NTNU
Norwegian University of
Science and Technology

TMR4345 - Marine Computer Science Lab
Project report

# An application for solving the two-dimensional discrete Poisson equation

Jørgen R. Høstmark

May 9, 2021

# Contents

# 1 Introduction

The Poisson equation is an important elementary partial differential equation (PDE for short), that is found in numerous applications related to theoretical physics. In particular, the Poisson equation appears in fluid dynamics as an equation describing the pressure field of an incompressible fluid. It is therefore important for engineers to have access to computer programs that can solve this equation efficiently, so that they may use them to simulate fluid flow in a variety of different applications and settings. While there already exist several programs for general computational fluid dynamics (such as OpenFOAM), it is still useful for engineers and engineering students to have some understanding of how the programs perform these computations, the necessary theory and the implementation behind them, in order to create good mathematical models of problems related to fluids. This is especially relevant for software developers and researchers.

In this project, I have learned about methods for solving a discrete version of the two-dimensional Poisson equation and implemented them in computer code. I have also created a simple program for visualising the solutions. This chapter provides the reader with the background and motivation for this project. In the **Theory** chapter, I will elaborate further about the Poisson equation, its discrete version, and the theory behind the methods I used to create the solver program. In the **Method** chapter, I will present the technologies and the key methods used, as well as describe their implementation in a concrete manner without being too specific about the code itself. (The source code already contains documentation in the form of comments.) In the **Results** chapter we will look at the resulting solutions from running the solver, in addition to the performance of the program itself, for various parameters. In the **Discussion** chapter I will discuss the results from the solver program, its efficacy, and potential improvements. Lastly, the **Conclusions** chapter provides a summary and some final words regarding the project. **Appendix A** presents an additional method that I have not implemented but will refer to briefly when discussing potential improvements.

# 2 Theory

## 2.1 Poisson's equation

In two-dimensional Cartesian coordinates, Poisson's equation can be expressed as

$$\nabla^2\varphi(x,y) = g(x,y) \ , \tag{2.1}$$

where $g(x,y)$ is a given function, $\varphi(x,y)$is the function sought, and $\nabla^2$ is the Laplace operator. By evaluating the dot product

$$\nabla^2 = \nabla \cdot \nabla = (\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}) \ , \tag{2.2}$$

we get the following expanded form for Poisson's Equation:

$$\frac{\partial \varphi^2}{\partial x^2} + \frac{\partial \varphi^2}{\partial y^2} = g(x,y). \tag{2.3}$$

A *second-order central finite difference* method can be used to approximate the second-order partial derivative of a function $f(x)$ as follows:

$$\frac{\partial^2 f}{\partial x^2} \approx \frac{f(x-dx) - 2f(x) + f(x+dx)}{dx^2} \ , \tag{2.4}$$

for some small distance dx. By applying (2.4) to the Poisson equation (2.3) with respect to both $x$ and $y$, and assuming $dx = dy = h$ , we get the following equation:

$$\nabla^2\varphi \approx \frac{\varphi(x-h) + \varphi(y-h) - 4\varphi(x,y) + \varphi(x+h) + \varphi(y+h)}{h^2} = g(x,y), \tag{2.5}$$

where $h$ is a small step size.

## 2.2 Discretisation and linear algebra

Our next step is to use (2.5) in order to discretise the Poisson equation into a system of linear equations. First, we need to define the domain $\mathcal{D}$ for the functions $\varphi$ and $g$. We define $\mathcal{D}$ to be all points $(x,y)$ for $x,y \in [0,1] \in \mathbb{R}$. The boundary $\Omega$ is a subset of $\mathcal{D}$ containing all points $(x,y)$ where $x = 0, x = 1, y = 0$, and $y = 1$. We call these four sides $\Omega_{X0}, \Omega_{X1}, \Omega_{Y0}, \Omega_{Y1}$ respectively. Now consider a finite set $\mathcal{M}$ containing all points $(x_i, y_j) \in \mathcal{D}$, where $x_i = ih$ ,and $y_j = jh$ ,for $i,j = 0,1,2,\ldots,n-1$. Here, $n$ equals the number of discrete equidistant points along the $x$- and $y$-axis from 0 to 1 inclusive, and $h$ is the mesh size equal to $\frac{1}{n-1}$ . Thus, $\mathcal{M}$ is the set of grid points of a $n \times n$ mesh grid over the domain $\mathcal{D}$, including the boundary $\Omega$. The cardinality of $\mathcal{M}$ is $n^2$. This constitutes our discrete computational domain. Let us now further define

$$\varphi_{i,j} = \varphi(x_i, y_j) \ \ and \ \ g_{i,j} = g(x_i, y_j) \ , \tag{2.6}$$

for $i,j = 0,1,2,\ldots,n-1$. If we substitute (2.6) into (2.5) for all $(x_i, y_j) \in \mathcal{M}$, then after some algebraic manipulation we obtain a system of $n^2$ linear equations of the form:

$$\varphi_{i-1,j} + \varphi_{i,j-1} - 4\varphi_{i,j} + \varphi_{i+1,j} + \varphi_{i,j+1} = h^2 g_{i,j} \ , \tag{2.7}$$

for $i, j = 0, 1, 2, \ldots, n-1$. These are all approximations to $h^2\nabla^2\varphi_{i,j}$. Further algebraic manipulation of (2.7) yields the following system of equations:

$$\varphi_{i,j} = \frac{1}{4}(\varphi_{i-1,j} + \varphi_{i,j-1} + \varphi_{i+1,j} + \varphi_{i,j+1} - h^2 g_{i,j}) \ . \tag{2.8}$$

Observe that the value of $\varphi$ at a mesh point is approximately equal to the average of its four immediate neighbours subtracted by a small term $\frac{h^2}{4}g_{i,j}$. By linear algebra, we can write the linear system (2.7) in matrix-vector form:

$$\mathbb{A}\Phi = \mathbb{b} \ , \tag{2.9}$$

where $\Phi = [\varphi_{0,0} \quad \varphi_{0,1} \quad \varphi_{0,2} \quad \cdots \quad \varphi_{n-1,n-2} \quad \varphi_{n-1,n-1}]^T = [\varphi_{i,j}]^T$ is the vector of $n^2$ unknown variables, $\mathbb{b} = h^2[g_{0,0} \quad g_{0,1} \quad g_{0,2} \quad \cdots \quad g_{n-1,n-2} \quad g_{n-1,n-1}]^T = h^2[g_{i,j}]^T$ is the vector of $n^2$ known constants, and $\mathbb{A}$ is a $n^2 \times n^2$ matrix of coefficients. By evaluating the matrix-vector multiplication $\mathbb{A}\Phi$ we obtain a vector containing the $n^2$ expressions on the left-hand side of Eq. (2.7). Now it is only a matter of solving (2.9). However, before we may begin exploring methods of solving the system of equations, we need to address two important hurdles: How we should deal with the values of $\varphi_{i,j}$ on the boundary $\Omega$, and the size of the coefficient matrix $\mathbb{A}$.

## 2.3 Boundary conditions and the matrix $\mathbb{A}$

Let's examine a value $\varphi_{0,j}$ on the boundary $\Omega_{X0}$. By Eq. (2.8) we obtain the value

$$\varphi_{0,j} = \frac{1}{4}(\varphi_{-1,j} + \varphi_{0,j-1} + \varphi_{1,j} + \varphi_{0,j+1} - h^2 g_{0,j}) \ . \tag{2.10}$$

Notice that computing $\varphi_{0,j}$ involves the value of $\varphi_{-1,j}$. This is problematic because $\varphi_{-1,j}$ lies on the point $(-h, jh)$, which is outside our computational domain $\mathcal{M}$. To rectify this, we need to make special considerations about the boundary values. We will look at two different ways of doing this: Prescribing *Dirichlet-* or *Neumann* boundary conditions.

