HIGH PERFORMANCE COMPUTING

PROJECT STAGE 1

Name: Dylan Scotney
Student number: 18138211
Submission date: 08/04/2019

1. INTRODUCTION

The aim of this coursework is to solve the Poisson equation,

$$-\nabla \cdot (\sigma(x,y)\nabla)u(x,y) = f(x,y) \tag{1.01}$$

For $(x,y) \in [0,1] \times [0,1]$ with boundary conditions $u(x,y) = 0$ by using the finite difference method. To simplify, we will use $f(x,y) = 1$ throughout.

To solve Equation 1.01 via an FDM implementation, a function will be written that applies a discrete version of the operator, $\nabla \cdot (\sigma(x,y)\nabla)$, to an input vector $u(x,y)$ using OpenCL. This function can then be converted to a linear operator class using the *LinearOperator* object from SciPy[i] so that iterative solvers *gmres* and *bicgstab* can be used to find a solution for $u(x,y)$.

In order to apply FDM to the Poisson Equation, we first need to define a mesh. For simplicity, we shall define this as a *uniform* grid of spatial points in the (x,y) plane. The number of discrete grid points in the x-axis is denoted by N_x and likewise for the y-axis, N_y. We can define each point on the mesh using:

$$x_i = i * dx \tag{1.02}$$

$$y_j = j * dy \tag{1.03}$$

Where i,j are integrers and dx, dy are the grid spacings in the x and y directions given by $dx = 1/N_x$ and $dy = 1/N_y$. An example 5x5 grid is illustrated in Figure 1.1 (i).

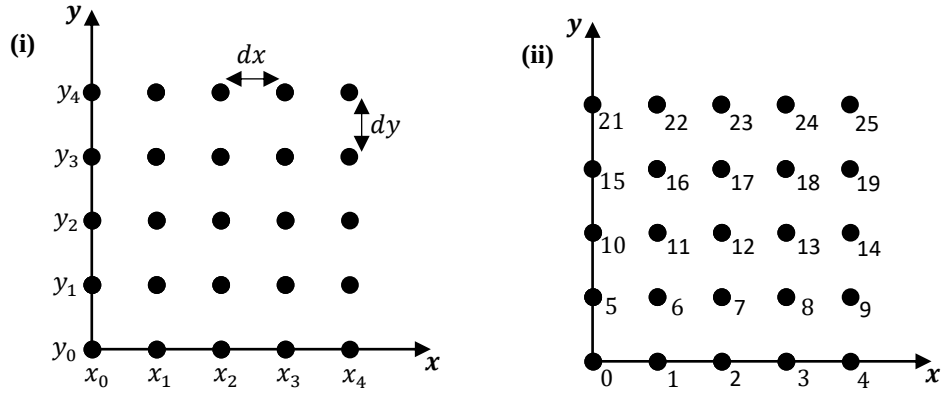

Figure 1.1. (i) Mesh points on a uniform FDM grid. (ii) The super indexing structure to represent 2D space using 1D vectors

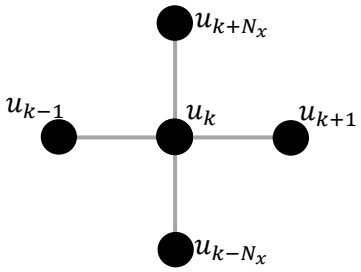

Figure 1.2. Grid points neighbouring u_k and their indices using the introduced super index notation

To apply the Poisson equation to a mesh like Figure 1.1 (i) we need to be able to express it in a discrete form. Using a standard finite difference scheme, the operator acting on $u(x,y)$, $\nabla \cdot (\sigma(x,y)\nabla)$, can be expressed approximately as:

$$\nabla \cdot (\sigma(x,y)\nabla)u \approx \left(\frac{\sigma_{i+\frac{1}{2},j}\left(\frac{u_{i+1,j}-u_{i,j}}{dx}\right) - \sigma_{i-\frac{1}{2},j}\left(\frac{u_{i,j}-u_{i-1,j}}{dx}\right)}{dx} \right) + \left(\frac{\sigma_{i,j+\frac{1}{2}}\left(\frac{u_{i,j+1}-u_{i,j}}{dy}\right) - \sigma_{i,j-\frac{1}{2}}\left(\frac{u_{i,j}-u_{i,j-1}}{dy}\right)}{dy} \right) \tag{1.04}$$

Where the notation represents $u_{i,j} \coloneqq u(x_i, y_i)$, $\sigma_{i+\frac{1}{2},j} \approx (\sigma_{i+1,j} + \sigma_{i,j})$, and $\sigma_{i-\frac{1}{2},j} \approx (\sigma_{i-1,j} + \sigma_{i,j})$.

