could be added, such as an optional maximum iteration parameter. It would also be a reasonable next step to code GMRES with restarts (we have programmed full GMRES here). Another obvious extension would be to program an effective error class to handle exceptions.
Aside form these natural extensions, there are a vast number of other linear solvers which could be programmed (CG, CGS, BICGStab etc). Each of these solvers often has a particular niche where they can perform better than the others, so combining all these methods into one bundle might be a reasonable challenge for a C++ programmer. Thinking even further ahead, it is certainly not impossible to conceive of designing a new GUI which would run these codes in a more user-friendly way.

8 Appendix - the codes in full

8.1 matrix.h

```
#ifndef MATRIXDEF
#define MATRIXDEF

//////////////////////////////////////////////////////////////////////////////
/////////////    This header file defines the 'matrix' class    /////////////
/////////////    All matrices created have 'double' entries     /////////////
//////////////////////////////////////////////////////////////////////////////

#include <math.h>
#include <string>
#include <iostream>
#include <cassert>

using std::ostream;


class matrix
{

/////////////////// Variables of the matrix class ////////////////////////
//private:
protected:
  // Define dimensions of matrix
    int rows, columns;
    // Pointer to first entry of the vector of pointers
    // (each of these pointers points to the first entry of each row)
    double **xx;


/////////////////// Constructors /////////////////////////////////////////

public:
  // Overwritten default constructor
    matrix();

    // Creates matrix of given dimension
    matrix(int no_of_rows,int no_of_columns);

  // Overwritten copy constructor
    matrix(const matrix& A);

////////////////// Destructor /////////////////////////////////////////////

    ¬matrix();

////////////////// Binary Operators ///////////////////////////////////////

    friend matrix operator +(const matrix& A, const matrix& B);
    friend matrix operator -(const matrix& A, const matrix& B);

    friend matrix operator *(const double& p, const matrix& A);
    friend matrix operator *(const matrix& A, const double& p);
    friend matrix operator *(const matrix& A, const matrix& B);

    friend matrix operator /(const matrix& A, const double& p);
    friend matrix operator /(const matrix& b, const matrix& A);
```

```
////////////////////// Unary operators ////////////////////////////////////
    friend matrix operator +(const matrix& A);
    friend matrix operator -(const matrix& A);

    // Overload ¬ to mean transpose
    friend matrix operator ¬(const matrix& A);

/////////////////////// Other operators ///////////////////////////////////

  // Overloads the assignment operator, '='
    matrix& operator =(const matrix &v);

  // Overloads (), so A(i,j) returns the i,j entry a la MATLAB
    double &operator() (int i, int j);

    // Returns the row dimension of the matrix
    friend int rows(matrix A);
    // Returns the column dimension of the matrix
    friend int columns(matrix A);

////////////////////// Functions that are friends ////////////////////////

    // Overloads the '<<' operator to allow easy printing of matrices
    friend ostream& operator<<(ostream& output, const matrix& A);

    // Create nxn Identity matrix
    friend matrix eye(int size);

  // computes an nxn permutation matrix which swaps rows i and j
    friend matrix permute_r(int n, int i, int j);

    // Locates largest number below the diagonal, for matrix, A
    friend int find_pivot(matrix A, int column);

    friend matrix resize(matrix A, int m, int n);
};
#endif
```

