Mathematical Institute
University of Oxford

Implementing a Matrix Class to Solve Linear Systems and Find Eigenvalues

Contents

1 Introduction

We implement a class of matrices and subclass of vectors in C++, overloading operators where necessary to allow for robust functionality. Upon implementing a definite linear system solver in Gaussian elimination, and indefinite solver in GMRES, we show their correctness by solving Poisson's equation for different forcing functions using the finite element method. We then consider eigenvalue problems, implementing the famous QR algorithm. We test our eigenvalue solver on an ODE based eigenvalue problem, again using the finite element method.

2 Matrices in C++

We begin my introducing the class of matrices that form the basis of this report. Inside our header file, **Matrix.h** we have defined our **protected**[1] and **public** variables and methods, key definitions can be seen in the following code.

```
 7  class Matrix
 8  {
 9  protected:
10          double* matrixVal = nullptr;
11          int rows;
12          int columns;
13  public:
14          // Used to allocate space
15          void createMatrix(int rows, int columns);
16          // Constructors
17          Matrix();
18          Matrix(int r, int c);
19          Matrix(const Matrix& m);
20          Matrix(int size) :Matrix(size, size) {};
21          // Destructor
22          ~Matrix();
```

Lines **11-12** show the expected **protected** variables for a class of matrices. Perhaps less conventionally, on line **10** we have defined a pointer to the first entry of a single array of doubles, rather than to the first entry of an array of pointers which each point to an array of doubles. In this construction we store a $m \times n$ matrix in memory as a single array of doubles of length mn, as opposed to the latter method which

[1]We note that some of our variables are defined **protected** instead of **private** as they must be accessible by our inheriting vector sub-class

would store the same matrix in memory as m distinct arrays of doubles of size n, a seemingly pointless difference however it is justified by the speed increase, which we shall demonstrate after introducing the constructors and destructor. Line **15** is the method called by the constructors defined on lines **17-19** to dynamically allocate memory, and line **20** allows for a single size parameter to be specified which allocates memory for a square matrix[2]. Line **22** defines our destructor to free the memory. The following code shows the **createMatrix** method located in our **Matrix.cpp** source file.

```
 5  void Matrix::createMatrix(int rows, int columns)
 6  {
 7      // Delete the contents of any non-empty matrix
 8          if (matrixVal != nullptr)
 9          {
10                  delete[] matrixVal;
11          }
12          // Allocate memory for single array format
13          matrixVal = new double[rows * columns];
14          this->rows = rows;
15          this->columns = columns;
16          if (rows == 0 || columns == 0)
17          {
18              // Assign an empty matrix NULL value
19                  matrixVal = NULL;
20          }
21          else
22          {
23                  for (int i = 0; i < rows * columns; i++)
24                  {
25                      // Assign the matrix with zeros
26                          matrixVal[i] = 0;
27                  }
28          }
29  }
```

The key line that demonstrates our method of storing the matrix is line **13**, where we simply allocate an array of doubles of the required size. The default constructor **Matrix()**[3] then simply calls **createMatrix(0,0)** and the basic overload of this,

[2]this simply calls the constructor that allocates memory for generic matrices with rows = columns = size, and so need not be defined in **Matrix.cpp**.

[3]We make use of this format of default constructor so we can define a matrix whose size we may not want to specify.

Matrix(r,c) calls **createMatrix(r,c)**. The copy constructor is similar, creating a matrix of the same size and simply assigning the value at each index.

2.1 Cache Friendly Memory Allocation and Locality

As stated already, we allocate memory using a pointer to the first entry in a single array of doubles of length $m \times n$, instead of a pointer to an array of m pointers which each point to the first entry in an array of doubles of length n. This is a matter of data locality[4], i.e. the locality of data entries which are likely to be accessed together. In the case where the matrices are stored as pointers to arrays of pointers, the data is not contiguous in memory since there is no reason why the secondary pointers should point to entries in any structured manner. In contrast to this, the method we employ points to the first entry in single array which is by definition contiguous in memory - a more cache friendly approach. Consequently, in any iteration over the matrix, the CPU avoids cache misses. In other words the CPU avoids having to wait for data that is not currently cached due to the lack of structure in the underlying data. It also allows the CPU to prefetch data[1], which helps reduce the natural bottleneck of transferring data from memory to the CPU. We can indeed demonstrate this, and to do so we implement a timer function. The code for this function can be found in appendix C.3, and it is aptly called by executing **tic();** prior to the code and **toc();** afterwards. We record the time taken for both methods in allocating memory and assigning values. For a matrix of size 8000×8000, and averaged over 30 executions, we find that the average time taken by our method is 0.1862 seconds, compared to 0.3288 seconds in the opposing method when timed on the same machine - a significant speed increase. A small downside to this method is that for a **Matrix** A, we are required to index using **[i+A.rows*j]** where we would otherwise use **[i][j]**. We refer the reader to this fact when reading the code displayed in this report.

2.2 Operator Overloading

We have overloaded a series of operators to allow for robust functionality. All of the basic mathematical binary ($+$, $-$, $/$, $*$) and unary $-$ operators have been overloaded to allow for computations between types **double**, **Matrix** and **Vector**. The equals ($=$) operator has been overloaded to allow for the assigning of **Matrix** types (and also **Vector** types)to each other. Similarly, the parentheses (($\cdot$ **,** $\cdot$)) operator has

[4]also referred to as spatial locality or memory locality

been overloaded to allow for the indexing of a **Matrix** or **Vector** , which we allow to start from 1. Finally the << operator has been overloaded to allow for the streaming of these types. A full list of these, as well as all implemented function, can be seen in our header file - see appendix C.1.

3 Solving $Ax = b$

An ample application of our matrix class is to solve some linear system $Ax = b$ that appears in a mathematical problem. In this section we discuss the algorithms implemented to solve such a problem, we consider these algorithms as described by Trefethen and Bau in [7], by Layton and Sussman in [3] and for GMRES in particular, by Saad[5] in [5] .

3.1 Gaussian Elimination

We first consider the implementation of a direct method to solving linear systems in Gaussian elimination (with partial pivoting to ensure stability). The full code for this can be found in appendix C.4. The implementation can be broken down into three main steps. Given an $m \times m$ matrix A and a $m \times 1$ Vector b, we seek the solution x to $Ax = b$. The first step is to form the augmented matrix $\tilde{A} = [\,A\,|\,b\,]$, where we have simply appended the vector b as an additional column on A. Once this has been formed, we perform suitable row operations to transform the augmented matrix into row reduced form. Finally we employ back substitution to solve for the vector x. In practice, the augmented matrix is formed by appending the $m \times 1$ vector b to the right of the $m \times m$ matrix A. This is a convenient step especially for the programming implementation since, in essence, it allows us to map the row transformations that we apply to A directly onto b so we need not store these in a separate matrix. We complete this task by introducing two new functions to the **Matrix** class, **addCols(int c)** and **setColumn(int col, Vector& v)**, these will also prove useful in the implementation of GMRES. We simply call the former to add a single column to (a copy of) A and the latter to append the vector b onto this new column. Once the augmented matrix has been computed we begin our loop through its columns. The first objective in our loop is to find the largest (absolute) value below the diagonal of the current column, we then use our implemented **swapRows** function, which takes in the indices of two rows, to swap this with the subdiagonal

[5]Saad is one of the co-publishers for the paper that introduced GMRES

entry (and thus its row). We call this new entry the pivot, and it is used to eliminate all of the elements below it. If the value of the largest pivot is 0, then the current column need not be reduced so the column index is increased and we move onto the next column. As Trefethen and Bau note, this method of partial pivoting introduces backwards stability to Gaussian elimination and is thus superior to the standard, non-pivoting algorithm [7]. Once we have the correct (non zero) pivot, the remainder of the loop simply performs row operations to reduce the other elements in the column to zero, thus changing the entries in the corresponding rows in the remainder of the matrix. After looping through all columns we arrive at a row reduced form, for example the final form when A is 4×4 may be

$$\left[\begin{array}{cccc|c} a_{11} & a_{12} & a_{13} & a_{14} & b_1 \\ & a_{22} & a_{23} & a_{24} & b_2 \\ & & a_{33} & a_{34} & b_3 \\ & & & a_{44} & b_4 \end{array}\right],$$

where we note that all non labelled elements are definitely zero, however those labelled could take any value (a zero element on the diagonal would constitute a singular matrix, in which case we throw an exception). From this point we compute the entries of the solution vector x by performing back substitution [3]. Using the above matrix as an example, back substitution would start with assigning $x_4 = b_4/a_{44}$, and then the row above would yield the equation $a_{33}x_3 + a_{34}x_4 = b_3$ and one could plug in the calculated x_4 and solve for x_3. This process is repeated in a backwards loop (starting at the final row and finishing at the first), appending values found to the **Vector** x at each stage. Once the loop has completed we have found our solution so we return its value.

3.2 GMRES

In contrast to direct algorithms, which aim to factorise the matrix A into a product of more simple matrices, thereafter immediately arriving at the exact solution , iterative algorithms rely on gradually refining the solution by minimising some form of error [7]. Krylov subspace methods are a subcategory of iterative algorithms that rely on approximating the solution to $Ax = b$ in some space defined by polynomials of A times the vector b. In particular we have that the *nth* Krylov subspace is defined by

$$\mathcal{K}_n = \langle b, Ab, \cdots, A^{n-1}b \rangle, \tag{1}$$

where $\langle \cdot \rangle$ denotes the linear span. We consider the implementation of the Generalised Minimal Residual Method [5] (GMRES), which is such an iterative method.
At iteration k of GMRES, we seek an an approximate solution $x_k \in \mathcal{K}_k$ to $Ax = b$ that minimises the residual $r_k = ||r_0 - Ax_k||_2$, that is $x_k = \text{argmin}_{\hat{x} \in \mathcal{K}_k} ||r_0 - A\hat{x}||_2$, where $r_0 = b - Ax_0$, and x_0 is an initial guess to the solution, often take as a vector of zeros. We implement a for loop to search for such solutions. We begin at iterate $k = 1$, where k denotes the loop for finding the minimising vector in the kth Krylov subspace. The termination condition is $k <$ **maxits**, where **maxits** is a user specified maximum allowed iterations of the algorithm. In each loop we want to search for a optimal solution in the kth Krylov subspace and so need a basis to check in. To this end we employ Arnoldi iteration to construct an orthonormal basis of the subspace ([5],[7]). The convenience in this method is that it generates a single vector on the current iteration using the vector generated from the previous iteration, where the vectors are mutually orthonormal - repeating this process generates, at iteration k, an orthogonal matrix Q_{k+1} whose columns are the orthonormal basis vector of $\mathcal{K}_{k+1}$. Along with this, Arnoldi iteration also generates an upper Hessenberg matrix $\tilde{H}_k$ at iteration k via the relation

$$AQ_k = Q_{k+1}\tilde{H}_k. \tag{2}$$

This Hessenberg matrix will be crucial for the remainder of the algorithm. Prior to our loop we set Q to be a **Matrix**[6] of size $m \times 1$, where m is the side length of the square matrix A. The first orthonormal basis vector of Q is set to be $r_0/||r_0||$ and we proceed to enter the main loop of GMRES. Within this main loop, we begin our Arnoldi iteration loop for the current GMRES iterate k. This follows the standard Arnoldi iteration algorithm [7], as listed in Appendix A.1 .

```
        // Arnoldi starts here ————————————————————
        // q_previous stores the previous column of q
                q_previous = getColumn(Q, k − 1);
                Q.addCols(1);
                // At this point H is (k+1)x(k) and Q is mxk
        // Initially set next column as A*(previous column)
                q = A * q_previous;
        // Initialise next column in the Hessenberg matrix
                Vector h(k + 1);
        // Main Arnoldi Loop ——————————————————
```