As mentioned, we are aiming to solve the Poisson equation by converting the above operator into a LinearOperator object using SciPy, i.e., we are looking to solve,

$$Au = f \tag{1.05}$$

Where A is a $[N_x * N_y] \times [N_x * N_y]$ matrix representing the operator in Equation 1.04, u and f are 1D vectors of length $N_x * N_y$ representing $u_{i,j}$ and $f_{i,j}$. To represent discrete 2D functions in a 1D vector we need to use a super index notation such that each node is represented with a single index, $k \in [0, N_x * N_y)$. The order in which nodes are numbered is not important but they must be kept *consistent* throughout so that all parameters of the Poisson equation are represented with the same index notation, i.e., u_k must corresponds to f_k and σ_k. An example of the super indexing structure used for this solution on a small 5x5 grid is shown in Figure 1.1 (ii). This changes how we will access neighbouring grid points of u_k. For example, in order to now access what was $u_{i,j+1}$ in the double index scheme we must use u_{k+N_x}. See Figure 1.2.

Implementation of the boundary conditions for this problem is simple. Since we are dealing with Dirichlet boundary conditions of $u(x,y) = 0$ at $x = 0, 1$ and $y = 0, 1$ we do not have to modify the operator acting on points on the boundary. Instead we must force the BCs onto f in Equation 1.05. That is, for every point, $u_k = 0$, that lies on the boundary, the corresponding f_k must also be set to $f_k = 0$.

2. FDM IMPLEMENTATION IN OPENCL

Using the super index notation described in the introduction, a function (which we will refer to as *poissionOp()*) will be written that takes a 1D input vector, $\boldsymbol{u}$, representing $u(x,y)$ and returns another 1D vector corresponding to the discrete form of $\nabla \cdot (\sigma(x,y)\nabla)\boldsymbol{u}$ from Equation 1.04. The input value of $\boldsymbol{u}$ is irrelevant since later we will convert *poissonOp()* into a linear operator using SciPy and solve

for $\boldsymbol{u}$. In parallel computing, this problem is what is known as *embarrassingly parallel* – one in which very little effort is required to separate the problem into parallel tasks. Here, because the problem is not time dependent nor do we modify $\boldsymbol{u}$ or $\boldsymbol{\sigma}$ when calculating the return vector of *poissonOp()*. We can easily calculate each element of the return vector of *poissonOp()* in parallel (so long as all thread have access to $\boldsymbol{u}$, $\boldsymbol{\sigma}$). Where $\boldsymbol{\sigma}$, like $\boldsymbol{u}$, is the 1D vector representing $\sigma(x, y)$ at each mesh point using the super index notation. It follows that, since we do not care about the order in which elements of the output are computed, we only require one work-group with $N_x * N_y$ work items. Refer to the accompanying *Jupyter Notebook* for the implementation of this.

3. SOLUTIONS AND VALIDATION USING FENICS

As mentioned, the parallelised OpenCL FDM implementation of the *poissonOp()* function was converted to a *LinearOperator* object using the SciPy library so that a solution for u could be found using iterative solvers, *gmres* and *bicgstab*. The performance of these solvers will be compared.

Using these iterative solvers, the solution for $u(x, y)$ was found using the FDM scheme for $N_x = 100$ and $N_y = 100$ and a field of $\sigma(x, y) = 1 + x^2 + y^2$. To validate the results FEnICS was used to solve the same system via a FEM scheme using 10,000 elements. The results of the *gmres*, *bicgstab* and FEnICS solutions are shown in figure 3.1.

To get a better sense for the accuracy of the solution, the values of $u(x, y)$ each mesh point was compared with that of the FEniCS solution. The average relative difference between these points was recorded i.e.,

$$average\ relative\ error = \sum_{i,j=0}^{i=N_x, j=N_y} \frac{\left|u_{FDM}(x_i, y_j) - u_{FEM}(x_i, y_j)\right|}{u_{FEM}(x_i, y_i)}$$

Where $u_{FDM}(x_i, y_i)$ represents the solution obtained via finite difference and u_{FEM} represents the FEnICs solution which uses finite elements.

For a 100×100 mesh grid this gave a the same relative error of 0.08411 for both iterative methods. Which is no surprise as *gmres* and *bicgstab* are expected to converge to the same solution.

