Implementing Finite Difference Schemes and GMRES to solve Poisson's Equation in One, Two and Three Dimensions

C++

Candidate Number: 1061996

1 Introduction

Finding analytical solutions to differential equations can be challenging, as such, methods have been developed to approximate solutions to them. One of the most common methods is using finite difference schemes. These can provide accurate solutions and are fairly easy to implement using programming languages such as MATLAB; the problem comes with the time taken to compute solutions for large schemes. In this essay, we hope to improve the computing time of MATLAB while still keeping a similar syntax to it so code can be easily translated from one to the other. We do this using C++ to represent vectors and matrices as classes and create methods and external functions in these classes to compute solutions.

We will study this through Poisson's equation attempting to solve it in 1, 2 and 3 dimensions using finite difference schemes. This will require us to implement a linear solver and we will initially use Gaussian elimination for this. Finite difference methods suffer from the curse of dimensionality in that the number of equations in the linear system which we must solve grows exponentially with the number of independent variables in the differential equation. To enable us to solve larger systems we will implement GMRES as it approximates the solutions to linear systems but is less computationally expensive than Gaussian elimination.

Throughout this essay we will look at the theory behind the problem, explain the code we have implemented and show the tests we have done to ensure we are getting a correct solution. Let us start by looking at the 1 dimensional problem.

2 One Dimensional Problem

The first problem we will study is the 1 dimensional Poisson equation with Dirichlet boundary conditions given by

$$\frac{\mathrm{d}^2u}{\mathrm{d}x^2} = f(x) \tag{1}$$

with $u(a^*) = \alpha$, $u(b^*) = \beta$ and $x \in [a^*, b^*]$, where $a^*, b^*, \alpha, \beta \in \mathbb{R}$. In order to solve this using a finite difference scheme we need to discretise the domain. We do this by taking N evenly spaced points such that the step size between adjacent points is given by $h = \frac{b^*-a^*}{N-1}$ and the mesh on which we get a solution is given by

$$\Omega_h = \{x_j \mid x_j = a^* + jh,\ j = 0,\ \ldots,\ N-1\}.$$

Further to this, we define the set of interior and boundary indices for the 1 dimensional problem by

$$\mathcal{I}_h = \{1,\ \ldots,\ N-2\} \quad \textit{and} \quad \mathcal{B}_h = \{0,\ N-1\},$$

respectively.

Now, we can approximate the second derivative using the second difference

$$D_x^2 u(x) \coloneqq \frac{u(x+h) - 2u(x) + u(x-h)}{h^2} \tag{2}$$

such that we have

$$\frac{u(x_{j+1}) - 2u(x_j) + u(x_{j-1})}{h^2} \approx f(x_j) \tag{3}$$

for all $j \in \mathcal{I}_h$, $u(x_0) = \alpha$ and $u(x_{N-1}) = \beta$. Writing U_j as our approximation of $u(x_j)$ for all $j \in \mathcal{I}_h \cup \mathcal{B}_h$ we get a system of linear equations which we can write in the form $AU_h = b$ where

$$A = \frac{1}{h^2}\begin{bmatrix} h^2 & 0 & 0 & \dots & 0 \\ 1 & -2 & 1 & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & 1 & -2 & 1 \\ 0 & \dots & 0 & 0 & h^2 \end{bmatrix} \in \mathbb{R}^{N\times N},$$
$$b = \begin{bmatrix} \alpha \\ f(x_1) \\ \vdots \\ f(x_{N-2}) \\ \beta \end{bmatrix} \in \mathbb{R}^N \text{ and } U_h = \begin{bmatrix} U_0 \\ U_1 \\ \vdots \\ U_{N-2} \\ U_{N-1} \end{bmatrix} \in \mathbb{R}^N. \tag{4}$$

Solving this linear system will then give us our numerical solution U_h for points on the mesh Ω_h.

To quantify how close our predicted solutions are to the analytical solution we use either the mesh dependent L^2 norm or the mesh dependent L^∞ norm given for a function on a mesh with d independent variables by

$$||u_h||_{2,h} \coloneqq \left(\sum_{j\in(\mathcal{I}_h \cup \mathcal{B}_h)} h^d |u(x_j)|^2 \right)^{1/2} \quad \text{and} \quad ||u_h||_{\infty,h} \coloneqq \max_{j\in(\mathcal{I}_h\cup\mathcal{B}_h)} |u(x_j)|$$

respectively, where $\mathcal{I}_h$ and $\mathcal{B}_h$ are the sets of interior and boundary indices, respectively [4].

To analyse the quality of our numerical solutions we will ensure they converge to the correct solution at the expected rate as h decreases. For this problem we would expect second order convergence in the mesh dependent L^2 norm when $u \in C^4([a^*, b^*])$. That is, if we define U by

$$U = \begin{bmatrix} u(x_0) \\ \vdots \\ u(x_{N-1}) \end{bmatrix} \in \mathbb{R}^N$$

then we we should have that [5]

$$||E_h||_{2,h} \coloneqq ||U - U_h||_{2,h} = \mathcal{O}(h^2) = \mathcal{O}(N^{-2}). \tag{5}$$

It is often the case there is the same rate of convergence in both of these norms however it can be difficult to find proofs for both norms. Hence, we will reference the results we have proofs for and test if our code is working correctly using that norm. However, we will also include a line generated using the other norm in the plots to see how the convergence compares.

Now that we have the theory out of the way we can get onto the implementation. We started with a class of matrices and a class of vectors that we had developed as part of the assignments. Within each of these classes we had overloaded many of the binary operators, such as $+$, $-$ and $*$. We had overloaded the assignment operator $=$ and created external functions to give the dimensions of the vector or matrix such that we could make the class member storing the dimensions private. In addition, we had created a constructor which can dynamically allocate memory and a destructor to free the memory when the object is no longer in use.

The first thing we wanted to add to our code was a function to access the data stored in the matrix and vectors with the indexing starting at 0. We do this so we can make the data stored in these private as it is good software engineering practice. The choice to start the indexing at 0 is simply because that is what I find most intuitive as I am used to using Python. We also added an exception to prevent us from picking an index too small or large and accessing data which is not stored in the array.

Next, we wanted to overload the / operator in the class of matrices such that when a non-singular $n \times n$ dimensional matrix and an n dimensional vector are input it will output the solution to the linear system. We decided to do this by Gaussian elimination using the algorithm from [2]. Here, the main challenges came in us using a variable but changing it without realising. For example, this can happen when subtracting one row of a matrix from another row with the first row being multiplied by a number from the second row. We may change the number we are multiplying by in the subtraction and therefore start multiplying by a different number. However, once we realised we made sure to be careful with our loops and set extra variables to store numbers that we want to stay constant if they may be changed in the iteration. Our code for this can be seen in listing 5 in appendix A.

For a linear system to have a unique solution the matrix representing it must be non-singular. The matrix is non-singular if and only if it becomes the identity matrix after row reduction. Therefore, if we apply our Gaussian elimination solver to a singular matrix we will not get an answer and in fact will get an indexing error due to our function trying to find a non-zero entry in a column of zeroes and eventually iterating outside of the matrix.