8.2 matrix.cpp

```
#include "matrix.h"
#include "vector.h"
#include "error.h"

/////////////////// Constructors //////////////////////////////////////////

// Constructor that overrides compiler generated default constructor
matrix::matrix()
{
  // constructs an empty object with rows = columns = 0
    rows = 0;
    columns = 0;
    xx = NULL;
}

// Constructor for basic matrix with specified dimensions
matrix::matrix(int no_of_rows, int no_of_columns)
{
  // Matrix dimension fields
    rows = no_of_rows;
    columns = no_of_columns;

  // A is an array of m pointers, each pointing to the
  // first entry in the vector (of length, 'no_of_columns')
    xx = new double *[no_of_rows];

    // Allocate the memory for the entries of the matrix
    for (int i=0; i<no_of_rows; i++)
    {
      // Creates 'no_of_rows' rows of length, 'no_of_columns'
        xx[i] = new double[no_of_columns];
    }

    for (int i=0; i<no_of_rows; i++)
    {
        for (int j=0; j<no_of_columns; j++)
        {
          // Initialise the entries of the matrix to zero
            xx[i][j]=0.0;
        }
    }
}

// Copy constructor - creates matrix with the same entries as input, A
matrix::matrix(const matrix& A)
{
    rows = A.rows;                          // Matrix dimension fields
    columns = A.columns;

  // A is an array of m pointers, each pointing to the
  // first entry in the vector (of length, 'A.columns')
    xx = new double *[A.rows];

  // Allocate the memory for the entries of the matrix
    for (int i=0; i<A.rows; i++)
    {
      // Creates 'A.rows' rows of length, 'A.columns'
        xx[i] = new double[A.columns];
```

```
    }

    for (int i=0; i < A.rows; i++)
    {
        for (int j=0; j < A.columns; j++)
        {
          // Copy across the entries from matrix A
            xx[i][j]=A.xx[i][j];
        }
    }
}

// Destructor
matrix::¬matrix()
{
    if ( rows > 0 || columns > 0 )
    {
        for (int i=0; i < rows; i++)
        {
            delete[] xx[i];
        }
        delete[] xx;
    }
}

///////////////////// Binary Operators //////////////////////////////////

// Overload the + operator to evaluate: A + B, where A and B are matrices
matrix operator +(const matrix& A, const matrix& B)
{
    int m = rows(A), n = columns(A), p = rows(B), q = columns(B);

    if ( m != p || n !=q )
    {
        std::cout << "Error: Matrices of different dimensions";
        std::cout << "Returned first argument";
        return A;
    }
    else
    {
        matrix C(m,n);
        for (int i=0; i<m; i++)
        {
            for (int j=0; j<n; j++)
            {
                C.xx[i][j]=A.xx[i][j]+B.xx[i][j];
            }
        }
        return C;
    }
}



// Overload the − operator to evaluate: A − B, where A and B are matrices
matrix operator −(const matrix& A, const matrix& B)
{
    int m,n,p,q;
    m = rows(A);
    n = columns(A);
    p = rows(B);
    q = columns(B);
```

```

    if ( m != p || n !=q )
    {
        std::cout << "Error: Matrices of different dimensions";
        std::cout << "Returned first argument";
        return A;
    }
    else
    {
        matrix C(m,n);
        for (int i=0; i<m; i++)
        {
            for (int j=0; j<n; j++)
            {
                C.xx[i][j]=A.xx[i][j]-B.xx[i][j];
            }
        }
        return C;
    }
}


// Definition of multiplication between a scalar, p and a matrix, A
matrix operator *(const double& p, const matrix& A)
{
  // Create a matrix with the same dimensions as A
    matrix B(A.rows,A.columns);

    for (int i=0; i<A.rows; i++)
    {
        for (int j=0; j<A.columns; j++)
        {
            B.xx[i][j]= p * A.xx[i][j];     // Multiply each entry by p
        }
    }
    return B;
}


// Definition of multiplication between a matrix, A and a scalar, p
matrix operator *(const matrix& A, const double& p)
{
  // Create a matrix with the same dimensions as A
    matrix B(A.rows,A.columns);

    for (int i=0; i<A.rows; i++)
    {
        for (int j=0; j<A.columns; j++)
        {
            B.xx[i][j]= p * A.xx[i][j];     // Multiply each entry by p
        }
    }
    return B;
}


// Definition of division of a matrix, A by a scalar, p i.e. A/p
matrix operator /(const matrix& A, const double& p)
{
  // Create a matrix with the same dimensions as A
    matrix B(A.rows,A.columns);