[6]An important note here is that we set Q to be a **Matrix** of size $m \times 1$ instead of a **Vector** of length m, this is because throughout the loop we append our newest othonormal basis vector as an additional column, thereby changing the shape.

```
11                  for (int i = 0; i < k; i++)
12                  {
13              // This follows the standard Arnoldi algorithm
14                      qi = getColumn(Q, i);
15                      qt = transpose(q);
16                      double hi = (qt * qi).matrixVal[0];
17                      h.matrixVal[i] = hi;
18                      q = q - hi * qi;
19                  }
20          // ------------------------------------------------
21          // Assign the final value of new H column
22                  h.matrixVal[k] = norm(q);
23          // Final value of the newest column of Q
24                  q = q * (1.0 / norm(q));
25          // add the new basis vector to the matrix
26                  Q.setColumn(k + 1, q); // --------------> Q_{k+1}
27          // add new column to Hessenberg matrix
28                  H.setColumn(k, h);     // --------------> H_{k}
29          // Arnoldi ends here ------------------------------------
```

Line **7** simply initialises the newest orthonormal basis vector **q** as the product of A and the previous basis vector, which is taken from the orthonormal matrix Q where these are stored. The vector which shall be appended to this iterations' Hessenberg matrix is initialsied on line **9** . Lines **11-19** perform the Arnoldi iteration to compute the newest vectors to be appended to Q_{k+1} and $\tilde{H}_k$, and the code on lines **26** and **28** does exactly that. Upon creating these matrices, Q_{k+1} and $\tilde{H}_k$, we can re-wrtite the objective of GMRES in this loop. By setting x_k, the required approximate solution in this iteration, to a linear combination of the basis vectors[7] Q_k (i.e. the the first k columns of Q_{k+1}), $x_k = x_0 + Q_k y$, our goal is to find the minimiser y to the kth error, or residual; $||r_k||_2 = ||Ax_0 + AQ_k y - b||_2 = ||AQ_k y - r_0||_2 = ||Q_{k+1}\tilde{H}_k y - r_0||_2$. Where we have used the recurrence relation 2. Since we initialise the first column of Q, to be $r_0/||r_0||_2$, it follows that we can write $r_0 = ||r_0||_2 Q_{k+1} e_1$, where e_1 is the first standard basis vector of $\mathbb{R}^m$ (1 in the first entry and 0s in the remaining $m-1$ entries). Thus we want to find the minimiser y to $||r_k||_2 = ||Q_{k+1}\tilde{H}_k y - ||r_0||_2 Q_{k+1} e_1||_2 = ||\tilde{H}_k y - ||r_0||_2 e_1||_2$, by the unitary invariance [7] of the Euclidean norm and the orthogonality of Q_{k+1}. All of this derivation to say - we can solve for x_k by solving for coefficients of its basis

[7]Recall that the goal of GMRES is simply that - to find the optimal solution in the Krylov subspace, and these vectors span it so naturally we construct x as a linear combination of them and find the optimal coefficients, given by y. The addition of x_0 is included since we are calculating residuals relative to some initial guess x_0.

expansion, y, and this boils down to solving a least squares problem involving known quantities - the $(k+1) \times k$ Hessenberg matrix computed from Arnoldi iteration, $\tilde{H}_k$, and the term easily calculated from the initial residual, $||r_0||_2 e_1$. In practice we solve the least squares problem by computing the QR factorisation of $\tilde{H}_k$, a task whose cost is very cheap due to the structure of an upper Hessenberg matrix being almost upper triangular already. For brevity and sake of repetition, we omit the details of the QR factorisation process here, but note that it will be covered in section 4 where we discuss the QR algorithm for finding eigenvalues. The important part for GMRES is that we can essentially compute what are known as Givens rotation matrices[5] for $\tilde{H}_k$, that upon left multiplication, knock off the subdiagonal entries turn by turn, thus forming an upper triangular matrix. Of course $\tilde{H}_k$ is not square; it is $(k+1) \times k$, and so the resulting matrix would be a square, upper triangular matrix with an extra row of zeros which we can easily remove. We note that this process is done iteratively, and throughout the iterations we track the total product of all Givens rotation matrices as these will constitute the relevant transformation from Hessenberg to upper triangular on the termination of the loop. For brevity we omit the code for the decomposition of $\tilde{H}_k$, but note that it can be found between lines **100-193** in appendix C.5. The main GMRES loop terminates if the current error, calculated by a product of the previous errors and the current Givens rotation matrix, is lower than some user specified tolerence, **tol**, or if, as stated previously the iterations surpass the maximum allowed. Say we terminate at iteration $k = N$. Upon termination we simply take our elongated, $(N+1) \times N$ upper triangular matrix, **UT** in the code, and remove the final row, consisting of zeros, naming the new **Matrix U**, for upper triangular. We then take Q_N by removing the final column of the currently stored Q_{N+1}. We use Gaussian elimination to solve $Uy = \beta$, where we have tracked residuals in the **Vector** β. This problem is very cheap since U is upper triangular and thus the cost is only in the necessary back substitution. Upon finding y we easily calculate the approximation $x_N = x_0 + Q_N y$. The implementation of computing the final value is fairly trivial and can be found between lines **237-273** in appendix C.5.

3.2.1 Convergence of GMRES

We can check our implementation of GMRES by investigating the number of iterations required for the solution of $Ax = b$ for increasingly well conditioned matrices A. We apply GMRES to an $M \times M$ matrix of random entries uniformly distributed between 0 and 1, using our implemented **rand(int n)** function. This takes in an integer **n** and returns a square **Matrix** of side length **n** filled with such random entries. For the sake of this experiment, we temporarily modify our **gmres** function to return the **Vector** of errors, $||r_k||/||b||$ over a specified number of iterations. We then compute the errors for our random matrix A, and for $A + nI$, with $n = 10, 20, \cdots, 100$. We consider Theorem 35.2 by Trefethen and Bau, as stated in [7]. We first note that the kth residual can be written $||r_k|| = ||p_k(A)b||$, where $p_k \in P_k$. Here P_k is the set of polynomials of degree (at most) k. Then, the theorem states that at the kth iteration of GMRES, r_k satisfies

$$\frac{||r_k||}{||b||} \leq \kappa(V) \inf_{p_k \in P_k} \sup_{\lambda \in \Lambda(A)} |p_k(\lambda)|, \tag{3}$$

where $\Lambda(A)$ is the set of eigenvalues of A and V is the matrix of eigenvectors of A. Without going into the technical details of the results of the theorem, it roughly tells us that we expect the convergence to speed up when eigenvalues of the matrix are clustered away from the origin. Trefethen and Bau go on to give examples[7] of random matrices whose conditioning is improved by adding on multiples of the identity (i.e. clustering the eigenvalue spectrum of A away from the origin). We demonstrate similar results, shown in figure 1. We see a drastic decrease in iterations required for the convergence of our **gmres** function. The code used to create this figure can be found in appendix C.5.1.

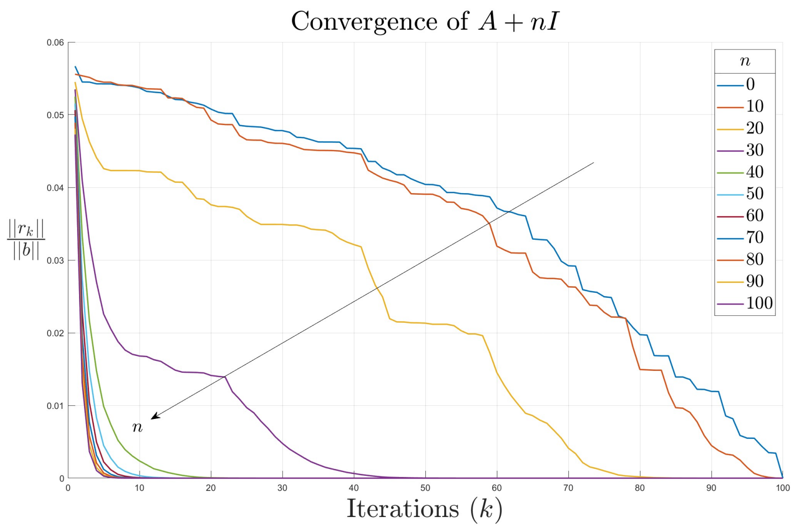

Figure 1: Convergence results for a random 100×100 matrix A, taking **gmres(A,b,guess,1.0e-10,1000)**, where b is a **Vector** of ones, **guess** is a **Vector** of zeros, and the final two entries are the tolerance and maximum iterations respectively. The arrow shows the direction of increasing n, for values $n = 0, 10, 20, \cdots, 100$

.

3.3 Application: Finite Element Solution to the Poisson Equation

We test our linear solvers on the Poisson equation with homogeneous Dirichlet boundary conditions and yet unspecified forcing function f. We follow the derivation[8] of the discretization by [6]. We want to find $u : \Omega \mapsto \mathbb{R}$ that satisfies

$$\begin{aligned} -\Delta u =& f \text{ in } \Omega, && (4) \\ u =& 0 \text{ on } \Gamma && (5) \end{aligned}$$

numerically. Multiplying by a test function $v \in V$ (where V is some function space which we shall choose to match the requirements of the variational formulation) and integrating gives

$$-\int_\Omega v\Delta u \, dx = \int_\Omega vf \, dx. \tag{6}$$

Applying the divergence theorem and noticing that $\nabla \cdot (v\nabla u) = v\Delta u + \nabla u \cdot \nabla v$ yields

$$-\int_\Omega v\Delta u \, dx = \int_\Omega \nabla u \cdot \nabla v \, dx - \int_\Gamma v\nabla u \cdot n ds, \tag{7}$$

where n is the unit outward normal on Ω to Γ and ds is a line element on Γ. Now, by choosing $V = H_0^1(\Omega)$, the Sobolev space that permits once weakly differentiable functions (so we can work with ∇u and ∇v) that satisfy the boundary condition (5), v vanishes on Γ and so the final term in (7) vanishes. We now have the variational (weak[9]) form of our problem; find $u \in V$ such that

$$\int_\Omega \nabla u \cdot \nabla v \, dx = \int_\Omega vf \, dx \tag{8}$$

for all $v \in V$. Though a weakened formulation, we are still trying to find a solution in a infinite dimensional function space V, thus we are seemingly still not in sight of a numerical solution, let alone a linear system to solve. The key is in the next step. We search for solutions in a finite dimensional subspcae V_h of V. Such a task is described by the Galerkin approximation[6] to our problem; find $u_h \in V_h$ such that

$$\int_{\Omega_h} \nabla u_h \cdot \nabla v_h \, dx = \int_{\Omega_h} v_h f \, dx \tag{9}$$

for all $v_h \in V_h$. We look for solutions u_h with general v_h in a basis of V_h. Recalling the definition of $V = H_0^1(\Omega)$, we require once (weakly) differentiable functions that vanish

[8]The derivation will also be relevent when solving the eigenvalue problem in the next section.