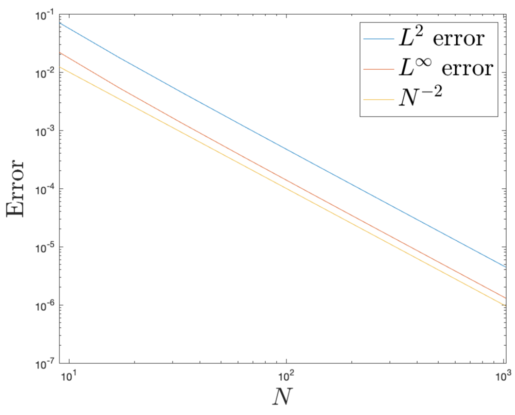
Now we have all the tools we need we just need to set up the system in C++. Since we will be testing this with quite a few different problems and then generalising this code so it can deal with higher dimensional problems we want to make the code as easy to modify as possible. As such, we set up functions for f and the exact solution u and variables for the parameters a^*, b^*, α, β and N at the start of the code. The matrix and vectors are then created using these functions and parameters such that they represent equation (4). If we change any of these functions or variables the code should still give the correct solution to the new problem. The code to define these vectors and matrices is quite simple so we will not write it here and instead save space for the more challenging higher dimensional implementations later.

To test if our implementation is working correctly we will calculate the L^2 error $||E_h||_{2,h} = ||U - U_h||_{2,h}$. As a simple first check we will make sure that the L^2 error is small, say less than 0.1 for $N = 11$, and that it decreases as N increases. We can then vary N, calculate the error and create a log-log plot of the L^2 error against N. This can be implemented using a simple for loop and storing the different N and errors in an array before writing them to a file such that we can read it and plot the data in MATLAB. The plot will be log-log so we will increase N exponentially with data such that $N = 2^j + 1$ for all $j \in \{2, \ldots, 10\}$. Given that $||E_h||_{2,h} = \mathcal{O}(N^{-2})$ (from equation (5)) we should have a straight line of gradient -2 for the log-log plot so we will plot a straight line of gradient -2 on the same graph to check we get the correct convergence.

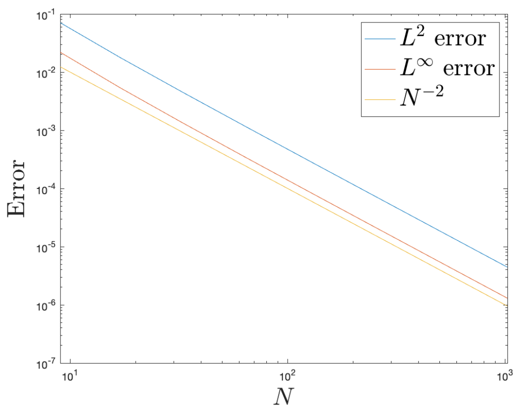
The first problem we will use to test convergence is

$$\frac{\mathrm{d}^2u}{\mathrm{d}x^2} = \sin x + \cos x \tag{6}$$

with $u(0) = 0$, $u(2\pi) = 2\pi$ and $x \in [0, 2\pi]$. This has an analytical solution given by



(a) For DE with domain $[0, 2\pi]$



(b) For DE with domain $[2\pi, 4\pi]$

Figure 1: Plots of the errors against N for the 1D Poisson problem with inhomogeneous boundary conditions.

$u(x) = 1+x-\sin x-\cos x$. Running our code which does not loop over different values of N and instead setting $N = 11$ we find that it has an L^2 error of 0.0136. Increasing N we see that the L^2 error decreases. We therefore decide to run the code which loops over different values of N, write the data to a file and plot it using MATLAB to see the rate of convergence. This gives us the plot shown in figure 1a. Clearly, the lines are parallel so we have the required rate of convergence. This suggests our implementation is working correctly, but let us do one more test.

To test our code on problems on a different domain we use the equation given by equation (6) but change the domain to $[2\pi, 4\pi]$ and the boundary conditions to $u(2\pi) = 2\pi$ and $u(4\pi) = 4\pi$. This gives the same analytical solution $u(x) = 1 + x - \sin x - \cos x$ thus we only have to change the parameters in our code. We can see from figure 1b that the plot is practically indistinguishable from figure 1a, hence we will assume our implementation is working correctly and move on to a harder problem: implementing a finite difference scheme to solve 2 dimensional problems.

3 Two Dimensional Problem

The second problem we want to look at is the 2 dimensional Poisson equation with Dirichlet boundary conditions given by

$$\Delta u(x,y) = f(x,y), \quad \text{for all } (x,y) \in \Omega, \tag{7}$$

$$u(x,y) = g(x,y), \quad \text{for all } (x,y) \in \partial\Omega, \tag{8}$$

where Ω is a rectangular open set and $\partial\Omega$ is the boundary of Ω. Again, the first thing we need to do to apply a finite difference scheme is discretise the domain. In this section, we will only be working with rectangular domains, that is domains of the form $[a^*, b^*] \times [c^*, d^*]$ and therefore they will be simple to discretise. If we create a grid of N_x evenly spaced points in the x-direction and N_y evenly spaced points in the y-direction we get the mesh

$$\Omega_h = \{(x_j, y_k) \mid x_j = a^* + jh_x,\ y_k = c^* + kh_y,\ j = 0, \ldots, N_x - 1 \text{ and } k = 0, \ldots, N_y - 1\}$$

where $h_x = \frac{b^* - a^*}{N_x - 1}$ and $h_y = \frac{d^* - c^*}{N_y - 1}$. We can then define the set of interior and boundary indices by

$$\mathcal{I}_h = \{(j, k) \mid j \in \{1, \ldots, N_x - 2\},\ k \in \{1, \ldots, N_y - 2\}\}, \tag{9}$$

$$\mathcal{B}_h = \{(j, k) \mid j \in \{0, \ldots, N_x - 1\},\ k \in \{0, \ldots, N_y - 1\}, \tag{10}$$

$$\text{and } j \in \{0,\ N_x - 1\} \text{ or } k \in \{0,\ N_y - 1\}\}, \tag{11}$$

respectively.

This time we have two second order derivatives each of which we can approximate with the second difference on the interior of the mesh. Writing $u_{j,k} := u(x_j, y_k)$ for all $j \in \{0, \ldots, N_x - 1\}$, $k \in \{0, \ldots, N_y - 1\}$ we get the system of equations given by

$$\frac{u_{j+1,k} - 2u_{j,k} + u_{j-1,k}}{h_x^2} + \frac{u_{j,k+1} - 2u_{j,k} + u_{j,k-1}}{h_y^2} \approx f(x_j, y_k) \quad \text{for all } (j, k) \in \mathcal{I}_h, \tag{12}$$

$$u_{j,k} = g(x_j, y_k) \quad \text{for all } (j, k) \in \mathcal{B}_h. \tag{13}$$

This time, we write $U_{j,k}$ as our approximation of $u_{j,k}$ for all $(j, k) \in \mathcal{I}_h \cup \mathcal{B}_h$. Implementing this as the linear system $AU_h = b$ in C++ is much more challenging than the previous system of equations due to the indexing.