### 2.3.1 Dirichlet boundary condition

A simple way of solving this dilemma is to prescribe known values of $\varphi_{i,j}$ on the boundary $\Omega$, and then only compute the unknown values of $\varphi_{i,j}$ for all $(x_i, y_j) \in \mathcal{M} \setminus \Omega$. This means that in practice, the indices $i, j$ can only have values in range $1, 2, \ldots, n-2$. As for values of$\varphi_{i,j}$ on the boundary, we can move them over to the right-hand side of (2.9) by subtraction. The condition of prescribing $\varphi$ on the boundary like this, is called a *Dirichlet boundary condition*. When we combine a PDE such as Poisson's equation (2.1), with a boundary condition, we get a *boundary value problem* (BVP). We will later consider several different BVPs, and how to solve them numerically with a computer.

### 2.3.2 Neumann boundary condition

Another type of boundary condition is the *Neumann boundary condition*. Here, we don't prescribe a value of $\varphi_{i,j}$. Instead, we know the value of $\frac{\partial\varphi}{\partial n}$ on the boundary, that is, the partial derivative with respect to the direction pointing normal to the boundary $\Omega$. We will only consider $\frac{\partial\varphi}{\partial n} = 0$ for the rest of this report. Let us again examine a value $\varphi_{0,j}$ on the boundary $\Omega_{X0}$. The direction normal to $\Omega_{X0}$ is $y$, hence we know that

$$\frac{\partial}{\partial y}\varphi(x_0, y_j) = \frac{\partial}{\partial y}\varphi(0, y_j) = 0 \ . \tag{2.11}$$

A *first-order central finite difference* method can be used to approximate the derivative of a function $f(x)$ as follows:

$$\frac{df}{dx} \approx \frac{f(x+dx) - f(x-dx)}{2dx} \ , \tag{2.12}$$

for some small distance $dx$. Thus, by applying (2.12) to Eq. (2.11) we obtain that

$$\frac{\varphi_{1,j} - \varphi_{-1,j}}{2h} = 0 \Leftrightarrow \varphi_{-1,j} = \varphi_{1,j} \ . \tag{2.13}$$

This allows us to compute $\varphi_{0,j}$ by replacing $\varphi_{-1,j}$ with $\varphi_{1,j}$ in Eq. (2.10), obtaining that:

$$\varphi_{0,j} = \frac{1}{4}(0\varphi_{-1,j} + \varphi_{0,j-1} + 2\varphi_{1,j} + \varphi_{0,j+1} - h^2 g_{0,j}) \ . \tag{2.14}$$

Notice how we let $\varphi_{-1,j}$ stay and replaced its coefficient with 0, and $\varphi_{1,j}$ with 2. We can think of $\varphi_{-1,j}$ as a *ghost node* in our computations, one whose value is never directly used.

There is one more thing to consider regarding the use of the Neumann boundary condition. If we *only* have Neumann conditions for the entirety of the boundary $\Omega$, then we need some more information about the function $\varphi$ if we are to find a particular solution. The function $\varphi(x, y)$ can be written on the form $f(x, y) + C$, where $C$ is an arbitrary constant. When we take the derivative of $\varphi(x, y)$, the constant $C$ disappears. Therefore, if we only have Neumann conditions on boundary, then there exists an infinite set of functions for $\varphi(x, y)$ that fit this criterium. However, if we know the value $\varphi_0$ at some point $P$, say the origin for example, then we can unambiguously decide the solution of $\varphi(x, y)$, since we can just add a constant $C$ that makes the solution align with the value $\varphi_0$ at point $P$.

### 2.3.3 Mixed boundary condition

With boundary conditions, we don't have to choose one over the other. It is possible to have both Dirichlet- *and* Neumann conditions. This is rather trivial to implement if the boundary can be partitioned into continuous parts that are non-smooth at the points connecting them, which our square boundary $\Omega$ (conveniently) is. Recall that we partitioned $\Omega$ into four parts, $\Omega_{X0}, \Omega_{X1}, \Omega_{Y0}, \Omega_{Y1}$ for $x = 0, x = 1, y = 0, y = 1$ respectively. These are all straight continuous lines that together form a closed curve $\Omega$ with non-smooth points at the corners (0,0),(1,0),(1,1),(0,1). Each of the four borders can be prescribed either a Dirichlet condition, or a Neumann condition. We only need to check near the border which one it is, so that we may compute the points accordingly as described in sections 2.3.1 and 2.3.2.

### 2.3.4 The size of the matrix of coefficients $\mathbb{A}$

As mentioned earlier, the size of the matrix $\mathbb{A}$ in (2.9) is $n^2 \times n^2 = n^4$. This matrix is sparse, which means that most of its entries are zero. By examining (2.7), we see that each row of $\mathbb{A}$ contains a maximum of five non-zero entries, which is either -4, 1, or in the case of Neumann boundaries, 2. This gives us a storage problem if we want to store the entirety of $\mathbb{A}$ in computer memory. If we want to solve (2.9) for a mesh grid of $1000 \times 1000$ points, and assuming we need only one byte per entry, then the matrix $\mathbb{A}$ would require $1000^4$ bytes $= 10^{12}$ bytes $= 1$ terabyte of memory, most of which is zero! It goes without saying that trying to store the entirety of $\mathbb{A}$ is ludicrous. Without it however, solving (2.9) directly using Gauss elimination becomes difficult. This is where we turn our heads towards using iterative methods for solving linear systems, which is especially useful when the system is large and sparse.

## 2.4 Iterative methods for solving linear systems of equations

Unlike *direct methods* such as Gauss elimination, an *iterative method* starts by approximating the solution, and obtain better and better approximations for every iteration. The number of

computations then depends on the problem, and the desired accuracy. One such method is the *Jacobi method* (Kreyszig, p. 859)[1]. In matrix notation it can be written as

$$\Phi^{m+1} = \mathbb{D}^{-1}(\mathbb{b} + (\mathbb{D} - \mathbb{A})\Phi^{(m)}) \ , \tag{2.15}$$

where $\Phi^{(m)}$ is the approximated solution of (2.9) at iteration $m$, $\mathbb{I}$ is the identity matrix, and $\mathbb{D}$ is the diagonal matrix of $\mathbb{A}$, which in this case only has entries of -4. When we apply (2.15) to our system of equations denoted by (2.7), we get the following Jacobi iteration scheme for our system:

$$\varphi_{i,j}^{(m+1)} = \varphi_{i,j}^{(m)} + \frac{1}{4}[\varphi_{i-1,j}^{(m)} + \varphi_{i,j-1}^{(m)} + \varphi_{i+1,j}^{(m)} + \varphi_{i,j+1}^{(m)} - 4\varphi_{i,j}^{(m)} - h^2 g_{i,j}] \ , \tag{2.16}$$

Note that if $\Phi^{(m)}$ is exactly equal to the solution of (2.9), the expression in the brackets (called the *residual* $R_{i,j}$) of (2.16) will be zero, thereby making $\Phi^{(m+1)} = \Phi^{(m)}$, thus we have reached complete convergence. If $\Phi^{(m)}$ is not equal to the solution however, the residual $R_{i,j}$ constitute our correction of $\Phi^{(m)}$(Hellevik, p. 58)[2].

This method can be improved by substituting the results of previous computations of $\Phi^{(m+1)}$ into the right-hand side of (2.16), achieving a method of *successive corrections*. This method is called the *Gauss-Seidel iteration method*. Essentially what this means for our iteration scheme (2.16) is that the residual $R_{i,j}$ becomes dependent on the approximations from the previous iteration as well as values from the *current* iteration that has been computed up to this point. We can write this new scheme in terms of the residual as follows:

$$\varphi_{i,j}^{(m+1)} = \varphi_{i,j}^{(m)} + \frac{1}{4}R_{i,j} \ , \tag{2.17}$$

where

$$R_{i,j} = [\varphi_{i-1,j}^{(m+1)} + \varphi_{i,j-1}^{(m+1)} + \varphi_{i+1,j}^{(m)} + \varphi_{i,j+1}^{(m)} - 4\varphi_{i,j}^{(m)} - h^2 g_{i,j}] \tag{2.18}$$

Since our goal is to reduce the residual $R_{i,j}$ to zero, our modification of the approximate solution can be considered as *relaxing* its components, thus making the Gauss-Seidel iteration a member of a class of methods called *relaxation methods*.