[9]We also refer to this as the weak formulation, since we have weakened the requirements on u as a solution to our problem.

on the boundary. We define a mesh of a domain to be a series of non intersecting cells whose union is the entire space. Considering V_h as such a mesh, we define a nodal basis $V_h = \text{span}\{\phi_1, \cdots, \phi_N\}$ of functions ϕ_i which vanish both on the boundary and on the intersection points between cells of the mesh, called the degrees of freedom. Writing u_h and v_h as a linear combination of basis functions, $u_h = \sum_{i=1}^N U_i\phi_i$ and $v_h = \sum_{j=1}^N V_j\phi_j$. After plugging these into (9) and rearranging, we arrive at

$$\sum_{j=1}^N V_j \sum_{i=1}^N U_j \int_{\Omega_h} \nabla\phi_i \nabla\phi_j dx = \sum_{j=1}^N V_j \int_{\Omega_h} \phi_j f dx. \tag{10}$$

We can write this as $AU = F$ where

$$A_{ij} = \int_{\Omega_h} \nabla\phi_i \nabla\phi_j dx, \quad \text{and} \quad F_j = \int_{\Omega_h} \phi_j f dx, \tag{11}$$

and U is the vector of coefficients that will be our discrete solution. Choosing the one-dimensional domain $\Omega = [0, 1]$ (and thus the boundary conditions $u(0) = u(1) = 0$), we can take the uniform mesh as a linear spacing of $[0, 1]$, with spacing h. With this formulation we can consider the domain as a mesh defined by degrees of freedom at interior points with spacing h, and we can consider the discretized solution space V_h as having dimension equal to the number of interior mesh points, $N = 1/h - 1$, where the numerator is simply the domain length, and we take one less to account for the extra exterior node included in the quotient. We omit the discussion regarding the rigorous formulation of the basis functions and the master element mapping for brevity however, essentially, we evaluate our integrals over the master element $[0, 1]$ instead of over each local finite element defined by degrees of freedom. In doing this we also calculate a local matrix for A and local vector for F defined by the mapping between the local element and the master element. This is done via the transformation $x \mapsto \xi = (x - x_L)/(x_R - x_L)$, where the local element defined by degrees of freedom x_L and x_R is mapped to the master element defined by $[0, 1]$. We note that in this case our basis functions become simply $\phi_1(\xi) = 1 - \xi$ and $\phi_2(\xi) = \xi$ for the one dimensional case. Applying the master element transformation, the matrix assembly requires that we compute the local matrix

$$\tilde{A} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} = \begin{bmatrix} \int_0^1 h^{-1}\phi_1'\phi_1' d\xi & \int_0^1 h^{-1}\phi_1'\phi_2' d\xi \\ \int_0^1 h^{-1}\phi_1'\phi_2' d\xi & \int_0^1 h^{-1}\phi_2'\phi_2' d\xi \end{bmatrix} \tag{12}$$

for each cell and append these 2×2 matrices on the main diagonal of A. Then the global matrix A is assembled as

$$A = \begin{bmatrix} a_{11} & a_{12} & & & & & \\ a_{21} & \mathbf{a_D} & a_{12} & & & & \\ & a_{21} & \mathbf{a_D} & a_{12} & & & \\ & & a_{21} & \mathbf{a_D} & \ddots & & \\ & & & \ddots & \ddots & & \\ & & & & & \mathbf{a_D} & a_{12} \\ & & & & & a_{21} & a_{22} \end{bmatrix}, \tag{13}$$

where we have defined $\mathbf{a_D} = a_{22} + a_{11}$. We note that in contrast to A, the computation of F is dependent on a forcing function f, to which we also must apply the mapping. We have that $f(x) = f(x_L + \xi(x_R - x_L))$, thus we must define F in a more general sense. We have the local vector

$$\tilde{F}_i = \begin{bmatrix} F_i \\ F_{i+1} \end{bmatrix} = \begin{bmatrix} \int_0^1 h\phi_1 f(x_i + \xi(x_{i+1} - x_i) d\xi \\ \int_0^1 h\phi_2 f(x_i + \xi(x_{i+1} - x_i) d\xi \end{bmatrix}, \tag{14}$$

where x_i and x_{i+1} are the values of the degrees of freedom i and j. We then have the global vector assembled as

$$F = \begin{bmatrix} F_1 \\ \mathbf{F_2 + F_3} \\ \mathbf{F_4 + F_5} \\ \vdots \\ \mathbf{F_N - 2 + F_N - 1} \\ F_N \end{bmatrix}. \tag{15}$$

We note that the entries in red shall be set to 0, and the entries in blue shall be set to 1 so as to satisfy the boundary conditions. From here we can implement the problem in to our **main()** method using the built up **Matrix** class. For convenience, we implement a **linspace** function, which takes as parameters two **double** types denoting the initial values and final values respectively, and an **int** type which denotes the number of points in our spacing. This function returns a **Vector** object which stores the linear spacing. The following code then generates the required variables for 100 mesh points.

```
int meshpts = 100;
Vector mesh = linspace(0.0,1.0,meshpts);
Vector dofs = linspace(1,meshpts,meshpts);
double h = 1.0 / ((double) meshpts - 1.0); // spacing
// Initialise A matrix (stiffness matrix) - square of size meshpts
Matrix A(meshpts);
// Initialise F vector (forcing vector) - size meshpts
Vector F(meshpts);
// Initialise DOFS
int dof1, dof2;
```

We next assemble our matrix A and vector F. By noticing the symmetry of $\tilde{A}$, we need only compute a_{11}, a_{12} and a_{22}, and these values are constant so do not need to be evaluated in a loop. To calculate integrals over $[0,1]$ we apply Simpson's rule[10], namely

$$\int_0^1 p(x)dx \approx \frac{1}{6}\left(p(0) + 4p\left(\frac{1}{2}\right) + p(1)\right). \tag{16}$$

Thus, by defining functions $f, \phi_1, \phi_2, \phi_1', \phi_2'$,

```
double f(double x){return -1;}  // Example forcing function f(x) = -1
double phi1(double x){return 1.0 - x;}
double phi2(double x){return x;}
double gradphi1(double x){return -1.0;}
double gradphi2(double x){return 1.0;}
```

we can compute the three distinct elements of $\tilde{A}$ by using 16, a task whose code we omit for brevity though refer the reader to appendix C.6. We then loop through the elements of the mesh, bar the last, and let our degrees of freedom be the current iterate and the subsequent iterate in the loop. These degrees of freedom denote the indices in the global matrix A to which we append our submatrix $\tilde{A}$ (and similarly for F). The matrix and vector assembly is then simply written in the loop as the following code.

```
        for (int element = 1; element < dofs(meshpts); element++)
        {
                // dofs give the index in A that we add to
                dof1 = element;
                dof2 = element + 1;
                // evaluate Simpsons rule points on the local cells
                x1 = masterToLocal(0.0, mesh(dof1), mesh(dof2));
                x2 = masterToLocal(0.5, mesh(dof1), mesh(dof2));
                x3 = masterToLocal(1.0, mesh(dof1), mesh(dof2));
```

[10]Of course we could easily calculate these integrals by hand and simply assign the values, however Simpson's rule will be necessary in the eigenvalue problem example and thus we introduce it here. It also turns out that in this example, the Simpson's rule approximation is exact

```
10                  // evaluate F elements at correct dofs
11                  // using Simpsons rule
12                  F1 = (double)1.0 / 6.0 * (h * f(x1) * phi1(0.0)) +
13                           +4.0 / 6.0 * (h * f(x2) * phi1(0.5)) +
14                           +1.0 / 6.0 * (h * f(x3) * phi1(1.0));
15                  F2 = (double)1.0 / 6.0 * (h * f(x1) * phi2(0.0)) +
16                           +4.0 / 6.0 * (h * f(x2) * phi2(0.5)) +
17                           +1.0 / 6.0 * (h * f(x3) * phi2(1.0));
18                  // Matrix and vector assembly
19                  A(dof1, dof1) += A11;
20                  A(dof1, dof2) += A12;
21                  A(dof2, dof1) += A12;
22                  A(dof2, dof2) += A22;
23                  F(dof1) += F1;
24                  F(dof2) += F2;
25          }
```

In lines **7-9** we have used our mapping between the master element and the local element. Upon completion of the assembly loop, we assign the boundary conditions by setting the values of A and F to 0 and 1 in the relvent locations (seen in red and blue previously). At this point in the code we have our final matrix A and vector F and thus it is finally time to put our linear solvers to the test. We compute solutions using both Gaussian elimination,

```
        Vector U_ge(meshpts);
        U_ge = A / F;
```

and GMRES,

```
        Vector U_gm(meshpts);
        Vector guess = 0 * U_gm;
        int maxits = meshpts;
        double tol = 1.0e-10;
        U_gm = gmres(A, F, guess, tol, maxits);
```

where we have manually set the tolerance to be 10^{-10} and max iterations to be the size of the matrix (GMRES should converge by this point). We then use **std::ofstream** to stream[11] our results to a **.dat** file which we can import into MATLAB to verify our results. For the forcing function $f(x) = -1$, it is not hard to see that we expect the solution $u(x) = x(x-1)/2$. Sure enough we see the correct results for both linear solvers.

[11]This is possible since we have overloaded the << operator to stream **Matrix** types.

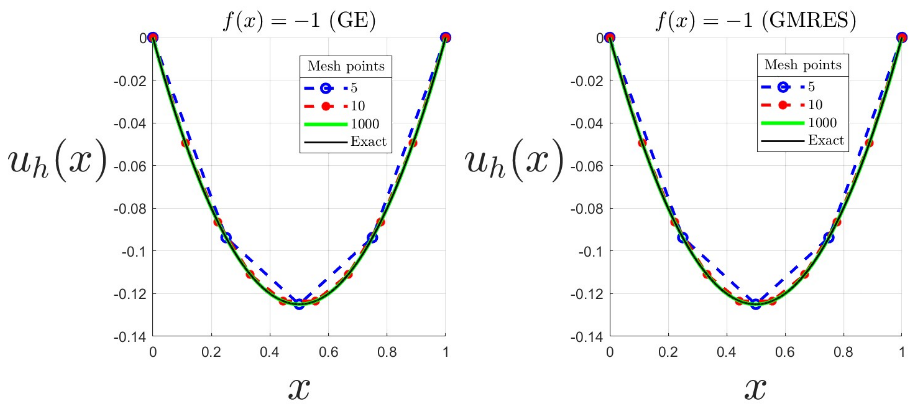

Figure 2: Finite element approximations to the Poisson equation with forcing function $f(x) = -1$. We see solutions for both Gaussian elimination and GMRES being used to solve the resulting linear system, at $5, 10$ and 1000 mesh points. We also see the exact solution

Now that we have verified our solvers for a simple forcing function, we can see how it holds out on a more complicated example. An easy way to do this is to pick a u, solve for f and use that as our forcing function. Choosing $u(x) = 2/3(x^5 - x)$ satisfies the boundary conditions and gives $f = -40/3x^3$. Changing our code to:

```
double f(double x){return -40.0 / 3.0 * pow(x,3);}
```

and rerunning for mesh sizes $5, 10$ and 1000 gives us back our solution.