The matrix is too complex to write out in LaTeX, however we will describe its structure and explain why it looks like it does. The matrix will be $N_xN_y \times N_xN_y$ dimensional. The first and last N_y rows' entries will all be 0 aside from on the main diagonal of the matrix, where the entries will be 1. Furthermore, if the top row of the matrix is considered to be row 1 then each row number which is a multiple of N_y and the row below it will also have all 0 entries aside from having the entry 1 on the diagonal. This is so we can set the boundary conditions. All the other rows will be set in the same way: the entry on the main diagonal will be $-2/h_x^2 - 2/h_y^2$, the adjacent column's entries on either side of the main diagonal will be $1/h_y^2$ and the entries in the columns N_y steps away from the main diagonal will be $1/h_x^2$. This is to satisfy equation (12) when

$$U_h = \begin{bmatrix} U_{0,0} \\ U_{0,1} \\ \vdots \\ U_{0,N_y-1} \\ U_{1,0} \\ U_{1,1} \\ \vdots \\ U_{N_x-1,N_y-1} \end{bmatrix} \in \mathbb{R}^{N_xN_y} \text{ and } b = \begin{bmatrix} g(x_0,y_0) \\ \vdots \\ g(x_0,y_{N_y-1}) \\ g(x_1,y_0) \\ f(x_1,y_1) \\ \vdots \\ f(x_1,y_{N_y-2}) \\ g(x_1,y_{N_y-1}) \\ \vdots \\ g(x_{N_x-1},y_0) \\ \vdots \\ g(x_{N_x-1},y_{N_y-1}) \end{bmatrix} \in \mathbb{R}^{N_xN_y}.$$

The last of the theory that we will look at before we get in to the implementation will be the convergence. For this scheme and this problem we would expect the method to converge at a rate of 2 with h on a square domain when $N = N_x = N_y$, $h = h_x = h_y$ and $u \in C^4(\Omega)$. This should be the same for both the L^2 norm and the L^∞ norm. That is,

$$\begin{aligned} ||E_h||_{2,h} &\coloneqq ||U - U_h||_{2,h} = \mathcal{O}(h^2) = \mathcal{O}(N^{-2}), \\ ||E_h||_{\infty,h} &\coloneqq ||U - U_h||_{\infty,h} = \mathcal{O}(h^2) = \mathcal{O}(N^{-2}) \end{aligned} \tag{14}$$

(see [4]) where U is the vector of the exact solution following the same indexing as U_h and b.

First, we want to test on Laplace's equation with inhomogeneous Dirichlet boundary conditions to check the implementation of the boundary conditions is working correctly. We do this by means of the problem

$$\Delta u(x,y) = 0 \tag{15}$$

with $u(0,y) = -y^2$, $u(1,y) = 1 - y^2$, $u(x,0) = x^2$, $u(x,1) = x^2 - 1$ and $(x,y) \in [0,1]^2$. This has an analytical solution given by $u(x,y) = x^2 - y^2$.

Similar to the 1 dimensional case, we want to make the code easy to modify for different problems. Initially, we do the most obvious step of initialising new variables to define the domain and the number of grid points in either dimension. We then update the functions for f and the exact solution to take two variables and give the correct outputs. The main challenge is implementing the matrix representing A. Carefully following the structure we laid out a few paragraphs ago and testing by printing the matrix for small values of N_x and N_y leads us to getting a matrix of the correct form and what looks like the correct entries.

The final two challenges are implementing a function to set the entries representing the boundary conditions in the vector for b and implementing the vector U which gives the exact solution at each of the mesh points. Initially, we choose to implement the vector U. We can calculate the entries for this vector using the function for the exact solution but the main difficulty with this is getting the indexing correct such that the correct values take the correct places. We do this using the `floor` function and the % operator to calculate the values of x and y being represented by the single index of the vector and then substituting x and y into the function for the exact solution u. That is, we use `exact_vec[i] = u_exact(x_0 + floor(i/N_y) * h_x, y_0 + (i%N_y)*h_y);`. This then makes setting the boundary values in the vector for b easy as we have a way to take an index and return the exact solution at that point. Using this we develop the function given in listing 1.

Listing 1: Code to define the boundary conditions

```
double boundary(int i, int N_x, int N_y, double h_x,
                double h_y, double x_0, double y_0)
{
        return u_exact(x_0 + floor(i/N_y) * h_x,
                       y_0 + (i%N_y)*h_y);
}
```

We then test this by using `boundary` to set the entries of the vector for b when they represent a boundary condition. We do this using the code in listing 2, then printing our vector for b for some small N_x and N_y and using our calculator to check that we are getting the correct boundary conditions. We are, hence we can now easily implement new examples by only changing the functions for f and the exact solution and the variables which define the domain. As such, let us move on to solving the finite difference problem.

Listing 2: Code to define the matrix and vector in 2D

```
//      Defining the matrix and vector st A*u_pred = y
for(int i=0; i<N_x*N_y; i++)
{
```

```
//      when x = x_0 (ignoring last entry as included later)
        if (i<N_y-1)
        {
            A[i][i] = 1;
            y[i] = boundary(i, N_x, N_y, h_x, h_y, x_0, y_0);
        }

//      when x = x_N (ignoring first entry as included later)
        else if (i> N_x*N_y - N_y)
        {
            A[i][i] = 1;
            y[i] = boundary(i, N_x, N_y, h_x, h_y, x_0, y_0);
        }

//      when y = y_0
        else if ((i+1)%N_y == 0)
        {
            A[i][i] = 1;
            y[i] = boundary(i, N_x, N_y, h_x, h_y, x_0, y_0);
        }

//      when y = y_N
        else if (i%N_y == 0)
        {
            A[i][i] = 1;
            y[i] = boundary(i, N_x, N_y, h_x, h_y, x_0, y_0);
        }

//      all other entries
        else
        {
            A[i][i] = -2*(1/pow(N_x,2) + 1/pow(N_y,2));
            A[i][i-1] = 1/ pow(N_y,2);
            A[i][i+1] = 1/ pow(N_y,2);
            A[i][i-N_y] = 1/ pow(N_x,2);
```

```
                A[i][i+N_y] = 1/ pow(N_x,2);
                y[i] = f(x_0 + floor(i/N_y) * h_x,
                                y_0 + (i%N_y)*h_y);
        }
}
```

Given that the exact solution to this problem is quadratic, if we derive the second difference operator for u using Taylor series expansions we find that the second difference operator for u is the same as the second derivative of u at any point in the domain and for any step size. As such, we would expect our finite difference method to give the exact solution up to rounding errors. Running our code we find that we have an L^2 error of 1.2×10^{-16} and an an L^∞ error of 5.5×10^{-16} when $N = N_x = N_y = 11$, both of which increase as N increases. We would not have expected the increase, however as the exact solution exists in $[-1, 1]$ we will assume for now that this a result of the errors at each point of the mesh being so close to machine precision that we cannot get an accurate picture of convergence. Thus, we move on to the more complex examples: Poisson's equation with homogeneous Dirichlet boundary conditions and Poisson's equation with inhomogeneous Dirichlet boundary conditions. Ultimately, we will check the rate of convergence using the latter example to ensure our implementation is working correctly.

For our test using Poisson's equation with homogeneous Dirichlet boundary conditions we use the equation

$$\Delta u(x, y) = 2x^2 + 2y^2 - 2x - 2y \tag{16}$$

with $u(0, y) = 0$, $u(1, y) = 0$, $u(x, 0) = 0$, $u(x, 1) = 0$ and $(x, y) \in [0, 1]^2$. This has an analytical solution given by $u(x, y) = x(1 - x)y(1 - y)$. As our exact solution has no terms greater than degree 2 in either variable, each of our second difference operators should be equal to their respective second order derivative and we should again get the exact solution up to rounding errors.