```

```
    for (int i=0; i<A.rows; i++)
    {
        for (int j=0; j<A.columns; j++)
        {
            B.xx[i][j]= A.xx[i][j]/p;       // Divide each entry by p
        }
    }
    return B;
}


// Define multiplication for matrices:

matrix operator *(const matrix& A, const matrix& B)
{
  // Use assertion to check matrix dimensions are consistent
    assert(A.columns==B.rows);

  // Create a result matrix, C with the correct dimensions
    matrix C(A.rows,B.columns);

    double temp = 0;

        // rows (m)
        for (int i=0; i < A.rows ; i++)
        {
          // columns (q)
            for (int j=0; j < B.columns ; j++)
            {
              // dot product step (n sums)
                for (int k=0; k < A.columns ; k++)
                {
                    temp = temp + A.xx[i][k]*B.xx[k][j];
                }

                // Set the C matrix values
                C.xx[i][j]=temp;

                // reset temp
                temp = 0;
            }
        }

    return C;
}

////////////////////// Unary operators ////////////////////////////////////

matrix operator +(const matrix& A)      // Define the unary operator, '+'
{
  // Create a temporary matrix with the same dimensions as A
    matrix B(A.rows,A.columns);

  // Set the entires of B to be the same as those in A
    for (int i=0; i < A.rows; i++)
    {
        for (int j=0; j < A.columns; j++)
        {
            B.xx[i][j] = A.xx[i][j];
        }
    }
    return B;
```

```
}

matrix operator -(const matrix& A)      // Define the unary operator, '-'
{
  // Create a temporary matrix with the same dimensions as A
    matrix B(A.rows,A.columns);

  // Set the entires of B to be the same as those in A
    for (int i=0; i < A.rows; i++)
    {
        for (int j=0; j < A.columns; j++)
        {
            B.xx[i][j] = -A.xx[i][j];
        }
    }
    return B;
}

// Overload the ¬ operator to mean transpose
matrix operator ¬(const matrix& A)
{
  // Create a temporary matrix with reversed dimensions
    matrix B(A.columns,A.rows);

  // Set the entires of B to be the same as those in A
    for (int i=0; i < A.columns; i++)
    {
        for (int j=0; j < A.rows; j++)
        {
            B.xx[i][j] = A.xx[j][i];
        }
    }
    return B;
}

/////////////////////////// Other operators /////////////////////////////

// Definition of matrix operator '='
// Operator returns a matrix equal to the RHS
matrix& matrix::operator =(const matrix &A)
{
  // Destruct previous entries
    for (int i=0; i < rows; i++)
        {
            delete[] xx[i];
        }
    delete[] xx;

  // Assign new dimensions to be equal to that of the RHS
    rows = A.rows;
    columns = A.columns;

  // Allocate the memory as in the constructor
    xx = new double *[A.rows];

    for (int i=0; i < A.rows ; i++)
    {
        xx[i] = new double[A.columns];
    }

  // Copy the values across from the RHS
    for (int i=0; i < A.rows; i++)
```

```
    {
        for (int j=0; j < A.columns ; j++)
        {
          // Set entries to be the same as the RHS matrix
            xx[i][j]=A.xx[i][j];
        }
    }
    return *this;
}

// Allows reference to the entries of a matrix in the same way as MATLAB
// Can call or assign values.
double &matrix::operator() (int i, int j)
{
    if (i < 1 || j < 1)
    {
    std::cout << "Error: One of your indices may have been too small \n\n";
    }

    else if (i > rows || j > columns)
    {
    std::cout << "Error: One of your indices may have been too large \n\n";
    }
    return xx[i-1][j-1];
}


//////////////////////// Function prototypes /////////////////////////////

int rows(matrix A);
int columns(matrix A);

//////////////////////// Function definitons /////////////////////////////

// Returns the private field, 'rows'
int rows(matrix A)
{
    return A.rows;
}

// Returns the private field, 'columns'
int columns(matrix A)
{
    return A.columns;
}

///////////////////////// Functions that are friends ////////////////////

// Overloads the '<<' operator to allow easy printing of matrices
ostream& operator << (ostream& output, const matrix &A)
{
    for (int i=0; i<A.rows; i++)
    {
        for (int j=0; j < A.columns; j++)
        {
            output << " " << A.xx[i][j];
        }
        output << "\n";
    }
    output << "\n";
    return output;
}
```

```

matrix eye(int size)
{
  // Create a temporary matrix with the same dimensions as A
    matrix temp_eye(size,size);

  // Set the entries of B to be the same as those in A
    for (int i=0; i < size; i++)
    {
            temp_eye.xx[i][i] = 1;
    }
    return temp_eye;
}

// Function that returns an nxn permutation matrix which swaps rows i and j
matrix permute_r(int n, int i, int j)
{
  // Create nxn identity matrix
    matrix I=eye(n);

    // Zero the diagonal entries in the given rows
    I(i,i)=0;
    I(j,j)=0;

  // Set the appropriate values to be 1
    I(i,j)=1;
    I(j,i)=1;

    return I;
}

// Function that returns the row number of the largest
// sub-diagonal value of a given column

int find_pivot(matrix A, int column)
{
  // Initialise maxval to be diagonal entry in column, 'column'
    double maxval=fabs(A(column,column));

    // Initialise rowval to be column
    int rowval=column;

        for (int i=column+1; i <= A.rows; i++)
        {
            if ( fabs(A(i,column)) > maxval)
            {
              // Update maxval and rowval if bigger than previous maxval
                maxval = fabs(A(i,column));
                rowval = i;
            }
        }
    return rowval;
}


// Function that returns an mxn matrix with entries that
// are the same as matrix A, where possible

matrix resize(matrix A, int m, int n)
{
    int p,q;
    matrix Mout(m,n);
```

```

    // select lowest of each matrix dimension
      if (m≤A.rows)
      {
          p=m;
      }
      else
      {
          p=A.rows;
      }

      if (n≤A.columns)
      {
          q=n;
      }
      else
      {
          q=A.columns;
      }

    // copy across relevant values
      for (int i=1; i ≤ p; i++)
      {
          for (int j=1; j ≤ q; j++)
          {
              Mout(i,j) = A(i,j);
          }
      }

      return Mout;
  }
```