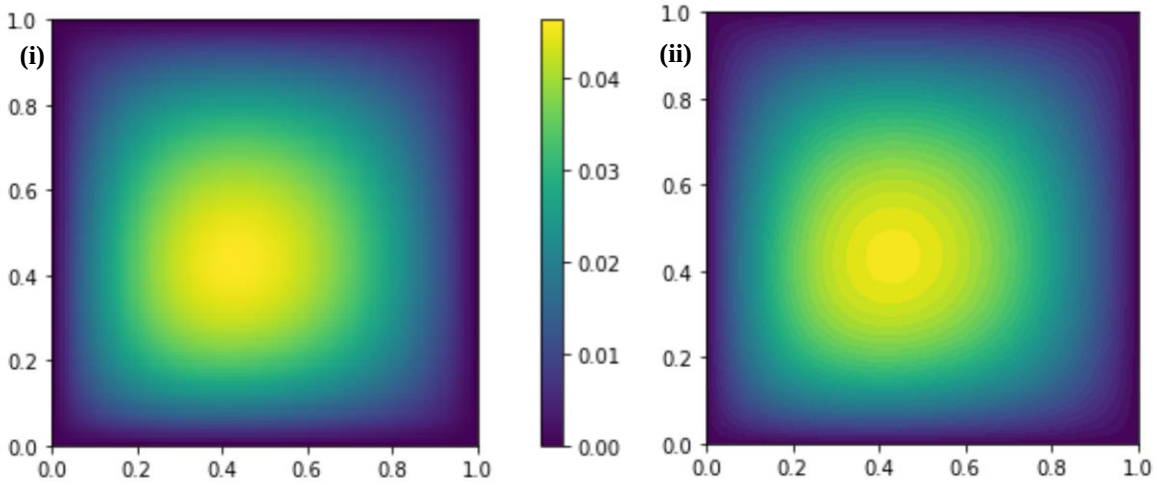

Figure 3.1. (i) The *gmres/bicgstab* solutions (the plots are indistinguishable by eye). (ii) The FEnICS solution.

4. PERFORMANCE OF DIFFERENT ITERATIVE SOLVERS

Now that we have validated method, this section will study the performance of each iterative method when calculating the solution for a field $\sigma(x, y) = \exp(S(x, y))$ where $S(x, y) \sim N(0, 0.1)$. A small variance was used to increase the accuracy and speed of the computations (A higher variance led to higher required iterations). The solution for this system is shown in Figure 4.1.

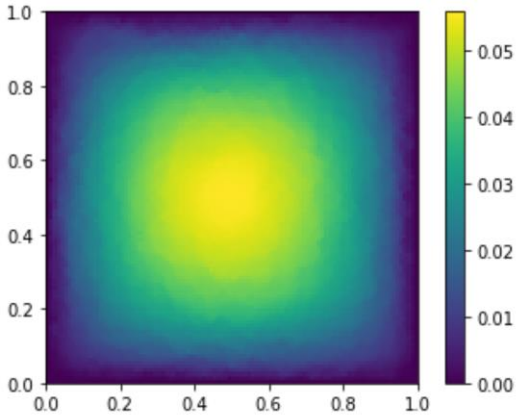

Figure 4.1. (i) The *gmres/bicgstab* solution for a 100×100 grid and $\sigma(x, y) = \exp(S(x, y))$

Firstly, in order to compare the convergence rates of the methods. The residual at each iteration was plotted for the solution of a 100×100 grid. Referring to figure 4.2 it's clear that *bicgstab* converges in far fewer iterations than *gmres*. Note, sometimes this does not necessarily mean that *bicgstab* is the faster overall implementation. It may be that *bicgstab* has very computationally expensive iterations and therefore be slower overall. This is not the case for this situation however, as any extra computational expense (if any) is vastly outweighed by the massive reduction in iterations.

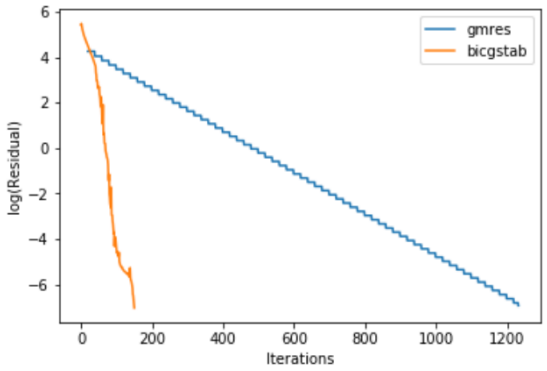

Figure 4.2. The *gmres/bicgstab* solution for a 100×100 grid and $\sigma(x, y) = \exp(S(x, y))$

If we define the rate of convergence as *the magnitude of the gradient* in Figure 4.2 we can they easily study the relationship between number of discretisation points and the rate of convergence. Firstly, the effect of increasing the number of discretisation points in just one spatial dimension was studied. $N_x = 100$ was held at a constant and the convergence rate was measured for $N_y \in (10, 200)$. Due to the symmetry of the problem the results are similar if N_x was varied whilst N_y was held constant. Figure 4.3 illustrates the fact that for both methods, the rate of convergence decreases with an increasing number of grid points.