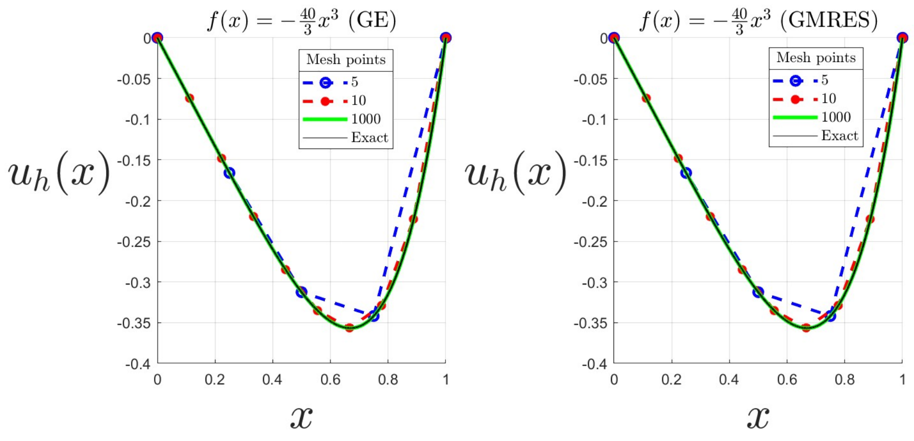

Figure 3: Finite element approximations to the Poisson equation with forcing function $f(x) = -40/3x^3$. We see solutions for both Gaussian elimination and GMRES being used to solve the resulting linear system, at $5, 10$ and 1000 mesh points. We also see the exact solution $u(x) = 2/3(x^5 - x)$

4 Solving $Ax = \lambda x$

Though seemingly similar to $Ax = b$, the solution to the eigenvalue problem $Ax = \lambda x$ requires a different approach. We implement the (pure) QR algorithm [7] which at first glance is strikingly simple - see appendix 2. Essentially, the algorithm repeats finding the QR decomposition of A, multiplying RQ, finding the QR decomposition of the result and so on. This converges to the Shur form of A [7]. Once this has been computed, the entries on the main diagonal are the eigenvalues. The only difficulty of the implementation of this lies within the step of actually computing the QR factorisation, for clarity we create a separate function, **qr**, to do this. An important note here is the return type. We would like to return both the orthogonal **Matrix** Q and the upper triangular **Matrix** R from the function that computes the decomposition, thus we use the **tuple** return type. After including ⟨**tuple**⟩ , the declaration in our header file[12] **Matrix.h** reads

```
        friend std::tuple<Matrix, Matrix> qr(const Matrix& A);
```

and one possible way we could call the function is **std::tie(Q, R) = qr(A)**. The actual return statement in the **qr** functin reads

```
        return std::make_tuple(Q, R);
```

Onto the implementation of the QR decomposition. We use the Givens rotation matrices to remove non zero entries from A to eventually make it upper triangular [7]. Since Givens rotation matrices are orthogonal, the product and inverse of them is also orthogonal, and thus the process of multiplying by them shall also yield an orthogonal matrix [3]. We initialise $Q = I_m$ and $R = A$ where A is the $m \times m$ matrix we are interested in. The Givens rotation matrix that we use to zero a subdiagonal entry (i, j) in A is given by $\mathrm{G}_{ij}, where$

$$\underbrace{\begin{bmatrix} I_{i-2} & & & \\ & c & s & \\ & -s & c & \\ & & & I_{m-i} \end{bmatrix}}_{G_{ij}\ (m\times m)} \times \underbrace{\begin{bmatrix} * & \dots & * & \dots & * \\ & & \vdots & & \\ \vdots & \dots & \mathbf{t} & \dots & \vdots \\ \vdots & \dots & \mathbf{b} & \dots & \vdots \\ & & \vdots & & \\ * & \dots & 0 & \dots & * \end{bmatrix}}_{A\ (m\times m)} = \begin{bmatrix} * & \dots & * & \dots & * \\ & & \vdots & & \\ \vdots & \color{red}{\dots} & \color{red}{*} & \color{red}{\dots} & \vdots \\ \vdots & \color{red}{\dots} & \color{red}{\mathbf{0}} & \color{red}{\dots} & \vdots \\ & & \vdots & & \\ * & \dots & 0 & \dots & * \end{bmatrix} \qquad (17)$$

[12] we also declare the function outside the scope of the **Matrix** class so as to allow for access in our **main()**.

and $m \geq i \geq j+1 \geq 2$ since we are dealing with subdiagonal entries. The lower right c of the Givens submatrix is entry (i,i) in the larger matrix[13]. In (17) we have noted the effected rows by the transformation in red. Upon calculating c and s using

$$c = \frac{t}{\sqrt{t^2+b^2}}, \quad s = \frac{b}{\sqrt{t^2+b^2}}, \tag{18}$$

we implement (17) with the following code in our loop through subdiagonal entries in all columns. We note that in each loop, prior to this calculation, we initialise **givens = eye(m)**.

```
1          // assign values to givens matrix
2          givens.matrixVal[i+m*i] = c;
3          givens.matrixVal[i+m*(i-1)] = -s;
4          givens.matrixVal[(i-1)+m*i] = s;
5          givens.matrixVal[(i-1)+m*(i-1)] = c;
6          // apply givens matrix to zero subdiagonal entries
7          R = givens * R;
8          // track the orthogonal vectors
9          Q = Q * transpose(givens);
```

This process looped over all subdiagonal elements will convert R (initialised as A) into an upper triangular matrix, and right multiplying Q (initialised as **eye(m)**) by the transpose of the Givens matrix will yield the corresponding orthogonal matrix. This is due to the relations [7]

$$G_{m-1}G_{m-2}\cdots G_1 A = R \tag{19}$$

$$\implies A = G_1^T \cdots G_{m-2}^T G_{m-1}^T R = QR, \tag{20}$$

which holds since each Givens matrix is orthogonal, $G^{-1} = G^T$, and thus the product of the inverses (transposes) of Givens matrices is also orthogonal. Over the course of the loop through all subdiagonal elements, line **7** in the code above generates equation (19) and line **9** generates equation (20).

4.1 QR algorithm

Now that we can compute the QR decomposition of a square matrix, we can implement the QR algorithm[7] stated in appendix 2. We do this by creating another

[13]It is important to note that this matrix seemingly only depends only on the row of the entry we want to zero, however we have implicitly defined the Givens rotation matrix with respect to the entry to be removed and the entry in the column above (b and t, for bottom and top) , thus the 2×2 structure of the submatrix is preserved. If we were considering zeroing with respect to a different entry (not directly adjacent) then our matrix would depend on j.

function, **eigMatrix**, to handle the generation of the Schur form of A, and a final, MATLAB styled function **eig** which extracts the diagonal elements (eigenvalues) from the Schur form and returns a **Vector** containing them. In practice this implementation is easily done in a loop since we have our **qr** function, and so we can repeatedly find the QR decomposition of the previously calculated product $R \times Q$. We terminate once our Schur form has been found - that is when the matrix is diagonal (for symmetric entries) or upper triangular (for general entries), which we can detect by checking the size of the off diagonals[7]. The **eig**[14] function then simply creates a **Vector** of size equal to that of the rows (or columns) of the matrix and assigns the values from the diagonal entries. We note that in this construction eigenvalues are returned in decreasing order, just as in MATLAB. We can now simply print the eigenvalues of a matrix A with the following code in our **main()**.

```
std::cout << "The eigenvalues of " << A << " are " << eig(A) << "\n";
```

4.2 Application: Finite Element Solution to an Eigenvalue Problem

We note that the purpose of the QR algorithm is to find eigenvalues, and though some eigenvectors may be generated, we limit our discussions to finding the former. An interesting extension would be to implement a more efficient way to calculate eigenvectors. To verify our eigenvalue calculator, we consider finding the eigenvalues λ_n that satisfy the equation[15][2]

$$-u''(x) + c(x)u(x) = \lambda u(x) \tag{21}$$

$$u(a) = u(b) = 0. \tag{22}$$

We use a similar finite element discretisation as when considerring the Poisson equation, as such we omit the derivation of the generalised eigenvalue problem here, but refer the reader instead to appendix B. We end up with the task to find eigenvalues that satisfy

$$AU = \lambda MU. \tag{23}$$

Left multiplying both sides by M^{-1}, gives the eigenvalue problem

$$M^{-1}AU = \lambda U. \tag{24}$$

[14]**eig** calls **eigMatrix** which repeatedly calls **qr**.

[15]For the case $c = 0$, this is known as the one dimensional Laplace eigenvalue problem [2].

Thus, to find our eigenpairs, we need to assemble the matrices A and M, compute the inverse[16] M^{-1}, and then find eigenvalues of the product $M^{-1}A$. The boundary conditions are incorporated by setting the entries in the first and last rows of A to zero and the same for M except the first entry of the first row and the final entry of the final row. This has the equivalent effect to enforcing that the product $M^{-1}A$ when right multiplied by U, zeros the initial and final values, thereby satisfying the homogeneous Dirichlet boundary conditions. We consider the case $a = 0$ and $b = \pi$. Using the master element transformation as before, we obtain

$$\tilde{a}_{ij} = \int_0^1 \frac{1}{h}\phi_i'\phi_j' + h\phi_1\phi_2 c(x_I + \xi(x_{I+1} - x_I)\, d\xi \tag{25}$$

where $i = 1, 2$, $j = 1, 2$ and $h = \pi/(N-1)$. Here x_1 and x_2 denote the x values of the degrees of freedom I and $I+1$, where the local matrix

$$\tilde{A} = \begin{bmatrix} \tilde{a}_{11} & \tilde{a}_{12} \\ \tilde{a}_{21} & \tilde{a}_{22} \end{bmatrix}$$

is added onto the global matrix at indices $(I, I), (I, I+1), (I+1, I)$ and $(I+1, I+1)$. Clearly I takes values between 1 and $N-1$, where N is the number of degrees of freedom. The matrix M is defined only in terms of basis functions and therefore all necessary values can be constructed prior to a loop, we have

$$\tilde{M} = \begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix} = \begin{bmatrix} \int_0^1 \phi_1\phi_1 h\, d\xi & \int_0^1 \phi_1\phi_2 h\, d\xi \\ \int_0^1 \phi_2\phi_1 h\, d\xi & \int_0^1 \phi_2\phi_2 h\, d\xi \end{bmatrix}. \tag{26}$$

We implement the solution to this problem in a very similar manner to the Poisson equation, and the full code can be found in appendix C.8. We note that for the coded solution, it was also convenient to implement an inverse function to calculate M^{-1}. We did this using our Gaussian elimination operator and calculating A/e_i where A is the $m \times m$ matrix to be inverted and e_i is the ith standard basis function of $\mathbb{R}^m$, iterating i over all rows. By appending the vector obtained from the calculation at each iteration to a new **Matrix** we obtain the inverse. Upon completing the matrix assembly, we simply execute the following code.

```
    Matrix B = inverse(M) * A;
    std::cout << "eigenvalues: " << eig(B) << "\n";
```

[16]A glaring issue in the simplification of the generalised eigenvalue problem to the standard eigenvalue problem is the possibility of a singular M. We fortunately do not run into this issue here since we choose appropriate functions $c(x)$, however we note that a possible extension would be to implement a solver of generalsied eigenvalue problems, such as the QZ algorithm, see [4].