8.3 vector.h

```
#ifndef VECTORDEF
#define VECTORDEF

#include "matrix.h"

// 'vector' inherits from 'matrix' in a public way
class vector: public matrix
{

public:

/////////////////////// Constructors ////////////////////////////////////

  // default constructor
    vector();
    // Constructor that takes 1 argument (size of the vector)
    vector(int no_of_elements);

///////////////// Binary Operators ////////////////////////////////////

    friend vector operator +(const vector& A, const vector& B);
    friend vector operator -(const vector& A, const vector& B);

    friend vector operator *(const double& p, const vector& A);
    friend vector operator *(const vector& A, const double& p);

    friend vector operator /(const vector& A, const double& p);


///////////////////// Unary operators /////////////////////////////////

    friend vector operator +(const vector& A);
    friend vector operator -(const vector& A);


////////////////////// Other operators ////////////////////////////////

  // Overloads (), so x(i) returns the ith entry a la MATLAB
    double &operator() (int i);

    // Overloads the assignment operator, '=' for vector RHS
    vector& operator =(const vector &v);

///////////////////// Functions that are friends /////////////////////

    friend vector mat2vec(matrix A);

  // Default call is norm(v) and returns 2-norm
    friend double norm(vector v, int p=2);

    friend vector GMRES(matrix A, matrix b, matrix x0, double tol=1e-6);

    friend vector GMRESout(matrix A, matrix b, matrix x0, double tol=1e-6);

    //friend vector resize(vector v, int m);

};
#endif
```

8.4 vector.cpp

```
#include "vector.h"                  // include the header file


/////////////////////// Constructors /////////////////////////////////////

// in the class, 'vector' there is a constructor of the same name
vector::vector()

// runs the default matrix constructor
: matrix()
{

}

vector::vector(int no_of_elements)

: matrix(no_of_elements,1)
{

}

////////////////////// Other operators /////////////////////////////////

// Overloads (), so x(i) returns the ith entry a la MATLAB
double &vector::operator() (int i)
{                                            // Can call or assign values.
    if (i < 1)
    {
        std::cout << "Error: Your index may be too small \n\n";
    }

    else if (i > rows)
    {
        std::cout << "Error: Your index may be too large \n\n";
    }
    return xx[i-1][0];
}

// Operator returns a matrix equal to the RHS
vector& vector::operator =(const vector &v)
{
  // Destruct previous entries
    for (int i=0; i < rows; i++)
        {
            delete[] xx[i];
        }
    delete[] xx;

  // Assign new dimensions to be equal to that of the RHS
    rows = v.rows;
    columns = v.columns;

  // Allocate the memory as in the constructor
    xx = new double *[v.rows];

    for (int i=0; i < v.rows ; i++)
    {
        xx[i] = new double[v.columns];
```

```
    }

  // Copy the values across from the RHS
    for (int i=0; i < v.rows; i++)
    {
        for (int j=0; j < v.columns ; j++)
        {
          // Set entries to be the same as the RHS matrix
            xx[i][j]=v.xx[i][j];
        }
    }
    return *this;

}
////////////////// Binary Operators ///////////////////////////////////////

vector operator +(const vector& A, const vector& B)
{
    int m,n;
    m = rows(A);
    n = rows(B);

    if ( m != n )
    {
        std::cout << "Error: Matrices of different dimensions.";
        std::cout << " Returned first argument";
        return A;
    }
    else
    {
        vector v(m);
        for (int i=0; i<m; i++)
        {
            v(i+1) = A.xx[i][0]+B.xx[i][0];
        }
        return v;
    }
}

vector operator -(const vector& A, const vector& B)
{
    int m,n;
    m = rows(A);
    n = rows(B);

    if ( m != n )
    {
        std::cout << "Error: Matrices of different dimensions.";
        std::cout << " Returned first argument";
        return A;
    }
    else
    {
        vector v(m);
        for (int i=0; i<m; i++)
        {
            v(i+1) = A.xx[i][0]-B.xx[i][0];
        }
        return v;
    }
}

```

```
//
vector operator *(const double& p, const vector& A)
{
        int m = rows(A);
        vector v(m);
        for (int i=0; i<m; i++)
        {
            v(i+1) = p*A.xx[i][0];
        }
        return v;
}
//
vector operator *(const vector& A, const double& p)
{
        int m = rows(A);
        vector v(m);
        for (int i=0; i<m; i++)
        {
            v(i+1) = p*A.xx[i][0];
        }
        return v;
}
//
vector operator /(const vector& A, const double& p)
{
        int m = rows(A);
        vector v(m);
        for (int i=0; i<m; i++)
        {
            v(i+1) = A.xx[i][0]/p;
        }
        return v;
}
//

/////////////////////// Unary operators ////////////////////////////////

vector operator +(const vector& A)
{
        int m = rows(A);
        vector v(m);
        for (int i=0; i<m; i++)
        {
            v(i+1) = A.xx[i][0];
        }
        return v;
}
//
vector operator -(const vector& A)
{
        int m = rows(A);
        vector v(m);
        for (int i=0; i<m; i++)
        {
            v(i+1) = -A.xx[i][0];
        }
        return v;
}

///////////////////// Functions that are friends //////////////////////

// Function that returns the first column of a matrix, A as a vector
```

```

vector mat2vec(matrix A)
{
  // create vector with same no of rows as A and 1 column
    vector v(rows(A));

    for (int i=1; i ≤ rows(A); i++)
    {
        v(i)=A(i,1);                            // copy only first column
    }

    return v;
}

double norm(vector v, int p)
{
  // define variables and initialise sum
    double temp, value, sum = 0.0;

    for (int i=1; i ≤ rows(v); i++)
    {
        // floating point absolute value
        temp = fabs(v(i));
        sum += pow(temp, p);
    }

    value = pow(sum, 1.0/((double)(p)));

    return value;
}
```