By introducing an *overrelaxation factor* $\omega > 1$, and multiplying it with the relaxation term $\frac{1}{4}R_{i,j}$, we obtain the so-called *Successive overrelaxation* (SOR) method for Gauss-Seidel:

$$\varphi_{i,j}^{(m+1)} = \varphi_{i,j}^{(m)} + \frac{\omega}{4}R_{i,j} \ . \tag{2.19}$$

It is possible to calculate the theoretical optimal $\omega$ for the Poisson equation in a rectangular domain (Hellevik, p. 58)[2]. We have that

$$\omega = \frac{2}{1 + \sqrt{1 - \rho^2}} \ , \tag{2.20}$$

where

$$\rho = \cos(\pi h) \ , \ h = \frac{1}{n-1} = \textit{mesh size} \tag{2.21}$$

We will consider SOR as our method of choice for solving the discrete Poisson equation for the remainder of this report.

## 2.5 Multigrid preconditioning

*Preconditioning* is used to transform a problem into a form more suitable for numerical solving. One such way, which is quite relevant for our problem, is to solve the equation on a coarser grid, then interpolate the result onto our finer grid and use it as an initial approximation for our iteration process. This is called *multigrid* preconditioning, because the coarser grid can also

be solved using an even coarser grid as initial approximation, and so on, with multiple grids. This recursive way of solving the equation resembles a bottom-up approach, where the first and coarsest grid is simple enough that it can be solved using SOR very quickly regardless of initial approximations. In the next chapter, we will look at how this can be implemented.

# 3 Method

## 3.1 Technology Stack

In this project, the solver application programming interface (API), example-, and the visualisation program, have been written in the C programming language. The reason for this choice, is due in part that it efficiently maps to machine instructions and provide low-level memory access and manipulation. Another part is that the language is cross-platform, with a wide variety of different compilers for different processor architectures and operating systems, making it somewhat easy to create portable code.

All the programs created specifically for this project have been developed and tested on Windows 10 and Manjaro Linux running on x86-64 architecture, as well as macOS running on ARM. To compile the code, the GNU Compiler Collection (GCC) is used, alongside the GNU C Library (glibc) and GNU make. For package management, MSYS2 is used for Windows and Homebrew is used for macOS. On Linux, the natively provided package manager is used. The solver API does not rely on third-party libraries. The example program makes use of POSIX threads (for solving each different BVP concurrently), and GNU-getopt for parsing command-line options. The visualisation demo program uses FreeGLUT, GLU and OpenGL for window management and graphics. Additionally, Gnuplot is used to generate images of the solution, but is by no means necessary in order to use the other programs.

## 3.2 Data structures

The solver API provides a type for creating, solving and saving Poisson BVPs. This type, called *bvp_t*, is an *opaque data type*, meaning that the implementation is hidden from the user, and is only meant to be passed as pointers in conjunction with the solver API's subroutines. When creating a BVP, one may specify the function to be used for prescribing Dirichlet boundaries, the function $g(x, y)$, and Neumann boundaries if there are any. Additionally, one needs to provide the number of desired discrete points $n$ that makes up the $n \times n$ mesh grid. However, the API may choose a different, albeit close, number instead (more on that later).

The solver makes use of double-precision floating-point numbers. The discretised $x$- and $y$-values are stored as arrays, with values ranging from 0 to 1. In order to avoid memory fragmentation, the solution grid is stored as a single array-block of memory. An array of pointers to the columns of the grid is used to access the values using two indices $i$ and $j$. Some extra memory is allocated to prevent out-of-bounds access for ghost nodes. A similar array-structure is used for storing the corresponding values of $g(x, y)$ for every point on the grid. Neumann boundaries are stored as an *int* in the form of a bit field, where the bits represent Neumann condition at $\Omega_{X0}$, $\Omega_{X1}$, $\Omega_{Y0}$, and $\Omega_{Y1}$, from least significant digit and up. For example, 0101 may represent Neumann boundaries at $x = 0$ and $y = 0$.

## 3.3 The iteration process

This project implements a version of SOR for solving the Poisson equation. During every iteration, the solver loops through all unknown points, checks for Neumann boundaries and changes the coefficients accordingly, computes the residual, and relaxes the components. It also checks for convergence after every iteration. When calling the solver subroutine, one must

specify the *relative tolerance* and whether to use multigrids. The relative tolerance is used to decide convergence; when the *relative residual* (sum of absolute residuals divided by sum of absolute values of the result) is less than the relative tolerance, the iteration ends. An example of such a value is $10^{-5}$. If using multigrids, and if the current grid is fine enough (for example more than $100 \times 100$), then the solver will solve a coarser grid with the same parameters, except the discretisation number is $n/2+1$. It then copies and interpolates the result from the coarser grid onto the finer grid, then sends the finer one through the SOR iteration process.

## 3.4 The implementation of multigrid preconditioning

As mentioned in Section 3.2, the solver API may choose a different discretisation number $n$ than the user provides. The API will choose an admissible $n$ as follows:

*If $n$ is sufficiently small, say $< 100$, then it is admissible.*
*Else if $n$ is odd and $n/2+1$ is admissible, then $n$ is also admissible.*
*Otherwise, it isn't admissible, so try $n+1$ instead.*

The reason why we don't want $n$ to be even, is that it simplifies the interpolation step. In short, the values from the coarser grid of size $n/2+1$ can then be evenly distributed on the finer grid from corner to corner in one simple iteration. The remaining points are calculated in three more batches of iterations, each of which calculates the average of the neighbouring points from the previous batches. But before that, if there are any Neumann boundaries, the values of those gets interpolated simply as the average of the values on the boundary. Dirichlet boundaries need not be interpolated, as they are already prescribed. Diagrams that illustrate the basic idea behind the interpolation process developed exclusively for this project, is provided below.

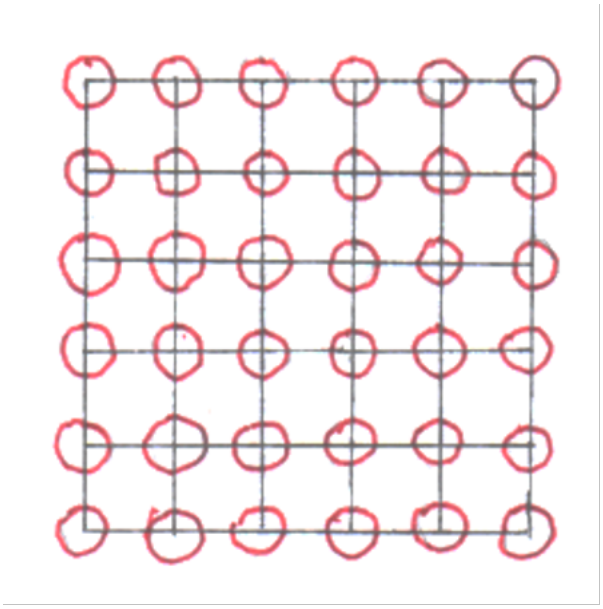


Figure 3.1: A $6 \times 6$ "coarse" grid.

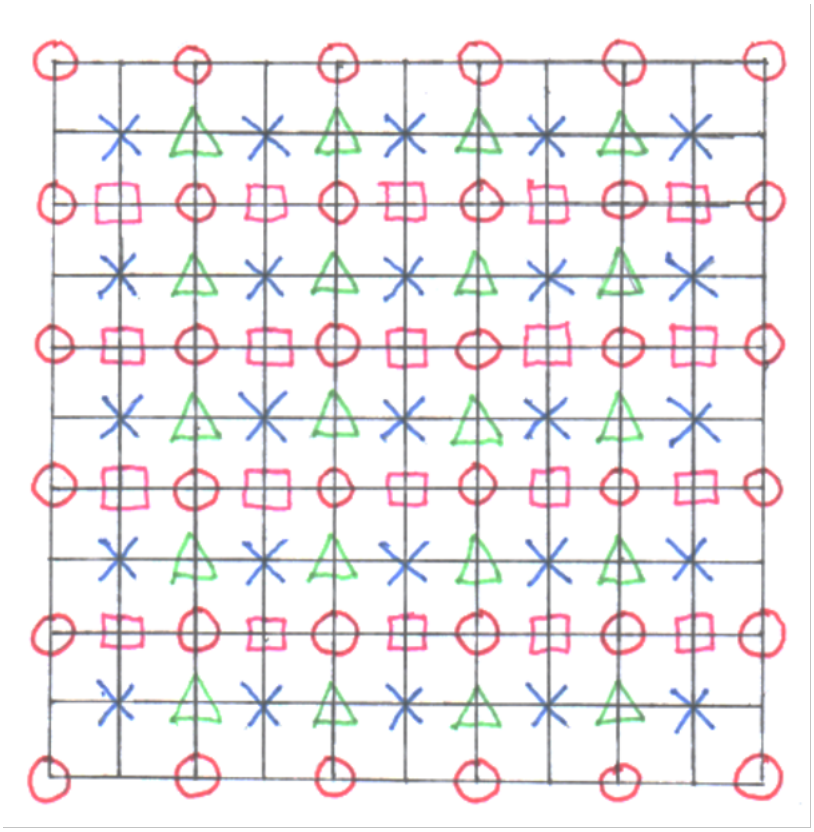


Figure 3.2: A $11 \times 11$ "fine" grid. Observe the even distribution of the points from Fig. 3.1.