To adapt our code from the previous example we simply have to change our functions for f and the exact solution. This will in turn update our boundary conditions. If we then run the code for $N = 11$ and we get an L^2 error of 366 and an L^∞ error of 756 so our approximation is extremely poor given that the function u varies between 0 and 0.0625 on our domain. Furthermore, we find that increasing N_x and N_y only make the errors larger. Therefore, there must be a problem with our implementation.

Initially, as we saw that the code approximated the solution so accurately in the example we did before this, we assume that the error must be in something we have changed. As such, we start looking at how we have defined the vector for b in our code. Running the code for some small $N = N_x = N_y$ and printing the vector that represents b we can clearly see it is of the correct form in that the zero and non-zero entries are in the correct positions. To check the non-zero entries are being defined correctly we pick a few random ones, find their indices and calculate the value that should be there using $f(x, y)$. All of these come back correct so we believe the problem is elsewhere in the code.

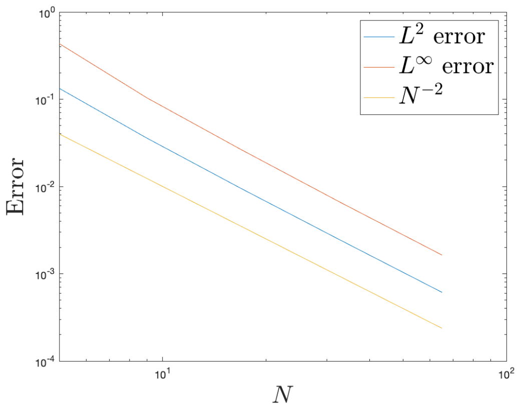
To test our linear solver is working correctly we run the code to calculate our approximate solution then left multiply this vector by the matrix representing A. This returns b so the linear solver must be working correctly. It therefore seems as though the problem is in the matrix A itself. Printing it again for some small $N = N_x = N_y$ we can see it is of the correct form in that the zero and non-zero entries are in the correct positions. Looking more closely, we see that in our implementation we have accidentally used `N_x` and `N_y` instead of `h_x` and `h_y` when defining entries of the matrix A. This can be seen in listing 2. Correcting this in our code we now get an L^2 error of 1.3×10^{-17} and an L^∞ error of 4.2×10^{-17} for $N_x = N_y = 11$. This is much more accurate but we again find that as $N = N_x = N_y$ increases the error seems to increase, even though it does still stay very small. We assume that this is due to the individual errors being so small again. Next time, we will pick a problem such that the second difference operator is not equal to the second derivative at all points in the domain so that we get larger errors and can hopefully see convergence.

Before we go onto the next example we will quickly go back and correct the code for the previous two 2 dimensional example. For Laplace's equation with inhomogeneous Dirichlet boundary conditions we find that for $N = N_x = N_y \leq 35$ that errors are of the same order of magnitude for the incorrect and the correct code which seems quite strange as you would assume getting this wrong would have a much larger effect on the solution. This is why we test on many different examples. Again, this example has the pattern that increasing $N = N_x = N_y$ actually makes the errors slightly larger.

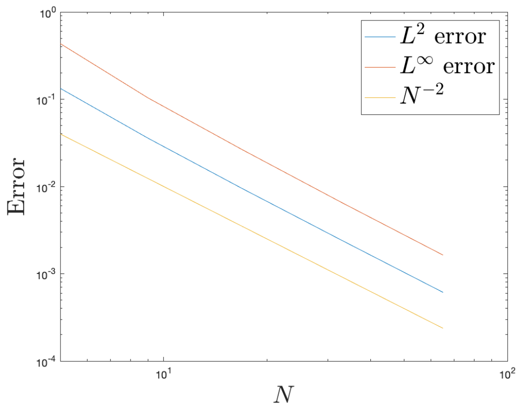
For the next test we will use the equation

$$\Delta u(x, y) = -\sin x - \cos y \tag{17}$$

with $u(0, y) = \cos(y)$, $u(2\pi, y) = \cos(y)$, $u(x, 0) = \sin(x) + 1$, $u(x, 2\pi) = \sin(x) + 1$ and $(x, y) \in [0, 2\pi]^2$. This has an analytical solution given by $u(x, y) = \sin x + \cos y$. We can easily implement this from our previous examples by changing the parameters that define the domain and the functions for f and the exact solution of the problem.



(a) For PDE with domain $[0, 2\pi]^2$



(b) For PDE with domain $[2\pi, 4\pi]^2$

Figure 2: Plots of the errors against N^{-2} for the 2D Poisson problem with inhomogeneous boundary conditions alongside the line $y = N^2$.

Running the code using $N = N_x = N_y = 11$ we get an L^2 error of 0.029 and an L^∞ error of 0.080. Running the code for different N we see that the errors decrease as N increases so to test if we have the required rate of convergence we plot loss against N, as shown in figure 2a. Here, we see we have the expected rate of convergence in both norms so the finite difference scheme working as we would like it to.

To test the code on a different domain we use the same equation but simply change the variables which define the domain. Due to the way our boundary conditions are defined we do not have to change anything else. The convergence plot using exactly the same implementation but with domain $[2\pi, 4\pi]^2$ is given in figure 2b. Again, we get the required convergence with the plot looking practically identical to figure 2a.

As a final test we want to see if we can find the solution to a PDE on a rectangular, but not square domain with different step sizes in the x and y directions. To do this we will again use equation (17) with $u(x, y) = \sin x + \cos y$ and simply change the domain to $[0, 6\pi] \times [0, \pi]$ and the number of points in each spatial dimension to $N_x = 18$ and $N_y = 9$. Checking the L^∞ error we see that it is 0.0065 and it decreases as the step sizes decreases, hence such a small error implies that our implementation also works for this. As such, we will now move on to the 3 dimensional problem.

4 Three Dimensional Problem

The next problem we want to look at is the 3 dimensional Poisson equation with Dirichlet boundary conditions given by

$$\Delta u(x, y, z) = f(x, y, z), \quad \text{for all } (x, y, z) \in \Omega, \tag{18}$$

$$u(x, y, z) = g(x, y, z), \quad \text{for all } (x, y, z) \in \partial\Omega, \tag{19}$$

where Ω is a cuboidal open set and $\partial\Omega$ is the boundary of Ω. First, we will discretise the domain. Here we will be working with cubic domains, that is on sets of the form $[a^*, b^*] \times [c^*, d^*] \times [e^*, f^*]$ where $a^*, b^*, c^*, d^*, e^*, f^* \in \mathbb{R}$. These can be easily discretised by taking a grid of N_x evenly spaced points in the x-direction, N_y evenly spaced points in the y-direction and N_z evenly spaced points in the z-direction to give us the mesh

$$\begin{aligned}\Omega_h = \{(x_j, y_k, z_l) \mid\ & x_j = a^* + jh_x,\ y_k = c^* + kh_y,\ z_l = e^* + lh_z, \\ & j = 0,\ \ldots,\ N_x - 1,\ k = 0,\ \ldots,\ N_y - 1 \text{ and } l = 0,\ \ldots,\ N_z - 1\}\end{aligned}$$

where $h_x = \frac{b^* - a^*}{N_x - 1}$, $h_y = \frac{d^* - c^*}{N_y - 1}$ and $h_z = \frac{f^* - e^*}{N_z - 1}$. We can then define the set of interior and boundary indices by

$$\begin{aligned}\mathcal{I}_h &= \{(j, k, l) \mid j \in \{1,\ \ldots,\ N_x - 2\},\ k \in \{1,\ \ldots,\ N_y - 2\},\ l \in \{1,\ \ldots,\ N_z - 2\}\} \\ \mathcal{B}_h &= \{(j, k, l) \mid j \in \{0,\ \ldots,\ N_x - 1\},\ k \in \{0,\ \ldots,\ N_y - 1\}, \\ &\qquad l \in \{0,\ \ldots,\ N_z - 1\} \text{ and } j \in \{0,\ N_x - 1\},\ k \in \{0,\ N_y - 1\} \\ &\qquad \text{or } l \in \{0,\ N_z - 1\}\},\end{aligned}$$

respectively.