For $c(x) = 0$, one can see that we require $u(x) = A\sin(\sqrt{\lambda}x) + B\cos(\sqrt{\lambda}x)$. Enforcing $a = 0$ and $b = \pi$ gives $u(x) = sin(\sqrt{\lambda}x)$ with the requirement that $\lambda = 1^2, 2^2, 3^2, \ldots$ ($\lambda_k = k^2$ for $k \in \mathbb{N}$) to satisfy the boundary conditions[17]. Assembling the matrices with these parameters, we find that the above code gives eigenvalues $\lambda_1 = 1.06161, \lambda_2 = 4.98914$ and $\lambda_3 = 12.7081$ with 5 mesh points. This begins to resemble the required result, however we can gain some confidence in our solution when using 50 mesh points; we obtain the following approximations to λ_i: 1.00034, 4.00548, 9.02778, 16.0882, 25.2225, 36.5188, 50.2117, 66.8098, $\ldots$, 880.766 where we clearly see the correct smaller eigenvalues, and an increasing error for larger eigenvalues.

5 Discussion

We have shown that our somewhat basic, from scratch, implementations of these well established numerical linear algebra methods and algorithms have been successful, in that they solve some non trivial problems, on a fairly sized scale. We have demonstrated the robustness of our implementation through our code snippets and have demonstrated a convergence property of our GMRES that gives confidence in our implementation. We have shown that our underlying memory allocation technique conforms to rigorous memory management conventions, in that they aim to minimise cache misses and reduce the bottlneck of the data transfer from memory to the CPU. By no means were the implemented algorithms the optimal choices for all cases, but rather a good choice for most cases. As such a natural extension would be to implement further, more specific solvers that can be chosen for different types of problems in a robust manner.

[17]Of course the trivial solution $\lambda = 0$ and thus $u = 0$ would also work.

References

[1] M. Ananth, S. Vishwas, and M.R. Anala. “Cache Friendly Strategies to Optimize Matrix Multiplication”. In: *2017 IEEE 7th International Advance Computing Conference (IACC)*. 2017, pp. 23–27. DOI: 10.1109/IACC.2017.0020.

[2] Daniele Boffi. “Finite element approximation of eigenvalue problems”. In: *Acta Numerica* 19 (2010), 1–120. DOI: 10.1017/S0962492910000012.

[3] William Layton and Myron Sussman. “Numerical linear algebra”. In: *University of Pittsburgh, Pittsburgh* (2014), pp. 28–39.

[4] Cleve B Moler and Gilbert W Stewart. “An algorithm for generalized matrix eigenvalue problems”. In: *SIAM Journal on Numerical Analysis* 10.2 (1973), pp. 241–256.

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

[6] “The Finite Element Method for the Poisson Equation”. In: *Numerical Methods for Elliptic and Parabolic Partial Differential Equations*. New York, NY: Springer New York, 2003, pp. 46–91. ISBN: 978-0-387-21762-8. DOI: 10.1007/0-387-21762-2_3. URL: https://doi.org/10.1007/0-387-21762-2_3.

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

A Algorithms

A.1 Arnoldi Iteration

Algorithm 1: Arnoldi Iteration [7]

```
**Data:** Matrix, $A$. Vector, $b$. Initial Guess to $Ax = b$, $x_0$.
**Result:** Orthonormal vectors $q_1, q_2, \cdots$
$r_0 \leftarrow b - Ax_0$
$q_1 \leftarrow r_0/||r_0||_2$
for $i = 1, 2, 3, \cdots$ do
    $v \leftarrow Aq_n$
    for $j = 1, \cdots, n$ do
        $h_{ji} \leftarrow q_j^T v$
        $v \leftarrow v - h_{ji} q_j$
    end
    $h_{i+1,i} \leftarrow ||v||_2$
    $q_{i+1} \leftarrow v/h_{i+1,i}$
end
```

A.2 QR algorithm

Algorithm 2: Pure QR Algorithm [7]

```
**Data:** Matrix, $A$.
**Result:** Schur form $\bar{A}$ of $A$
$A_0 \leftarrow A$
for $i = 1, 2, 3, \cdots$ do
    $Q_i R_i \leftarrow A_{i-1}$                Compute QR factorisation of A$_{i-1}$
    $A_i \leftarrow R_i Q_i$
end
```

B Derivation of eigenvalue problem

We consider the discretisation of the following problem.

$$-u''(x) + c(x)u(x) = \lambda u(x) \tag{27}$$

$$u(a) = u(b) = 0 \tag{28}$$

where $u_n \in [a, b]$, $c(x) \geq 0$ and $a < b$. We follow a very similar discretisation process as we did for the Poisson equation in section 3.3. Multiplying by a test function in a currently unknown function space $v \in V$, and integrating over our domain gives

$$\int_a^b -u''v + cuv\, dx = \lambda \int_a^b uv\, dx. \tag{29}$$

Integrating by parts and enforcing $V = H_0^1([a, b])$ then gives our weak form of the eigenvalue problem; find $u \in V = H_0^1([a, b])$ such that

$$\int_a^b u'v' + cuv\, dx = \lambda \int_a^b uv\, dx \tag{30}$$

for all $v \in V = H_0^1([a, b])$. Recall that this holds since the Sobolev space $H_0^1(\Omega)$ enforces that functions vanish on the boundary of Ω and the space permits functions whose first weak derivative exists, thus (30) is a well defined problem. We then, as before, search for a solution in the finite dimensional subspace $V_h \subset V$. Taking u_h and v_h as linear combinations of basis vectors $\phi_1, \phi_2, \cdots$. We have the problem to find the Galerkin approximation u_h to the eigenvalue problem by finding coefficients U_i of the basis expansion of u_h such that

$$\sum_{j=1}^{N} V_j \sum_{i=1}^{N} U_i \int_a^b \phi_i'\phi_j' + \phi_i\phi_j c\, dx = \lambda \sum_{j=1}^{N} V_j \sum_{i=1}^{N} U_i \int_a^b \phi_i\phi_j\, dx \tag{31}$$

for all coefficients V_j of the basis expansion of v_h. Since this must hold for all coefficients V_j, it is equivalent to finding the vector of coefficients U that satisfies the generalised eigenvalue problem

$$AU = \lambda MU. \tag{32}$$

where

$$A_{ij} = \int_a^b \phi_i'\phi_j' + \phi_i\phi_j c\, dx \tag{33}$$

and

$$M_{ij} = \int_a^b \phi_i\phi_j\, dx. \tag{34}$$

C C++ Code

C.1 Matrix.h

```cpp
#ifndef MATRIXDEF
#define MATRIXDEF

#include <cmath>
#include "Exception.h"
#include <tuple>
class Vector; //forward declaration
class Matrix
{
protected:
        double* matrixVal = nullptr;
        int rows;
        int columns;
public:

        // Used to allocate space
        void createMatrix(int rows, int columns);
        // Constructors
        Matrix(); // Default constructor
        Matrix(int r, int c);// Constructor given rows and columns
        Matrix(int size) :Matrix(size, size) {}; // Given size
        Matrix(const Matrix& m); // Copy constructor
        // Destructor
        ~Matrix();

        // Operators
        Matrix& operator=(const Matrix &m);      // equality
        double& operator()(int i, int j);        // indexing
        friend Matrix operator+(const Matrix& m1, const Matrix& m2);
        friend Matrix operator-(const Matrix& m1, const Matrix& m2);
        friend Matrix operator*(const Matrix& m1, const Matrix& m2);
        friend Matrix operator*(const double& a, const Matrix& m);
        friend Matrix operator*(const Matrix& m, const double& a);
        friend Matrix operator/(const Matrix& m, const double& a);
        friend Matrix operator-(const Matrix& m);
        friend Vector operator*(const Matrix& m, const Vector& v);
        friend Vector operator/(const Matrix& A, const Vector& b);
        // printing
        friend std::ostream& operator<<
        (std::ostream& output, const Matrix& m); // printing
        // Functions
        friend Matrix eye(int n);
        friend Vector getColumn(Matrix A, int c);
        friend Matrix getRow(Matrix A, int c);
        friend Matrix transpose(const Matrix& m);
        friend Vector gmres(const Matrix& A, const Vector& b,
            const Vector& x0, double tol, int maxits);
        friend std::tuple<Matrix, Matrix> qr(const Matrix& A);
        friend Matrix eigMatrix(const Matrix& A);
        friend Vector eig(const Matrix& A);
        friend Matrix inverse(const Matrix& A);
        friend Matrix hessenberg(const Matrix& A);
        friend Vector convertMatrixToVector(const Matrix& A);
        friend Matrix rand(int n);
        Matrix& swap_rows(int a, int b);
        Matrix& setColumn(int col, Vector& v);
        Matrix& addRows(int r);
        Matrix& addCols(int c);

};
// Accesible functions
Matrix rand(int n);
Matrix hessenberg(const Matrix& A);
Matrix inverse(const Matrix& A);
Matrix eigMatrix(const Matrix& A);
std::tuple<Matrix, Matrix> qr(const Matrix& A);
Vector gmres(const Matrix& A, const Vector& b,
    const Vector& x0, double tol = 1.0e-10, int maxits = 10000);
Matrix eye(int n);
Vector eig(const Matrix& A);
Vector getColumn(Matrix A, int c);
#endif
```

C.2 Vector.h

```
#include "Matrix.h"

class Vector : public Matrix
{
public:

        // Constructors
        Vector();
        Vector(int size);
        Vector(const Vector& v);

        // Operators
        Vector& operator=(const Vector& v);     // equality
        double& operator()(int i);                      // indexing
        friend Vector operator-(const Vector& m1, const Vector& m2);
        friend Vector operator+(const Vector& m1, const Vector& m2);
        friend Vector operator*(const Vector& m, const double& a);
        friend Vector operator*(const double& a, const Vector& m);

        friend double norm(Vector v, int p);
        friend Vector linspace(double a, double b, int n);
        friend double length(Vector v);

};
Vector linspace(double a, double b, int n);
double norm(Vector v, int p = 2);
double length(Vector v);
```

C.3 Memory Allocation Testing

C.3.1 Timer.h

```
#ifndef TIMERDEF
#define TIMERDEF

#include <chrono>

static std::chrono::
time_point<std::chrono::high_resolution_clock> startTime;

void tic();
void toc();

#endif // !TIMERDEF
```