8.5 backslash.cpp

```
#include "matrix.h"
#include "vector.h"

// Definition of division of a vector, b by a matrix, A i.e. y=b/A
matrix operator /(const matrix& b, const matrix& A)
{
    int n = A.rows;

  // Create empty matrices, P & L
    matrix P, L;

    // Create and intialise U & Atemp
    matrix Utemp = eye(n);
    matrix Atemp=A;

    //std::cout << U << "\n\n";

    for (int j=1; j < n ; j++)
    {

        //std::cout << "Need to permute row " << j << " with row ";
    //std::cout << find_pivot(Atemp,j) << "\n\n";

    // Create appropriate permutation matrix, P
        P = permute_r(n,find_pivot(Atemp,j),j);

        Utemp = P*Utemp;                                // Update U & Atemp

        Atemp = Utemp*A;

        //std::cout << "Permute rows \n\n" << Atemp;

        L = eye(n);

        for (int i=j+1; i <= n ; i++)
        {
          // Check for division by zero
            assert(fabs(Atemp(j,j))>1.0e-015);

            // Compute multiplier and store in sub-diagonal entry of L
            L(i,j)= -Atemp(i,j)/Atemp(j,j);
        }

        Utemp = L*Utemp;

        Atemp = Utemp*A;

        //std::cout << "Eliminate sub-diagonal entries \n\n" << Atemp;

    }

// Now loop through and set to zero any values which are almost zero

    for (int j=1; j < n ; j++)
    {
        for (int i=j+1; i <= n ; i++)
        {
            if (fabs(Atemp(i,j)) < 5.0e-016)
```

```
                {
                    Atemp(i,j)=0;
                }
            }
        }

        //std::cout << "The matrix U = Utemp*A is then: \n\n" << Atemp;

        // So, to solve Ax=b, we do: (Utemp*A)x=Utemp*b i.e.
        // Set U=Utemp*A=Atemp, compute y=Utemp*b and
        // solve Ux=y (upper triangular system -> back subs)

        matrix U = Utemp*A;               //Atemp; gives the same result
        matrix y = Utemp*b;

        //std::cout << "The RHS is then Utemp*b: \n\n" << y ;

        matrix x(n,1);                    // Create result vector

        // Solve Ux=y by back substitution:

      // Compute last entry of vector x (first step in back subs)
        x(n,1)=y(n,1)/U(n,n);

        double temp = 0;                  // Initialise temp

        for (int i=n-1; i≥1; i--)
        {
            temp = y(i,1);
            for (int j=n; j>i; j--)
            {
                temp = temp - U(i,j)*x(j,1);
            }
            x(i,1)=temp/U(i,i);
        }

        return x;
    }
```