This time we have three second order derivatives each of which we can approximate with the second difference on the interior of the mesh. Writing $u_{j,k,l} \coloneqq u(x_j, y_k, z_l)$ for all $j \in \{0,\ \ldots,\ N_x - 1\}$, $k \in \{0,\ \ldots,\ N_y - 1\}$, $l \in \{0,\ \ldots,\ N_z - 1\}$ we get the system of equations given by

$$\begin{aligned}&\frac{u_{j+1,k,l} - 2u_{j,k,l} + u_{j-1,k,l}}{h_x^2} + \frac{u_{j,k+1,l} - 2u_{j,k,l} + u_{j,k-1,l}}{h_y^2} \\ &\quad + \frac{u_{j,k,l+1} - 2u_{j,k,l} + u_{j,k,l-1}}{h_z^2} \approx f(x_j, y_k, z_l) \quad \text{for all } (j, k, l) \in \mathcal{I}_h, \\ &u_{j,k,l} = g(x_j, y_k, z_l) \quad \text{for all } (j, k, l) \in \mathcal{B}_h.\end{aligned}$$

Implementing this as the linear system $AU_h = b$ in C++ is even more challenging than the previous system of equations. The vectors for U_h and U are given by

$$U_h = \begin{bmatrix} U_{0,0,0} \\ U_{0,0,1} \\ \vdots \\ U_{0,0,N_z-1} \\ U_{0,1,0} \\ U_{0,1,1} \\ \vdots \\ U_{0,N_y-1,N_z-1} \\ U_{1,0,0} \\ \vdots \\ U_{N_x-1,N_y-1,N_z-1} \end{bmatrix} \in \mathbb{R}^{N_xN_yN_z} \text{ and } U = \begin{bmatrix} u(x_0,y_0,z_0) \\ u(x_0,y_0,z_1) \\ \vdots \\ u(x_0,y_0,z_{N-1}) \\ u(x_0,y_1,z_0) \\ u(x_0,y_1,z_1) \\ \vdots \\ u(x_0,y_{N_y-1},z_{N_z-1}) \\ u(x_1,y_0,z_0) \\ \vdots \\ u(x_{N_x-1},y_{N_y-1},z_{N_z-1}) \end{bmatrix} \in \mathbb{R}^{N_xN_yN_z}$$

while b follows the same indexing arrangement and has entries which are either given by $f(x_j, y_k, z_l)$ if (x_j, y_k, z_l) does not lie on the boundary or $g(x_j, y_k, z_l)$ if (x_j, y_k, z_l) lies on the boundary. Again, we will describe the matrix's structure then implement it.

The matrix will be $N_xN_yN_z \times N_xN_yN_z$ dimensional. If we map the single index of the row we are on to the three indices j, k and l given by the vectors U_h and b we can decide which rows of our matrix correspond to boundary points and which do not. If any of the indices are 0 or $N_p - 1$ where p is x for the index j, and so on, then the row corresponds to a boundary. As such, all entries of the row should be 0 except on the main diagonal where we should have 1. Then, all other rows correspond to interior points so we want the entry on the main diagonal to be $-2/h_x^2 - 2/h_y^2 - 2/h_z^2$, the entry either side of the main diagonal to be $1/h_z^2$, the entries in the columns N_z steps away from the main diagonal to be $1/h_y^2$ and, finally, the entries in the columns N_yN_z steps away from the main diagonal to be $1/h_x^2$.

For the rest of this essay let $N := N_x = N_y = N_z$. For this scheme we would expect the L^∞ error to converge at a rate of 2 with h on a cubic domain when $N = N_x = N_y = N_z$, $h = h_x = h_y = h_z$ and $u \in C^4(\Omega)$. That is,

$$||E_h||_{\infty,h} := ||U - U_h||_{\infty,h} = \mathcal{O}(h^2) = \mathcal{O}(N^{-2}) \tag{20}$$

(see [1]) where U is the $N_xN_yN_z$ dimensional vector of the exact solution at the corresponding positions given by the indices in U_h and b.

Now we can move on to the implementation. The most challenging part of the implementation will be creating the new matrix for A and vector for b. Last time we used an if statement and 3 else if statements to decide whether the row corresponded to a point on the boundary, however creating these conditions for the 3 dimensional problem now that there are even more boundaries would be quite challenging. Instead, we will apply a new idea. Using the single index which gives the row of the matrix representing A or the row of the vector representing b and creating three functions which give the corresponding value of x, y and z at this point then checking whether any of these are on the boundary of the domain.

In fact, we have actually already developed the formulas for the functions in the 2 dimensional case: they are used in listing 1. But we had not defined them as functions. These can be extended to get the code we see in listing 3. We can pick small values for N_x, N_y and N_z and run through the integers from 0 to $N_xN_yN_z$ as an index while printing the values of x, y and z we get with these functions. This works correctly giving points covering the whole mesh in the correct order. Using this to implement the matrix for A and vector for b we get the code given in listing 4.

Listing 3: Code to give the positions in space given an index.

```
//Gives the x position given an index i
double xfromindex(int i, double x_0, int N_y, double N_z,
                         double h_x)
{
        return x_0 + floor(i/(N_y*N_z)) * h_x;
}

//Gives the y position given an index i
double yfromindex(int i, double y_0, int N_y, int N_z,
                         double h_y)
{
        return y_0 + ((i/N_z)%( N_y))*h_y;
}

//Gives the z position given an index i
double zfromindex(int i, double z_0, int N_z, double h_z)
{
        return z_0 + (i%N_z)*h_z;
}
```

We will now start working to solve Laplace's equation with inhomogeneous Dirichlet boundary conditions on $[0,1]^3$. We start working with a duplicate of the file we used to solve the analogous problem in 2 dimensions and add the new code to define the matrix and vector. Printing the matrix for small N_x, N_y and N_z it looks as though it takes the correct form, however the main test for this will be if we get the correct rate of convergence for later examples.

Listing 4: Code to define the matrix and vector in 3D

```
//      Defining the matrix and vector st Au_pred = y_vec
for(int i=0; i<N_x*N_y*N_z; i++)
{
        double x = xfromindex(i, x_0, N_y, N_z, h_x);
        double y = yfromindex(i, y_0, N_y, N_z, h_y);
        double z = zfromindex(i, z_0, N_z, h_z);

        if ((x!=x_0 && x!=x_N) && (y!=y_0 && y!=y_N) &&
            (z!=z_0 && z!=z_N))
        {
                A[i][i] = -2*(1/pow(h_x,2) + 1/pow(h_y,2)
                            + 1/pow(h_z,2));
                A[i][i-1] = 1/ pow(h_z,2);
                A[i][i+1] = 1/ pow(h_z,2);
                A[i][i-N_y] = 1/ pow(h_y,2);
                A[i][i+N_y] = 1/ pow(h_y,2);
                A[i][i-N_y*N_z] = 1/ pow(h_x,2);
                A[i][i+N_y*N_z] = 1/ pow(h_x,2);
                y_vec[i] = f(x, y, z);
        }

        else
        {
                A[i][i] = 1;
                y_vec[i] = boundary(i, N_x, N_y, N_z, h_x,
                                    h_y, h_z, x_0, y_0, z_0);
        }
}
```

Since we have so many boundaries we will define our problems slightly differently from now on in that we will not give the boundary conditions. Instead, we will give the exact solution and the domain then the boundary conditions can be recovered easily. The first problem we want to solve is

$$\Delta u(x, y, z) = 0 \tag{21}$$

with $(x, y, z) \in [0, 1]^3$ and an analytical solution given by $u(x, y, z) = 2x^2 - y^2 - z^2$. We change the functions for f and the exact solution to take another variable and output the correct solution then we run the code. We get the usual case of a very small error which slightly increases as N increases.