C.3.2 Timer.cpp

```
include "Timer.h"
#include <stdio.h>
void tic()
{
        startTime = std::chrono::high_resolution_clock::now();
}

void toc()
{
        float dt = std::chrono::
        duration_cast<std::chrono::microseconds>
        (std::chrono::high_resolution_clock::now()-startTime).count()
        / 1000000.f;
        printf("Elapsed time is %f seconds.\n", dt);
}
```

C.3.3 Main

```
#include "Timer.h"
int main()
{

        // we test the allocation of memory
        // and assigning of entries for the two methods
        int rows = 8000;
        int columns = 8000;
        //  single array storage
        tic();
        double* A = new double[rows * columns];
        for (int i = 0; i < rows * columns; i++)
        {
                A[i] = 0;
        }
        delete[] A;
        toc();
        // multi array storage
        tic();
        double** B = new double* [rows];
        for (int i = 0; i < rows; i++)
        {
                B[i] = new double[columns];
        }
        for (int i = 0; i < rows; i++)
        {
                for (int j = 0; j < columns; j++)
                {
                        B[i][j] = 0;
                }
        }
        for (int i = 0; i < rows; i++)
        {
                delete[] B[i];
        }
        delete[] B;
        toc();


        return -1;
}
```

C.4 Gaussian Elimination

```
Vector operator/(const Matrix& A, const Vector& b)
{
        int m = A.rows;
        int n = m + 1; // Columns of Augmented matrix
        // We only accept square matrix input
        if (m != A.columns)
        {
                throw Exception(
                        "Bad input",
                        "A/b requires a sqaure matrix input");
        }
        // Form augmented matrix
        Vector bcopy(b);
        Matrix A_aug(A);
        A_aug.addCols(1);
        A_aug.setColumn(m + 1, bcopy);

        // solution vector
        Vector x(m);

        // Initialise variables
        double largest_pivot = 0;
        int largest_pivot_index = 0;
```

```cpp
    double q;       // variable to store a quotient in the loop

    int h = 0; // row index    (0 --> m - 1)
    int k = 0; // column index (0 --> n - 1 = m)


    // loop through the augmented matrix
    while ((h < m) && (k < n))
    {
        // clear previos pivot values
        largest_pivot = 0;
        largest_pivot_index = 0;
        // findng pivot in column k -
        // loop through all subdiagonal entries
        for (int i = h; i < m; i++)
        {
            // check if entry > largest pivot
            if(abs(A_aug.matrixVal[i+k*m])
                  >abs(largest_pivot))
            {
                // if entry > largest pivot,
                // set largest pivot = entry
                largest_pivot=A_aug.matrixVal[i+k*m];
                largest_pivot_index=i;

            }

        }
        // if largest pivot is 0 we can move
        // onto the next column
        if (largest_pivot == 0)
        {
            k += 1;
        }
        // otherwise we bring the pivot to the sub diagonal
        // and eliminate lower entries
        else
        {
            A_aug.swap_rows(h+1,largest_pivot_index+1);
            for (int i = h + 1; i < m; i++)
            {
                // find the ratio of each
                // value to the pivot
```

```
                                q = A_aug.matrixVal[i+k*m]/
                                    A_aug.matrixVal[h+k*m];
                                // elimnate entries below the pivot
                                A_aug.matrixVal[i+k*m]=0;
                                for (int j = k + 1; j < n; j++)
                                {
                                        // perform corresponding
                                        // operation on the rest
                                        // of the row (-=)
                                        A_aug.matrixVal[i+j*m] -=
                                        A_aug.matrixVal[h+j*m]*q;
                                }
                        }
                        // move to the next column, repeat the process
                        h += 1;
                        k += 1;
                }

        }

        // throw exception if matrix is singular
        for (int i=0;i<m;i++)
        {
            if(A_aug.matrixVal[i+i*m] == 0)
            {
                throw Exception("singular matrix",
                "A/b cannot solve when A is singular");
            }
        }

        // Now we find x using back substitution
        double sum = 0;

        // final value of x is simply the ratio of
        // the final two non zero elements in the row
        // reduced augmented matrix
        x.matrixVal[m - 1] = A_aug.matrixVal[m-1+(n-1)*m]/
                             A_aug.matrixVal[m-1+(n-2)*m];
        // we loop through the rest of the rows, each time
        // gaining another value in x
        for (int i = m - 2; i >= 0; i--)
        {
                sum = 0;
                for (int j = i + 1; j <= m - 1; j++)
                {
                        // sum the known values of x * coefficients.
                        // -= since these will be subtracted from
                        // the final column value
                        sum -= A_aug.matrixVal[i + j * m]
                             * x.matrixVal[j];
                }
                // next x value is given by adding the sum of
                // -(row elements * known x values) and the final
                // coefficient
                x.matrixVal[i] = (A_aug.matrixVal[i+m*(n-1)]+sum)
                               / (A_aug.matrixVal[i+i*m]);
        }
        // return the solution
        return x;
}
```

C.5 GMRES

```
Vector gmres(const Matrix& A,const Vector& b,const Vector& x0,
double tol,int maxits)
{

        int m = A.rows;
        if (m != A.columns)
        {
                throw Exception(
                "Bad input", "GMRES requires a sqaure matrix input");
        }
        // Residual given initial guess
        Vector r = b - A * x0;
        // We initialise Q as a matrix instead of a
        // vector since we shall add columns
        Matrix Q(m, 1);
        Vector q = r * (1.0 / norm(r)); // first entry of Q
        Q.setColumn(1, q); // q now added as first column of Q


        // Initialise the Hessenberg matrix obtained from Arnoldi
        Matrix H(2, 1); // starts as a 2x1

        /// Givens rotations values
        Matrix GivensTotal = eye(2);
        Matrix GivensCurrent;

        // variables used for Givens rotations
        Matrix Hcopy(H);
        double c;
        double s;
        double top;
        double bot;
        double div;
        Matrix UT(2);
        Vector errors(maxits);
        errors.matrixVal[0] = norm(r);
        double error;
        double bnorm = norm(b);

        // we will often require a temporary matrix
        Matrix temp;

        // variables for Arnoldi
        Matrix qt;
        Vector qi;
        Vector q_previous;


        // for gmres demonstration
        //Vector residuals(maxits);
        //int itsreq = 1;

        int k = 1; // errors iterations.
        // we define this outside the scope  of
        // the loop so we can access it afterwards
        for (k; k < maxits; k++)
        {

                if (k > 1)
                {
                        // on first iteration:
                        // Q is mx1 and H is 2x1,
                        // on further iterations we
                        // increase the size of H to
                        // retain Upper Hessenberg form
```

```
            H.addRows(1);
            H.addCols(1);
        }


        // Arnoldi starts here ------------------------------
        // q_previous stores the previous column of q
        q_previous = getColumn(Q, k - 1);
        Q.addCols(1);
        // At this point H is (k+1)x(k) and Q is mxk
        // Initially set next column as A*(previous column)
        q = A * q_previous;
        // Initialise next column in the Hessenberg matrix
        Vector h(k + 1);
        // Main Arnoldi Loop ------------------------------
        for (int i = 0; i < k; i++)
        {
            // This follows the standard Arnoldi algorithm
            qi = getColumn(Q, i);
            qt = transpose(q);
            double hi = (qt * qi).matrixVal[0];
            h.matrixVal[i] = hi;
            q = q - hi * qi;
        }
        // ------------------------------------------------
        // Assign the final value of new H column
        h.matrixVal[k] = norm(q);
        // Final value of the newest column of Q
        q = q * (1.0 / norm(q));
        // add the new basis vector to the matrix
        Q.setColumn(k + 1, q); // --------------> Q_{k+1}
        // add new column to Hessenberg matrix
        H.setColumn(k, h);     // -------------->  H_{k}

        // Arnoldi ends here ------------------------------



        // Uncommenting the below code gives a check
        // for orthonormality between basis vectors:
        /*
        // q1 and q1 are columns of Q
        Matrix q1 = getRow(transpose(Q), k);
```

```
                Vector q2 = getColumn(Q, k+1);
                std::cout << "Q is " << Q << " and " << q1 <<
                " x " << q2 << " = " << q1 * q2 << "\n";
                */

                // We now solve the minimisation problem -------------


                // Notes on implementation:
                // Need to compute the Givens rotation for
                //the input H, this depends on the final
                // two non zero entries of column k...
                // ... once we have multiplied the current
                // H by the total of the previous Givens rotations.
                // This will give a matrix...
                // ... which is upper triangular with an extra
                 //row of zeros
                // Need to store the value of the previous (total)
                // Givens as this is required for the update of
                // current givens
                // Inital value of total givens is 2x2 identity
                // as initial H is 2x1.
                // - need to change H and keep a copy
                // - want the current givens to be kxk

                // Initialise the givens matrix that will zero
                // sub diagonal entries of the newest column of H
                // top left corner of givens is identity.
                // Bottom right 2x2 is the interesting part
                GivensCurrent = eye(k - 1);
                GivensCurrent.addCols(2);
                GivensCurrent.addRows(2);

                // we will change the lower right 2x2 submatrix
                // in current givens matrix to hold values
                // [ c s ; -s c]...
                // ... where c and s are dependent on the final
                // two non zero entries of the kth column of
                // the current upper triangular form.

                // We want to store the H of this iteration
                // as it is used for the next Arnoldi step,
                // therefore make a copy that we change instead
```

```
            Hcopy = GivensTotal * H;

            // Need to multiply this by the current
            // Givens matrix to obtain the current
            // upper triangular matrix
            // So we need to form the givens matrix.
            // This depends on the element on the ,...
            //...   diagonal and the one below it.
            // Call these top and bot

            // Equivalent to QR factorising but
            // since it is Hessenberg initially we
            // need to zero much fewer values

            // calculate givens matrix constants for this iteration
            top = Hcopy.matrixVal[(k - 1) + Hcopy.rows * (k - 1)];
            bot = Hcopy.matrixVal[(k)+Hcopy.rows * (k - 1)];
            div = pow(pow(top, 2) + pow(bot, 2), 0.5);
            c = top / div;
            s = bot / div;



            // now we have the values of the 2x2 we can add
            // these to current givens to give our full
            // transformation matrix

            GivensCurrent.matrixVal[(k-1)+GivensCurrent.rows*(k-1)]
            =c;
            GivensCurrent.matrixVal[(k)+GivensCurrent.rows*(k-1)]
            =-s;
            GivensCurrent.matrixVal[(k-1)+GivensCurrent.rows*(k)]
            =s;
            GivensCurrent.matrixVal[(k)+GivensCurrent.rows*(k)]
            =c;

            // we track the composition of all givens rotations
            temp = GivensCurrent * GivensTotal;
            GivensTotal = temp;

        // current upper triangular matrix
            UT = GivensTotal * H;