8.6 GMRES.cpp

```
#include "matrix.h"
#include "vector.h"

// GMRES
vector GMRES(matrix A, matrix b, matrix x0, double tol)
{
    //double tol=1e-6;

  // determine initial residual, r0 in vector form
    vector r0 = mat2vec(b - A*x0);

    //std::cout << "initial residual vector, r0 = b-A*x0 : \n\n" << r0;

  // need this in least square part later
    double normr0 = norm(r0);

    // initialise to enter while loop
    double residual=1.0;

    // intialise vector v
    vector v= r0/normr0;

    //std::cout << "initial vector v = r0/||ro|| : \n\n" << v;

  // Arnoldi/GMRES step index
    int k=1;

    // Declare Givens rotation matrix, initialise Jtotal;
    matrix J, Jtotal;
    Jtotal=eye(2);

  // intialise H, declare tempMat, V, w
    matrix H(1,1), Htemp, HH, bb(1,1), c, cc;
    matrix tempMat, V, Vold, hNewCol;
    vector w, vj(rows(v));

    bb(1,1)=normr0;

  // initialise matrix V (matrix of orthogonal basis vectors)
    V=v;

    while (residual>tol)
    {
        //std::cout<< "-----------------------------------------------\n\n";

    // update Vold (used for checking Arnoldi later)
        Vold=V;

        H=resize(H,k+1,k);

    // Arnoldi steps (using Gram-Schmidt process)
        w = mat2vec(A*v);
        //std::cout<< "(k = " << k <<") : vector w=Av : \n\n" << w;

            for (int j=1; j<=k; j++)
            {
                for (int i=1; i<=rows(V); i++)
                {
```

```
       // set the vector vj to be jth column of V
        vj(i)=V(i,j);
      }

      tempMat = ¬vj*w;

      // these two lines calculate the inner product
      H(j,k)= tempMat(1,1);
      //std::cout<< "H("<<j<<","<<k<<")= "<<H(j,k)<<"\n\n";

      w = w − H(j,k)*vj;
      //std::cout<< "Gramm−Schmidt update of vector w: \n\n" << w;
    }

  H(k+1,k)=norm(w);
  //std::cout<< "H(" << k+1 << "," << k << ")= " << H(k+1,k) << "\n\n";

  v=w/H(k+1,k);
  //std::cout<< "(k = " << k <<") :new vector v: \n\n" << v;

// add one more column to matrix V
  V=resize(V,rows(V),k+1);

  for (int i=1; i≤rows(V); i++)
  {
   // copy entries of v to new column of V
    V(i,k+1)=v(i);
  }

  //std::cout<< "(k = " << k << ") :latest matrix, V: \n\n" << V;

  //std::cout << "(k = " << k <<") :latest matrix, H: \n\n" << H;

  //std::cout << "check:    AV[k] = V[k+1]H: \n\n" << A*Vold << V*H;

/////////////////////////////// Least squares step //////////////////////

  if (k==1)
  {
   // First pass through, Htemp=H
    Htemp=H;
  }
  else
  {
   // for subsequent passes, Htemp=Jtotal*H
    Jtotal=resize(Jtotal,k+1,k+1);
    Jtotal(k+1,k+1)=1;
    Htemp=Jtotal*H;
  }

// Form next Givens rotation matrix
  J = eye(k−1);
  J = resize(J,k+1,k+1);

J(k,k)=Htemp(k,k)/pow(pow(Htemp(k,k),2)+pow(Htemp(k+1,k),2),0.5);
J(k,k+1)=Htemp(k+1,k)/pow(pow(Htemp(k,k),2)+pow(Htemp(k+1,k),2),0.5);
J(k+1,k)=−Htemp(k+1,k)/pow(pow(Htemp(k,k),2)+pow(Htemp(k+1,k),2),0.5);
J(k+1,k+1)=Htemp(k,k)/pow(pow(Htemp(k,k),2)+pow(Htemp(k+1,k),2),0.5);

  //std::cout<< "J: \n\n" << J;

// combine together with previous Givens rotations
```

```
        Jtotal=J*Jtotal;

        //std::cout<< "Check orthogonality of Jtotal \n\n" << ¬Jtotal*Jtotal;

        HH=Jtotal*H;

        for (int i=1; i≤k+1; i++)
        {
            for (int j=1; j≤k; j++)
            {
              // set all 'small' values to zero
                if (fabs(HH(i,j))<1e-15)
                {
                    HH(i,j)=0;
                }
            }
        }

        //std::cout<< "Check Jtotal*H is upper triangular: \n\n" << HH;

        bb=resize(bb,k+1,1);

        //std::cout<< "bb: \n\n" << bb;

        c=Jtotal*bb;

        //std::cout<< "c=J*bb: \n\n" << c;

        residual=fabs(c(k+1,1));

        //std::cout<< k << "th residual: \n\n" << residual << "\n\n";

        k++;
    }

    std::cout<< "GMRES iteration converged in " << k-1 << " steps\n\n";

  // Extract upper triangular square matrix
    HH=resize(HH,rows(HH)-1,columns(HH));

    //std::cout<< "HH: \n\n" << HH;

    cc=resize(c,rows(HH),1);

    //std::cout<< "cc: \n\n" << cc;

    matrix yy = cc/HH;                                // solve linear system

    vector y = mat2vec(yy);

    //std::cout<< "y: \n\n" << y;

  // chop the newest column off of matrix V
    V=resize(V,rows(V),columns(V)-1);

    vector x= mat2vec(x0+V*y);

    return x;
}
```