The next test will be on a Poisson problem with homogeneous Dirichlet boundary conditions to ensure the rows relating to the interior of the mesh are being defined correctly. We use the equation

$$\Delta u(x, y, z) = -2x(1-x)y(1-y) - 2y(1-y)z(1-z) - 2z(1-z)x(1-x) \tag{22}$$

with $(x, y, z) \in [0, 1]^3$ and an analytical solution given by $u(x, y, z) = x(1-x)y(1-y)z(1-z)$. Running this again gives a tiny L^∞ error of 6.3×10^{-18} for $N = 11$ so we assume the implementation works. This is despite the fact that increasing N causes the error to slightly increase.

Now, we want to solve a Poisson problem with inhomogeneous boundary conditions and such that the second difference operator is not equal to the second derivative at all points in the domain so we can get larger errors and better convergence plots. The equation we will use to do this is will pick a problem next time so that we get larger errors and can hopefully see convergence

$$\Delta u(x, y, z) = -\sin x - \cos y - \sin z \tag{23}$$

with $(x, y, z) \in [0, 2\pi]^3$ and with an analytical solution given by $u(x, y, z) = \sin x + \cos y + \sin z$. Running our code for this problem and $N = 11$ we get an L^∞ error of 0.078. We then also run the code for $N = 5$ and 17 and from this we see that the error is decreasing as N increases, as we would have hoped. The problem is, running the code for much larger N takes too long so we cannot get solutions using my laptop. However, we need values of L^∞ for larger N if we want to make our plot of L^∞ against N more reliable. The part of the our code which is taking the most time to run is our linear solver so we will implement GMRES which should approximate the solution to our linear system giving this solution much more quickly.

5 GMRES

First, we will briefly discuss how GMRES works before getting into the implementation and testing. We follow the theory and algorithms given in the Oxford University Numerical Linear Algebra course [3] and our code is shown in listing 6 in appendix B.

Suppose we want to solve the linear system $Ax = b$ with $A \in \mathbb{R}^{n\times n}$ and $x, b \in \mathbb{R}^n$. GMRES approximates the solution to a linear system by attempting to minimise the residual in the Krylov subspace. That is, we want to find x such that

$$x = \underset{x\in\mathcal{K}_k(A,b)}{\arg\min}\, ||Ax - b|| \tag{24}$$

where $||.||$ is the Euclidean norm and $\mathcal{K}_k(A, b) := \operatorname{span}([b, Ab, \dots, A^{k-1}b])$. This takes three steps. First, we need to choose a $k \in \mathbb{N}$ and apply the Arnoldi iteration for k steps. This gives us an upper Hessenberg matrix $\tilde{H}_k \in \mathbb{R}^{(k+1)\times k}$ and two more matrices whose columns form orthonormal bases of $\mathcal{K}_k(A, b)$ and $\mathcal{K}_{k+1}(A, b)$, these are $Q_k \in \mathbb{R}^{n\times k}$ and $Q_{k+1} \in \mathbb{R}^{n\times k+1}$, respectively. Furthermore, the matrices are such that $AQ_k = Q_{k+1}\tilde{H}_k$. Substituting this into $\min_{x\in\mathcal{K}_k(A,b)} ||Ax - b||$, writing $x = Q_k y$ and taking advantage of the structure afforded by Arnoldi we find that minimising $||Ax - b||$ over $x \in \mathcal{K}_k(A, b)$ is equivalent to minimising $||\tilde{H}_k y - Q^T_{k+1} b||$ over $y \in \mathbb{R}^k$. This is a least-squares problem and can therefore be solved by computing the QR decomposition of $\tilde{H}_k$ such that $\tilde{H}_k = QR$ where $Q \in \mathbb{R}^{(k+1)\times(k+1)}$ is orthogonal and $R \in \mathbb{R}^{(k+1)\times k}$ is such that the top k rows give an upper triangular matrix and the bottom row only contains zeros. We can find the QR decomposition by applying k Givens rotations to $\tilde{H}_k$, then we can find y by solving $Ry = Q^T Q^T_{k+1} b$ while ignoring the bottom row of both R and the right-hand side. We solve this using triangular solve [3]. Lastly, we compute $x = Q_k y$ to find the solution to the original problem.

Now on to the implementation. We want to make the data types of the inputs and output the same as that of our Gaussian elimination solver so we can just swap this function in for the old one. The first thing we want to implement is the Arnoldi iteration. We initially get an error running our code since we cannot use the [ ] operator on our inputs as we labelled them as constant. Thus, we make copies of our input vector and matrix and try again copying the algorithm given in [3] and creating a norm function for matrices so we can use it in our algorithm.

To test our algorithm we created a 20×20 matrix such that all off diagonal entries are given by 0 and all diagonal entries are given by the row number where we say the top row is row 1. We also defined a dimension 20 vector such that the entry on the

ith row is i. Running this for $k = 4$ we initially find that the first two columns are orthogonal but none of the others. This problem reoccurs when we test for different matrices and values of k. This suggests that there is some problem in the way our loop works and, in fact, it turns out we did not include a line to update a vector just before the end of the loop. Correcting this and calculating $Q_k^T Q_K$ shows the columns are orthonormal for $k \in \{1, \ldots, 19\}$.

As this is working, we try using the matrix and vector given in our last 3 dimensional example with $N = 17$. For k below a certain number the columns of Q_k are orthonormal, however as we increase k the columns eventually stop being orthonormal. The specific point this happens is hard to tell given that when we calculate the dot product of two columns we never actually get 0, we tend to just get very small numbers. The loss in orthogonality will be because the Krylov subspace $\mathcal{K}_k(A, b)$ may not always be k dimensional in that the vectors $\{b, Ab, \ldots, A^{k-1}b\}$ may lose linear independence. If this happens then we cannot form an orthonormal basis of $\mathcal{K}_k(A, b)$ so we start to get columns of Q_k which are not orthonormal. Although we have not strictly showed that the vectors of Q_k span the space $\mathcal{K}_k(A, b)$ we will see if we can get the whole implementation of GMRES to work and if we cannot get it to we will come back and check this.

Now, we need to find the QR factorisation of $\tilde{H}_k$. As this is upper Hessenberg we can apply k Givens rotations. We then calculate $Q^T = G_k G_{k-1} \ldots G_1$ and $R = G_k G_{k-1} \ldots G_1 \tilde{H}_k$ where G_i is the ith Givens rotation applied. To test this we print the upper Hessenberg matrices with the rotations applied to ensure they have become upper triangular.