```

```
                // append newest residual based on previous
                // residual and the givens transformation
                errors.matrixVal[k] = errors.matrixVal[k - 1] * (-s);
                errors.matrixVal[k - 1] = errors.matrixVal[k - 1] * (c);
                error = fabs(errors.matrixVal[k] / bnorm);
                // for gmres demo
                //residuals.matrixVal[k] = error;


                // check termination condition
                if (error < tol)
                {
                        std::cout << "GMRES_CONVERGED_IN_"
                        << k << "_ITERATIONS_WITH_RESIDUAL_"
                        << error << "\n";

                        //for gmres demo
                        //itsreq = k;
                        //
                        // exit loop
                        k = maxits;
                }
                else
                {
                        // Increase the size of the Givens total
                        // so it is ready for the next loop
                        GivensTotal.addCols(1);
                        GivensTotal.addRows(1);
                        // New givens total = | |-----------------|  0 |
                        //                    | |old givens total| ...|
                        //                    | |-----------------|  0 |
                        //                    |  0      ...      0   1 |
                        GivensTotal.matrixVal
                            [(k+1)+GivensTotal.rows*(k+1)]=1;

                }
                // Uncomment this to print the current iteration and error
                //std::cout << " Current iteration: " << k
                //<< " Error : " << error << "\n";

        }
        // need to remove final row of temp * H to give UT matrix
        Matrix U(UT.rows - 1, UT.columns);
        Vector beta(U.rows);
        Matrix Qf(Q.rows, Q.columns - 1);
        // remove final column of Q (since we terminate with
        // Q_{k+1} but only the upper triangular matrix
        // gained from the QR decomposition of H_{k})
        for (int i = 0; i < Qf.rows; i++)
        {
                for (int j = 0; j < Qf.columns; j++)
                {
                        Qf.matrixVal[i+Qf.rows*j]=
                            Q.matrixVal[i+Q.rows*j];
                }
        }

        // Extract the  upper triangular matrix from UT
        // since it has an extra row of zeros
        // simultaneously extract all but the last rows
        // of the vector of residuals
        for (int i = 0; i < U.rows; i++)
        {
                //yy.matrixVal[i] = y.matrixVal[i];
                beta.matrixVal[i] = errors.matrixVal[i];
                for (int j = 0; j < U.columns; j++)
                {
                        U.matrixVal[i+U.rows*j]=
                            UT.matrixVal[i+UT.rows*j];
                }
        }
        // solve the minimisation problem using Gaussian
        // elimination for the upper triangular matrix U.
        //  This is cheap since it only uses back substitution
        Vector yu = U / beta;

        // convert back to x
        Vector kk = Qf * yu;
        return x0 + kk;

        /* Used for testing gmres convergence
        Vector residuals1(100);
        for (int j = 0; j < itsreq; j++ )
        {
                residuals1.matrixVal[j] = residuals.matrixVal[j+1];
        }
        return residuals1;
        */

}
```

C.5.1 Testing GMRES convergence

```
#include <iostream>
#include  <stdlib.h>
# include <cassert>
#include "Exception.h"
#include "Matrix.h"
#include "Vector.h"
#include <fstream>
int main()
{
        int M = 100;
        // Create random matrix of size M
        Matrix A = rand(M)*10.0;
        // Create RHS TO Ax=b
        Vector b(M);
        // Assign 1s to b
        for (int i = 1; i < M + 1; i++)
        {
                b(i) = 1;
        }
        // set gmres parameters
        int maxits = 1000;;
        double tol = 1.0e-10;
        Vector guess = 0.0 * b;
        // initialise results vector,
        // gmres has currently been modified to return
        // the vector of residuals norm(r_k)/norm(b)
        Vector results;
        // stream out results to .dat file for MATLAB
        // processing
        std::ofstream outfile("gmres_convergence.dat");
        assert(outfile.is_open());
        for (int i = 0; i <= 100; i += 10)
        {
                // we consider convergence for A + multiples
                // of the identity
                results = gmres(A + i * eye(M), b, guess, tol, maxits);
```

```
                    outfile << results;
          }
          outfile.close();

}
```

```
% MATLAB PROCESSING
T = readtable("gmres_convergence.dat");
figure
hold on;
colormap parula
for i=1:11
     ind1 = 100*(i-1)+1;
     plot(1:100,table2array(T(ind1:ind1+99,:)),"LineWidth",1.5);
end
X = [.7 .22];
Y = [.7 .22];
xlabel("Iterations ($k$)",Interpreter="latex",FontSize=30)
ylabel("$\frac{||r_k||}{||b||}$  \quad ...
...  ",Interpreter="latex",Rotation=360,FontSize=30)
annotation('arrow',X,Y);
annotation('arrow',X,Y);
legend("$0$","$10$","$20$","$30$","$40$","$50$","$60$"...
...,"$70$","$80$","$90$","$100$",Interpreter="latex",fontsize=20)
title("Convergence of $A+nI$",Interpreter="latex",FontSize=30)
title(legend,"$n$",Interpreter="latex",FontSize=20)
txt = '$n$';
text(9,.007,txt,Interpreter="latex",FontSize=20)
grid on
```

C.6 Finite Element Solution to Poisson Equation

```
#include <iostream>
#include  <stdlib.h>
#include <cassert>
#include "Exception.h"
#include "Matrix.h"
#include "Vector.h"
#include <fstream>
#include <tuple>

// 1D basis functions ( and derivatives ):
```

```
double phi1(double x);
double phi2(double x);
double gradphi1(double x);
double gradphi2(double x);
// forcing function:
double f(double x);
double masterToLocal(double x, double xl, double xr);
int main()
{
        int meshpts = 1000;
        Vector mesh = linspace(0, 1, meshpts);
        Vector dofs = linspace(1, meshpts, meshpts);
        double h = 1.0 / ((double)meshpts - 1.0);

        // Initialise A matrix (stiffness matrix)
        Matrix A(meshpts); // square of size meshpts
        // Initialise F vector (forcing vector)
        Vector F(meshpts); // size meshpts

        // Initialise DOFS
        int dof1, dof2;

        // Given basis functions 1-x and x,
        // the local matrix will be symmetric
        // so need 3 distinct values:
        double A11, A12, A22;
        A11 = (double)
          1.0/6.0*(1.0/h)*(gradphi1(0.0))*(gradphi1(0.0))
        + 4.0/6.0*(1.0/h)*(gradphi1(0.5))*(gradphi1(0.5))
        + 1.0/6.0*(1.0/h)*(gradphi1(1.0))*(gradphi1(1.0));

        A12 = (double)
          1.0/6.0*(1.0/h)*(gradphi1(0.0))*(gradphi2(0.0))
        + 4.0/6.0*(1.0/h)*(gradphi1(0.5))*(gradphi2(0.5))
        + 1.0/6.0*(1.0/h)*(gradphi1(1.0))*(gradphi2(1.0));

        A22 = (double)
          1.0/6.0*(1.0/h)*(gradphi2(0.0))*(gradphi2(0.0))
        + 4.0/6.0*(1.0/h)*(gradphi2(0.5))*(gradphi2(0.5))
        + 1.0/6.0*(1.0/h)*(gradphi2(1.0))*(gradphi2(1.0));
        // Vector values
        double F1, F2;

        // initialise transformed coordinates
        double x1;
        double x2;
        double x3;

        // loop through the elements in the domain,...
        // ... adding the submatrix to A in the appropriate locations
        for (int element = 1; element < dofs(meshpts); element++)
        {
                // dofs give the index in A that we add to
                dof1 = element;
                dof2 = element + 1;
                // evaluate Simpsons rule points on the local cells
                x1 = masterToLocal(0.0, mesh(dof1), mesh(dof2));
                x2 = masterToLocal(0.5, mesh(dof1), mesh(dof2));
                x3 = masterToLocal(1.0, mesh(dof1), mesh(dof2));
                // evaluate F elements at correct dofs
                F1 = (double)1.0 / 6.0 * (h * f(x1) * phi1(0.0)) +
                        +4.0 / 6.0 * (h * f(x2) * phi1(0.5)) +
                        +1.0 / 6.0 * (h * f(x3) * phi1(1.0));
                F2 = (double)1.0 / 6.0 * (h * f(x1) * phi2(0.0)) +
                        +4.0 / 6.0 * (h * f(x2) * phi2(0.5)) +
                        +1.0 / 6.0 * (h * f(x3) * phi2(1.0));
                // Matrix and vector assembly
                A(dof1, dof1) += A11;
                A(dof1, dof2) += A12;
                A(dof2, dof1) += A12;
                A(dof2, dof2) += A22;
                F(dof1) += F1;
                F(dof2) += F2;
        }
        // Apply boundary conditions
        A(1, 1) = 1;
        A(1, 2) = 0;
        A(meshpts, meshpts) = 1;
        A(meshpts, meshpts - 1) = 0;
        F(1) = 0;
        F(meshpts) = 0;

        // Printing the computed matrix and vector
        //std::cout << "A: " << A << "\n";
        //std::cout << "F: " << F << "\n";

```

```cpp
        // Solving the problem AU = f

        // Using Gaussian elimination
        Vector U_ge(meshpts);
        U_ge = A / F;

        // Using GMRES
        Vector U_gm(meshpts);
        Vector guess = 0 * U_gm;
        int maxits = meshpts;
        double tol = 1.0e-8;
        U_gm = gmres(A, F, guess, tol, maxits);

        // Write results in columns | x | u(x) | to .dat
        // file for MATLAB processing


        // Write Guassian elimination solution for 1000 mesh points

        std::ofstream outfile1("ge_solution_1000.dat");
        assert(outfile1.is_open());
        outfile1 << mesh << " " << U_ge << "\n";
        outfile1.close();

        // Write GMRES solution for 1000 mesh points
        std::ofstream outfile2("gm_solution_1000.dat");
        assert(outfile2.is_open());
        outfile2 << mesh << " " << U_gm << "\n";
        outfile2.close();

}
// basis functions
double phi1(double x){return 1.0 - x;}
double phi2(double x){return x;}
double gradphi1(double x){return -1.0;}
double gradphi2(double x){return 1.0;}
// forcing function
double f(double x)
{
        return -40.0 / 3.0 * pow(x, 3);
}
// master to local mapping
double masterToLocal(double x, double xl, double xr)
```

```
{
        return xl + x * (xr - xl);
}
```