Secondly, the number of points in both spatial coordinates where increased symmetrically and again the rate of convergence (which we define above) was plotted against $N_x * N_y$. Figure 4.4. (i) illustrates the results.

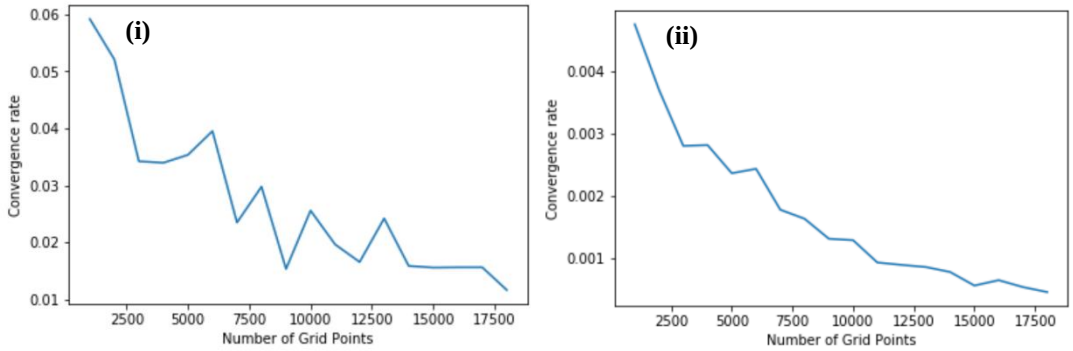

Figure 4.3. The relationship between number of grid points and the convergence rate measured as the magnitude of the gradient in Figure 4.2 for $N_x = 100$ and $N_y \in (20{,}200)$ (i) The *bicgstab* method and (ii) the *gmres* method.

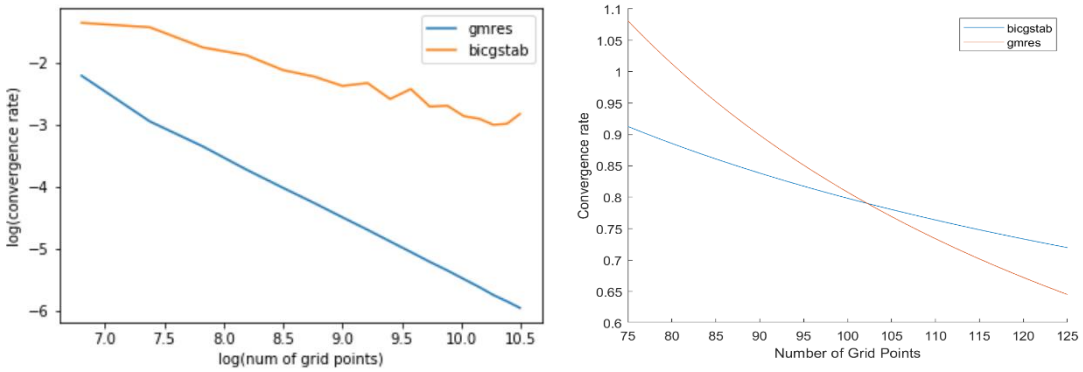

Figure 4.4. (i) The relationship between number of grid points and the convergence rate measured as the magnitude of the gradient in Figure 4.2 for $N_x, N_y \in (20, 150)$. (ii) The experimental relationship extracted from Figure 4.4 (i) as described by Equations 4.01, 4.02.

Figure 4.4 illustrates linear relationships between $\log(convergence\ rate)$ and $\log(N_x N_y)$. It follows that by approximating the gradient for both methods and extrapolating for an intercept we can find an experimental relationship between the two for both *gmres* and *bicgstab* iterative methods, which are:

$$C_b = 6.61N^{-0.46} \tag{4.01}$$

$$C_g = 79.8N^{-0.98} \tag{4.02}$$

Where C_b, C_g represent the convergence rate of the *bicgstab* and *gmres* method respectively and N is the total number of grid points. From Figure 4.4 (ii) it follows that bicgstab becomes more efficient for greater than ~ 100 grid points. Although this information isn't very useful in this case, since we likely want more than 100 grid points to achieve a greater accuracy for the solution. Again, it's also worth noting that faster convergence does not necessarily mean faster computational time as one method may have more computationally expensive iterations than the other. In theory this should be easy to check by comparing the computational time vs number of iterations completed.

[i] Scipyorg. 2019. [Online]. [7 March 2019]. Available from: https://docs.scipy.org/doc/scipy/reference/