Lastly, we need to solve $Ry = Q^T Q_{k+1}^T b$ but ignoring the bottom row of R and the right-hand side. The main problem we have here is making sure the data is of the correct type such that we can apply certain functions to it. We could solve this by overloading operators and functions, such as the multiplication operator to work with more data types. In the end, this may have been easier however we chose instead to just change vectors into matrices and matrices into vectors as and when we needed to so the code is slightly messy. Another problem with this part of the implementation is that the NLA lecture notes [3] were not clear about what matrix we have to use for the matrix we call Q_{k+1} in this part of the problem. As such, first I tried to use Q_k however I caught an exception from the overloaded $*$ operator saying that the product $Q^T Q_k^T$ could not be computed as the matrices dimensions are not compatible. Deriving the last part of the implementation myself we find that we can use Q_{k+1} instead of Q_k and to get the correct solution. We then left multiply the vector for y

we get by Q_k to get a solution for x.

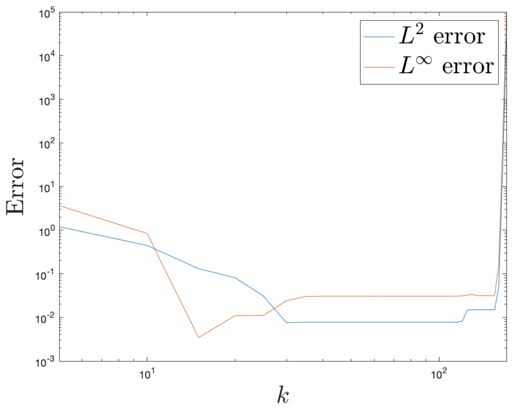
In [3] it says to use triangular solve to solve this last system. Looking at our implementation of the Gaussian elimination solver it will automatically apply triangular solve for an upper triangular matrix.

To test our implementation we use the 20×20 matrix and 20 dimensional vector that we used the for Arnoldi iteration. Running this for $k = 5, 10, 15, 19$ and also solving with our Gaussian elimination function then calculating the L^∞ norm of the difference between our solutions we see that the approximation becomes more accurate at k increases. Ultimately, at $k = 19$ the L^∞ norm of the difference becomes 1.4×10^{-6}. Hence, GMRES provides an accurate approximation. Running our code again with $k = 4$ but with b having 1 as its first entry and 0 as all other entries we find that the program returns a vector of all nan entries. This is expected as $A^t b = b$ for all $t \in \{1, 2, 3\}$ hence we instantly lose linear independence so cannot calculate a 4 dimensional orthogonal basis for this.

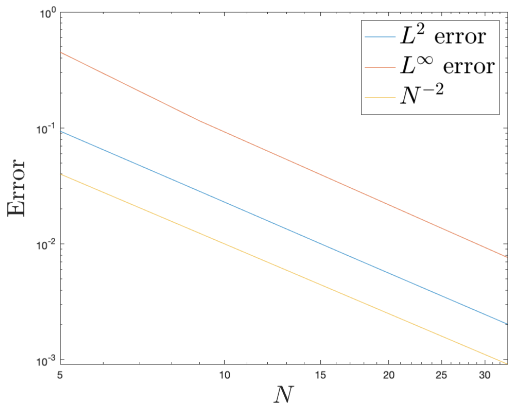
We will now try to use GMRES in our 3 dimensional Poisson problem with inhomogeneous Dirichlet boundary conditions. To implement this we can simply replace the Gaussian elimination solver with the GMRES solver and choose our desired k. Given that we know the exact solution to the problem we can calculate the L^∞ and L^2 error using our predicted solution and the exact solution.

Running our code for different values of k and trying to minimise both of the errors we find that after the error initially decreases it then increases before stabilising for a while then blowing up, as shown in figure 3a. We expect the initial decrease, as we have more vectors in our Krylov subspace, and the final blow-up, due to us losing linear independence of the vectors spanning the Krylov subspace, however we did not expect the minimum to not be directly before the blow-up. Furthermore, we found that the minimum for both errors was found at different k. The error that we are making is that we should not be minimising the errors over k, we should be waiting until they stabilise. This is because at larger k (but before the blow up) GMRES is providing a more accurate approximate solution to the linear system, it is just that a more accurate solution to the linear system is not necessarily a better prediction of the solution to the PDE. In fact, if we take the solutions from when GMRES has stabilised for $N = 5, 9, 17$ and compare them to the solutions found using Gaussian elimination then we see they both have the same L^∞ error to 3 significant figures.

Repeating our previous experiments with GMRES for $N = 5, 9, 17, 33$ and taking the value of our errors to 3 significant figures after the error has stabilised we get the plots shown in figure 3b. The lines follow a similar gradient so we have the required



(a) Errors against k for $N = 17$.



(b) Errors against N.

Figure 3: Plots for the 3D Poisson problem with inhomogeneous boundary conditions.

convergence indicating that both our GMRES and 3 dimensional finite difference implementation is working. We would have liked to try this for larger N as I believe it would have made the lines look even closer to being parallel. Unfortunately, the size of these matrices makes the program take too long to run on my laptop.

We then repeat this experiment for equation (23) with the same solution $u(x, y, z) = \sin x + \cos y + \sin z$ but on the domain $[2\pi, 4\pi]^3$. This too gives us the expected convergence so we can assume our code works on different domains.

Although there is not space left to cover this we can also easily modify the code to work on cuboid, rather than cubic domains and with different step sizes in each direction. These modifications can be done easily by setting the variables at the start of the program differently, similar to what we did for the 2 dimensional problem.

6 Conclusion

In this essay, we introduced finite difference schemes to allow us to solve Poisson's equation in 1, 2 and 3 dimensions. We did this by developing classes to represent vectors and matrices therefore allowing us to improve computing time compared to MATLAB while keeping a similar syntax. To solve the systems generated by the finite difference schemes we developed two linear solvers: one which works via Gaussian elimination and the other using GMRES.

We tested thoroughly using assertions, exceptions and the printing of objects to help debug the code. Ultimately, we used convergence plots as the final test to show our schemes work correctly.

References

[1] Salwa Abd-El-Hafiz, Gamal Ismail, and Berlant Mattit. A numerical technique for the 3-d poisson equation. *International Journal of Pure and Applied Mathematics*, 7:263–270, January 2003.

[2] Diane Maclagan and Damiano Testa. Lecture notes in MA106 Linear Algebra from the Mathematical Institute, University of Warwick, January 2010.

[3] Yuji Nakatsukasa. Lecture notes in C6.1 Numerical Linear Algebra from the Mathematical Institute, University of Oxford, December 2021.

[4] Björn Stinner and Susana Gomes. Lecture notes in MA3H0 Numerical Analysis and PDEs from the Mathematical Institute, University of Warwick, February 2020.

[5] Xiangxiong Zhang. Lecture notes in MA615 Numerical Methods for PDEs from the Department of Mathematics, University of Purdue, Spring 2022.