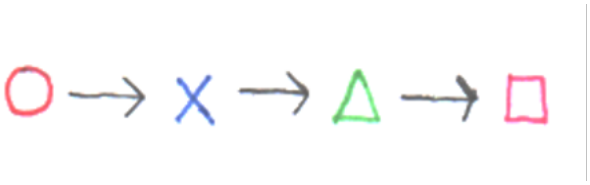


Figure 3.3: The order of dependencies for the interpolated points in Fig. 3.2.

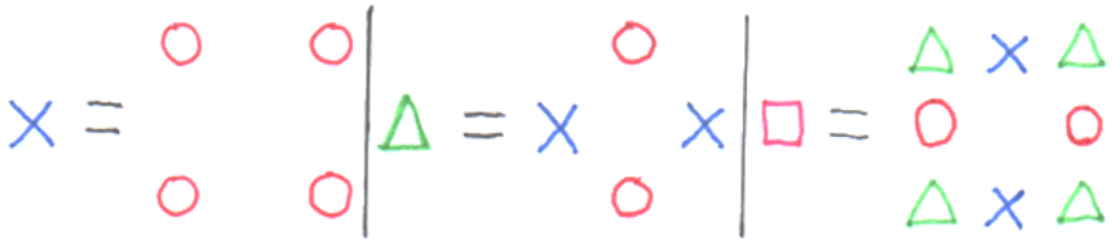


Figure 3.4: The neighbourhood of points that each type of interpolation point depends on.

## 3.5 Example BVPs for testing the solver

For this project, we define five different Poisson boundary value problems for the program to solve.

### 3.5.1 Problem Mixed

This BVP uses Neumann conditions for $x = 0$ and $y = 0$, and Dirichlet boundaries for $x = 1$ and $y = 1$. The boundary function is defined as:

$$\varphi(x, y) = \begin{cases} 1, & y = 1 \\ 0, & otherwise \end{cases}$$

And the function $g(x, y)$ as:

$$g(x, y) = 0$$

### 3.5.2 Problem Dirichlet1

This BVP uses Dirichlet conditions only. The boundary function is defined as:

$$\varphi(x, y) = \frac{1}{4}(x^2 + y^2) \ , \tag{3.1}$$

and the function $g(x, y) = g_1$ as:

$$g(x, y) = g_1(x, y) = 12 - 12x - 12y \ . \tag{3.2}$$

### 3.5.3 Problem Dirichlet2

This BVP also uses Dirichlet conditions only. The boundary function is the same as (3.1). The function $g(x, y) = g_2$ is defined as:

$$g(x, y) = g_2(x, y) = (6 - 12x)(3y^2 - 2y^3) + (3x^2 - 2x^3)(6 - 12y) \ . \tag{3.3}$$

### 3.5.4 Problem Neumann1

This BVP uses Neumann conditions only. The function $g(x, y) = g_1$ is the same as (3.2). We know the value $\varphi(0, 0) = 0$.

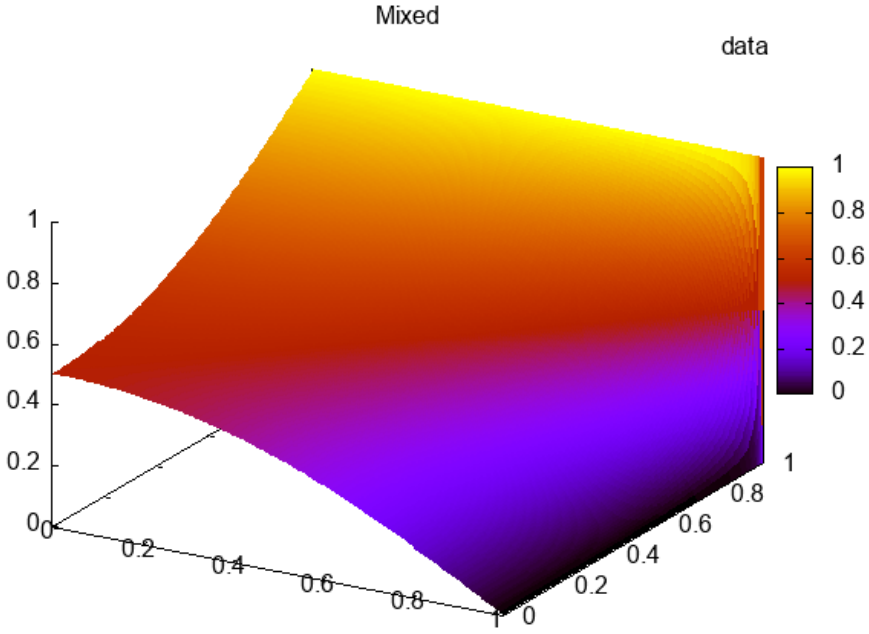
### 3.5.5 Problem Neumann2

This BVP also uses Neumann conditions only. The function $g(x, y) = g_2$ is the same as (3.3). We know the value $\varphi(0, 0) = 0$.

# 4 Results

## 4.1 Solutions

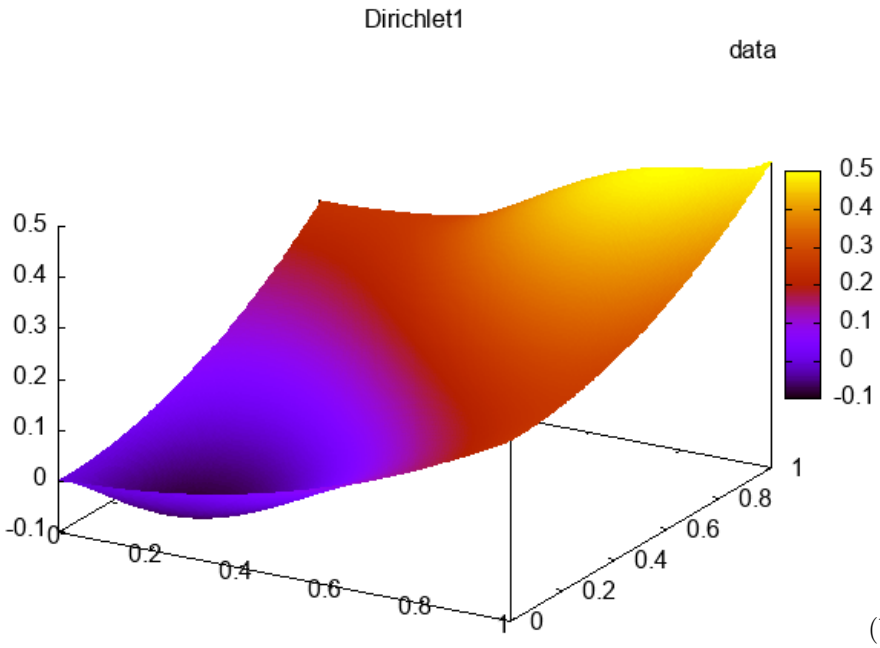
This section presents visualisations of the results from running the example program as follows: `./example -n 100 -s -r 1e-5`. This solves the problems given in section 3.5 for a $100 \times 100$ mesh grid and saves the solution to file. The relative tolerance is set to $10^{-5}$.