C.7 QR algorithm

```
\subsubsection{QR decomposition}
std::tuple<Matrix, Matrix> qr(const Matrix& A)
{
        // We assume A is square
        int m = A.rows;
        // Throw exception if A is non square
        if (m != A.columns)
        {
                throw Exception(
                "Bad␣input", "QR␣requires␣a␣sqaure␣matrix␣input");
        }
        // Initialise R as A and Q as mxm identity
        Matrix R(A);
        Matrix Q = eye(m);

        // Initialise givens rotataion matrix
        Matrix givens;

        // Initialise variables for givens rotations
        double top, bot, div, c, s;

        // the 2x2 givens matrix will be [c s ; -s c ]...
        // ...where c and s depend on top and bot

        // Loop through each column
        for (int j = 0; j < m; j++)
        {
                // Loop from the final entry to the first...
                // ... subdiagonal entry in each column
                for (int i = m - 1; i > j; i--)
                {
                        // define top and bot to be the pair  ...
                        // ... of adjacent entries on the ...
                        // ... given column at the given row index
                        top = R.matrixVal[(i - 1) + m * j];
                        bot = R.matrixVal[(i)+m * j];

                        // we only want to zero the bottom entry...
                        // ... if it is non zero
                        if (bot != 0)
                        {
```

```
                                        // clear previous givens matrix,
                                        // initialise new one
                                        givens = eye(m);
                                        // compute givens matrix constants
                                        div = pow(pow(top,2)+pow(bot,2),0.5);
                                        c = top / div;
                                        s = bot / div;
                                        // assign values to givens matrix
                                        givens.matrixVal[i+m*i] = c;
                                        givens.matrixVal[i+m*(i-1)] = -s;
                                        givens.matrixVal[(i-1)+m*i] = s;
                                        givens.matrixVal[(i-1)+m*(i-1)] = c;
                                        // apply givens matrix to zero...
                                        // ...subdiagonal entries
                                        R = givens * R;
                                        // track the orthogonal vectors
                                        Q = Q * transpose(givens);
                              }
                    }
          }

          // FOR TESTING
          // we can uncomment below to make it more clear when...
          // ... printing R that it is correct ...
          // ... since often rounding error adds up.
          /*
          double tol = 1e-10;
          for (int i = 0; i < R.rows * R.columns; i++)
          {
                    if (fabs(R.matrixVal[i]) < tol)
                    {
                              R.matrixVal[i] = 0;
                    }
          }
          */

          // return tuple (Q,R)
          return std::make_tuple(Q, R);
}
```

C.7.1 Schur Form

```
Matrix eigMatrix(const Matrix& A)
```

```
{

        int n = A.rows;
        // Initialise QR matrices
        Matrix Q, R;
        // Initialise Schur form matrix
        Matrix E(n);
        // QR already throws for non square input
        // compute QR
        std::tie(Q, R) = qr(A);
        // We impose a hidden maximum iterations for QR
        int maxits = 1000;
        // termination condition counter for subdiagonal
        // zeros
        int ctr = 0;
        int i = 0;
        // tolerence of finding zeros
        double tol = 1.0e-10;
        while (i < maxits && ctr < n - 1)
        {
                std::cout << "it:␣" << i << "\n";
                // mutliply the RxQ
                E = R * Q;
                //evecs = evecs * Q;
                for (int j = 0; j < n-1 ; j++)
                {
                        // check subdiagonal sizes as termination
                        // condition
                        if (fabs(E.matrixVal[(j + 1) + n * j]) < tol)
                        {
                                ctr++;
                        }
                }
                // compute next QR
                std::tie(Q, R) = qr(E);
                i++;
        }
        // We need to compute one final multiplication
        E = R * Q;
        /*
        * Uncomment for testing - zeros elements close to zero
        * easier to track
        for (int i = 0; i < A.rows * A.columns; i++)
        {
                if (fabs(E.matrixVal[i]) < tol)
                {
                        E.matrixVal[i] = 0;
                }
        }
        */
        return E;
}
```

C.7.2 eig()

```
Vector eig(const Matrix& A)
{
      // extracts diagonal elements
      // from the Schur form of A
        int n = A.rows;
        // calculate Schur form
        Matrix eigenvalueMatrix = eigMatrix(A);
        Vector eigVec(n);
        for (int i = 0; i < n; i++)
        {
               // get diagonal elements
                eigVec.matrixVal[i]=eigenvalueMatrix.matrixVal[i+n*i];
        }
        return eigVec;
}
```

C.8 Eigenvalue problem

```
#include <iostream>
#include  <stdlib.h>
# include <cassert>
#include "Exception.h"
#include "Matrix.h"
#include "Vector.h"
#include <fstream>
#include <tuple>
# define M_PI 3.14159265358979323846

// 1D basis functions ( and derivatives ):
```

```
double phi0(double x);
double phi1(double x);
double gradphi0(double x);
double gradphi1(double x);

// xi -> x coordinates caluclator
double masterToLocal(double x, double xl, double xr);

// c function
double c(double x);
int main()
{
        int meshpts = 50;
        Vector mesh = linspace(0.0, M_PI, meshpts);
        Vector dofs = linspace(1, meshpts, meshpts);
        double h = M_PI / ((double)meshpts - 1.0);

        // initialise problem matrices and vector
        Matrix A(meshpts);
        Matrix M(meshpts);
        Vector F(meshpts);
        // initialise dofs
        int dof1, dof2;

        // initialise Matrix values
        double Aii, Aij, Ajj;
        double Mii, Mij, Mjj;

    // Initialise transformed coordinates
        double x1, x2, x3;

        // M can be calculated prior to the loop
        Mii = 1.0 / 6.0 * (h * phi0(0.0) * phi0(0.0))
                + 4.0 / 6.0 * (h * phi0(0.5) * phi0(0.5))
                + 1.0 / 6.0 * (h * phi0(1.0) * phi0(1.0));

        Mij = 1.0 / 6.0 * (h * phi1(0.0) * phi0(0.0))
                + 4.0 / 6.0 * (h * phi1(0.5) * phi0(0.5))
                + 1.0 / 6.0 * (h * phi1(1.0) * phi0(1.0));

        Mjj = 1.0 / 6.0 * (h * phi1(0.0) * phi1(0.0))
                + 4.0 / 6.0 * (h * phi1(0.5) * phi1(0.5))
                + 1.0 / 6.0 * (h * phi1(1.0) * phi1(1.0));
    for (int element = 1; element < dofs(meshpts); element++)
    {
        // assign degrees of freedom - indices for
        // the global matrix
            dof1 = element;
            dof2 = element + 1;

            // convert to local coordinates for the calculation
            // of c(x)
            x1 = masterToLocal(0.0, mesh(dof1), mesh(dof2));
            x2 = masterToLocal(0.5, mesh(dof1), mesh(dof2));
            x3 = masterToLocal(1.0, mesh(dof1), mesh(dof2));


            // calculate local elements using Simpsons rule
            Aii = (double)
            1.0/6.0*((1.0/h)*(gradphi0(0.0))*(gradphi0(0.0))
            +h*c(x1)*phi0(0.0)*phi0(0.0))
            +4.0/6.0*((1.0/h)*(gradphi0(0.5))*(gradphi0(0.5))
            +h*c(x2)*phi0(0.5)*phi0(0.5))
            +1.0/6.0*((1.0/h)*(gradphi0(1.0))*(gradphi0(1.0))
            +h*c(x3)*phi0(1.0)*phi0(0.0));

            Aij = (double)
            1.0/6.0*((1.0/h)*(gradphi1(0.0))*(gradphi0(0.0))
             +h*c(x1)*phi1(0.0)*phi0(0.0))
            +4.0/6.0*((1.0/h)*(gradphi1(0.5))*(gradphi0(0.5))
            +h*c(x2)*phi1(0.5)*phi0(0.5))
            +1.0/6.0*((1.0/h)*(gradphi1(1.0))*(gradphi0(1.0))
            +h*c(x3)*phi1(1.0)*phi0(1.0));

            Ajj = (double)
            1.0/6.0*((1.0/h)*(gradphi1(0.0))*(gradphi1(0.0))
            +h*c(x1)*phi1(0.0)*phi1(0.0))
            +4.0/6.0*((1.0/h)*(gradphi1(0.5))*(gradphi1(0.5))
            +h*c(x2)*phi1(0.5)*phi1(0.5))
            +1.0/6.0*((1.0/h)*(gradphi1(1.0))*(gradphi1(1.0))
            +h*c(x3)*phi1(1.0)*phi1(0.0));

            // append local elements to global matrices
            A(dof1, dof1) += Aii;
            A(dof1, dof2) += Aij;
            A(dof2, dof1) += Aij;
```

```
                A(dof2, dof2) += Ajj;

                M(dof1, dof1) += Mii;
                M(dof1, dof2) += Mij;
                M(dof2, dof1) += Mij;
                M(dof2, dof2) += Mjj;
        }
        // apply boundary conditions
        A(1, 1) = 0;
        A(1, 2) = 0;
        A(meshpts, meshpts) = 0;
        A(meshpts, meshpts - 1) = 0;
        M(1, 1) = 1;
        M(1, 2) = 0;
        M(meshpts, meshpts) = 1;
        M(meshpts, meshpts - 1) = 0;

        // compute the matrix we want the eigenvalues of
        Matrix B = inverse(M) * A;
        // print eigenvalues
        std::cout << "eigs: " << eig(B) << "\n";

}
// master to local map
double masterToLocal(double x, double xl, double xr)
{
        return xl + x * (xr - xl);
}
// c(x) function, we take 0.0 as an example
double c(double x)
{
        return 0.0;
}
// basis functions and their derivatives
double phi0(double x)
{
        return 1.0 - x;
}
double phi1(double x)
{
        return x;
}
double gradphi0(double x)
```

```
{
        return -1.0;
}
double gradphi1(double x)
{
        return 1.0;
}
```

C.9 Miscellaneous Functions

C.9.1 eye()

```
Matrix eye(const int n)
{
        Matrix m(n);
        for (int i = 0; i < n; i += 1)
        {
                m.matrixVal[i + i * n] = 1;
        }
        return m;
}
```

C.9.2 transpose()

```
Matrix transpose(const Matrix& m)
{
        Matrix mt(m.columns, m.rows);
        for (int i = 0; i < mt.rows; i++)
        {
                for (int j = 0; j < mt.columns; j++)
                {
                        mt.matrixVal[i + j * mt.rows] =
                        m.matrixVal[j + i * m.rows];
                }
        }
        return mt;
}
```

C.9.3 rand()

```
Matrix rand(int n)
{
        Matrix A(n);
        srand(time(NULL)); // random seed
        for (int i = 0; i < n * n; i++)
        {
                A.matrixVal[i] =  ((double) rand() / (RAND_MAX));
        }
        return A;
}
```

C.9.4 inverse()

```
Matrix inverse(const Matrix& A)
{
        // uses Gaussian elimination to
        // construct inverse by solving
        // Ax=e_i
        int n = A.rows;
        Matrix Ainv(n);
        Vector col;
        for (int i = 0; i < n; i++)
        {
                Vector b(n);
                b.matrixVal[i] = 1;
                col = A / b;
                Ainv.setColumn(i+1, col);
        }
        return Ainv;
}
```

C.9.5 convertMatrixToVector()

```
Vector convertMatrixToVector(const Matrix& A)
{
        int n = A.rows;
        int m = A.columns;
        if (m != 1)
        {
                std::cout << "BAD_CONVERSION_TO_VECTOR";
                return Vector();
        }
        Vector x(n);
        for (int i = 0; i < n; i++)
        {
                x.matrixVal[i] = A.matrixVal[i];
        }
        return x;
}
```

C.9.6 linspace()

```
Vector linspace(double a, double b, int n)
{
        Vector v(n);
        double length = b - a;
        double h = length / (n-1);
        for (int i = 0; i < n ; i++)
        {
                v.matrixVal[i] = a + h * i;
        }
        return v;
}
```

C.9.7 norm()

```
double norm(Vector v, int p)
{
        double norm_val = 0.0;
        double temp;
        for (int i = 0; i < v.rows; i++)
        {
                temp = fabs(v.matrixVal[i]);
                norm_val += pow(temp, p);
        }
        return pow(norm_val, 1.0 / ((double)(p)));
}
```