8.7 use_matrix.cpp

```cpp
#include "matrix.h"

#include "vector.h"
#include <stdlib.h>

int use_matrix()
{
    int m = 4, n = 4;

    matrix A(m,n);

    std::cout << "Matrix A has " << rows(A) << " rows" <<"\n";
    std::cout << "Matrix A has " << columns(A) << " columns" <<"\n\n";

    std::cout << "The newly created matrix, A looks like: \n\n";

    std::cout << A ;

    std::cout << "Now, we will assign some values to the entries: \n\n";

    for (int i=1; i<=m; i++)
    {
        for (int j=1; j<=n; j++)
        {
            A(i,j)=i+j;
        }
    }

    std::cout << A ;

    std::cout << "The (1,1) entry of matrix A is: " << A(1,1) << "\n\n";

    std::cout << "matrix, B has 5 times the entries of A: \n\n";

    matrix B(m,n);

    for (int i=1; i<=m; i++)
    {
        for (int j=1; j<=n; j++)
        {
            B(i,j)=5*(i+j);
        }
    }

    std::cout << B ;

    std::cout << "Add together the two matrices (A+B): \n\n";

    std::cout << A+B;

    std::cout << "Subtract the matrices (A-B): \n\n";

    std::cout << A-B;

    std::cout << "Create matrix C = A + A + A  using the '=', \n\n";

    matrix C(m,n);
    C = A + A + A;
```

```

    std::cout << C;

    std::cout << "Create matrix D such that D = +C \n\n";

    matrix D(m,n);
    D = +C;
    std::cout << D;

    std::cout << "Create matrix E such that E = -C \n\n";

    matrix E(m,n);
    E = -C;
    std::cout << E;

    std::cout << "Try copying the above matrix: \n\n";

    matrix F(m,n);
    F = matrix(E);
    std::cout << F;

    std::cout << "G = A is automatically sized: \n\n";

    matrix G = A;

    std::cout << G;

    std::cout << "Create matrix H = 6*G: \n\n";

    matrix H = 6 * G;

    std::cout << H;

    std::cout << "Create matrix J = H*0.5: \n\n";

    matrix J = H * 0.5;

    std::cout << J;

    std::cout << "Overwrite matrix B = J/10: \n\n";

    B = J/10;

    std::cout << B;

    //std::cout << "Now consider multiplying two Matrices together \n";
    //std::cout << "Define L=A and M= \n\n";

    //std::cout << A*B;

    std::cout << "Create a 'vector', x: \n\n";

    matrix x(n,1);

    for (int i=1; i<=n; i++)
    {
        x(i,1)=i;
    }

    std::cout << x;

    std::cout << "Multiply the Matrix A by x (Ax)  \n\n";
```

```
    std::cout << A*x ;

    /////////////////////////// Vector Stuff //////////////////////////////

    std::cout << "Create an empty vector using default constructor \n\n";

    vector a;

    std::cout << "Vector a has " << rows(a) << " rows" <<"\n";
    std::cout << "Vector a has " << columns(a) << " columns" <<"\n\n";

    std::cout << "Create a vector of size " << n << "\n\n";

    vector b(n);

    std::cout << "Vector b has " << rows(b) << " rows" <<"\n";
    std::cout << "Vector b has " << columns(b) << " columns" <<"\n\n";

    std::cout << b;

    std::cout << "Put some values into the entries: \n\n";

    for (int i=1; i≤n; i++)
    {
        //b(i,1)=i;
        b(i)=i;
    }

    std::cout << b;

    std::cout << "Multiply matrix A by vector b \n\n";

    std::cout << A*b;

    std::cout << "Create a 7 x 7 identity matrix \n\n";

    B = eye(7);

    std::cout << B;

    matrix T(4,4); A=T;

    std::cout << "Perform GE w. PP on the following 4x4 matrix: \n\n";

    A(1,1)=2; A(1,2) = 1; A(1,3) = 1; A(1,4) = 0;
    A(2,1)=4; A(2,2) = 3; A(2,3) = 3; A(2,4) = 1;
    A(3,1)=8; A(3,2) = 7; A(3,3) = 9; A(3,4) = 5;
    A(4,1)=6; A(4,2) = 7; A(4,3) = 9; A(4,4) = 8;

    std::cout << "Matrix A is: \n\n" << A;

    std::cout << "Vector b is: \n\n" << b << "Solve x=b/A: \n\n";

    x=b/A;

    std::cout << "The solution to the problem, x is: \n\n" << x;

    std::cout << "And as a check, multiply A*x: \n\n";

    std::cout << A*x;
```

```
    std::cout << "Try another example: \n\n";

    matrix AA(3,3);

    AA(1,1)=3; AA(1,2) = 17; AA(1,3) = 10;
    AA(2,1)=2; AA(2,2) = 4; AA(2,3) = -2;
    AA(3,1)=6; AA(3,2) = 18; AA(3,3) = -12;

    vector c(3);
    c(1)=1; c(2)=2; c(3) =3;

    std::cout << "Matrix AA is: \n\n" << AA << "Vector c is \n\n" << c;

    matrix y=c/AA;

    std::cout << "The solution to the problem, y is: \n\n" << y;

    std::cout << "And as a check, multiply A*y: \n\n";

    std::cout << AA*y;

    /*x=matrix(2,2);
    x(1,1)=1;x(1,2)=2;x(2,1)=3;x(2,2)=4;
    x=2*x;*/

    std::cout<<A;

    A=resize(A,10,3);

    std::cout<<AA;

    std::cout<<¬AA;

    vector d(10);
    for (int i=1; i≤10; i++)
    {
        d(i)=i;
    }

    std::cout<<d;

    std::cout<<¬d*d;

  exit(0);
}
```