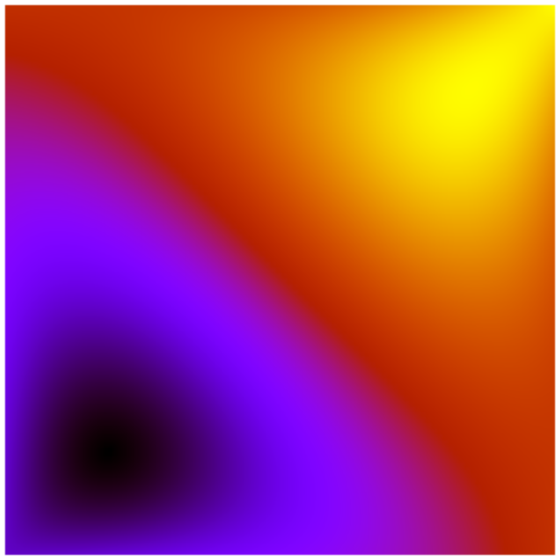

(a) Visualisation of Mixed produced by Gnuplot

(b) Visualisation of Mixed produced by our program

Figure 4.1: The solution to our Mixed problem

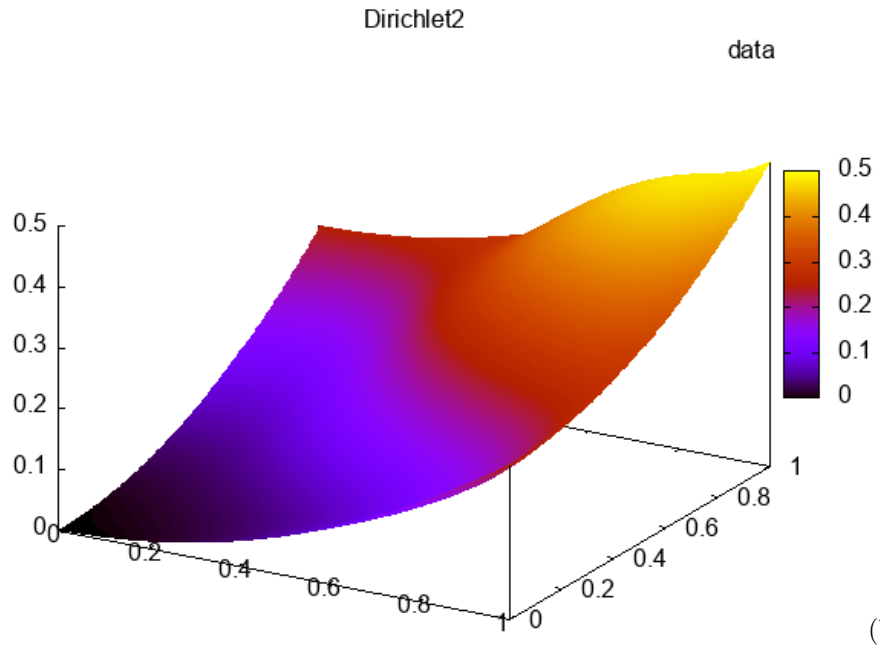



(a) Visualisation of Dirichlet1 produced by Gnuplot

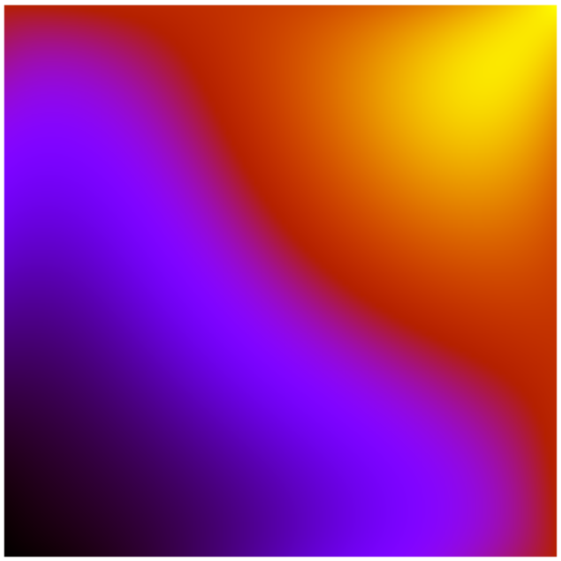


(b) Visualisation of Dirichlet1 produced by our program

Figure 4.2: The solution to our Dirichlet1 problem

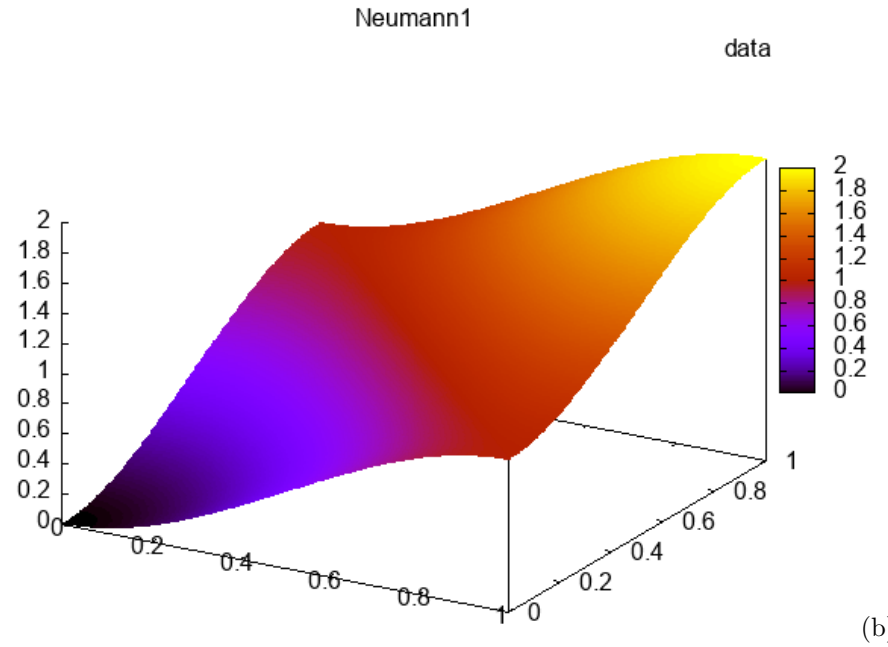



(a) Visualisation of Dirichlet2 produced by Gnuplot

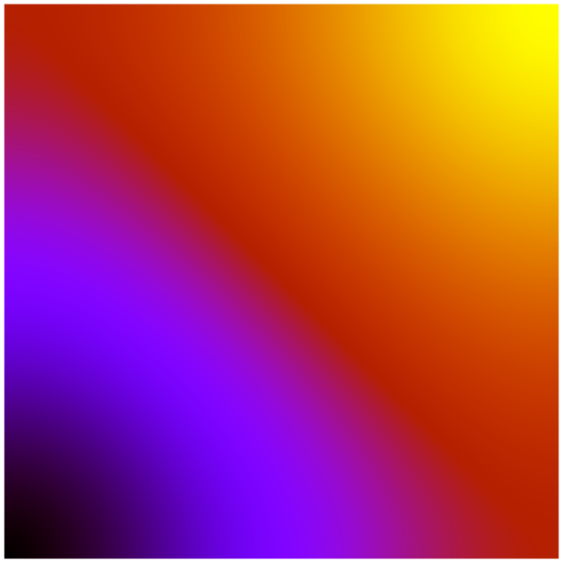


(b) Visualisation of Dirichlet2 produced by our program

Figure 4.3: The solution to our Dirichlet2 problem

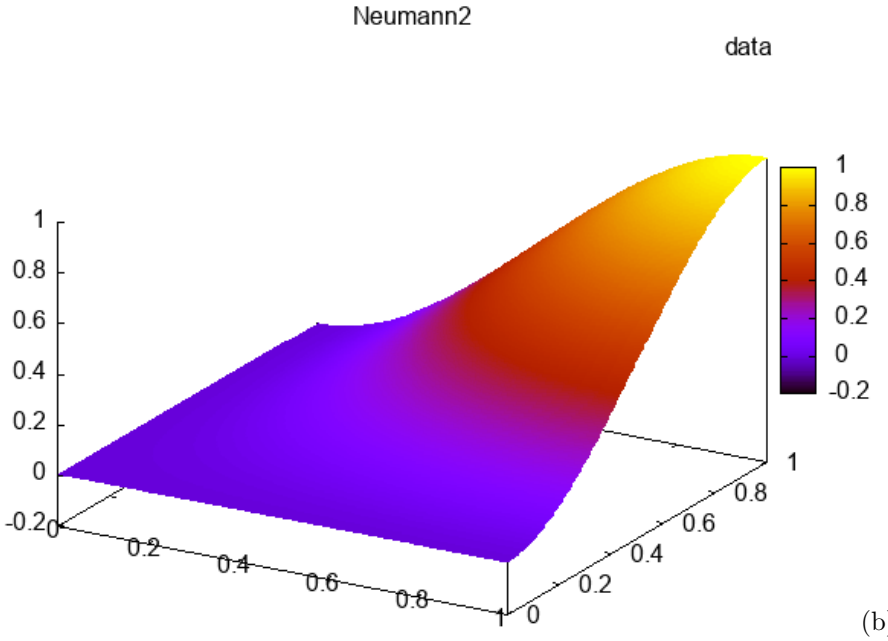