A Gaussian Elimination Code

Listing 5: Code for the Gaussian elimination solver

```
// Solving a linear system using Gaussian elimination
// (for matrix and vector)
Vector operator/(const Matrix& A, const Vector& b)
{
        // check vectors of correct dimensions
        assert ((size(A)[0] == length(b)) &&
        (size(A)[1] == length(b)));

        int i=0,j=0;
        int n = size(A)[0];

        //      Create matrix and vector to work with
        Matrix A_copy(A);
        Vector b_copy(b);

        // Use j to decide when we finish.
        // If non-singular should be when j=n
        while (j<n)
        {
                // make A[i][j] nonzero
                int homerow = i;
                while (A_copy[i][j] == 0)
                {
                        i += 1;
                }

                // swapping row i with row pivot row
                double* swap;
                swap = new double [n];
                for(int k=j; k<n; k++)
                {
                        swap[k] = A_copy[i][k];
                        A_copy[i][k] = A_copy[homerow][k];
```

```
          A_copy[homerow][k] = swap[k];
}
swap[0] = b_copy[i];
b_copy[i] = b_copy[homerow];
b_copy[homerow] = swap[0];


// set i back to the top row
i=homerow;

// set first entry of row to 1
if (A_copy[i][j] != 1)
{
          // set first1 to divide through the
          // other elements as A[i][j] would
          // change
          double first1 = A_copy[i][j];
          for(int k=0; k<n; k++)
          {
                    A_copy[i][k] /= first1;
          }
          b_copy[i] /= first1;
}


// subtract top row from lower rows
for(int k=i+1; k<n; k++)
{
          double first2 = A_copy[k][j];
          for(int l=0; l<n; l++)
          {
          A_copy[k][l] -= A_copy[i][l]*first2;
          }
          b_copy[k] -= b_copy[i]*first2;
}
```

```
                // subtract top row from higher rows
                for(int k=0; k<i; k++)
                {
                        double first = A_copy[k][j];
                        for(int l=0; l<n; l++)
                        {
                        A_copy[k][l] -= A_copy[i][l]*first;
                        }
                        b_copy[k] -= b_copy[i]*first;
                }

                // move pivot diagonally
                i += 1;
                j += 1;
        }
        return b_copy;
}
```

B GMRES Code

Listing 6: Code for the GMRES solver

```
// Solving a linear system using GMRES
Vector GMRES(const Matrix& A, const Vector& b, int k)
{
// check vectors of correct dimensions
assert ((size(A)[0] == length(b)) &&
        (size(A)[1] == length(b)));

//      Check k is viable
        assert (k<=size(A)[0]);

//      Make copy of A and b so can keep type const
        Matrix A_copy(A);
        Vector b_copy(b);
```

```
//      Setting the size of the vectors to n
        int n = size(A)[1];

////////////////////////////////////////
//////// Arnoldi iteration /////////////
////////////////////////////////////////

//      creating the matrix Q_{k+1} but we just call it Q_k
//  for brevity
        Matrix Q_k(n, k+1);


//      Defining the vector we will use for q_i (as a matrix
//  so can do operations more easily)
        Matrix q(n,1);
        for(int i = 0; i<n; i++)
        {
                q[i][0] = b_copy[i];
        }
        q = q/norm(q);

//      Setting first column of Q_k to q_i
        for(int l=0; l<n; l++)
        {
                Q_k[l][0] = q[l][0];
        }



//      creating the matrix H which is (k+1) by k
        Matrix H(k+1, k);



//      Calculating the Arnoldi iteration k times
        for(int i=0; i<k; i++)
        {
                Matrix v(n,1);
                v = A*q;


                for(int j=0; j<i+1; j++)
                {
//              calculating dot product of Q_j and v. Can
//      do like this as all matrix data initialised to 0
                        for(int l=0; l<n; l++)
                        {
                                H[j][i] += Q_k[l][j]*v[l][0];
                        }

//                      Calculating the new vector v
                        for(int l=0; l<n; l++)
                        {
                                v[l][0] = v[l][0]
                                        - H[j][i]*Q_k[l][j];
                        }
                }
                H[i+1][i] = norm(v);

//              setting new column of Q_k
//              if  (i<n-1)
//              {
                        for(int l=0; l<n; l++)
                        {
                            Q_k[l][i+1] = v[l][0] / H[i+1][i];
                            q[l][0] = Q_k[l][i+1];
                        }
//              }
        }
//      checking the output of the arnoldi iteration
        std::cout << "Q_k = \n" << Q_k << "\n";
        std::cout << "H = \n" << H << "\n";

//      checking the columns of Q_k are orthogonal
```

```
          std::cout << "Q_k^T * Q_k = \n" <<
                    transpose(Q_k)*Q_k << "\n";


//////////////////////////////////////////////////
//////// QR by Givens rotations of H ////////
//////////////////////////////////////////////////

//        define matrices for R and Q_T for the QR
//        factorisation of H Q_T stands for Q^T
//        with Q from the QR factorisation
          Matrix R(H);
          Matrix Q_T(k+1,k+1);
          Q_T = eye(k+1);

//        We need to apply k Givens rotations
          for(int i=0; i<k; i++)
          {
//                  Defining each Givens rotation
                    Matrix G(k+1,k+1);
                    for(int j=0; j<k+1; j++)
                    {
                              G[j][j] = 1;
                    }

                    double c = R[i][i] / pow( pow(R[i][i],2)
                              + pow(R[i+1][i],2), 0.5);
                    double s = R[i+1][i] / pow( pow(R[i][i],2)
                              + pow(R[i+1][i],2), 0.5);
                    G[i][i] = c;
                    G[i+1][i] = -s;
                    G[i][i+1] = s;
                    G[i+1][i+1] = c;

                    R = G*R;
                    Q_T = G*Q_T;
```

```
//                  testing the Givens rotations work
//                  std::cout<< "R = " << "\n" << R << "\n";
//                  std::cout.flush();
          }


//////////////////////////////////////////////////
/////// Solving Ry = Q^T Q_{k+1}^T b ////////
//////////////////////////////////////////////////

//        Write b as a matrix so we can multipy by it
          Matrix b_mat(n,1);
          for(int i = 0; i<n; i++)
          {
                    b_mat[i][0] = b_copy[i];
          }

//        To solve the Hessenberg least squares problem we
//        solve Ry = Q_T Q_{k+1}^T b but ignoring the bottom
//        row of the matrices on either side where x = Q_k*y
          Vector y(k);

//        Creating Vector and matrix without the bottom row
          Matrix R2(k,k);
          Matrix RHS(k+1,1);
          Matrix RHS2(k,1);

          RHS = Q_T*(transpose(Q_k)*b_mat);

          for(int i=0; i<k; i++)
          {
                    for(int j=0; j<k; j++)
                    {
                              R2[i][j] = R[i][j];
                    }
```

```
            RHS2[i][0] = RHS[i][0];
        }

        y = R2 / RHS2;

//      Writing y as a matrix so we can multiply by it
        Matrix y_mat(k,1);
        for(int i = 0; i<k; i++)
        {
            y_mat[i][0] = y[i];
        }


//      creating Q_k so we can calculate x from y
        Matrix Q_k2(n,k);
        for(int i=0; i<n; i++)
        {
            for(int j=0; j<k; j++)
            {
                Q_k2[i][j] = Q_k[i][j];
            }
        }

//      checking the Gaussian elimination solver is working
//      correctly
//      std::cout<< R2*y_mat << "\n";
//      std::cout<< RHS2 << "\n";
//      std::cout.flush();


        Matrix output_mat(n,1);
        Vector output(n);

//      calculating x = Q_k*y
//      changing from y back to x
        output_mat = Q_k2*y_mat;
```

```
//          converting to a Vector
            for(int i=0; i<n; i++)
            {
                                  output[i] = output_mat[i][0];
            }

            return output;
}
```