8.8 useGMRES.cpp

```
#include "matrix.h"
#include "vector.h"
//#include "error.h"
#include <stdlib.h>

int useGMRES()
{

  matrix A(5,5);
  //A(1,5)=1;
  //A(2,1)=1;
  //A(3,2)=1;
  //A(4,3)=1;
  //A(5,4)=1;

A(1,1)=0.8780;A(1,2)=0.8316;A(1,3)=0.2663;A(1,4)=0.9787;A(1,5)=0.0239;
A(2,1)=0.1159;A(2,2)=0.2926;A(2,3)=0.2626;A(2,4)=0.7914;A(2,5)=0.2085;
A(3,1)=0.9857;A(3,2)=0.5109;A(3,3)=0.5826;A(3,4)=0.2115;A(3,5)=0.2943;
A(4,1)=0.8573;A(4,2)=0.7512;A(4,3)=0.4431;A(4,4)=0.9486;A(4,5)=0.3660;
A(5,1)=0.4416;A(5,2)=0.3803;A(5,3)=0.4465;A(5,4)=0.0586;A(5,5)=0.8501;

  vector b(5);
  //b(1)=1;

  b(1)=1;b(2)=1;b(3)=1;b(4)=1;b(5)=1;

  vector x0(5);

  std::cout << "matrix A: \n\n" << A;
  std::cout << "vector b: \n\n" << b;
  std::cout << "initial guess vector, x0: \n\n" << x0;

  vector x = GMRES(A,b,x0);

  std::cout << "GMRES solution, x is: \n\n" << x;

  std::cout << "Check: Ax = \n\n" << A*x;

  std::cout << "Backslash solution b/A \n\n" << b/A;

  srand(7);

  std::cout << "Random number 1:   " << - 1000 + rand() 2000  << " \n\n";
  std::cout << "Random number 2:   " << - 1000 + rand() 2000  << " \n\n";
  std::cout << "Random number 3:   " << - 1000 + rand() 2000  << " \n\n";

  exit(0);
}
```

8.9 useGMRES2.cpp

```
#include "matrix.h"
#include "vector.h"
//#include "error.h"
#include <stdlib.h>

int useGMRES2()
{
  matrix A(4,4);
  //A(1,5)=1;
  //A(2,1)=1;
  //A(3,2)=1;
  //A(4,3)=1;
  //A(5,4)=1;

A(1,1)=0.8780;A(1,2)=0.8316;A(1,3)=0.2663;A(1,4)=0.9787;//A(1,5)=0.0239;
A(2,1)=0.1159;A(2,2)=0.2926;A(2,3)=0.2626;A(2,4)=0.7914;//A(2,5)=0.2085;
A(3,1)=0.9857;A(3,2)=0.5109;A(3,3)=0.5826;A(3,4)=0.2115;//A(3,5)=0.2943;
A(4,1)=0.8573;A(4,2)=0.7512;A(4,3)=0.4431;A(4,4)=0.9486;//A(4,5)=0.3660;
//A(5,1)=0.4416;A(5,2)=0.3803;A(5,3)=0.4465;A(5,4)=0.0586;A(5,5)=0.8501;

  vector b(4);
  //b(1)=1;

  b(1)=1;b(2)=1;b(3)=1;b(4)=1;//b(5)=1;

  vector x0(4);

  std::cout << "matrix A: \n\n" << A;
  std::cout << "vector b: \n\n" << b;
  std::cout << "initial guess vector, x0: \n\n" << x0;

  vector x = GMRES(A,b,x0);

  std::cout << "GMRES solution, x is: \n\n" << x;

  std::cout << "Check: Ax = \n\n" << A*x;

  std::cout << "Backslash solution b/A \n\n" << b/A;

  exit(0);
}
```

8.10 useGMRES3.cpp

```
#include "matrix.h"
#include "vector.h"
//#include "error.h"
#include <stdlib.h>

int useGMRES3()
{
  int n=100;

  matrix A(n,n);

  srand(1);

  for (int i=1; i<=n; i++)
  {
    for (int j=1; j<=n; j++)
    {
        A(i,j) = - 1000 + rand() % 2000;
    }
  }

  A=A/1000;

  A=A/pow(n,0.5);

  A=A+5*eye(n);

  vector b(n);

  for (int i=1; i<=n; i++)
  {
        b(i) = 1.0;
  }

  vector x0(n);

  //std::cout << "matrix A: \n\n" << A;
  //std::cout << "vector b: \n\n" << b;
  //std::cout << "initial guess vector, x0: \n\n" << x0;

  vector x = GMRESout(A,b,x0);

  std::cout << "GMRES solution, x is: \n\n" << x;

  std::cout << "Check: Ax = \n\n" << A*x;

  //std::cout << "Backslash solution b/A \n\n" << b/A;

  exit(0);
}
```

References

[1] Lloyd N. Trefethen & David Bau III
Numerical Linear Algebra
SIAM 1997

[2] Youcef Saad & Martin H. Schultz GMRES: A generalised minimal residual algorithm for solving nonsymmetric linear systems
SIAM Journal of Scientific Computing., Vol. 7, No. 3, July 1986

[3] Joe Pitt-Francis
Lecture Notes
C++ for Scientifc Computing Course 2009
Computing Laboratory, Oxford University