(a) Visualisation of Neumann1 produced by Gnuplot

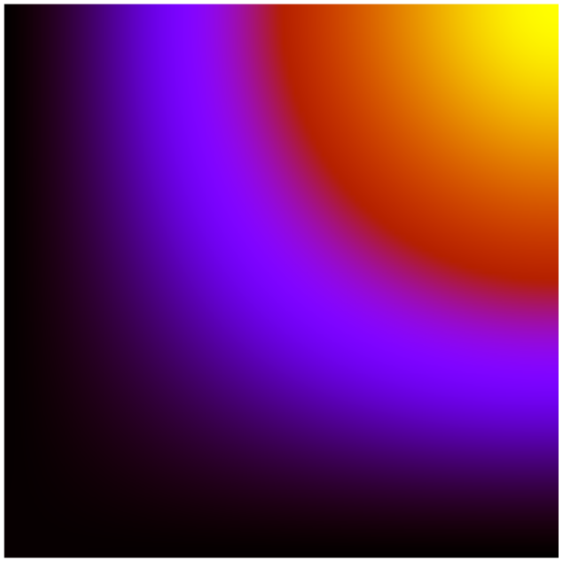


(b) Visualisation of Neumann1 produced by our program

Figure 4.4: The solution to our Neumann1 problem




(a) Visualisation of Neumann2 produced by Gnuplot



(b) Visualisation of Neumann2 produced by our program

Figure 4.5: The solution to our Neumann2 problem

## 4.2 Tests

This section presents visualisations of the results from running the test program. The relative tolerance is set to $10^{-5}$.

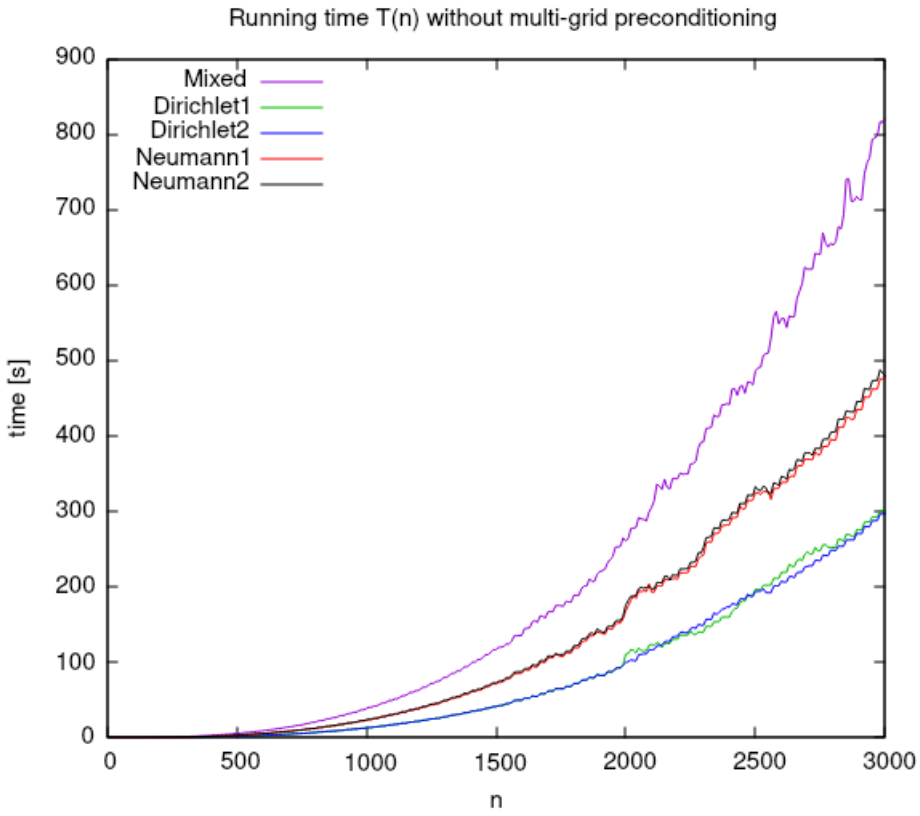



Figure 4.6: Time in seconds spent solving various BVPs for increasing $n$ without multigrid.

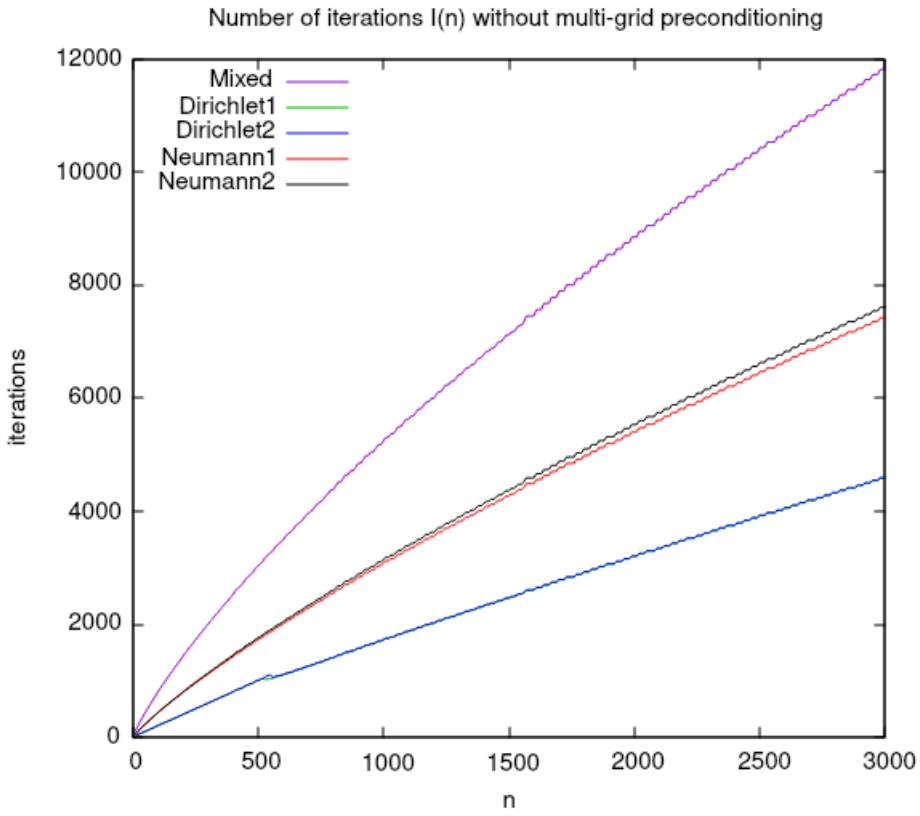



Figure 4.7: Iterations required to solve various BVPs for increasing $n$ without multigrid.

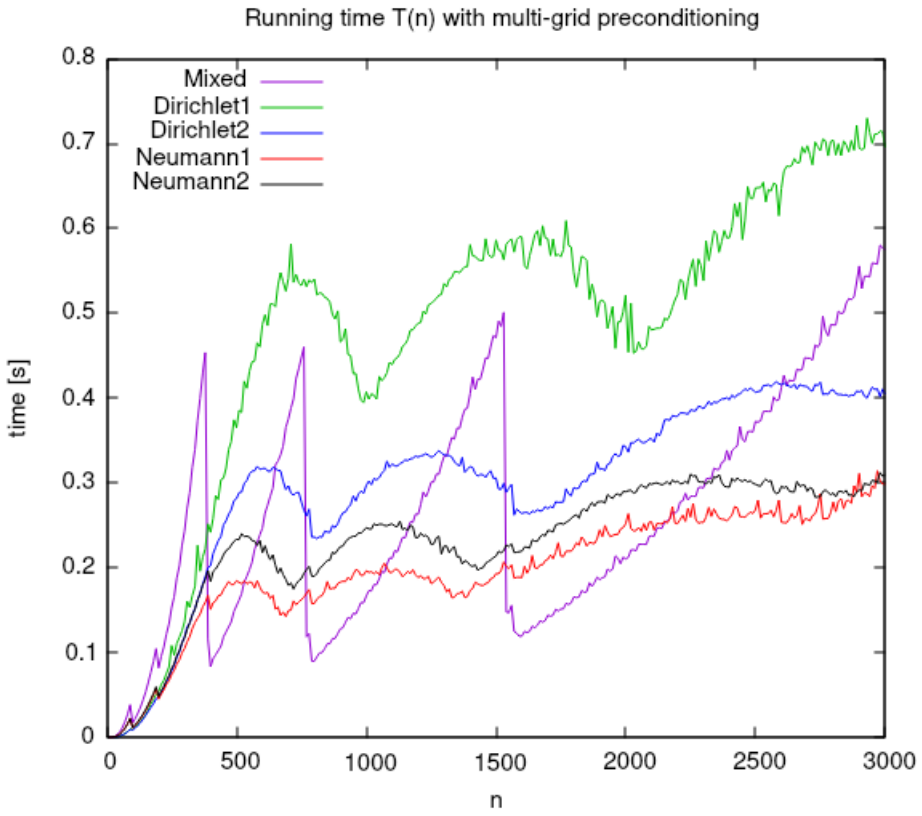



Figure 4.8: Time in seconds spent solving various BVPs for increasing $n$ with multigrid.

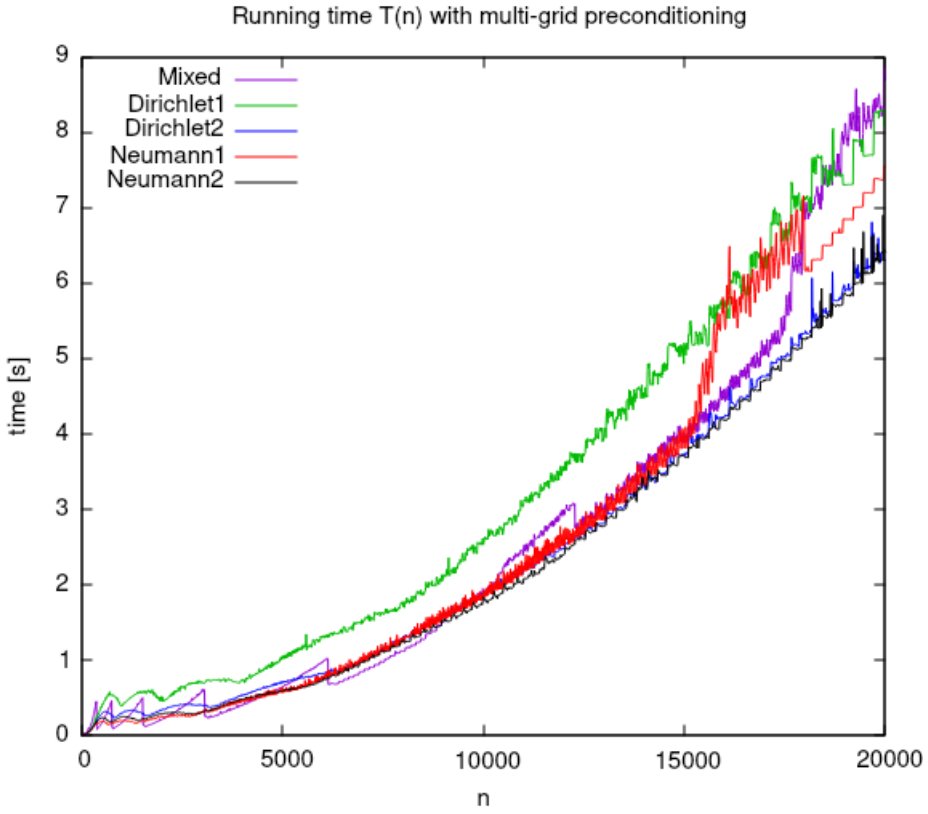



Figure 4.9: Same as figure 4.8 but with $n$ up to $2 \times 10^4$.

# 5 Discussion

Although hundreds of hours have been spent on this project, most of the time went towards creating the solver API and learning C at the same time. The visualisation demo program does not display text nor numbers, but by using a colour plot and rotations in three dimensions, we can still get a good idea about how the solutions look. Notably, the plot produced by the visualisation demo program matches that of the plot produced by Gnuplot well.

## 5.1 The results

By looking at the Figure 4.1a of the solution to our Mixed problem 3.5.1, it becomes abundantly clear that we got the Dirichlet boundary conditions right, where it is equal to 1 for $y = 1$, and 0 for $x = 1$. As expected, there is a noticeable discontinuity in the corner $(1, 1)$ where the two different Dirichlet boundaries meet. The fact that the corner $(0, 0)$ has a value of 0.5, the average between the two, is not surprising. How the problem was defined, is like that of the stationary heat equation, with the side $y = 1$ kept at temperature 1, side $x = 1$ at temperature 0, and the two other sides kept isolated. Heat can flow in and out of the Dirichlet sides, which is why we see a bit of a gradient normal on those sides. For the two other boundaries, by definition do not have a gradient in the normal direction on them, which we can see in Figure 4.1a as well as by studying the colour gradients of Figure 4.1b.

Figures 4.2a and 4.2b shows the solution to our Dirichlet1 problem 3.5.2. We can see that the boundaries match our definition here also. Its 0 at the origin, 0.25 at the corners (0,1) and (1,0), and its 0.5 at corner (1,1). Another thing to note here is that the solution is symmetric for x and y, which correspond well with the functions we used, both for the boundaries and $g(x, y)$. Lastly, there is an apparent antisymmetric relation between the points on either side of the hypotenuse line from corner (0,1) to (1,0), particularly regarding the sign and magnitude of the curvatures. This is to be expected considering the function $g(x, y) = g_1$ is also antisymmetric for the same xy coordinates. As the Poisson equation is defined in Eq. (2.3), the solution function's curvature is equal to the given function $g(x, y)$. That is why the solution we see here has this antisymmetric property.

Figures 4.3a and 4.3b shows the solution to our Dirichlet2 problem 3.5.3. Here we used the same boundary conditions as Dirichlet1, and this is apparent in Figure 4.3a. Unlike Dirichlet1 however, this one does not exhibit the antisymmetric property for the curvatures. This is unsurprising, since the function $g(x, y) = g_2$ used here is not antisymmetric like the other one. However, it is symmetric for x and y, which we can clearly see in both figures, mostly in Figure 4.3b.

Figures 4.4a and 4.4b shows the solution to our Neumann1 problem 3.5.4. This one has only Neumann boundary conditions, so we have little preconceived idea about what it should look like. One thing we know, is that the solution should be equal to 0 at the origin, which we can see in Figure 4.4a. Another thing we know, is that the gradient normal to the boundary should be 0 at the boundary, and this is especially apparent by studying the colour gradients of Figure 4.4b. Lastly, we know that the function used for $g(x, y) = g_1$ has an antisymmetric property. If we look closely at Figure 4.4a, we can see this antisymmetric property for the sign of the curvature on either side of same hypotenuse as for Dirichlet1. It is also symmetric for x and y, due to $g(x, y)$.

Figures 4.5a and 4.5b shows the solution to our Neumann2 problem 3.5.4. Imminently, we see

the lack of the antisymmetric property we saw in Neumann1, due to $g(x, y) = g_2$ lacking this property as well. We know that it should be zero at the origin, which we can see in Figure 4.5a. By studying the colour gradient in Figure 4.5b near the boundaries, we can confirm that the Neumann condition holds. Lastly, there is a symmetry for x and y, which is caused by our choice of $g(x, y)$. Notably, the colour plot from our visualisation program differs slightly from that of Gnuplot. That is, Gnuplot doesn't seem to use black in this one, even though it theoretically should for the lowest values.

## 5.2 The efficacy of the solver

When assessing the efficacy of the solver program, we take both its *correctness* as well as its *efficiency* into account. In the previous section we focused only on the correctness by looking at how the results conform to our knowledge about the equations and parameters used. We concluded for the various different problems that they each conform reasonably well with our expectations, so it is not unsensible to assume that the solver is sufficiently correct, at least for our specific cases. While it is vitally important for the solver to provide solutions that are correct, it is also of great practical importance for the solver to be efficient as well. Engineers are interested in programs that don't use more computational power than necessary, as this saves time *and* money. In order to benchmark the efficiency of the solver, one can run the solver for the BVPs in section 3.5 using increasingly larger $n$, and record the time and iterations required for each problem.

Figure 4.6 shows the time in seconds it took to solve the various different BVPs *without* using multigrid preconditioning. We know that as $n$ gets larger, the number of unknowns scales with $n^2$. Therefore, we can expect the amount of computations to scale similarly or more than $n^2$. Unlike direct methods, with iterative methods it is difficult to predict the number of required iterations in advance, as this depends on the problem and the desired accuracy. If we assume that the number of iterations required to reach convergence scales with $n$, then the expected running time for the solver is in the order of $\mathcal{O}(n^3)$. This is not an unreasonable assumption considering what we can see in Figure 4.7. A potential improvement for predicting the run time may require a more thorough analysis using curve-fitting methods.

Figure 4.8 shows the time in seconds it took to solve the various different BVPs *with* multigrid preconditioning. It's apparent that the multigrid preconditioning method is quite powerful, resulting in a running time less than one second for $n = 3000$. Compare this with the hundreds of seconds required without it in Figure 4.6. It does lead to some strange behaviour, where solving for a large value $n$ is more efficient than solving for a significantly smaller $n$ in some cases. This strange behaviour is not so apparent for very large $n$ as seen in Figure 4.9, where the running time averages out to be increasing with $n$. Here it is important to consider the fact that the processor needs to move around increasingly larger amounts of data between memory and internal cache, which bottlenecks the program significantly in comparison to running the program for lower values of $n$. Near $n = 2 \times 10^4$, the processor needs to move around several gigabytes of data every iteration. If it wasn't for this bottleneck, we may have gotten a near linear average running time or perhaps an average running time of $\mathcal{O}(n \log n)$, from what we can see in Figure 4.9.

## 5.3 Potential improvements

It would be ideal for the visualisation demo program to be able to display numbers along the axes, as well as having a graphical interface for choosing which files to import the solutions from. The solver could also be further improved. The SOR iteration implementation checks for every point whether it is on the boundary or not, so it may update the coefficients accordingly. This leads to short and simple code, but also needlessly many checks inside the boundary. It would be

more optimal for the algorithm to process each boundary separately first, then the inner points afterwards, without needing to make checks or subroutine calls inside the iteration loop. This would require longer and more complicated code, and the GNU C Compiler already optimises the code quite a bit, so it strictly isn't necessary unless one absolutely needs to squeeze out as much performance as possible.

As described in Appendix A, it is possible to parallelise the Gauss-Seidel iteration by concurrently updating points that are independent of each other. This can be done with minimum overhead by providing each thread a range of columns of points to be responsible for, letting them first update the red points until all of them are finished, then update the black points. A mutual exclusion lock can be used to let each thread add the relative residuals to a common variable at the end of every iteration. A *synchronisation barrier* can be used to synchronise the threads, making them essentially rendezvous at the barrier after updating all the red points, then another barrier after updating all the black points. The POSIX standard specifies such a barrier[3]. This could speed up the solver by utilising multiple processor cores at the same time, in addition to utilising more of the processor's total cache, minimising the bottleneck discussed in the previous section.

# 6 Conclusions

During this project, I learned a lot about methods for solving the discrete Poisson equation, and got hands-on experience implementing them in code. I managed to come up with my own way of implementing multigrid preconditioning to significantly speed up the solver, and I'm very satisfied with the result. I also got some experience in creating visualisations of the solutions using OpenGL and Gnuplot. There is no doubt in my mind that the knowledge and practical experience I procured during this project will be highly relevant and useful for a future career in engineering and software development. As for the source code itself, I will be releasing it as completely free and open source software, and hopefully it will be useful to somebody.[4]

# Bibliography


[1] Kreyszig. *Advanced Engineering Mathematics, 10th Edition*. Wiley, 2011.

[2] Leif Rune Hellevik. Numerical methods for engineers. https://folk.ntnu.no/leifh/teaching/tkt4140/._main057.html, 2020. Downloaded: 23-04-2021.

[3] Oracle. Using barrier synchronization. https://docs.oracle.com/cd/E19253-01/816-5137/gfwek/index.html, 2010. Downloaded: 8-05-2021.

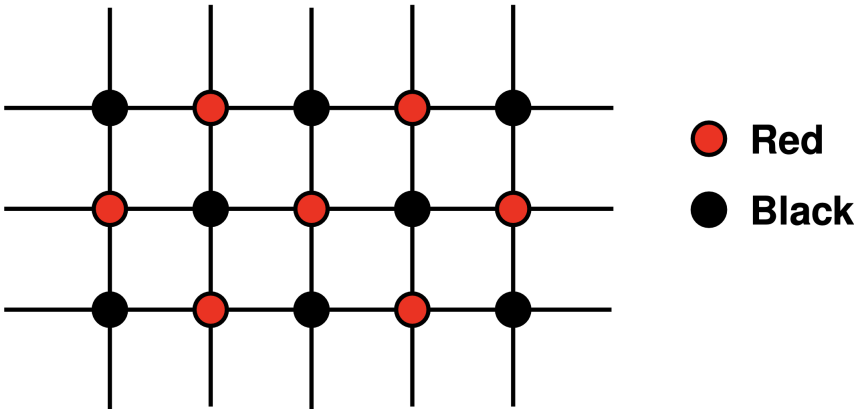
[4] Jørgen R. Høstmark. Poisson-solver. https://github.com/jorhostm/poisson-solver, 2021.

[5] MIT. Solution methods: Iterative techniques, lecture 6. https://ocw.mit.edu/courses/aeronautics-and-astronautics/16-920j-numerical-methods-for-partial-differential-equations-sma-5212-spring-2003/lecture-notes/lec6_notes.pdf, 2003. Downloaded: 18-04-2021.

# Appendix A

## Red-Black Gauss-Seidel

Unlike the Jacobi iteration method mentioned in Section 2.4, the Gauss-Seidel iteration method is successive in nature, due to its dependency of the order it updates the values. At first glance it may seem like a difficult task to parallelise this method without a huge amount of inter-thread communication, which defeats the entire purpose of parallelisation to begin with. However, further study of the iteration-scheme for Gauss-Seidel (2.17) and (2.18) reveals that a point $(x_i, y_j)$ depends only on $(x_{i-1}, y_j)$, $(x_i, y_{j-1})$, $(x_{i+1}, y_j)$, and $(x_i, y_{j+1})$, but not $(x_{i-1}, y_{j-1})$, $(x_{i-1}, y_{j+1})$, $(x_{i+1}, y_{j-1})$, and $(x_{i+1}, y_{j+1})$. This means that we can partition our computational domain $\mathcal{M}$ into two disjoint subsets $\mathcal{R}$ and $\mathcal{B}$, that is $\mathcal{R}, \mathcal{B} \subset \mathcal{M}$ and $\mathcal{R} \cap \mathcal{B} = \emptyset$. The points in $\mathcal{R}$ (red) only depends on the points in $\mathcal{B}$ (black), and vice versa. A diagram of the red and black points from sets $\mathcal{R}$ and $\mathcal{B}$ respectively, are shown below.




Red-black Gauss-Seidel[5]

This is called the red-black Gauss-Seidel method, because the red points may be updated independently of each other in parallel, then the black points in the same manner afterwards.